Table of contents

Abstract. Dotty-cps-async is an open-source package that consists of scala macro, which implements generic async/await via monadic cps transform and library, which provide monadic substitutions for high-order functions from the standard library. It allows developers to use direct control flow constructions of base language instead of monadic DSL for various applications. Behind well-known async/await operations, the package provides options for transforming high-order function applications, generating call-chain proxies, and automatic coloring

Project Paper: Embedding generic monadic transformer into Scala

Can we return monadic programming into mainstream?

Ruslan Shevchenko[0000−0002−1554−2019]

ruslan@shevchenko.kiev.ua

1 Introduction

One of the barriers during industrial adoption of the Scala language is an unnecessary high learning curve. The tradition of using embedded DSL instead of base language leads to a situation when the 'cognitive load' of relatively simple development tasks, such as querying an extra resource, is higher than in mainstream languages. A programmer cannot use control flow constructions of base language but should learn a specific DSL and use a suboptimal embedding of this DSL, usually within monadic for comprehensions. Therefore, developers who are proficient in java or typescript cannot be immediately proficient in scala without additional training.

Can we provide a development environment that gives the programmer an experience comparable to the state-of-the-art mainstream back-end programming? Dotty-cps-async intends to be an element of the possible answer. It provides the way to embed monadic expressions into base scala language using well-known async/await constructs, existing for nearly all mainstream programming languages. Although the main idea is not new, dotty-cps-async provides behind well-known interfaces a set of novel features, such as support of the generic monads, transformation of high-order function applications, generation of call-chain proxies, and automatic coloring.

The package is open-source and can be downloaded from github repository `https://github.com/rssh/dotty-cps-async`.

2 Embedding generic monadic cps transform into Scala.

Dotty-cps-async implements a similar to scala-async[7] interface based on optimized monadic cps transform. It is implemented as scala macros and provides a simple generic interface with a well-known async/await signature, slightly changed to support monad parametrization:

```
def async[F[_]][T](using m: CpsMonad[F])(T): F[T]
```

Where inside async block we can use an await macro:

```
def await[G[_],T](x:G[T])(using CpsAwaitable[G]):T
```

Note that F and G can be different; if the given instance of CpsMonadConversion morphism from $F[\_]$ to $G[\_]$ is defined in the current scope, then await[F] can be used inside async[G].

Underlying source transformation is an optimized version of monadification [4], similar to translating terms into continuations monad[8]. This translation is limited to the code block inside an async argument.

monad type parameter is represented as typeclass with the next interface:

```
trait CpsMonad[F[_]] {
  def pure[A](v:A): F[A]
  def map[A,B](fa:F[A])(f: A⇒B): F[B]
  def flatMap[A,B](fa:F[A])(f: A⇒F[B]): F[B]
}
```

Optionally extended by error generation and handling operations:

```
trait CpsTryMonad[F[_]] extends CpsMonad[F] {
  def error[A](e: Throwable): F[A]
  def flatMapTry[A,B](fa:F[A])(f: Try[A] ⇒ F[B]):F[B]
}
```

We will use notations *F.op* as shortcut for appropriative operation over monad typeclass for F. $C_F[code]$ is a translation of code *code* in the context of F.

Let us recap the basic monadification transformations adopted to scala controlflow construction:

- trivia: $C_F[t] = F.pure(t)$ if $t \in Constant \vee t \in Identifier$
- sequential composition: $C_F[\{a; b\}] = F.flatMap(C_F[a])(\_ \Rightarrow C_F[b])$
- variable definition: $C_F[val a = b; c] = F.flatMap(C_F[a])(a' \Rightarrow C_F[b_{x/x'}])$
- condition: $C_F[if\ a\ then\ b\ else\ c] =$

$$F.flatMap(C_F[a])(a' \Rightarrow if\ (a')\ then\ C_F[b]\ else\ C_F[c])$$

- match: $C_F[a\ match\{\ case\ r_1 \Rightarrow v_1 \ldots r_n \Rightarrow v_n\}] =$

$$F.flatMap(C_F[a])\{a' \Rightarrow match\{case\ r_1 \Rightarrow C_F[v_1] \ldots r_n \Rightarrow C_F[v_n]\}\}$$

- while: $C_F[while(a)\{b\}] =$

$$\begin{array}{l}\{\\ def\ tmpFun(x : F[Boolean]) : F[Unit] = \\ \quad F.flatMap(x)\{c \Rightarrow \\ \quad\quad if(c)\ then\ F.flatMap(C_F[b])(\_ \Rightarrow tmpFun(C_F[a]))\ else\ F.pure(()) \\ \quad \} \\ tmpFun(C_F[a])\end{array}$$

– try/catch: $C_F[try\{a\}\{catch\, e \Rightarrow b\}\{finally\, c\}] =$

$$
\begin{array}{l}
F.flatMap(\\
\quad F.flatMapTry(C_F[a])\{ \\
\quad\quad case\, Success(v) => F.pure(v) \\
\quad\quad case\, Failure(e) => C_F[b] \\
\quad \} \\
)\{x \Rightarrow F.map(C_F[c], x)\}
\end{array}
$$

– throw: $C_F[throw\, ex] = F.error(ex)$
– lambda function: $C_F[a \Rightarrow b] = a \Rightarrow C_F[b]$. Note, that type of lamba function is changed after this transformation.
– functional application: $C_F[f(a)] =$
  - $F.flatMap(C_F[a])(x => f(x))$ if x is non-functional type
  - $F.flatMap(C_F[a])(x => C_F[f](x))$ if x is a lamba-function which can be transformed inline.
  - $F.flatMap(C_F[a])(x => f'(x))$ where f' is an external-provided shifted variant of f. The mechanism for definition and substitution of shifted functions is described in 2.2.
– await: $C_F[await_G(a)] =$
  - a if $F == G$ and F is not context-depended.
  - $F.adoptContext(a)$ if $F == G$ and F evaluation is context-depended.
  - $CpsMonadConversion[F, G](a)$ if $F == G$ and F

Implementation is differ from basic transformation, by few optimizations:

– each translation is specializated for cases when transformations of some subterms are trivial. For example rules for if taking into account optimizations will looks like:
  - $C_F[if\, a\, then\, b\, else\, c] =$
  - $F.flatMap(CF[a])(v => if(v) then\, C_F[b]\, else\, C_F[c])$ if $C_F[A] \neq F.pure(a) \wedge (C_F[b] \neq F.pure(b) \vee C_F[c] \neq F.pure(c))$
  - $if\, a\, then\, C_F[b]\, else\, C_F[c]$ if $CF[a] = F.pure(a) \wedge (C_F[b] \neq F.pure(b) \vee C_F[c] \neq F.pure(c))$
  - $F.pure(if\, a\, then\, b\, else\, c)$ if $CF[a] = F.pure(a) \wedge C_F[b] = F.pure(b) \wedge C_F[c] = F.pure(c)$
– few sequential blocks with trivial CPS transformations are merged into one:

$$F.flatMap(F.pure(a))(x => F.pure(b(x)) = F.pure(b(a))$$

In the resulting code, the number of monad bounds is usually the same as a number of awaits in the program, which made performance characteristics of code, written in a direct style and then transformed to monadic, the same, as in monadic style, writtend by hands.

2.1 Monads parametrization

Async expressions are parameterized by monads, which allows the CPS macro to support behind the standard case of asynchronous processing other more exotic applications, such as processing effects[16], [2], logical search[10], or probabilistic programming[1].

Let's look on the next example:

```
val prg = async[[X] =>> Resource[IO,X]] {
  val input = open(Paths.get(inputName),READ)
  val output = open(outputName,WRITE, CREATE, TRUNCATE_EXISTING)
  var nBytes = 0
  while
     val buffer = await(read(input, BUF_SIZE))
     val cBytes = buffer.position()
     await(write(output, buffer))
     nBytes += cBytes
     cBytes == BUF_SIZE
  do ()
  nBytes
}
```

Here inside async, we construct an async expression for monad [X] =>> Resource[IO,X] which represent an abstraction over computations with resources acquiring and releasing logic.

Analogical expression without async/await will look as

```
(
 for{
  input <- open(Paths.get(inputName),READ)
  output <- open(outputName,WRITE, CREATE, TRUNCATE_EXISTING)
 } yield (input, output)
).evalTap{ case (input, output) =>
   var nBytes = 0
   def step(): IO[Unit] = {
     read(input, BUF_SIZE).flatMap{ buffer =>
       val cBytes = buffer.position()
       write(output, buffer).flatMap{ _ =>
          nBytes += cBytes
          if (cBytes == BUFF_SIZE)
             step()
          else
             IO.pure(())
       }
     }
   }
   step().map{ _ => nBytes }
}
```

The next example illustrating a monadic representation of combinatorical search. Monad [X] =>> ReadChannel[Future,X] represent a csp-like channel[9],

where monadic combinators applying the functions over the stream of a possible states.

```
def putQueen(state:State): ReadChannel[Future,State] =
  val ch = makeChannel[State]()
  async[Future] {
    val i = state.queens.length
    if i < N then
      for{ j <- 0 until N  if !state.isBusy(i,j) }
        ch.write(state.put(i,j))
    ch.close()
  }
  ch

def solutions(state: State): ReadChannel[Future,State] =
  async[[X] =>> ReadChannel[Future,X]] {
    if(state.queens.size < N) then
      val nextState = await(putQueen(state))
      await(solutions(nextState))
    else
      state
  }
```

Here we see one async[Future] in putQueen which spawns a concurrent process for enumerating the next possible steps in N-Queens solution, and solution function recursivelly explore all possible steps.

The computation is directed by reading from the stream of solutions. The process is switched to advance for each state after writing an element to the channel (await is hidden inside ch.write inside for loop).

ch.write is defined in ReadChannel[F,A] as

```
transparent inline def write(inline a:A): Unit =
    await(awrite(a))(using CpsMonad[F])
```

transparent inline macros in scala are substituted in code at the same compiler phase before enclosing macro, so async code transformer process this expression in for loop instead ch.write.

In such way solutions(State.empty).take(2) will return the first two solutions without performing a breadth-first search.

Note, that logical search can not be represented using async/await over one-shot continuations semantics, since ReadChannel monad processing involve multiple values.

2.2 Translation of high-order functions

Support of cps-transformation of hight-order functions is important for functional language, because it allows using await expression inside loops and in arguments of common collection operators. As example, in previous section await inside for loop was used for asynchronious channel write. Using await inside hight-order function enable idiomatic functional style, such as

```
val v = cache.getOrElse(url, await fetch(url))
```

Local cps transform change the type of a lambda function. If the runtime platform supports continuations, we can keep the shape of the arguments in application unchanged by defining 'monad-escape' function transformers, which can restore the view of $cps(f) : A \Rightarrow F[B]$ back to $A \to B$.

But for platform whithout continuation support, high-order functions from other module is a barrier for local async transformations. For those runtimes and for cases when semantic of monad does not allow us to build such escape function, doty-cps-async implements limited support of hight-order functions. Macro performs a set of transformations, which allows developers to describe the substitution for the origin high-order function in their code.

Let us have a first-order function: $f : A \Rightarrow B$ which have form $\lambda x : code_f(x)$ and high-order method $o.m : (A \Rightarrow B) \Rightarrow C$. For simplicity, let's assume that o is reference to external symbol and not need cps-transformation itself, since we want to show only function call transaltions here. Async transformation transform $code : X$ into $cps(code) : F[X]$, where F is our monad.

Let us informally describe a set of transformations used to translate function call:

- $cps(code_f)$ have form $F.pure(code_f)$. We can leave the call unchanged in such a case because no cps transformation was needed: $cps(o.m(f)) = F.pure(o.m(f))$
- B have form $G[B']$, where G is compatible with F (i.e. exists monad conversion $G[\_] \Rightarrow F[\_]$). In such case it is possible to reshape function arguments, to keep the same signature to receive $cps(o.m(f)) =$

$$F.pure(o.m(\lambda x : A \Rightarrow \text{CpsMonad[F].flatMap}(cps(code_f(x)))(identity)))$$

- Exists given instance of marker typeclass AsyncShift[O], which provide a substitution methods in one of the forms:
  - m[F](f: CpsMonad[F], o:O)(f: A=>F[B]) Then we can substitute $o.m(f)$ to the call of

    $$\text{summon[AsyncShift[O]].m[F](summon[CpsMonad[F]], o)}(x => cps(code_f(x))$$

  - m(o:O, f: A=>F[B]) - substitute to

    $$\text{summon[AsyncShift[O]].m(o)}(x => cps(code_f(x))$$

  Such substitutors for the most of high-order functions from Scala standard library is supplied with dotty-cps-async runtime. Also developers can provide their own substitution for third-party libraries. The return type of substituted function can be:
  - C, the same as the origin
  - $F[C]$ origin return type wrapped into the monad.
  - CalChainAsyncShiftSubst[F,C,F[C]]. This is a special marker interface for call chain substitution, which wich will be described later.

– Exists method in O with name m_async or mAsync which accept shifted argument $f : A \Rightarrow F[B]$. The conventions for the return type are the same as in the previous case. This case is helpful for the fluent development of API, which is accessible in both synchronous and asynchronous forms.
– If none of the above is satisfied, the macro generates a compile-time error.

These rules are extended to multiple parameters and multiple parameters list, assuming that if we have one high-order async parameter, then all other parameters should also be transformed, having only one representation of the asynchronous method.

2.3 Call-chain substitutions

As shown in previous section, one of the possible variant of return method of substituted high-order function is
CallChainAsyncShiftSubst[F[_],B,F[B]]. The developer can use this substitution when he/she wants to delay applying $F[\_]$ until the call of all the methods in the call chain.

For example, let's look on the next block of code:

```
for { url <- urls if await(score(url)) > limit)
    yield await(fetchData)
```

wich is desugared as

```
urls.withFilter(
       url => await(score(url)) > limit
     ).map(url => await(fetchData))
```

The programmer expects that list of URL-s will be iterated once. However, if the result of withFilter has form F[List.WithFilter], two iterations are performed - one for filtering list of URLs and the other over the filtered list to perform fetching data. User objects for call-chain substitution can accumulate the sequence of high-order functions in one batch and perform iteration once. After transform this block of code will be looks as:

```
summon[AsyncShift[List[String]].withFilter[F](urls,m)(
      url => m.map(score(url))(x=>x>limit)
    )              // CallChainAsyncShiftSubst[F, WithFilter,F[A]]
   .mapAsync(url => fetchData)     // function added to builder
   ._finish()                      // finally eval all.
```

2.4 Automatic coloring

Automatic coloring is the way to free the developer from writing boilerplate await statements. Since most industrial code is built with some asynchronous framework, await expressions often situated literally in each line. Those expressions do not carry out business logic; in general, when writing code, we should not care how an object is coming to code, synchronously or asynchronously, the

same as we do not care how memory to our objects should be allocated and deallocated. Exists some relatively rare optimization points, where low-level information is valuable, but most of the time, we prefer to think in more high-level terms than memory or concurrency management. I.e. if developer writing a core of a web-server, than concurrency low-level details is important. During writing a business logic using some low-level system framework, we can expect that framework give us a reasonable generic concurrency model and abstract from manual coloring.

We can provide implicit conversion from $F[T]$ to T. Can we make such conversion safe and preserve semantics with automatic coloring? It is safe when $F[\_]$ is a cached monad with eager evaluation, such as $Future$. We can extend such conversion to monads, which can provide memoization of execution, by embedding the memoization into the transformation of val definitions.

Let we have block of code { **val** v =expr; $tail_v$ }, $expr$ return value of type $F[T]$ and exists CpsMemoization[F] with method apply[T](F[T]):F[F[T]].

Cps transformer can check the variable type and rewrite this to.

$$\text{summon[CpsMonad[F]].flatMap(CpsMemoization[F]}(expr))(v1 \Rightarrow cps(tail_{v1}))$$

Implicit conversions often criticized as unsafe technique, which can be a source of bugs and maintability problems. In our case, uncontrolled usage of such conversion can broke semantics of building complex effects, where some building parts can be automatically memoized. To prevent such situation, dotty-cps-async implement preliminary analysis of automatically generated conversion, which emit errors when detecting potentially unsafe usage.

To make transformation safe, we should check that developer cannot pass memoized value to API, which expects a delayed effect. Preliminary analysis ensures that all usages of memoized values are in synchronous contexti by forsing the next rules:

- If some variable is used only in a synchronous context (i.e., via await), the macro will color it as synchronous (i.e., cached if used more than once).
- If some variable is passed to other functions as effect - it is colored as asynchronous (i.e., uncached).
- If the variable is simultaneously used in synchronous and asynchronous contexts, we cannot deduce the programmer's intention, and the coloring macro will report an error.
- If the variable, defined outside of the async block, is used in synchronous context more than once - the macro also will report an error.

Behind providing implicit conversion, automatic coloring should also care about value discarding: expressions that provide only side-effects are not an assignment to some value but discarded. When we do automatic coloring, the monad with side-effect generation becomes the value of an expression. So, we should also transform statements with value discard to insert awaits there. Dotty-cps-async interfaces has a ValueDiscard[T] typeclass. The statement inside async block can discard value of type T only if exists implementation

of ValueDiscard[T] interfaces: in such case macro transforms value discard into summon[ValueDiscard[T]].discard(t). A special marker typeclass AwaitValueDiscard[F[T]] is used when this value discard should be a call to await.

If we will apply automatic coloring to our example with copying file, we will see that difference between synchronous and asynchronous code become invisible.

```
val prg = asyncScope[IO] {
  val input = open(Paths.get(inputName),READ)
  val output = open(outputName,WRITE, CREATE, TRUNCATE_EXISTING)
  var nBytes = 0
  while
      val buffer = read(input, BUF_SIZE)
      val cBytes = buffer.position()
      write(output, buffer)
      nBytes += cBytes
      cBytes == BUF_SIZE
  do ()
  log.info(s"transformed ${nBytes} from ${inputName} to ${outputName}")
  nBytes
}
```

3 Related work

The idea of 'virtual' program flow encapsulated in a monad is tracked to[3], which become a foundation for Haskell concurrent library. Later `F#` computation expressions were implemented as further development of do-notation. Furthermore, `C#` moves async/await from virtual monadic control-flow to 'normal control-flow,' which becomes a pattern for other languages[15]. [11] provides an overview of computation expression usage in different areas.

Generic monadic operation pairs [reify/reflect] and links between monadic and cps transformations described in [5].

In scala land, the first cps transformer was implemented as a compiler plugin[14]. It provides quite a powerful but complex interface based on delimited continuations. Scala-Async[7] provides a more familiar interface for developers for organizing asynchronous processing by compling async control flow to state machines. The main limitation is the absence of exception handling. Last year, a Lightbend team moved implementation of scala-async from macro to compiler plugin and extended one to support external 'Future systems' such as IO or Monix. In [6] scala-async model is extended to handle reactive streams. Scala coroutines [12] provides a model which allows to build async/await interface on top of coroutines. Scala Virtualized[13] devotes to solving a more general problem: provide deep embedding not only for monadic costructions but for arbitrary language. Scala Effekt [2] allows interpretation of effect handlers inside control monad whith delimited continuations.

4 Conclusion and further work

Ability to use direct control-flow on top of some library is a one half of programming experience. The other part is the library itself. Currently, we have a set of asynchronous scala runtimes with a different sets of capabilities and it would be interesting to build some uniform facilities for concurrency programming. One of the open questions is to extend eager Future runtime to support structured concurrency; Problem from the other side – users of effect stacks, such as IO, need to wrap impure API into effects. Can we automate this process? Also we plan to extend integration with existing asynchronous streaming interfaces.

Another direction is the expressivity of internal language, which can be extended by building appropriative wrapper control monad.

References

1. Adam, Ghahramani, Z., Gordon, A.D.: Practical probabilistic programming with monads. SIGPLAN Not. **50**(12), 165–176 (Aug 2015). https://doi.org/10.1145/2887747.2804317, https://doi.org/10.1145/2887747.2804317
2. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. J. Funct. Program. **30**, e8 (2020). https://doi.org/10.1017/S0956796820000027, https://doi.org/10.1017/S0956796820000027
3. Claessen, K.: A poor man's concurrency monad. Journal of Functional Programming **9**(3), 313–323 (1999). https://doi.org/10.1017/S0956796899003342
4. Erwig, M., Ren, D.: Monadification of functional programs. Sci. Comput. Program. **52**(1–3), 101–129 (Aug 2004). https://doi.org/10.1016/j.scico.2004.03.004, https://doi.org/10.1016/j.scico.2004.03.004
5. Filinski, A.: Representing monads. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 446–457. POPL '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/174675.178047, https://doi.org/10.1145/174675.178047
6. Haller, P., Miller, H.: A formal model for direct-style asynchronous observables. CoRR **abs/1511.00511** (2015), http://arxiv.org/abs/1511.00511
7. Haller, P., team, L.: scala-async (2013), https://github.com/scala-async/scala-async
8. Hatcliff, J., Danvy, O.: A generic account of continuation-passing styles. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 458–471. POPL '94, Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/174675.178053, https://doi.org/10.1145/174675.178053
9. Hoare, C.: Communicating Sequential Processes. Prentice-Hall International Series in Computer Science, Prentice Hall (1985), http://www.usingcsp.com/cspbook.pdf
10. Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving, and terminating monad transformers: (functional pearl). SIGPLAN Not. **40**(9), 192–203 (Sep 2005). https://doi.org/10.1145/1090189.1086390, https://doi.org/10.1145/1090189.1086390

11. Petricek, T., Syme, D.: The f# computation expression zoo. In: Proceedings of Practical Aspects of Declarative Languages. PADL 2014 (2014)
12. Prokopec, A., Liu, F.: Theory and Practice of Coroutines with Snapshots. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 3:1–3:32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.3, http://drops.dagstuhl.de/opus/volltexte/2018/9208
13. Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-virtualized: Linguistic reuse for deep embeddings. Higher Order Symbol. Comput. **25**(1), 165–207 (Mar 2012). https://doi.org/10.1007/s10990-013-9096-9, https://doi.org/10.1007/s10990-013-9096-9
14. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: Hutton, G., Tolmach, A.P. (eds.) Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009. pp. 317–328. ACM (2009). https://doi.org/10.1145/1596550.1596596, https://doi.org/10.1145/1596550.1596596
15. Syme, D.: The early history of f#. Proc. ACM Program. Lang. **4**(HOPL) (Jun 2020). https://doi.org/10.1145/3386325, https://doi.org/10.1145/3386325
16. Wadler, P., Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Logic **4**(1), 1–32 (Jan 2003). https://doi.org/10.1145/601775.601776, https://doi.org/10.1145/601775.601776

Towards a Language for Defining Reusable Programming Language Components
(Project Paper, Extended Abstract)

Cas van der Rest[1] and Casper Bach Poulsen[2]

c.r.vanderrest@tudelft.nl[1] c.b.poulsen@tudelft.nl[2]
Delft University of Technology, Delft, The Netherlands

1 Introduction

Our goal is to build reusable programming language components from algebraic data types and pattern matching functions. Most functional programming languages, however, are less than ideal for this purpose, because they lack built-in solutions for the *Expression Problem* [13]: while adding functionality to existing data types is easily done by writing a new function, we cannot extend data definitions themselves without modifying existing code. For instance, consider the following implementation of a small expression language in Haskell:

```
data Expr = Lit Int | Div Expr Expr
eval :: Expr → Maybe Int
eval (Lit x)      = return x
eval (Div e1 e2) = do {x ← eval e1; y ← eval e2; safeDiv x y}
```

To extend this expression language with support for pretty-printing, we can simply define a new function *pretty* :: *Expr* → *String*. But what if we want to extend it with a new construct corresponding to, for example, addition? Such a change would require us to go back and add a constructor *Add* to the definition of *Expr*, and extend all functions that match on *Expr* with new clauses accordingly.

This problem is amplified if we want to extend *Expr* with constructs that introduce new side-effects other than exceptions arising from division by zero. In that case, we also have to modify *eval*'s type signature, and potentially even the implementation of clauses for existing constructors. Clearly, if we intend *Expr* and *eval* as reusable components this is an undesirable situation.

We improve upon this state of affairs by introducing CS (working title), a functional meta-language for defining reusable programming language components. In CS, we can define components that describe part of a language's syntax, semantics or side effects, such that they can safely be composed into larger languages without requiring modification of existing code. The key features of CS that make this possible are (1) built-in support for per-case definitions of data types and pattern matching functions in the style of Data Types à la Carte [12], and (2) an effect system based on Plotkin and Pretnar's effect handlers [9] for the modular definition of side-effects.

CS is work in progress. There is a prototype implementation of an interpreter and interactive programming environment which we can us to define and run the examples from this abstract. We are, however, still in the process of developing and implementing a type system. In particular, we should statically prevent errors resulting from missing implementations of function clauses.

The name CS is an abbreviation of "Compositional Semantics". It is also the initials of Christopher Strachey, whose pioneering work [10] initiated the development of denotational semantics. In *Fundamental Concepts in Programming Languages* [11], Strachey wrote that "the urgent task in programming languages is to explore the field of semantic possibilities", and that we need to "recognize and isolate the central concepts" of programming languages. Today, five decades later, the words still ring true. The CS language aims to address this urgent task in programming languages, by supporting the definition of reusable (central) programming language concepts, via compositional denotation functions that map the syntax of programming languages to their meaning.

2 CS by Example

To showcase CS's design, we consider how to define the previous example as a reusable language component in CS.

Signatures and Modules The first step is to define a **signature** that announces the existence of an extensible data type *Expr*, and extensible function *eval*:

```
signature Eval (FX : Effects) where
  sort Expr : Set
  alg  eval  : Expr → {[FX] Int}
end
```

The braces ('{' and '}') in the type of *eval* indicate that it returns a suspended computation. Effects in CS happen eagerly, meaning that the side-effects of an expression occur *then and there* unless we suspend them. For *eval*, we do want suspension, leaving it up to the caller to decide when its effects take place.

The *Eval* signature has an effect row parameter, *FX*, describing which side effects may occur during evaluation. With the **alg** keyword, we allude to the initial algebra semantics for data types [5] on which CS's semantics for extensible types and functions is based. Indeed, we will see shortly that function clauses for *eval* are not implemented as regular functions, but as algebras instead.

We inhabit *Expr* and *eval* by defining **module**s that instantiate the *Eval* signature. We do this for the *Lit* and *Div* constructors:

```
module Lit : Eval where
  cons Lit : Int → Expr
  case eval (Lit n) = {n}
end
```

```
module Div : Eval where
  cons Div : Expr → Expr → Expr
  case eval (Div m1 m2) = { x ← m1; y ← m2; safeDiv x y }
```

Let us take a closer look at the implementation of *eval* in the module *Div*. There are two things worth noting here. First, we do not invoke *eval* recursively on the sub-expressions *m1* and *m2*. This is because we define function clauses as algebras, meaning that we assume that any recursive subtrees have already been replaced with the result of evaluating those subtrees. Second, the implementation uses the function *safeDiv* that guards against errors resulting from devision by zero. We find its implementation later on in the same module:

```
  fun safeDiv : Int → Int → [Abort] Int where
   | x 0 = abort !
   | x y = ...
end
```

The function *safeDiv* is annotated with the *Abort* effect, which supplies the abort operation, signalling abrupt termination. By invoking *safeDiv* in the defintion of *eval*, which from the definition of *Eval* has type $Expr \to [FX]\ Int$, we are implicitly imposing a constraint on the module parameter *FX* that it contains at least the *Abort* effect. In other words, whenever we import the module *Div* we better make sure that we instantiate *FX* with a row that has *Abort* in it.

We must say a few words about the braces ('{' and '}') that surround the implementation of *eval* for *Div*. Their purpose is to introduce a suspended computation. The opposite of suspension is enactment, which is denoted by postfixing with an exclamation mark (!). We see it in action in the definition of *safeDiv* (indeed, we abort immediately). Our use of braces is inspired by the similar language feature found in Frank [3].

Now, how do we use these modules to construct an interpreter for a language with integer literals and division? In CS, it is not necessary to explicitly compose constructors and clauses into data types and functions. Instead, the language manages this for us by automatically merging constructors and clauses whenever we import multiple instances of the same signature.

```
module Test where
  import Abort
       , Eval [Abort]
       , Lit, Div
  fun run : Expr → [Abort] Int where
   | e = eval e
   -- Evaluates to 3
  fun test : [Abort] Int
     = run (Div (Lit 6) (Lit 2))
end
```

We are allowed to invoke *eval* in the body of *run* here, because the sole constraint (imposed by importing *Div*) on its effect annotation of is that it contains *Abort*.

When importing the *Eval* signature we instantiated its effect row parameter with the singleton row $[Abort]$, which satisfies this constraint.

Effects and Handlers To use the *run* function, we must first invoke a handler for the *Abort* effect. To understand handlers, let us look at the module that implements the *Abort* effect together with its handler.

```
module Abort where
  import Prelude
  effect Abort where
   | abort : [Abort] a
  handler hAbort : [Abort | FX] a → [FX] (Maybe a) where
   | abort    k = Nothing
   | return x = Just x
end
```

With the **effect** keyword we declare a new effect together with its operations. Effect declarations are much like data type declarations, but instead of constructors they define the different ways in which we can construct effectful computations containing a particular effect.

We use the **handler** keyword to declare a handler for the *Abort* effect, *hAbort*, which removes it from the annotation of an effectful computation. The type of *hAbort* contains a free type variable (a) and a free row variable (FX), both of which are implicitly universally quantified, as is any free type or row variable. The result of handling the *Abort* effect is a *Maybe* value. *Maybe*, along with its constructors *Just* and *Nothing* is defined in the *Prelude* module.

Handlers must have a branch for each operation of the handled effect, plus a **return** branch that decorates pure values to match the handler's co-domain type. All branches corresponding to operations have an extra parameter that binds the *continuation*, representing the computation that succeeds the operation we are currently handling. By convention, we name this parameter k. In the abort case of *hAbort*, however, we ignore this continuation altogether, because the semantics of this operation should correspond to abrupt termination.

We use the continuation parameter in a more interesting way when defining a handler for a *State* effect:

```
module State (s : Set) where
  effect State where
   | get :      [State] s
   | put : s → [State] ()
  handler hState : [State | FX] a → s → [FX] (a×s) where
   | get        st k = k st st
   | (put st')  st k = k () st'
   | return x st    = (x, st)
end
```

For both the get and put operations, we use the continuation parameter k to implement the corresponding branch in *hAbort*. The continuation expects a value whose type corresponds to the return type of the current operation, and produces a computation with the same type as the co-domain type of the handler. For the put operation, for example, this means that k is of type $() \to s \to [FX]\,(a \times s)$. The implementation of *hState* for get and put then simply invokes k, using the current state as both the value and input state (get), or giving a unit value and using the given state st' as the input state (put).

2.1 Implementing Functions as a Reusable Effect

CS's effect system can describe much more sophisticated effects than *Abort* and *State*, as it permits fine-grained control over the semantics of operations that affect a program's control flow, even in the presence of other effects. To illustrate its expressiveness, we will now consider how to define function abstraction as a reusable effect, and implement two different handlers for this effect corresponding to a call-by-value and call-by-name semantics. We start by declaring the *Abstracting* effect and its operations:

```
effect Abstracting where
  | lam   : String → [Abstracting] Value → [Abstracting] Value
  | app   : Value  → Value               → [Abstracting] Value
  | var   : String                       → [Abstracting] Value
  | thunk : [Abstracting] Value          → [Abstracting] Value
```

The *Abstracting* effect has four operations, of which three correspond to the usual constructs of the λ-calculus. The thunk operation has no syntactical counterpart, but will be used for implementing a call-by-value and call-by-name evaluation strategy. *Value* is the type of values in our language; we will see shortly how it is defined.

When looking at the lam and thunk operations, we find that they both have parameters annotated with the *Abstracting* effect. This annotation indicates that they construct effectful computations from effectful computations, a pattern sometimes referred to as *higher-order effects*. Effectively, this means that any effects belonging to a value we wrap in a closure or thunk are postponed, leaving it up to the handler to decide when these take place.

Using the Abstracting effect To define a langue with function abstractions using the *Abstracting* effect, we define constructors *Abs*, *App*, and *Var* for *Expr*, and evaluate them by mapping onto the corresponding operation.

```
module Lambda : Eval where
  cons Abs : String → Expr → Expr
  |    App : Expr   → Expr → Expr
  |    Var : String        → Expr
```

```
  case eval (Abs x m)      = {lam x m}
   |   eval (App m1 m2) = {t ← thunk m2; app m1! t}
   |   eval (Var x)         = {var x}
end
```

Crucially, in the case for *Abs* we pass the effect-annotated body m, which has type $\{[FX]\ Value\}$, to the lam operation directly without extracting a pure value first. This prevents any effects in the body of a lambda from being enacted at the definition site, and instead leaves the decision of when these effects should take place to the used handler for the *Abstracting* effect. Similarly, in the case for *App*, we pass the function argument *m2* to the thunk operation directly, postponing any side-effects until we force the constructed thunk. We do, however, enact the side-effects of evaluating the function itself (i.e., *m1*), because the app operation expects its arguments to be a pure value.

We define the call-by-value and call-by-name handlers for *Abstracting* in a new module, that also defines the type of values for our language. Consequently, we adapt the *Eval* signature to use this value type in the signature for *eval*. To keep the exposition simple, we do not define *Value* as an extensible **sort**, but it is possible to do this in CS.

The values in this language are either numbers (*Num*), functions (*Clo*), or thunked computations (*Thunk*):

```
module HLambda (FX : Effect) where
  import Abstracting
  type Env = List (String× Value)
  data Value = Num Int
             | Clo String (Env → [Abort | FX] Value) Env
             | Thunk ([Abort | FX] Value)
  -- ... (handler for the Abstracting effect) ...
end
```

Call-by-value We are now ready to define a hander for the *Abstracting* effect that implements a call-by-value evaluation strategy. Figure 1 shows its implementation.

The **return** case is unremarkable: we simply ignore the environment *nv* and return the value *v*. The cases for lam and thunk are similar, as in both cases we do not enact the side-effects associated with the stored computation f, but instead wrap this computation in a *Closure* or *Thunk* which is passed to the continuation k. For variables, we resolve the identifier x in the environment and pass the result to the continuation.

A call-by-value semantics arises from the implementation of the app case. The highlights (e.g., *t*!) indicate where the thunk we constructed for the function argument in *eval* is forced. In this case, we force this argument thunk immedi-

```
handler hCBV : [Abstracting | FX] Value
                 → Env → [Abort | FX] Value where
| (lam x f)                          nv k = k (Clo x f nv) nv
| (app (Clo x f nv') ( Thunk t )) nv k = v' ← f ((x, t!) :: nv')
                                        ; k v' nv
| (app _ _)                          _  _ = abort !
| (var x)                            nv k = k (lookup nv x) nv
| (thunk f)                          nv k = k (Thunk {f nv}) nv
| return v                           nv   = v
```

Fig. 1. A Handler for the *Abstracting* effect, implementing a call-by-value semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

ately when encountering a function application, meaning that any side-effects of the argument take place *before* we evaluate the function body.

Call-by-name The handler in Figure 2 shows an implementation of a call-by-name semantics for the *Abstracting* effect. The only cases differences with the call-by-value handler in Figure 1 are the app and var cases.

In the case for app, we now put the argument thunk in the environment immediately, without forcing it first. Instead, in the case for var, we check if the variable we look up in the environment is a *Thunk*. If so, we force it and pass the resulting value to the continuation. In effect, this means that for a variable that binds an effectful computation, the associated side-effects take place every time we use that variable, but not until we reference it for the first time.

Example To illustrate the difference between *hCBV* (Figure 1) and *hCBN* (Figure 2), we combine the *Lambda* module with a module that uses the *State* effect. It defines expressions for reading and incrementing a single memory cell containing an integer:

```
module Mem : Eval where
  import State Int
  cons Incr   : Expr
  |    Recall : Expr
  case eval Incr   = {put (get ! +1); Num get!}
  |    eval Recall = get
end
```

When combining *Mem* and *Lambda*, we can observe the difference between a call-by-value and call-by-name evaluation strategy. Figure 3 shows an example of this.

```
handler hCBN : [Abstracting | FX] Value
                 → Env → [Abort | FX] Value where
| (lam x f)                nv k = k (Clo x f nv) nv
| (app (Clo x f nv') v)    nv k = v' ← f ((x, v) :: nv')
                                 ; k v' nv
| (app _ _)               _  _ = abort !
| (var x)                 nv k = match lookup x nv with
                                   | ( Thunk t ) → k t! nv
                                   | v           → k v nv
                                 end
| (thunk f)               nv k = k (Thunk {f nv}) nv
| return v                nv   = v
```

Fig. 2. A Handler for the *Abstracting* effect, implementing a call-by-name semantics for function arguments. The gray highlights indicate where thunks constructed for function arguments are forced.

3 Outlook

CS is an ongoing research project. Here, we briefly summarize the current state, and some of the challenges that still remain.

Current state There is a prototype implementation of CS, consisting of an implementation of the operational semantics, a declarative type checker written in Statix [1], and an interactive environment through which we can interact with the language. The operational semantics is inspired by a recently-proposed flavor of effect handlers called *Latent Effects* [2], which unlike plain effects and handlers can describe many advanced control-flow mechanisms. The *Abstracting* effect and its different evaluation strategies are an example of how we can benefit from this extra expressivity. With this prototype, it is possible to define and run the examples shown in this abstract.

Static Semantics We are still in the process of developing a type system for CS. Our plan is to use row types for both effect annotations (e.g., à la Frank [3] and Koka [6]), and for typing extensible data types and functions. The motivation for the latter is that CS's static semantics should prevent problems arising from missing function clause declarations. Row types seem to be a good fit for this requirement, since they allow pattern matching functions to reflect in their type for which constructors they are defined. By assigning a row type to extensible functions, we statically make the necessary information available to check that they are not applied to an input for which there does not exist a corresponding **case** declaration. We draw inspiration from the Rose [8] language, which applies row types to type extensible data types and records.

```
module Test where
  import Prelude
         , Abstracting
         , State Int
         , HLambdaCBV [State]
         , HLambdaCBN [State]
         , Lambda
         , Mem
  fun execCBV : Expr → (Maybe Value×Int) where
   | e = hState {hAbort {hCBV (eval e) []}} 0
  fun execCBN : Expr → (Maybe Value×Int) where
   | e = hState {hAbort {hCBN (eval e) []}} 0
  fun expr : Expr = App (Abs "x" Recall) Incr
   -- evaluates to (Just (Num 1) , 1)
  fun result1 : (Maybe Value×Int) = execCBV expr
   -- evaluates to (Just (Num 0) , 0)
  fun result2 : (Maybe Value×Int) = execCBN expr
end
```

Fig. 3. Examples of different outcomes when using a call-by-value or call-by-name evaluation strategy.

Core Language Parallel to developing CS, we are also working on developing a row-typed core language, which is intended as a minimal calculus to which we can desugar programs written in full CS. We base this core language on Rose [8], adapting it where necessary to encode (extensible) recursive data types and row-typed effects. Because the core language is much smaller than the surface language, it becomes more feasible to give a full formal specification of its semantics, and verify meta-theoretical properties such as type safety. The core language is still under development, but we hope to use it as a well-understood foundation for CS in the future. Of course, this will introduce additional challenges with respect to usability, such as how to provide decent error messages when type checking CS by going through the core language.

Semantics of extensible functions The current semantics of extensible functions is given by a catamorphism (fold) over the input type. This is a limiting factor when we try to implement traversals with a more complex recursive structure as an extensible function. To make CS's extensible functions in more expressive, we could switch to a richer model of extensible functions. For this we could explore, for example, more expressive recursion schemes [7], or *mixin algebras* [4].

References

1. van Antwerpen, H., Poulsen, C.B., Rouvoet, A., Visser, E.: Scopes as types. Proc. ACM Program. Lang. **2**(OOPSLA), 114:1–114:30 (2018). https://doi.org/10.1145/3276484, https://doi.org/10.1145/3276484
2. van den Berg, B., Schrijvers, T., Bach-Poulsen, C., Wu, N.: Latent effects for reusable language components: Extended version. CoRR **abs/2108.11155** (2021), https://arxiv.org/abs/2108.11155
3. Convent, L., Lindley, S., McBride, C., McLaughlin, C.: Doo bee doo bee doo. J. Funct. Program. **30**, e9 (2020). https://doi.org/10.1017/S0956796820000039, https://doi.org/10.1017/S0956796820000039
4. Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 207–218. ACM (2013). https://doi.org/10.1145/2429069.2429094, https://doi.org/10.1145/2429069.2429094
5. Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Rocca, S.R.D. (ed.) Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4583, pp. 207–222. Springer (2007). https://doi.org/10.1007/978-3-540-73228-0_16, https://doi.org/10.1007/978-3-540-73228-0_16
6. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 486–499. ACM (2017). https://doi.org/10.1145/3009837.3009872, https://doi.org/10.1145/3009837.3009872
7. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Lecture Notes in Computer Science, vol. 523, pp. 124–144. Springer (1991). https://doi.org/10.1007/3540543961_7, https://doi.org/10.1007/3540543961_7
8. Morris, J.G., McKinna, J.: Abstracting extensible data types: or, rows by any other name. Proc. ACM Program. Lang. **3**(POPL), 12:1–12:28 (2019). https://doi.org/10.1145/3290325, https://doi.org/10.1145/3290325
9. Plotkin, G.D., Pretnar, M.: Handling algebraic effects. Log. Methods Comput. Sci. **9**(4) (2013). https://doi.org/10.2168/LMCS-9(4:23)2013, https://doi.org/10.2168/LMCS-9(4:23)2013
10. Strachey, C.: Towards a formal semantics (1966)
11. Strachey, C.S.: Fundamental concepts in programming languages. High. Order Symb. Comput. **13**(1/2), 11–49 (2000). https://doi.org/10.1023/A:1010000313106, https://doi.org/10.1023/A:1010000313106
12. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758, https://doi.org/10.1017/S0956796808006758
13. Wadler, P.: The expression problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (1998), accessed: 2020-07-01

Deep Embedding with Class

Mart Lubbers[0000−0002−4015−4878]

Institute for Computing and Information Sciences,
Radboud University Nijmegen, Nijmegen, The Netherlands
mart@cs.ru.nl

Abstract The two flavours of DSL embedding are shallow and deep embedding. In functional languages, shallow embedding models the language constructs as functions in which the semantics are embedded. Adding semantics is therefore cumbersome while adding constructs is a breeze. Upgrading the functions to type classes lifts this limitation to a certain extent.
Deeply embedded languages represent their language constructs as data and the semantics are functions on it. As a result, the language constructs are embedded in the semantics, hence adding new language constructs is laborious where adding semantics is trouble free.
This paper shows that by abstracting the semantics functions in deep embedding to type classes, it is possible to easily add language constructs as well. So-called classy deep embedding results in DSLs that are extensible both in language constructs and in semantics while maintaining a concrete abstract syntax tree. Additionally, little type-level trickery or complicated boilerplate code is required to achieve this.

Keywords: functional programming, embedded domain specific languages, Haskell

1 Introduction

The two flavours of DSL embedding are deep and shallow embedding [4]. In functional programming languages, shallow embedding models language constructs as functions in the host language. As a result, adding new language constructs—extra functions—is easy. However, the semantics of the language is embedded in the functions, thus it is troublesome to add semantics since it requires updating all existing language constructs.

On the other hand, deep embedding models language constructs as data in the host language. The semantics of the language are represented by functions over the data. Consequently, adding new semantics, i.e. novel functions, is straightforward. It can be stated that the language constructs are embedded in the functions that form a semantics. If one wants to add a language construct, all semantics functions must be revisited and revised to avoid ending up with partial functions.

This juxtaposition has been known for many years [22] and discussed by many others [14] but most famously dubbed the *expression problem* by Wadler [25]:

> The *expression problem* is a new name for an old problem. The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety (e.g., no casts).

In shallow embedding, abstracting the functions to type classes disentangles the language constructs from the semantics, allowing extension both ways. This technique is dubbed tagless-final embedding [5], nonetheless it is no silver bullet. Some semantics that require an intensional analysis of the syntax tree, such as transformation or optimisations, are difficult to implement in shallow embedding due to the lack of an explicit data structure representing the abstract syntax tree. Thus, either the semantics of the DSL must be a state or context has to be maintained so that structural information is not lost [13].

1.1 Research contribution

This paper shows how to apply the technique observed in tagless-final embedding to deep embedding. The presented basic technique, christened *classy deep embedding*, does not require advanced type system extensions to be used. However, it is suitable for type system extensions such as generalised algebraic data types. While the examples are written in Haskell [21] using some minor extensions provided by GHC [9], the idea is applicable to other languages as well.

2 Deep embedding

Take a DSL, take any DSL, take the language of expressions in which literal integers and addition can be expressed. In deep embedding, terms in the language are represented by data in the host language. Hence, defining the constructs is as simple as creating the following algebraic data type. The suffixed to indicate the evolution.

$$\begin{aligned}&\textbf{data } \mathit{Expr}_0 = \mathit{Lit}_0 \;\; \mathit{Int}\\ &\qquad\qquad\;\; | \;\; \mathit{Add}_0 \; \mathit{Expr}_0 \; \mathit{Expr}_0\end{aligned}$$

Semantics are defined as functions on the Expr_0 data type. For example, a function transforming the term to an integer—an evaluator—is implemented as follows.

$$\begin{aligned}&\mathit{eval}_0 :: \mathit{Expr}_0 \to \mathit{Int}\\ &\mathit{eval}_0 \; (\mathit{Lit}_0 \; e) \qquad = e\\ &\mathit{eval}_0 \; (\mathit{Add}_0 \; e_1 \; e_2) = \mathit{eval}_0 \; e_1 + \mathit{eval}_0 \; e_2\end{aligned}$$

Adding semantics—e.g. a printer—just means adding another function and the existing functions remain untouched. I.e. the key property of deep embedding. The following function, transforming the Expr_0 data type to a string, defines a simple printer for our language.

$$
\begin{aligned}
&print_0 :: Expr_0 \to String \\
&print_0\ (Lit_0\ \ v) \quad\ = show\ v \\
&print_0\ (Add_0\ e_1\ e_2) = \texttt{"("} +\!\!+ print_0\ e_1 +\!\!+ \texttt{"-"} +\!\!+ print_0\ e_2 +\!\!+ \texttt{")"}
\end{aligned}
$$

While the language is concise and elegant, it is not very expressive. Traditionally, extending the language is achieved by adding a case to the $Expr_0$ data type. However, this means that we have to touch, and thus recompile, the original datatype. So, adding subtraction to the language results in the following revised data type.

$$
\begin{aligned}
\textbf{data}\ Expr_0 = &\ Lit_0\ \ Int \\
\mid &\ Add_0\ Expr_0\ Expr_0 \\
\mid &\ Sub_0\ \ Expr_0\ Expr_0
\end{aligned}
$$

Extending the DSL with language constructs exposes the Achilles' heel of deep embedding. Adding a case to the data type means that all semantic functions need to be updated to be able to handle this new case. This does not seem like an insurmountable problem, but it does pose a problem if either the functions or the data type itself are written by others or are contained in a closed library.

3 Shallow embedding

Conversely, let us see how this would be done in shallow embedding. First, the data type is represented by functions in the host language with embedded semantics. Therefore, the evaluators for literals and addition both become a function in the host language as follows.

$$
\begin{aligned}
&\textbf{type}\ Sem_s = Int \\
&lit_s :: Int \to Sem_s \\
&lit_s\ i = i \\
&add_s :: Sem_s \to Sem_s \to Sem_s \\
&add_s\ e_1\ e_2 = e_1 + e_2
\end{aligned}
$$

Adding subtraction to the language is as simple as adding a new function.

$$
\begin{aligned}
&sub_s :: Sem_s \to Sem_s \to Sem_s \\
&sub_s\ e_1\ e_2 = e_1 - e_2
\end{aligned}
$$

Adding semantics on the other hand—e.g. a printer—is not that simple because the semantics are part of the functions representing the language constructs. One way to add semantics is to change all functions to execute both semantics at the same time. In our case this means changing the type of Sem_s to be $(Int, String)$ so that all functions operate on a tuple containing the result of the evaluator and the printed representation at the same time. Alternatively, a single semantics can be defined that represents a fold over the language constructs [10], delaying the selection of semantics to the moment the fold is applied.

3.1 Tagless-final embedding

Tagless-final embedding overcomes the limitations of standard shallow embedding. To upgrade to this embedding technique, the language constructs are changed from functions to type classes. For our language this results in the following class definition.

```
class Expr_t s where
  lit_t  :: Int → s
  add_t :: s → s → s
```

Semantics become data types implementing these type classes, resulting in the following instance for the evaluator.

```
newtype Eval_t = E_t Int
instance Expr_t Eval_t where
  lit_t  v                = E_t v
  add_t (E_t e_1) (E_t e_2) = E_t (e_1 + e_2)
```

Adding constructs—e.g. subtraction—just results in an extra type class and corresponding instances.

```
class Sub_t s where
  sub_t :: s → s → s
instance Sub_t Eval_t where
  sub_t (E_t e_1) (E_t e_2) = E_t (e_1 − e_2)
```

Finally, adding semantics such as a printer over the language is achieved by providing a data type representing the semantics accompanied by instances for the language constructs.

```
newtype Printer_t = P_t String
instance Expr_t Printer_t where
  lit_t  i                = P_t (show i)
  add_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "+" ++ e_2 ++ ")")
instance Sub_t Printer_t where
  sub_t (P_t e_1) (P_t e_2) = P_t ("(" ++ e_1 ++ "-" ++ e_2 ++ ")")
```

4 Lifting the backends

Let us rethink the deeply embedded DSL design. Remember that in shallow embedding, the semantics are embedded in the language construct functions. Obtaining extensibility both in constructs and semantics was accomplished by abstracting the semantic functions to type classes, making the constructs overloaded in the semantics. In deep embedding, the constructs are embedded in the

semantics functions instead. So, let us apply the same technique, i.e. make the semantics overloaded in the language constructs by abstracting the semantics functions to type classes. The same effect may be achieved when using similar techniques such as explicit dictionary passing or ML style modules. In our language this results in the following type class.

```
class Eval_1 v where
  eval_1 :: v → Int


data Expr_1 = Lit_1  Int
            | Add_1 Expr_1 Expr_1
```

Implementing the semantic type class instances for the $Expr_1$ data type is an elementary exercise. By a copy, paste and some modifications, we come to the following implementation.

```
instance Eval_1 Expr_1 where
  eval_1 (Lit_1  v)      = v
  eval_1 (Add_1 e_1 e_2) = eval_1 e_1 + eval_1 e_2
```

Subtraction can now be defined in a separate data type, leaving the original data type intact. Instances for the additional semantics can now be implemented separately as instances of the type classes.

```
data SubExpr_1 = Sub_1 Expr_1 Expr_1


instance Eval_1 SubExpr_1 where
  eval_1 (Sub_1 e_1 e_2) = eval_1 e_1 − eval_1 e_2
```

5 Existential data types

The astute reader might have noticed that we have dissociated ourselves from the original data type. It is only possible to create an expression with a subtraction on the top level. The recursive knot is left untied and as a result, $SubExpr_1$ can never be reached from an $Expr_1$.

Luckily, we can reconnect them by adding a special constructor to the $Expr_1$ data type for housing extensions. To allow it to house not just subtraction but any extension, it contains an existentially quantified [18] type with class constraints [15,16] for all semantics type classes [9, Chp. 6.4.6].

```
data Expr_2 =                                  Lit_2  Int
            |                                  Add_2 Expr_2 Expr_2
            | ∀x.(Eval_2 x, Print_2 x) ⇒ Ext_2  x
```

The implementation of the extension case in the semantics type classes is in most cases just a matter of calling the function for the argument as can be seen in the semantics instances shown below.

$$
\begin{aligned}
&\textbf{instance } \mathit{Eval}_2\ \mathit{Expr}_2 \textbf{ where}\\
&\quad \mathit{eval}_2\ (\mathit{Lit}_2\ \ v) \quad\ = v\\
&\quad \mathit{eval}_2\ (\mathit{Add}_2\ e_1\ e_2) = \mathit{eval}_2\ e_1 + \mathit{eval}_2\ e_2\\
&\quad \mathit{eval}_2\ (\mathit{Ext}_2\ \ x) \quad\ = \mathit{eval}_2\ x
\end{aligned}
$$

Adding language construct extensions in different data types does mean that an extra *Ext* tag is introduced when using the extension. This burden can be relieved by creating a smart constructor for it that automatically wraps the extension with the *Ext* constructor so that it is of the type of the main data type.

$$
\begin{aligned}
&\mathit{sub}_2 :: \mathit{Expr}_2 \to \mathit{Expr}_2 \to \mathit{Expr}_2\\
&\mathit{sub}_2\ e_1\ e_2 = \mathit{Ext}_2\ (\mathit{Sub}_2\ e_1\ e_2)
\end{aligned}
$$

In our example this means that the programmer can write:

$$e_2 = \mathit{sub}_2\ (\mathit{Lit}_2\ 42)\ (\mathit{Lit}_2\ 1)$$

instead of having to write

$$e_2' = \mathit{Ext}_2\ (\mathit{Sub}_2\ (\mathit{Lit}_2\ 42)\ (\mathit{Lit}_2\ 1))$$

5.1 Unbraiding the semantics from the data

This approach does reveal a minor problem. Namely that all semantics type-classes are braided into our datatypes via the Ext_2 constructor. Say if we add the printer again, the Ext_2 constructor has to be updated to —$\forall x.(\mathit{Eval}_2\ x, \mathit{Print}_2\ x) \Rightarrow \mathit{Ext}_2\ x$. Thus, if we add semantics, the main data type's class constraints in the *Ext* constructor need to be updated. To avoid this, the type classes can be bundled in a class alias or class collection as follows.

$$\textbf{class } (\mathit{Eval}_2\ x, \mathit{Print}_2\ x) \Rightarrow \mathit{Semantics}_2\ x$$

$$
\begin{aligned}
\textbf{data } \mathit{Expr}_2 = \quad & \mathit{Lit}_2\ \ \mathit{Int}\\
\mid \quad & \mathit{Add}_2\ \mathit{Expr}_2\ \mathit{Expr}_2\\
\mid \ \forall x.\mathit{Semantics}_2\ x \Rightarrow\ & \mathit{Ext}_2\ \ x
\end{aligned}
$$

The class alias removes the need for the programmer to visit the main data type when adding an additional semantics. Unfortunately, the compiler does need to visit the main data type again. Some may argue that adding semantics happens less frequently than adding language constructs but in reality it means that we have to concede that the language is not as easily extensible in semantics

as in language constructs. More exotic type system extensions such as constraint kinds [3,26] can mitigate this issue by making the data types parametrised by the particular semantics. However, by adding some boilerplate, even without this extension the language constructs can be parametrised by the semantics by putting the semantics functions in a data type. First the data types for the language constructs are parametrised by the type variable c as follows.

```
data Expr3 c =     Lit3  Int
             |     Add3 (Expr3 c) (Expr3 c)
             | ∀x.Ext3  (c x) x

data SubExpr3 c = Sub3 (Expr3 c) (Expr3 c)
```

The c type variable is inhabited by an explicit dictionary for the semantics. Therefore, for all semantics type classes, a data type is made that contains the semantics function for the given semantics. This means that for $Eval_3$, a dictionary with the function $EvalDict_3$ is defined, a type class $HasEval_3$ for retrieving the function from the dictionary and an instance for $HasEval_3$ for $EvalDict_3$.

```
data EvalDict3 v = EvalDict3 (v → Int)
class HasEval3 d where
  hasEval3 :: d v → v → Int
instance HasEval3 EvalDict3 where
  hasEval3 (EvalDict3 e) = e
```

The instances for the type classes change as well according to the change in the datatype accordingly.

```
instance HasEval3 d ⇒ Eval3 (Expr3 d) where
  eval3 (Lit3  v)      = v
  eval3 (Add3 e1 e2) = eval3 e1 + eval3 e2
  eval3 (Ext3  d x)  = hasEval3 d x

instance HasEval3 d ⇒ Eval3 (SubExpr3 d) where
  eval3 (Sub3 e1 e2) = eval3 e1 − eval3 e2
```

Because the Ext_3 constructor from $Expr_3$ now contains a value of type c, the smart constructor for Sub_3 must somehow come up with this value. To achieve this, a class is introduced that allows the generation of such a dictionary.

```
class GDict a where
  gdict :: a
```

This class has instances for all semantics dictionaries.

```
instance Eval3 v ⇒ GDict (EvalDict3 v) where
  gdict = EvalDict3 eval3
```

With these instances, the semantics function can be retrieved from the Ext_3 constructor and in the smart constructors they can be generated as follows:

```
sub3 :: GDict (c (SubExpr3 c)) ⇒ Expr3 c → Expr3 c → Expr3 c
sub3 e1 e2 = Ext3 gdict (Sub3 e1 e2)
```

Finally we reached the end goal, orthogonal extension of both language constructs as shown by adding subtraction to the language and in language semantics. Adding the printer can now be done without touching the original code as follows. First the printer class, dictionaries and instances for *GDict* are defined.

```
class Print3 v where
  print3 :: v → String
data PrintDict3 v = PrintDict3 (v → String)
class HasPrint3 d where
  hasPrint3 :: d v → v → String
instance HasPrint3 PrintDict3 where
  hasPrint3 (PrintDict3 e) = e
instance Print3 v ⇒ GDict (PrintDict3 v) where
  gdict = PrintDict3 print3
```

Then the instances for $Print_3$ of all the language constructs can be defined.

```
instance HasPrint3 d ⇒ Print3 (Expr3 d) where
  print3 (Lit3  v)      = show v
  print3 (Add3 e1 e2) = "(" ++ print3 e1 ++ "+" ++ print3 e2 ++ ")"
  print3 (Ext3  d x)  = hasPrint3 d x


instance HasPrint3 d ⇒ Print3 (SubExpr3 d) where
  print3 (Sub3 e1 e2) = "(" ++ print3 e1 ++ "-" ++ print3 e2 ++ ")"
```

6 Transformation semantics

Most semantics convert a term to some final representation and can be expressed just by functions on the cases. However, some semantics such as transformation or optimisation require a so called intentional analysis of the abstract syntax tree. In shallow embedding, the implementation for these type of semantics is difficult because there is no tangible abstract syntax tree. In off-the-shelf deep embedding this is effortless since the function can pattern match on the constructor or structures of constructors.

To demonstrate intensional analyses in classy deep embedding we write an optimizer that removes addition and subtraction by zero In classy deep embedding, adding new semantics means first adding a new type class housing the function including the machinery for the extension constructor.

```
class Opt_3 v where
  opt_3 :: v → v
data OptDict_3 v = OptDict_3 (v → v)
class HasOpt_3 d where
  hasOpt_3 :: d v → v → v
instance HasOpt_3 OptDict_3 where
  hasOpt_3 (OptDict_3 e) = e
instance Opt_3 v ⇒ GDict (OptDict_3 v) where
  gdict = OptDict_3 opt_3
```

The implementation of the optimizer for the $Expr_3$ data type is no complicated task. The only interesting bit in occurs in the Add_3 constructor, where we pattern match on the optimised children to determine whether an addition with zero is performed. If this is the case, the addition is removed.

```
instance HasOpt_3 d ⇒ Opt_3 (Expr_3 d) where
  opt_3 (Lit_3  v)      = Lit_3 v
  opt_3 (Add_3 e_1 e_2) = case (opt_3 e_1, opt_3 e_2) of
    (Lit_3 0, e_2')     → e_2'
    (e_1', Lit_3 0)     → e_1'
    (e_1', e_2')        → Add_3 e_1' e_2'
  opt_3 (Ext_3  d x)    = Ext_3 d (hasOpt_3 d x)
```

Replicating this for the Opt_3 instance of $SubExpr_3$ seems a clear-cut task at first glance.

```
instance HasOpt_3 d ⇒ Opt (SubExpr_3 d) where
  opt_3 (Sub_3 e_1 e_2) = case (opt_3 e_1, opt_3 e_2) of
    (e_1', Lit 0) → e_1'
    (e_1', e_2')  → SubExpr_3 e_1' e_2'
```

Unsurprisingly, this code is rejected by the compiler. When a literal zero is matched as the right-hand side of a subtraction, the left hand side of type $Expr_3$ is returned. However, the type signature of the function dictates that it should be of type $SubExpr_3$. To overcome this problem we add a convolution constructor.

6.1 Convolution

Adding a loopback case or convolution constructor to the Sub_3 allows the removal of the Sub_3 constructor while remaining the $SubExpr_3$ type. It should be noted that a loopback case is *only* required if the transformation actually removes tags. This changes the $SubExpr_3$ data type as follows.

```
data SubExpr_3 c = Sub_3       (Expr_3 c) (Expr_3 c)
                 | SubLoop_3 (Expr_3 c)
```

```
instance HasEval_3 d ⇒ Eval_3 (SubExpr_3 d) where
  eval_3 (Sub_3       e_1 e_2) = eval_3 e_1 − eval_3 e_2
  eval_3 (SubLoop_3 e_1)       = eval_3 e_1
```

With this loopback case in the toolbox, the following *SubExpr* instance optimises away subtraction with zero literals.

```
instance HasOpt3 d ⇒ Opt3 (SubExpr3 d) where
  opt3 (Sub3 e1 e2)  = case (opt3 e1, opt3 e2) of
    (e1', Lit3 0) → SubLoop3 e1'
    (e1', e2')    → Sub3 e1' e2'
  opt3 (SubLoop3 e) = SubLoop3 (opt3 e)
```

6.2 Pattern matching

Pattern matching within datatypes and from an extension to the main data type works out of the box. Pattern matching on values with an existential type is not possible without leveraging dynamic typing [1,2]. To enable dynamic typing support, the *Typeable* type class as provided by *Data.Dynamic* [8] is added to the list of constraints in all places where we need to pattern match across extensions. Cross-extensional pattern matching on the other hand—matching on a particular extension—is something that requires a bit of extra care. Take for example the optimisations negation propagation and double negation elimination. As a result, the *Typeable* class constraints is added to the quantified type variable x of the Ext_4 constructor and to cs in the smart constructors.

```
data Expr4 c =                          Lit4  Int
             |                          Add4 (Expr4 c) (Expr4 c)
             | ∀x.Typeable x ⇒ Ext4 (c x) x
```

First let us add negation to the language by defining a datatype representing it. Negation elimination requires the removal of negation constructors so a convolution constructor is defined as well.

```
data NegExpr4 c = Neg4     (Expr4 c)
                | NegLoop4 (Expr4 c)
neg4 :: (Typeable c, GDict (c (NegExpr4 c))) ⇒ Expr4 c → Expr4 c
neg4 e = Ext4 gdict (Neg4 e)
```

The evaluation and printer instances for the $NegExpr_4$ datatype are defined as follows.

```
instance HasEval4 d ⇒ Eval4 (NegExpr4 d) where
  eval4 (Neg4     e) = negate (eval4 e)
  eval4 (NegLoop4 e) = eval4 e
```

```
instance HasPrint4 d ⇒ Print4 (NegExpr4 d) where
  print4 (Neg4     e) = "(~" ++ print4 e ++ ")"
  print4 (NegLoop4 e) = print4 e
```

The Opt_4 contains the interesting bit. If the sub expression of a negation is an addition, negation is propagated downwards. If the sub expression is again a negation, something that can only be found out by

```
instance (Typeable d, GDict (d (NegExpr_4 d)), HasOpt_4 d) ⇒
    Opt_4 (NegExpr_4 d) where
  opt_4 (Neg_4 (Add_4 e_1 e_2))
    = NegLoop_4 (Add_4 (opt_4 (neg_4 e_1)) (opt_4 (neg_4 e_2)))
  opt_4 (Neg_4 (Ext_4 d x))
    = case fromDynamic (toDyn (hasOpt_4 d x)) of
      Just (Neg_4 e) → NegLoop_4 e
      _              → Neg_4 (Ext_4 d (hasOpt_4 d x))
  opt_4 (Neg_4      e) = Neg_4 (opt_4 e)
  opt_4 (NegLoop_4 e) = NegLoop_4 (opt_4 e)
```

Loopback cases do make cross-extensional pattern matching less modular in general. For example, $Ext_4\ d\ (SubLoop_4\ (Lit_4\ 0))$ is equivalent to $Lit_4\ 0$ in the optimisation semantics and would just require an extra pattern match. Fortunately, this problem can be mitigated—if required—by just adding an additional optimisation semantics that removes loopback cases. Moreover, one does not need to resort to these arguably blunt matters a lot. Dependent language functionality often does not need to span extensions, i.e. it is possible to group them in the same data type.

6.3 Chaining semantics

Now that the data types are parametrised by the semantics a final problem needs to be overcome. The data type is parametrised by the semantics, thus, using multiple semantics, such as evaluation after optimising is not straightforwardly possible. Luckily, a solution is readily at hand: introduce an ad-hoc combining semantics.

```
data OptPrintDict_4 v = OptPrintDict_4 (OptDict_4 v) (PrintDict_4 v)
instance HasOpt_4    OptPrintDict_4 where
  hasOpt_4    (OptPrintDict_4 v _) = hasOpt_4 v
instance HasPrint_4 OptPrintDict_4 where
  hasPrint_4 (OptPrintDict_4 _ v) = hasPrint_4 v
instance (Opt_4 v, Print_4 v) ⇒ GDict (OptPrintDict_4 v) where
  gdict = OptPrintDict_4 gdict gdict
```

And this allows us to write $print_4\ (opt_4\ e_1)$ resulting in $((\sim 42) + (\sim 38))$ when e_1 represents $(\sim (42 + 38)) - 0$ and is thus defined as follows.

```
e_1 :: Expr_4 OptPrintDict_4
e_1 = sub_4 (neg_4 (Add_4 (Lit_4 42) (Lit_4 38))) (Lit_4 0)
```

7 Generalised algebraic data types (GADTs)

GADTs are enriched data types that allow the type instantiation of the constructor to be explicitly defined [7,11]. Leveraging GADTs, deeply embedded DSLs can be made statically type safe even when different value types are supported. Even when GADTs are not supported natively in the language, they can be simulated using embedding-projection pairs or equivalence types [6, Sec. 2.2]. Where some solutions to the expression problem do not easily generalise to GADTs (see Section 9), classy deep embedding does. Generalising the data structure of our DSL results in the following GADTs. Note that to make the DSL more general, the types of the constructors have been relaxed more. For example, operations on integers now work on all numerals instead. Moreover, the Lit_g constructor can be used to lift values of any type to the DSL domain as long as they can be shown which is required for the printer. Since the optimisations on Neg_g remove constructors and is a cross-extensional pattern match, *Typeable* constraints must be added to a. Furthermore, because the optimisations for Add_g and Sub_g are now more general, they do not only work for *Int* but for any a for which an *Num* instance is available, the *Eq* constraint is added to these constructors as well.

$$
\begin{aligned}
&\textbf{data } Expr_g\ c\ a \textbf{ where}\\
&\quad Lit_g\ :: Show\ a \Rightarrow a \to Expr_g\ c\ a\\
&\quad Add_g :: (Eq\ a, Num\ a) \Rightarrow Expr_g\ c\ a \to Expr_g\ c\ a \to Expr_g\ c\ a\\
&\quad Ext_g\ :: (Typeable\ x) \Rightarrow c\ x \to x\ a \to Expr_g\ c\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data } SubExpr_g\ c\ a \textbf{ where}\\
&\quad Sub_g :: (Eq\ a, Num\ a) \Rightarrow Expr_g\ c\ a \to Expr_g\ c\ a \to SubExpr_g\ c\ a\\
&\quad SubLoop_g :: \qquad\qquad\qquad\quad Expr_g\ c\ a \to SubExpr_g\ c\ a
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data } NegExpr_g\ c\ a \textbf{ where}\\
&\quad Neg_g :: (Typeable\ a, Num\ a) \Rightarrow Expr_g\ c\ a \to NegExpr_g\ c\ a\\
&\quad NegLoop_g :: \qquad\qquad\qquad\quad Expr_g\ c\ a \to NegExpr_g\ c\ a
\end{aligned}
$$

The smart constructors for the language extensions inherit the class constraints of their data types and include a *Typeable* constraint on the c type variable for it to be usable in the Ext_g constructor

$$
\begin{aligned}
&sub_g :: (Typeable\ c, GDict\ (c\ (SubExpr_g\ c)), Eq\ a, Num\ a) \Rightarrow\\
&\quad Expr_g\ c\ a \to Expr_g\ c\ a \to Expr_g\ c\ a\\
&sub_g\ e_1\ e_2 = Ext_g\ gdict\ (Sub_g\ e_1\ e_2)\\
&neg_g :: (Typeable\ c, GDict\ (c\ (NegExpr_g\ c)), Typeable\ a, Num\ a) \Rightarrow\\
&\quad Expr_g\ c\ a \to Expr_g\ c\ a\\
&neg_g\ e = Ext_g\ gdict\ (Neg_g\ e)
\end{aligned}
$$

Upgrading the semantics type classes to support GADTs is done by an easy textual search and replace. All occurances of v are now parametrised by type variable a.

```
class Eval_g  v where
  eval_g  :: v a → a
class Print_g v where
  print_g :: v a → String
class Opt_g   v where
  opt_g   :: v a → v a
```

Now that the shape of the classes has changed, the dictionary data types and the classes need to be adapted as well. The introduced type variable a is no argument to the class so it should not be an argument to the dictionary data type. To represent this class function, a rank-2 polymorph function is needed [9, Chp. 6.4.15][20].

```
data EvalDict_g v = EvalDict_g (∀a.v a → a)
class HasEval_g d where
  hasEval_g :: d v → v a → a
instance HasEval_g EvalDict_g where
  hasEval_g (EvalDict_g e) = e
data PrintDict_g v = PrintDict_g (∀a.v a → String)
class HasPrint_g d where
  hasPrint_g :: d v → v a → String
instance HasPrint_g PrintDict_g where
  hasPrint_g (PrintDict_g e) = e
data OptDict_g v = OptDict_g (∀a.v a → v a)
class HasOpt_g d where
  hasOpt_g :: d v → v a → v a
instance HasOpt_g OptDict_g where
  hasOpt_g (OptDict_g e) = e
```

The *GDict* class is general enough so the instances can remain the same.

```
instance Eval_g v ⇒ GDict (EvalDict_g v) where
  gdict = EvalDict_g eval_g
instance Print_g v ⇒ GDict (PrintDict_g v) where
  gdict = PrintDict_g print_g
instance Opt_g v ⇒ GDict (OptDict_g v) where
  gdict = OptDict_g opt_g
```

Finally, the implementations for the instances can be ported without complication.

```
instance HasEval_g d ⇒ Eval_g (Expr_g d) where
  eval_g (Lit_g  v)     = v
  eval_g (Add_g e1 e2) = eval_g e1 + eval_g e2
  eval_g (Ext_g  d x)  = hasEval_g d x
instance HasEval_g d ⇒ Eval_g (SubExpr_g d) where
```

```
  eval_g (Sub_g       e_1 e_2) = eval_g e_1 − eval_g e_2
  eval_g (SubLoop_g e)         = eval_g e

instance HasPrint_g d ⇒ Print_g (Expr_g d) where
  print_g (Lit_g  v)       = show v
  print_g (Add_g e_1 e_2) = "(" ++ print_g e_1 ++ "+" ++ print_g e_2 ++ ")"
  print_g (Ext_g  d x)     = hasPrint_g d x
instance HasPrint_g d ⇒ Print_g (SubExpr_g d) where
  print_g (Sub_g       e_1 e_2) = "(" ++ print_g e_1 ++ "-" ++ print_g e_2 ++ ")"
  print_g (SubLoop_g e)         = print_g e

instance HasOpt_g d ⇒ Opt_g (Expr_g d) where
  opt_g (Lit_g  v)       = Lit_g v
  opt_g (Add_g e_1 e_2) = case (opt_g e_1, opt_g e_2) of
    (Lit_g 0, e_2') → e_2'
    (e_1', Lit_g 0) → e_1'
    (e_1', e_2')    → Add_g e_1' e_2'
  opt_g (Ext_g  d x)     = Ext_g d (hasOpt_g d x)
instance HasOpt_g d ⇒ Opt_g (SubExpr_g d) where
  opt_g (Sub_g       e_1 e_2) = case (opt_g e_1, opt_g e_2) of
    (e_1', Lit_g 0) → SubLoop_g e_1'
    (e_1', e_2')    → Sub_g e_1' e_2'
  opt_g (SubLoop_g e)         = SubLoop_g (opt_g e)
instance (Typeable d, GDict (d (NegExpr_g d)), HasOpt_g d) ⇒
    Opt_g (NegExpr_g d) where
  opt_g (Neg_g (Add_g e_1 e_2))
    = NegLoop_g (Add_g (opt_g (neg_g e_1)) (opt_g (neg_g e_2)))
  opt_g (Neg_g (Ext_g d x))
    = case fromDynamic (toDyn (hasOpt_g d x)) of
      Just (Neg_g e) → NegLoop_g e
      _              → Neg_g (Ext_g d (hasOpt_g d x))
  opt_g (Neg_g       e) = Neg_g (opt_g e)
  opt_g (NegLoop_g e) = NegLoop_g (opt_g e)
```

8 Conclusion

Classy deep embedding is a novel organically grown embedding technique that alleviates deep embedding from the extensibility problem in most cases.

By abstracting the semantics functions to classes they become overloaded in the language constructs. Thus, making it possible to add new language constructs

in a separate type. These extensions are brought together in a special extension constructor residing in the main data type. This extension case is overloaded by the language construct using a data type containing the class dictionary. As a result, orthogonal extension is possible for language constructs and semantics using only little syntactic overhead or type annotations. The basic technique only requires—well established through history and relatively standard—existential data types. However, if needed, the technique generalises to GADTs as well, adding rank-2 types to the list of type system requirements as well.. Finally, the abstract syntax tree remains observable which makes it suitable for intensional analyses, albeit using occasional dynamic typing for truly cross-extensional transformations.

9 Related work

Embedded DSL techniques in functional languages have been a topic of research for many years and thus we do not claim a complete overview of related work.

Clearly, classy deep embedding bears most similarity to the Datatypes à la Carte [24]. In Swierstra's approach, semantics are lifted to type classes in a similar fashion to classy deep embedding. Each language construct is their own datatype parametrised by a type parameter. This parameter contains some type level representation of language constructs that are in use. In classy deep embedding, extensions do not have to be iterated at the type level but are captured in the extension case. Because all the constructs are expressed in the type system, nifty type system tricks need to be employed to convince the compiler that everything is type safe and the class constraints can be solved. Furthermore, it requires some boilerplate code such as functor instances for the data types. In return, pattern matching is easier and does not require dynamic typing. Classy deep embedding only strains the programmer with writing the extension case for the main data type and the occasional loopback constructor.

Löh et al. proposed a language extension that allows open data types and open functions, i.e. functions and data types that can be extended with more cases later on [17]. They hinted at the possibility of using type classes for open functions but had serious concerns that pattern matching would be crippled because constructors are becoming types, thus ultimately becoming impossible to type. In contrast, this paper shows that pattern matching is easily attainable—albeit using dynamic types—and that the terms can be typed without complicated type system extensions.

A technique similar to classy deep embedding was proposed by Najd and Peyton Jones to tackle a slightly different problem, namely that of reusing a data type for multiple purposes in a slightly different form [19]. For example to decorate the abstract syntax tree of a compiler differently for each phase of the compiler. They propose to add an extension descriptor as a type variable to a data type and a type family that can be used to decorate constructors with extra information and add additional constructors to the data type using an extension constructor. Classy deep embedding works similarly but uses existentially quan-

tified type variables to describe possible extensions instead of type variables and type families. In classy deep embedding, the extensions do not need to be encoded in the type system and less boilerplate is required. Furthermore, pattern matching on extensions becomes a bit more complicated but in return it allows for multiple extensions to be added orthogonally and avoids the necessity for type system extensions.

Tagless embedding is the shallowly embedded counterpart of classy deep embedding and was invented for the same purpose; overcoming the issues with standard shallow embedding [5]. Classy deep embedding was organically grown from observing the evolution of tagless embedding. The main difference between tagless embedding and classy deep embedding—and in general between shallow and deep embedding—is that intensional analyses of the abstract syntax tree is very difficult because there is no tangible abstract syntax tree data structure. In classy deep embedding, it is possible to define transformations even across extensions. On the other hand, tagless embedding does allow partial semantics, i.e. semantics that do not support every language construction. This restriction on classy deep embedding may be lifted by using a data structure for the class constraints [12] instead but this remains future work.

Hybrid approaches between deep and shallow embedding exist as well. For example, Svenningson et al. show that by expressing the deeply embedded language in a shallowly embedded core language, extensions can be made orthogonally as well [23]. This paper differs from those approaches in the sense that it does not require a core language in which all extensions need to be expressible.

Acknowledgements

To appear

References

1. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic Typing in a Statically Typed Language. ACM Trans. Program. Lang. Syst. **13**(2), 237–268 (Apr 1991). https://doi.org/10.1145/103135.103138, https://doi.org/10.1145/103135.103138, place: New York, NY, USA Publisher: Association for Computing Machinery
2. Baars, A.I., Swierstra, D.S.: Typing Dynamic Typing. In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 157–166. ICFP '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/581478.581494, https://doi.org/10.1145/581478.581494, event-place: Pittsburgh, PA, USA
3. Bolingbroke, M.: Constraint Kinds for GHC (Sep 2011), http://blog.omega-prime.co.uk/2011/09/10/constraint-kinds-for-ghc/
4. Boulton, R., Gordon, A., Gordon, M., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Stavridou, V., Melham, T.F., Boute, R.T. (eds.) IFIP TC10/WG. vol. 10, pp. 129–156. Elsevier, North-Holland (1992)

5. Carette, J., Kiselyov, O., Shan, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming **19**(05), 509 (Sep 2009). https://doi.org/10.1017/S0956796809007205, http://www.journals.cambridge.org/abstract_S0956796809007205
6. Cheney, J., Hinze, R.: A Lightweight Implementation of Generics and Dynamics. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 90–104. Haskell '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/581690.581698, https://doi.org/10.1145/581690.581698, event-place: Pittsburgh, Pennsylvania
7. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell University (2003), https://ecommons.cornell.edu/handle/1813/5614
8. GHC Team: Data.Dynamic (2021), https://hackage.haskell.org/package/base-4.14.1.0/docs/Data-Dynamic.html
9. GHC Team: GHC User's Guide Documentation (2021), https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf
10. Gibbons, J., Wu, N.: Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 339–347. ICFP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628138, https://doi.org/10.1145/2628136.2628138, event-place: Gothenburg, Sweden
11. Hinze, R.: Fun With Phantom Types. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, pp. 245–262. Cornerstones of Computing, Bloomsbury Publishing, Palgrave (2003)
12. Hughes, J.: Restricted data types in Haskell. Tech. Rep. UU-CS-1999-28, Department of Information and Computing Sciences, Utrecht University, Paris (1999)
13. Kiselyov, O.: Typed Tagless Final Interpreters. In: Gibbons, J. (ed.) Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures, pp. 130–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_3, https://doi.org/10.1007/978-3-642-32202-0_3
14. Krishnamurthi, S., Felleisen, M., Friedman, D.P.: Synthesizing object-oriented and functional design to promote re-use. In: Jul, E. (ed.) ECOOP'98 — Object-Oriented Programming. pp. 91–113. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
15. Läufer, K.: Combining type classes and existential types. In: Proceedings of the Latin American Informatic Conference (PANEL). ITESM-CEM, Monterrey, Mexico (1994)
16. Läufer, K.: Type classes with existential types. Journal of Functional Programming **6**(3), 485–518 (1996). https://doi.org/10.1017/S0956796800001817, publisher: Cambridge University Press
17. Löh, A., Hinze, R.: Open Data Types and Open Functions. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 133–144. PPDP '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1140335.1140352, https://doi.org/10.1145/1140335.1140352, event-place: Venice, Italy
18. Mitchell, J.C., Plotkin, G.D.: Abstract Types Have Existential Type. ACM Trans. Program. Lang. Syst. **10**(3), 470–502 (Jul 1988). https://doi.org/10.1145/44501.45065, https://doi.org/10.1145/44501.45065, place: New York, NY, USA Publisher: Association for Computing Machinery

19. Najd, S., Peyton Jones, S.: Trees that Grow. Journal of Universal Computer Science **23**(1), 42–62 (Jan 2017)
20. Odersky, M., Läufer, K.: Putting Type Annotations to Work. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 54–67. POPL '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237721.237729, https://doi.org/10.1145/237721.237729, event-place: St. Petersburg Beach, Florida, USA
21. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
22. Reynolds, J.C.: User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In: Gries, D. (ed.) Programming Methodology: A Collection of Articles by Members of IFIP WG2.3, pp. 309–317. Springer New York, New York, NY (1978). https://doi.org/10.1007/978-1-4612-6315-9_22, https://doi.org/10.1007/978-1-4612-6315-9_22
23. Svenningsson, J., Axelsson, E.: Combining Deep and Shallow Embedding for EDSL. In: Loidl, H.W., Peña, R. (eds.) Trends in Functional Programming. pp. 21–36. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
24. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758, publisher: Cambridge University Press
25. Wadler, P.: The expression problem (1998), https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
26. Yorgey, B.A., Weirich, S., Cretin, J., Peyton Jones, S., Vytiniotis, D., Magalhães, J.P.: Giving Haskell a Promotion. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation. pp. 53–66. TLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2103786.2103795, https://doi.org/10.1145/2103786.2103795, event-place: Philadelphia, Pennsylvania, USA

First-Class Data Types in Shallow Embedded Domain Specific Languages using Metaprogramming

Mart Lubbers[0000−0002−4015−4878], Pieter Koopman[0000−0002−3688−0957], and Rinus Plasmeijer

Institute for Computing and Information Sciences,
Radboud University Nijmegen, Nijmegen, The Netherlands
firstname@cs.ru.nl

Abstract. Functional programming languages are excellent candidates for hosting embedded domain specific languages (eDSLs) because of their rich type systems, minimal syntax, referential transparency and composability. However, data types defined in the host language are not automatically available in the embedded language. To do so, all the operations on the data type must be redefined by the programmer for the eDSL resulting in a lot of boilerplate.
This paper shows that with the use of metaprogramming, all first order user-defined data types can be automatically made first class in shallow embedded DSLs. It does so by providing an implementation in Template Haskell for a typical DSL with three different semantics for the language of which one is a compiling semantics. Furthermore, we show that by utilising quasiquotation, there is hardly any burden on the syntax. Finally, the paper also serves as a gentle introduction to Template Haskell.

Keywords: embedded domain specific languages, metaprogramming, Haskell, Template Haskell

1 Introduction

Functional programming languages are excellent candidates for hosting embedded domain specific languages (DSLs) because of their rich type systems, minimal syntax, referential transparency and composability. By expressing the language constructs in the host language, the parser, the type checker and sometimes even the compiler are inherited from the host language. Unfortunately, data types defined in the host language are not automatically available in the embedded language. To do so, all the operations on the data type must be redefined by the programmer for the eDSL.

The two main strategies for embedding DSLs in a functional language are deep embedding (sometimes called initial) and shallow embedding (final). Deep embedding represents the constructs in the language as data types and the semantics as functions over these data types. This makes extending the language with new semantics is effortless by adding another function Conversely, adding

a language construct requires changing the data type and updating all existing semantics individually to support this new constructor. Shallow embedding on the other hand models the language constructs as functions with the semantics embedded. Consequently, adding a construct is easy, i.e. it entails adding another function. Contrarily, adding semantics requires adapting all language constructs. Lifting the functions to typeclasses, i.e. parametrising the constructs over the semantics, allows extension of the language both in constructs and in semantics orthogonally and is called tagless final or class-based shallow embedding [15].

In tagless final embedding, the parser and the compiler are inherited from the host language. While it often is possible to lift values of a user-defined data type to a value in the DSL, it is not possible to interact with it using DSL constructs. In other words, it is not possible to 1. construct values from fields using a constructor, 2. deconstruct values deconstructors or pattern matching, 3. test which constructor the value holds. The functions for this are simply not available automatically in the embedded languages. For some semantics it is possible to directly lift the functions from the host language to the DSL domain, i.e. an interpreter. In other cases—e.g. compiling DSLs such as a compiler or a printer—this is not possible [10]. Thus they have to be defined by hand using a lot of boilerplate code.

To relieve the burden of adding all these functions, metaprogramming, accompanied by custom quasiquoters, can be used. Metaprogramming entails that some parts of the program are generated by a program itself. Quasiquotation is a metaprogramming technique for using the host language, or custom, parser to write syntax fragments verbatim instead of with data types. This allows functions to be added at compile time to the program based on the structure of a user-defined data type.

1.1 Contributions of the paper

This paper shows that with the use of metaprogramming, all first order user-defined data types can be automatically made first class in shallow embedded DSLs. It does so by providing an implementation in Template Haskell for a typical DSL with three different semantics for the language of which one is a compiling semantics. Furthermore, we show that by utilising quasiquotation (see Section 5), there is hardly any burden on the syntax. Finally, the paper also serves as a gentle introduction to Template Haskell.

2 Tagless-final embedding

Tagless-final embedding is an upgrade to standard shallow embedding achieved by lifting all language construct functions to typeclasses. As a result, views on the DSL are data types implementing these classes.

To illustrate the technique, a simple DSL, the language consisting of literals and addition, is outlined. This language, implemented in tagless-final style [4] in Haskell [29] consists initially only of one typeclass containing two functions. `lit`

lifts values from the host language to the DSL domain. The class constraint `Show` is enforced on the `a` type variable to make sure that the value can be printed. Secondly, $\oplus$ represents the addition of two expressions in the DSL.

```
class Expr v where
  lit  :: Show a ⇒ a → v a
  (⊕)  :: Num a ⇒ v a → v a → v a
infixl 6 ⊕
```

The implementation of a view on the DSL is achieved by implementing the typeclasses with the data type representing the view. In the case of our example DSL, an interpreter accounting for failure may be implemented as an instance for the `Either String` type. The standard infix functor application and infix sequential application are used[1].

```
instance Expr (Either String) where
  lit a = Right a
  (⊕) l r = (+) <$> l <*> r
```

2.1 Adding language constructs

To add an extra language construct, we define a new class housing it. For example, to add division we define a new class as follows:

```
class Div v where
  (⊘) :: Fractional a ⇒ v a → v a → v a
infixl 7 ⊘
```

Division is an operation that fails when the right operand is equal to zero. To capture this behaviour, the `Left` constructor from `Either` is used to represent errors. The right-hand side of the division operator is evaluated first. If the right-hand side is zero, the division is not performed and an error is returned instead:

```
instance Div (Either String) where
  (⊘) l r = r ≫= λlr→case lr of
    0 → Left "Division by zero"
    n → (/) <$> l <*> Right n
```

2.2 Adding semantics

To add semantics to the DSL, the existing classes are implemented with a novel data type representing the view on the DSL. First a data type representing the semantics is defined. In this case, the printer is very simple and defined as a *newtype* of a string to store the string representation. Since the language is typed, the printer data type has type variable which is only used during typing, it is a phantom type [17]:

[1] `<$> :: (a → b) → f a → f b;  infixl 4 <$>`
`<*> :: f (a → b) → f a → f b;  infixl 4 <*>`

```
newtype Printer a = P { runPrinter :: String }
```

The class instances for `Expr` and `Div` for the pretty printer are straightforward and as follows:

```
instance Expr Printer where
  lit a = P $ show a
  (⊕) l r = P $ "(" ++ runPrinter l ++ "+" ++ runPrinter r ++ ")"
instance Div Printer where
  (⊘) l r = P $ "(" ++ runPrinter l ++ "/" ++ runPrinter r ++ ")"
```

2.3 Functions

Adding functions to the language is achieved by adding a multi-parameter class to the DSL. The type of the class function allows for the implementation to only allow first order function by supplying the arguments in a tuple. Furthermore, by defining the `Main` type, the DSL forces all functions to be defined at the top level and with the `:-` operator the syntax becomes usable. Finally, by defining the functions as a higher order abstract syntax (HOAS) type safety is achieved [5]. The complete definition looks as follows:

```
class Function a v where
  fun :: ( (a → v s) → In (a → v s) (Main (v u)) ) → Main (v u)
newtype Main a = Main { unmain :: a }
data In a b = a :- b
infix 1 :-
```

Using the `Function` type class can be used to define functions with little syntactic overhead[2]. The following listing shows an expression in the DSL utilising two user-defined functions:

```
   fun λincrement→ (λx      →x ⊕ lit 1)
:- fun λdivide→    (λ(x, y)→x ⊘ y    )
:- Main { unmain = increment (divide (lit 38, lit 5)) }
```

The interpreter only requires one instance of the `Function` class that works for any argument type. In the implementation, the resulting function `g` is simultaneously provided to the definition `def`. Because the laziness of Haskell's lazy let bindings, this results in a fixed point calculation:

```
instance Function a (Either String) where
  fun def = Main $ let g :- m = def g in unmain m
```

The given `Printer` type is not sufficient to implement the `Function` instances, it must be possible to generate fresh function names. After extending the `Printer` type to contain some sort of state to generate fresh function names and a

[2] The *BlockArguments* extension of GHC is used to reduce the number of brackets that allows lambda's to be an argument to a function without brackets or explicit function application using $

`MonadWriter [String]`[3] to streamline the output, we define an instance for every arity. To illustrate this, the instance for unary functions is shown, all other arities are implemented in similar fashion.

```
instance Function () Printer where ...
instance Function (Printer a) Printer where ...
  fun def = Main $ freshLabel >>= λf →
    let g :- m = def $
      λa0 → const ⊥ <$> (tell ["f", show f, " ("] >> a0 >> tell [")"])
    in  tell ["let f", f, " a0 = "] >> g (const ⊥ <$> tell ["a0"])
    >>  tell [" in "] >> unmain m
instance Function (Printer a, Printer b) Printer where ...
```

2.4 Data types

Adding data types, e.g. a list, to the DSL requires the programmer to write functions for all the machinery—constructors, deconstructors and constructor predicate functions—to operate the data type. Lifting the values from the host language to the DSL is already possible using the `lit` function. However, this means that the data type has to have instances for all the class constraints that `lit` enforces, something that is not always possible. Furthermore, once lifted, it is not possible to do anything with values of a user-defined data type. It is not possible to construct new values from expressions in the DSL, nor to deconstruct a value into the fields or to test of which constructor the value is. The machinery for this must thus be added manually, resulting in the following class definitions:

```
data List a = Nil | Cons {hd :: a, tl :: List a}

class ListDSL v where
  -- Constructors
  nil    :: v (List a)
  cons   :: v a → v (List a) → v (List a)
  -- Deconstructors
  unnil  :: v (List a) → v b → v b
  uncons :: v (List a) → (v a → v (List a) → v b) → v b
  -- Predicates
  isNil, isCons :: v (List a) → v Bool
```

Furthermore, instances for all views on the DSL need to be created. We omit the instance for the printer for brevity because it is very similar to the interpreter. The instance for the interpreter is as follows:

```
instance ListDSL (Either String) where
  nil      = Right Nil
  cons l r = Cons <$> l <*> r
  unnil  _ f = f
  uncons d f = d >>= λ(Cons l r) → f (Right l) (Right r)
  isNil  d = d >>= λv → Right $ case v of Nil → True;  Cons _ _ → False
  isCons d = d >>= λv → Right $ case v of Nil → False; Cons _ _ → True
```

[3] `freshLabel :: Printer String; tell :: MonadWriter w m ⇒ w → m ()`

Adding these classes and their corresponding instances is tedious and results in boilerplate code. We therefore resort to metaprogramming, and in particular Template Haskell [33].

3 Template metaprogramming

In metaprogramming, programs have the ability to treat program or program fragments as data. There are several techniques to facilitate metaprogramming and it has been around for many years now [19]. While it has been around for many years, it is considered complex [32].

Template Haskell is GHC's de facto metaprogramming system and it is implemented as a compiler extension together with a library [33]. Readers familiar with Template Haskell can safely skip this section. It adds four main concepts to the language, namely AST data types (Section 3.1), splicing (Section 3.2), quasiquotation (Section 3.3) and reification (Section 3.4). With this machinery, regular Haskell functions can be defined that are called at compile time, inserting generated code into the AST. These functions are monadic functions operating in the **Q** monad. The **Q** monad facilitating failure, reification and fresh identifier generation for hygienic macros [16]. Within the **Q** monad, capturable and non-capturable identifiers can be generated using the `mkName` and `newName` functions respectively. The *Peter Parker principle*[4] holds for the **Q** monad because it executes at compile time. It is possible to (mis)use Template Haskell to for example subvert module boundaries, thus accessing constructors that were hidden, and it may cause side effects during compilation because it is possible to call **IO** operations [36].However, to achieve the goal of embedding data types in a DSL we refrain from using the *unsafe* features.

3.1 Data types

Firstly, for all of Haskell's AST elements, data types are available. With these data types, the entire syntax of a Haskell program can be specified. A selection of datatypes available in Template Haskell is given below:

```
data Dec    = FunD Name [Clause] | SigD Name Type | ClassD Cxt Name ...
data Clause = Clause [Pat] Body [Dec]
data Pat    = LitP Lit | VarP Name | TupP [Pat] | WildP | ...
data Body   = GuardedB [(Guard, Exp)] | NormalB Exp
data Guard  = NormalG Exp | PatG [Stmt]
data Exp    = VarE Name | LitE Lit | AppE Exp Exp | LamE [Pat] Exp Exp
    ...
data Lit    = CharL Char | StringL String | IntegerL Integer | ...
```

When operating in the **Q** monad, lowercase variants of these AST data types are available that lift the constructor to the **Q** monad as as follows:

```
lamE :: [Q Pat] → Q Exp → Q Exp
lamE ps es = LamE <$> sequence ps <*> es
```

[4] With great power comes great responsibility.

3.2 Splicing

Special splicing syntax (`$(...)`) marks functions for compile-time execution. Other than that they always produce a value of an AST data type, they are normal functions. Depending on the context of the splice, the result type is either a list of declarations, a type, an expression or a pattern. The result of this function, when successful, is then spliced into the code and treated as regular code by the compiler. The following listing shows an example of a Template Haskell function generating on-the-fly functions for arbitrary selection of a field in a tuple. When called as `$(tsel 2 4)` it expands at compile time to `(λ(_, f, _, _)→f)`

```
tsel :: Int → Int → Q Exp
tsel field total = do
    f ← newName "f"
    lamE [tupP [if i == total then varP f else wildP | i←[0..t-1]]]
        (varE f)
```

3.3 Quasiquotation

Another feature of Template Haskell is the dual of splicing: Quasiquotation [3]. While it is possible to construct entire programs using the provided data types, it is a little cumbersome. Using oxford-style brackets and single or double apostrophes, verbatim Haskell code is converted automatically to the corresponding AST nodes easing the creation of language constructs. Depending on the context, different quasiquotes are used: – $[\![\ldots]\!]$ or $[\![_e \ldots]\!]$ for expressions – $[\![_d \ldots]\!]$ for declarations – $[\![_p \ldots]\!]$ for patterns – $[\![_t \ldots]\!]$ for types – `'`... for function names – `''`... for type names It is possible to escape the quasiquotes again by splicing. Variables defined within quasiquotes are always fresh—as if defined with `newName`—but it is possible to capture identifiers using `mkName`. For example, $[\![\lambda$ `x→x`$]\!]$ translates to **do** `{ x ← newName "x"; lamE [varP x] (varE x)}` and does not interfere with other `x`'s already defined.

3.4 Reification

The final added construct is reification, querying the compiler for information about a certain name. For example, reifying a type name results in information about the type and the corresponding AST nodes of the type's definition. This information can then be used to generate code according to the structure of data types. Reification is done using the `reify :: Name → Q Info` function.

4 Metaprogramming for generating DSL functions

Metaprogramming can relieve us from writing the boilerplate code by generating it automatically at compile time. The `genDSL` function generates all required boilerplate for the provided name of the type. All type names that are passed

as arguments to this function are made available for the DSL. For example, for the list type, this results in the following definition and Template Haskell call.

```
data List a = Nil | Cons { hd :: a, tl :: List a }
$(genDSL ''List)
```

The `genDSL` function is defined in a different module and has type: `Name → Q Decs`, i.e. given a name, it produces a list of declarations in the **Q** monad. The `genDSL` function first reifies the name to retrieve the structural information. If the name matches a type constructor containing a data type declaration, the structure of the type—the type variables, the type name and information about the constructors—is passed to the `genDSL'` function. The structure sometimes needs some occasional scrubbing first using `genConsName`. Unsupported constructors such as generalised ADT constructors or constructors with universally quantified type variables are rejected. From this structure of the type, `genDSL'` generates a list of declarations containing a class definition (Section 4.1), instances for the interpreter (Section 4.2), and instances of the printer (Section 4.3) respectively.

```
genDSL :: Name → Q [Dec]
genDSL name = reify name >>= λinfo→case info of
  TyConI (DataD cxt typeName tvs mkind constructors derives)
    → mapM getConsName constructors >>= genDSL' tvs typeName
   where
    --Invent names for non record types
    getConsName :: Con → Q (Name, [(Name, Bang, Type)])
    getConsName (NormalC consName fs) = pure (consName,
      [(adtFieldName consName i, b, t) | (i, (b, t))←[0..] `zip` fs])
    getConsName (RecC consName fs) = pure (consName, fs)
    getConsName c = fail $ "genDSL does not support: " ++ show c
  t → fail $ "genDSL does not support: " ++ show t

genDSL' :: [TyVarBndr] → Name → [(Name, [(Name, Bang, Type)])] → Q [
    Dec]
genDSL' typeVars typeName constructors
  = sequence [ mkClass, mkInterpreter, mkPrinter ]
 where
  (consNames, fields) = unzip constructors
  ...

-- Helper function te generate a field name from a constructor name and an index
adtFieldName :: Name → Integer → Name
```

4.1 Classes

The class definition is the same for all types and the function to generate it are defined in the **where** clause of the `genDSL'` function. Using the `classD` constructor, a typeclass is created with a single type variable `v`. The `classD` function takes four arguments: 1. context, which is empty in this case 2. a name, generated from the type name using the `className` function that simply appends the text `DSL` 3. a list

of type variables, in this case the only type variable is the view on the DSL, i.e. `v`. 4. functional dependencies, empty in our case 5. a list of function declarations, the class members. The list of members is a concatenation of the list of constructors, deconstructors, field selectors and constructor predicate functions . Depending on the information needed, either **zipWith** or **map** is used to apply the generation function to all constructors.

```
mkClass :: Q Dec
mkClass = classD (cxt []) (className typeName) [PlainTV (mkName "v")] []
    (  zipWith mkConstructor   consNames fields
    ++ zipWith mkDeconstructor consNames fields
    ++ map     mkPredicate     consNames
    )
```

In all class members, the view `v` plays a crucial role. Therefore, a definition for `v` is accessible for all generation functions. Furthermore, the `res` type represents the *result* type, it is defined as the type including all type variables. This result type is derived from the type name and the list of type variables. In case of the `List` type, `res` is defined as `v (List a)` and is available for as well:

```
v = varT $ mkName "v"
res = v `appT` foldl appT (conT typeName) (map getName typeVars)
 where getName (PlainTV name) = varT name
       getName (KindedTV name _) = varT name
```

Constructors The constructor definitions are generated from just the constructor names and the field information. All class members are defined using the `sigD` constructor that represents a function signature. The first argument is the name of the constructor function, a lowercase variant of the actual constructor name generated using the `constructorName` function. The second argument is the type of the function. A constructor C_k of type T where $T\ tv_0 \ldots tv_n = \ldots \mid C_k\ a_0\ \ldots\ a_m \mid \ldots$ is defined as a DSL function $c_k :: v\ a_0 \rightarrow \ldots \rightarrow v\ a_m \rightarrow v\ (T\ v_0 \ldots v_n)$. In the implementation, first the view `v` is applied to all the field types. Then, the constructor type is constructed by folding over the lifted field types with the result type as the initial value using `mkCFun`.

```
mkConstructor :: Name → [(Var, Bang, Type)] → DecQ
mkConstructor n fs = sigD (constructorName n) $ mkCFun fs res

mkCFun :: [(Var, Bang, Type)] → Q Type → Q Type
mkCFun fs res = foldr (λx y→ [|t $x → $y|]) res
    $ map (appT v . pure . thd3) fs
```

Deconstructors The deconstructor is generated similarly to the constructor as the function for generating the constructor is the second argument modulo a result type change. A deconstructor C_k of type T is defined as a DSL function $c_k :: v\ (T\ v_0 \ldots v_n) \rightarrow (v\ a_0 \rightarrow \ldots \rightarrow v\ a_m \rightarrow v\ b) \rightarrow v\ b$. In the implementation, `mkCFun` is reused to construct the type of the deconstructor as follows:

```
mkDeconstructor :: Name → [(Var, Bang, Type)] → Q Dec
mkDeconstructor n fs = sigD (deconstructorName n)
  [|t $res → $(mkCFun fs [|t $v $b|]) → $v $b|]
  where b = varT $ mkName "b"
```

Constructor predicates The last part of the class definition are the constructor predicates. A function that checks whether the provide value of type T contains a value with constructor C_k. A constructor predicate for constructor C_k of type T is defined as a DSL function $v\ (T\ v_0 \dots v_n) \to v\ Bool$. A constructor predicate—name prefixed by `is`—is generated for all constructors but they all have the same type:

```
mkPredicate :: Name → Q Dec
mkPredicate n = sigD (predicateName n) [|t $res → $v Bool|]
```

4.2 Interpreter

Generating the interpreter for the DSL means generating the class instance for the `Interpreter` data type using the `instanceD` function. The first argument of the instance is the context, this is left empty. The second argument of the instance is the type, the `Interpreter` data type applied to the class name. Finally, the class function instances are generated using the information derived from the structure of the type. The structure for generating the function instances is very similar to the definitions, other than that for the constructor predicates, the field information is required as well as the names.

```
mkInterpreter :: Q Dec
mkInterpreter = instanceD (cxt []) [|t $(conT $ className typeName)
    Interpreter|]
  $  zipWith mkConstructor consNames fields
  ++ zipWith mkDeconstructor consNames fields
  ++ zipWith mkPredicate consNames fields
 where λldots
```

Constructors The interpreter is a view on the DSL that immediately executes all operations in the `Either String` monad. Therefore, the constructor function is implemented by lifting the actual constructor to the monad using sequential application. I.e. `ck a0 ... a1 = pure Ck <*> a0 <*> ... <*> a` To avoid accidental shadowing, fresh names for all the arguments are generated.

```
mkConstructor :: Name → [(Var, Bang, Type)] → Q Dec
mkConstructor consName fs = do
  fresh ← sequence [newName "a" | _←fs]
  fun (constructorName consName) (map varP fresh)
    $ foldl (ifx "<*>") [|pure $(conE consName)|] (map varE fresh)
```

Deconstructors

```
mkDeconstructor :: Name → [(Var, Bang, Type)] → Q Dec
mkDeconstructor consName [] = fun (deconstructorName consName) [] ⟦const
    id⟧
mkDeconstructor consName fs = do
    d ← newName "d"
    f ← newName "f"
    fresh ← mapM (newName . nameBase . fst3) fs
    fun (deconstructorName consName) [varP d, varP f]
        ⟦$(varE d) >>= λ($(pure $ ConP consName (map VarP fresh))) → $(
    fapp f fresh)⟧
  where fapp f = foldl appE (varE f) . map (λf → ⟦pure $(varE f)⟧)
```

Constructor predicates Constructor predicates evaluate the argument and make a case distinction on the result to determine the constructor. To be able to generate a valid pattern in the case distinction, the total number of fields must be known. To avoid having to explicitly generate a fresh name for the first argument, a lambda function is used. In general, the constructor selector for C_k results in the following code

```
isCk f = f >>= λx → case x of
  Ck _ ... ai ... _ → pure True
  _                 → pure False

mkPredicate :: Name → [(Var, Bang, Type)] → Q Dec
mkPredicate n fs = fun (predicateName n) []
  ⟦λx → x >>= λp → case p of
    $(conP n [wildP | _ ← fs]) → pure True
    _ → pure False
  ⟧
```

4.3 Pretty printer

...

5 Pattern matching

It is possible to construct and deconstruct values from other DSL expressions, and to perform tests on the constructor but with a clunky and unwieldy syntax. They have have become first-class citizens in a grotesque way. For example, writing a list-based factorial function in our DSL would be done as follows:

— List.hs

```
data List a = Nil | Cons { hd :: a, tl :: List a }
$(genDSL ''List)
```

— *Main.hs*
```
factorial
  =  fun λfromto→(
       λ(a, b)→if' (a >. b) nil (cons a (a ⊕ lit 1, b))
  :- fun λfacl→(
       λl→if' (isNil l) (lit 1) (unCons λhd tl→hd *. facl tl)
  :- fun λfac→(
       λn→facl (fromto (lit 1, n))
  :- Main{unmain=fac (lit 10)}
```

A similar Haskell implementation is much more consice and less cluttered because of the support for pattern matching. Pattern matching offers a convenient syntax for doing deconstruction and constructor tests at the same time.

— *List.hs*
```
data List a = Nil | Cons { hd :: a, tl :: List a }
```

— *Main.hs*
```
fromto :: Int → Int → [Int]
fromto fro to
  | fro > to = []
  | otherwise = fro : fromto (fro+1) to

factorial :: Int → Int
factorial n = facl (1 `fromto` n)
 where
  facl :: List Int → Int
  facl Nil = 1
  facl (Cons x xs) = x * facl xs
```

5.1 Custom quasiquoters

The syntax burden of DSLs can be reduced using quasiquotation. In Template Haskell, quasiquotation is a convenient way to create Haskell language constructs by entering them verbatim using oxford brackets. However, it is also possible to create so-called custom quasiquoters [23]. If the programmer writes down a fragment of code between *tagged* oxford brackets, the compiler executes the associated quasiquoter functions at compile time. A quasiquoter is a value of the following data type:

```
data QuasiQuoter = QuasiQuoter
    { quoteExp  :: String → Q Exp,  quotePat  :: String → Q Pat
    , quoteType :: String → Q Type, quoteDec  :: String → Q Dec
    }
```

Listing 1.1. The data type for the quasiquoter, containing parsers for all contexts

I.e. code between *dsl* brackets ($[\![_{dsl} \ldots]\!]$) is preprocessed by the `dsl` quasiquoter. Because the functions are executed at compile time, errors—thrown using the `MonadFail` instance of the **Q** monad—in these functions result in compile

time errors. The code produced by the quasiquoter is inserted into the location and (type)checked as if it was written by the programmer.

To illustrate writing a custom quasiquoter, we show the implementation of a quasiquoter for adding binary literals to Haskell. The `bin` quasiquoter is only defined for expressions and parses subsequent zeros and ones as a binary number and splices it back in the code as a regular integer. Thus, $[\![_{bin}\texttt{101010}]\!]$ results in the literal integer expression 42. If an invalid character is used, a compile time error is shown. The quasiquoter is defined as follows:

```
bin :: QuasiQuoter
bin = QuasiQuoter { quoteExp = parseBin }
  where parseBin :: String → Q Exp
        parseBin s = LitE . IntegerL <$> foldM bindigit 0 s
          where bindigit :: Integer → Char → Q Integer
                bindigit acc '0' = pure $ 2*acc
                bindigit acc '1' = pure $ 2*acc + 1
                bindigit acc c   = fail $ "invalid char: " ++ show c
```

5.2 Quasiquotation for pattern matching

Custom quasiquoters allow the DSL user to enter fragments verbatim, bypassing the syntax of the host language. Pattern matching in general is not suitable for a custom quasiquoter because it does not really fit in one of the four syntactic categories for which custom quasiquoter support is available. However, a concrete use of pattern matching interesting enough to be benificial but simple enough for a demonstration is the *simple case expression*. Simple case expressions are expressions that match a single variable. As they are not nested and cover all constructors, they are suitable for immediate conversion to existing DSL primitives [28, Chp. 4.4].

In contrast to the binary literal quasiquoter example, we do not parse the `String` by hand. The parser combinator library *parsec* is used instead to ease the creation of the parser [18]. First the location of the quasiquoted code is retrieved using the `location` function that operates in the **Q** monad. This location is inserted in the parsec parser so that errors are localised in the source code. Then, the `expr` parser is called that returns an `Exp` in the **Q** monad. The `expr` parser uses parsec's commodity expression parser primitive `buildExpressionParser`. The resulting parser translates the string directly into Template Haskell's AST data types in the **Q** monad. The most interesting parser is the parser for a case expression that is an alternative in the basic expression parser `basic`. A case expression is parsed when a keyword **case** is followed by an expression that is in turn followed by a list of matches. A match is parsed when a pattern (`pat`) is followed by an arrow and an expression. The results of this parser are fed into the `mkCase` function that transforms the case into an expression using DSL primitives such as conditionals, deconstructors and constructor predicates. The above translates to the following skeleton implementation:

```
expr :: Parser (Q Exp)
```

```
expr = buildExpressionParser [...] basic
 where basic :: Parser (Q Exp)
       basic
           =   ...
           <|> mkCase <$ reserved "case" <*> expr
                      <* reserved "of"   <*> many1 match
           <|> ...

       match :: Parser (Q Pat, Q Exp)
       match = (,) <$> pat <* reserved "→" <*> expr
```

The `mkCase` function transforms a case expression into let bindings, constructors, deconstructors and constructor predicates. For every pattern except for the wildcard, either a constructor predicate, a literal comparison or a let binding is introduced.

For every case, the generated AST node checks whether the structure of the pattern matches the structure of the value using constructor predicates or comparisons. If the structure matches, let bindings and deconstructors are used to bind all variables to the correct fields. Finally, the expression on the right of the arrow is copied verbatim.

```
mkCase :: ExpQ → [(PatQ, ExpQ)] → ExpQ
mkCase ...
```

6 Discussion

Functional programming languages are especially suitable for embedding DSLs but adding user-defined data types is still an issue. The tagless final style of embedding offers great modularity, extensibility and flexibility. However, user-defined data types are awkward to handle because the built-in operations on them—construction, deconstruction and constructor tests—are not inherited from the host language. We showed that by calling a Template Haskell function with the data type as the argument, the required definitions and views on the novel DSL functions can be generated. Furthermore, by writing a custom quasiquoter, pattern matches in natural syntax are automatically converted to the internal representation of the DSL, thus removing the syntax burden. The use of a custom quasiquoter does require the DSL programmer to write a parser for their DSL, i.e. the parser is not inherited from the host language as is often the case in an embedded DSL. However, by making use of modern parser combinator libraries, this overhead is limited and errors are already caught at compilation.

6.1 Future work

For future work, it would be interesting to see how generating boilerplate for user-defined data types translates from shallow embedding to deep embedding.

In deep embedding, the language constructs are expressed as data types in the host language. Adding new constructs, e.g. constructors, deconstructors and constructor tests, for the user-defined data type therefore requires extending the data type. Techniques such as data types à la Carte [35] and open data types [22] show that it is possible to extend data types orthogonally but whether metaprogramming can still readily be used is something that needs to be researched.

Another venue of research is to try to find the limits of this technique in regards to richer data type definitions. It would be interesting to see whether it is possible to apply the technique on data types with existentially quantified type variables or full-fledged generalised ADTs [13]. It is not possible to straightforwardly lift the deconstructors to typeclasses because existentially quantified type variables will escape. Rank-2 polymorphism offers tools to define the types in such a way that this is not the case anymore. However, implementing compiling views on the DSL is complicated because it would require inventing values of an existentially quantified type variable to satisfy the type system which is difficult.

6.2 Related work

Generic or polytypic programming is a promising technique at first glance for automating the generation of function implementations [21]. However, while it is possible to define a function that works on all first-order types, adding a new function with a new name to the language is not possible.

Much research is going into optimising EDSL techniques but embedding data types and pattern matching is mostly uncharted territory. Atkey et al. first describe embedding pattern matching in a DSL by giving patterns an explicit representation in the DSL by using pairs, sums and injections [2, Section 3.3]. McDonell et al. extend on this idea and use it in deep embedding, again by purely within the concrete syntax of the host language [24] Their approaches differ from this work in the sense that all its functionality is expressed in terms of the concrete syntax of the host language, resulting in an overhead. While it does not require an extra metaprogramming step, the syntax is clunky and it is only possible to match on data types using the structure of the type and not the name of the fields. Furthermore, Young et al. added pattern matching to a deeply embedded DSL using a compiler plugin [39]. This plugin implements a `externalise :: a → E a` function that allows lifting all machinery required for pattern matching automatically from the host language to the DSL. Under the hood, this function translates the pattern match to constructors, deconstructors, constructor predicates. The main difference with this work is that it requires a compiler plugin while our metaprogramming approach works on any compiler supporting a metaprogramming system similar to Template Haskell.

6.3 Related work on Template Haskell

Metaprogramming in general is a very broad research topic and has been around for years already. We therefore do not claim an exhaustive related work on all

aspects of metaprogramming but have tried to present all research on metaprogramming in Template Haskell. Czarnecki et al. provide a more detailed comparison of different metaprogramming techniques. They compare staged interpreters, metaprogramming and templating by comparing MetaOCaml, Template Haskell and C++ templates [7]. Template Haskell has been used to implement related work. They all differ slightly in functionality from our domain and can be divided into several categories.

Generating extra code Using Template Haskell or other metaprogramming systems it is possible to add extra code to you program. The original Template Haskell paper showed that it is possible to create variadic functions such as `printf` using Template Haskell that would be almost impossible to define without [33]. Hammond et al. used Template Haskell to generate parallel programming skeletons [12]. In practise, this means that the programmer selects a skeleton and, at compile time, the code is massaged to suit the pattern and information about the environment is inlined for optimisation.

Polak et al. implemented automatic GUI generation using Template Haskell [30]. Duregård et al. wrote a parser generator using template haskell and the custom quasiquoting facilities [8]. From a specification of the grammar, given in verbatim using a custom quasiquoter, a parser is generated at compile time. Shioda et al. used metaprogramming in the D programming language to create a DSL toolkit [34]. They also programmatically generate parsers and a backend for either compiling or interpretering the IR.

Optimisation Besides generating code, it is also possible to analyse existing code and perform optimisations. Yet, this is dangerous territory because unwantedly the semantics of the optimised program may be slightly different than the original program. For example, Lynagh implemented various optimisations in Template Haskell such as automatic loop unrolling [20]. The compile time executed functions analyse the recursive function and unroll the recursion to a fixed depth to trade execution speed for program space. Also, O'Donnoll embedded Hydra, a hardware description language, in Haskell utilising Template Haskell [27]. Using intensional analysis of the AST, it detects cycles by labelling nodes automatically so that it can generate netlists. Alternatively this could be done using a monad but this hampers equational reasoning greatly, which is a key property of Hydra. Finally, Viera et al. present an a way of embedding attribute grammars in Haskell in a staged fashion [38]. Checking several aspects of the grammar is done at compile time using Template Haskell while other safety checks are performed at runtime.

Compiler extension Sometimes, expressing certain functionalities in the host languages requires a lot of boilerplate, syntax wrestling or other pains. Metaprogramming can relief some of this stress by performing this translation to core constructs automatically. For example, implementing generic—or polytypic—

functions in the compiler is a major effort. Norell et al. used Template Haskell to implement the generic machinery required to implement generic functions compiletime [26]. Adams et al. explores also implement generic programming using Template Haskell to speed things up considerably compared to regular generic programming [1]. Clifton et al. used Template Haskell with a custom quasiquoter to offer skeletons for workflows and embed foreign function interfaces in a DSL [6]. Eisenberg et al. showed that it is possible to programmatically lift some functions from the function domain to the type domain, i.e. type families[9]. Furthermore, Seefried et al. argued that it is difficult to do some optimisations in EDSLs and that metaprogramming can be of use there [31]. They use Template Haskell to change all types to unboxed types, unroll loops to a certain depth and replace some expressions by equivalent more efficient ones. Torrano et al. showed that it is possible to use Template Haskell to perform a strictness analysis and perform let to case translation [37]. Both applications are examples of compiler extensions that can be implemented using Template Haskell. Another example of such a compiler extension is shown by Gill et al. [11]. They created a meta level DSL to describe rewrite rules on Haskell syntax that are applied on the source code at compile time.

Quasiquotation By means of quasiquotation, the host language syntax that usually seeps through the embedding can be hidden. The original Template Haskell quasiquotation paper [23] shows how this can be done for regular expressions, not only resulting in a nicer syntax but syntax errors are also lifted to compile time instead of run time. Also, Kariotis et al. used Template Haskell to automatically construct monad stacks without having to result to the monad transformers library which requires advanced type system extensions [14].

Najd use the compiletime to be able to do normalisation for a DSL, dubbing is QDSLs [25]. They utilise the quasiquation facilities of Template Haskell to convert Haskell DSL code to constructs in the DSL,p applying optimisations such as eliminating lambda abstractios and function applications along the way.

Acknowledgements

This research is partly funded by the Royal Netherlands Navy. Furthermore, we would like to thank the anonymous reviewers for their invaluable comments.

References

1. Adams, M.D., DuBuisson, T.M.: Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming. In: Proceedings of the 2012 Haskell Symposium. pp. 13–24. Haskell '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364506.2364509, event-place: Copenhagen, Denmark

2. Atkey, R., Lindley, S., Yallop, J.: Unembedding Domain-Specific Languages. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. pp. 37–48. Haskell '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1596638.1596644, event-place: Edinburgh, Scotland
3. Bawden, A.: Quasiquotation in Lisp. In: O. Danvy, Ed., University of Aarhus, Dept. of Computer Science. BRICS Notes Series, vol. NS-99-1, pp. 88–99. BRICS, Aarhus, Denmark (1999). https://doi.org/10.1.1.22.1290
4. CARETTE, J., KISELYOV, O., SHAN, C.C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205, publisher: Cambridge University Press
5. Chlipala, A.: Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming. pp. 143–156. ICFP '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1411204.1411226, event-place: Victoria, BC, Canada
6. Clifton-Everest, R., McDonell, T.L., Chakravarty, M.M.T., Keller, G.: Embedding Foreign Code. In: Flatt, M., Guo, H.F. (eds.) Practical Aspects of Declarative Languages. pp. 136–151. Springer International Publishing, Cham (2014)
7. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers, pp. 51–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25935-0_4, https://doi.org/10.1007/978-3-540-25935-0_4
8. Duregård, J., Jansson, P.: Embedded Parser Generators. In: Proceedings of the 4th ACM Symposium on Haskell. pp. 107–117. Haskell '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2034675.2034689, event-place: Tokyo, Japan
9. Eisenberg, R.A., Stolarek, J.: Promoting Functions to Type Families in Haskell. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 95–106. Haskell '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2633357.2633361, event-place: Gothenburg, Sweden
10. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(3), 455–481 (2003). https://doi.org/10.1017/S0956796802004574, publisher: Cambridge University Press
11. Gill, A.: A Haskell Hosted DSL for Writing Transformation Systems. In: Taha, W.M. (ed.) Domain-Specific Languages. pp. 285–309. Springer Berlin Heidelberg, Cham (2009)
12. Hammond, K., Berthold, J., Loogen, R.: AUTOMATIC SKELETONS IN TEMPLATE HASKELL. Parallel Processing Letters **13**(03), 413–424 (2003). https://doi.org/10.1142/S0129626403001380, _eprint: https://doi.org/10.1142/S0129626403001380
13. Hinze, R.: Fun With Phantom Types. In: Gibbons, J., de Moor, O. (eds.) The Fun of Programming, pp. 245–262. Cornerstones of Computing, Bloomsbury Publishing, Palgrave (2003)
14. Kariotis, P.S., Procter, A.M., Harrison, W.L.: Making Monads First-Class with Template Haskell. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell. pp. 99–110. Haskell '08, Association for Computing Machinery, New York,

NY, USA (2008). https://doi.org/10.1145/1411286.1411300, event-place: Victoria, BC, Canada
15. Kiselyov, O.: Typed Tagless Final Interpreters. In: Gibbons, J. (ed.) Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures, pp. 130–174. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_3
16. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic Macro Expansion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 151–161. LFP '86, Association for Computing Machinery, New York, NY, USA (1986). https://doi.org/10.1145/319838.319859, event-place: Cambridge, Massachusetts, USA
17. Leijen, D., Meijer, E.: Domain Specific Embedded Compilers. In: Proceedings of the 2nd Conference on Domain-Specific Languages. pp. 109–122. DSL '99, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/331960.331977, event-place: Austin, Texas, USA
18. Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators For The Real World. Tech. Rep. UU-CS-2001-27, Universiteit Utrecht, Utrecht (2001)
19. Lilis, Y., Savidis, A.: A Survey of Metaprogramming Languages. ACM Comput. Surv. **52**(6) (Oct 2019). https://doi.org/10.1145/3354584, place: New York, NY, USA Publisher: Association for Computing Machinery
20. Lynagh, I.: Unrolling and Simplifying Expressions with Template Haskell (May 2003), http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/
21. Lämmel, R., Jones, S.P.: Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation. pp. 26–37. TLDI '03, Association for Computing Machinery, New York, NY, USA (2003). https://doi.org/10.1145/604174.604179, event-place: New Orleans, Louisiana, USA
22. Löh, A., Hinze, R.: Open Data Types and Open Functions. In: Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 133–144. PPDP '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1140335.1140352, event-place: Venice, Italy
23. Mainland, G.: Why It's Nice to Be Quoted: Quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 73–82. Haskell '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1291201.1291211, event-place: Freiburg, Germany
24. McDonell, T.L., Meredith, J.D., Keller, G.: Embedded Pattern Matching (2021), _eprint: 2108.13114
25. Najd, S., Lindley, S., Svenningsson, J., Wadler, P.: Everything Old is New Again: Quoted Domain-Specific Languages. In: Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 25–36. PEPM '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2847538.2847541, event-place: St. Petersburg, FL, USA
26. Norell, U., Jansson, P.: Prototyping Generic Programming in Template Haskell. In: Kozen, D. (ed.) Mathematics of Program Construction. pp. 314–333. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
27. O'Donnell, J.T.: Embedding a Hardware Description Language in Template Haskell. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers, pp. 143–164. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25935-0_9

28. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice Hall (Jan 1987), https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/
29. Peyton Jones, S.: Haskell 98 language and libraries: the revised report. Cambridge University Press (2003)
30. Polak, G., Jarosz, J.: Automatic Graphical User Interface Form Generation Using Template Haskell (2006)
31. Seefried, S., Chakravarty, M., Keller, G.: Optimising Embedded DSLs Using Template Haskell. In: Karsai, G., Visser, E. (eds.) Generative Programming and Component Engineering. pp. 186–205. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
32. Sheard, T.: Accomplishments and Research Challenges in Meta-programming. In: Taha, W. (ed.) Semantics, Applications, and Implementation of Program Generation. pp. 2–44. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
33. Sheard, T., Jones, S.P.: Template Meta-Programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 1–16. Haskell '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/581690.581691, event-place: Pittsburgh, Pennsylvania
34. Shioda, M., Iwasaki, H., Sato, S.: LibDSL: A Library for Developing Embedded Domain Specific Languages in d via Template Metaprogramming. In: Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences. pp. 63–72. GPCE 2014, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2658761.2658770, event-place: Västerås, Sweden
35. Swierstra, W.: Data types à la carte. Journal of functional programming **18**(4), 423–436 (2008)
36. Terei, D., Marlow, S., Peyton Jones, S., Mazières, D.: Safe Haskell. In: Proceedings of the 2012 Haskell Symposium. pp. 137–148. Haskell '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364506.2364524, event-place: Copenhagen, Denmark
37. Torrano, C., Segura, C.: Strictness Analysis and let-to-case Transformation using Template Haskell (2008)
38. Viera, M., Balestrieri, F., Pardo, A.: A Staged Embedding of Attribute Grammars in Haskell. In: Proceedings of the 30th Symposium on Implementation and Application of Functional Languages. pp. 95–106. IFL 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3310232.3310235, event-place: Lowell, MA, USA
39. Young, D., Grebe, M., Gill, A.: On Adding Pattern Matching to Haskell-Based Deeply Embedded Domain Specific Languages. In: Morales, J.F., Orchard, D. (eds.) Practical Aspects of Declarative Languages. pp. 20–36. Springer International Publishing, Cham (2021)

Creating Interactive Visualizations of TopHat Programs

Mark Gerarts[1], Marc de Hoog[1], Nico Naus[2], and Tim Steenvoorden[1]

[1] Open University, Heerlen, The Netherlands mark.gerarts@gmail.com, mladehoog@gmail.com, tim.steenvoorden@ou.nl
[2] Virginia Tech, Blacksburg VA, United States niconaus@vt.edu

Abstract. Many companies and institutions have automated their business process in workflow management software. The novel programming paradigm Task-Oriented Programming (TOP) provides an abstraction for such software. The largest framework based on TOP, iTasks, has been used to develop real-world software.
Workflow software often includes critical systems. In such cases it is important to reason over the software to ascertain its correctness. The lack of a formal iTasks semantics makes it unsuitable for formal reasoning. To this end TopHat has been developed as a TOP language with a formal semantics. However, TopHat lacks a graphical user interface(GUI), making it harder to develop practical TopHat systems.
In this paper we lay the foundation for TopHat to support GUIs. By combining an existing server framework and user interface framework, we have developed a fully functioning proof of concept implementation in Haskell, on top of TopHat's semantics. We show that implementing a TOP framework is possible using a different host language than iTasks uses. None of TopHat's formal properties have been compromised, since the UI framework is completely separate from TopHat. We run several example programs and evaluate their generated GUI. The results of this paper show that one can have a TOP system with a formal semantics and a user interface. Having such a system improves the quality and verifiability of TOP software in general.

Keywords: Task Oriented Programming · User Interface · Functional Programming

1 Introduction

Workflow software is present in most businesses and institutions nowadays. From health care and first responders, to commerce and industrial processes. Businesses use workflow software to streamline their processes, increase efficiency and reduce costs. In these sectors, reliability of software is crucial.

Previous research into workflow software in the functional programming community aimed to improve reliability, while at the same time reducing the effort of development. This led to the development of Task-Oriented Programming (TOP), a programming paradigm that aims to facilitate working with multiple

people towards a shared goal over the internet. TOP separates the *what* from the *how*. This separation allows programmers to focus on the work that has to be done (*what*) instead of paying attention to design issues, implementation details, operating system limitations, and environment requirements (*how*) [1, 20].

iTasks [1], implemented in the functional programming language Clean [7], is the main TOP framework that has been around for a long time. iTasks has been used to create real-world applications, such as an incident coordination tool for the Dutch coast guard [14]. While this proves its practical usability, iTasks lacks in formalization. The iTasks' semantics are given by its implementation, making it much harder to formally reason about iTasks programs. Previous attempts to mitigate this issue by some of iTasks' creators involved developing a separate iTasks semantics, which allowed them to perform model-based testing, but no formal verification [13]. Formal program verification is necessary to ensure the correctness of critical software, like the incident coordination tool. TopHat is a Domain-Specific Language (DSL) that paves the way to formally reason about task-oriented programs [26], by defining a formal TOP semantics. These semantics have been implemented in Haskell and Idris [3]. Idris is a programming language that features dependent types and a totality checker, which is used to prove properties of TopHat programs. Even though TopHat has an implementation in Haskell, it lacks an interactive user interface.

Motivation

In this paper, we develop such an interactive UI for TopHat as a goal in and of itself, what challenges arise here? But it also answers several fundamental research questions. Before the development of TopHat, it was the case that iTasks, TOP and Clean were tied together very strongly. Previous research even suggests that certain specific Clean features are essential to the implementation of TOP[20]: uniqueness typing, data generic programming, dynamics [30] and a sophisticated backend using interpreted ABC bytecode on clients [18], to name a few. We aim to determine if it is possible to implement a true TOP framework in a different host language, Haskell. TopHat is geared towards formal reasoning, which begs the question, does the addition of a UI to TopHat jeopardize the formal reasoning properties? In other words: is a formal TOP framework that is useful in practice possible? Can we have the best of both worlds?

Besides these research questions and challenges, we expect this work to bring TOP to a bigger audience. The current Clean user base is quite small. Haskell is being used in production code, has a huge number of packages available online and an active online community. Task-oriented programming could benefit from being ported to Haskell, making it available to a large community of both developers and researchers. Developing an interactive UI for TopHat brings this one step closer.

[3] `https://github.com/timjs/tophat-proofs`

Motivated by the above, this paper presents a prototype framework written on top of TopHat's Haskell implementation that is able to create interactive graphical user interfaces of TopHat programs.

Structure

The remainder of this paper is structured as follows: we first provide some background about TOP, including iTasks and TopHat in Section 2. Section 3 introduces our TopHat UI prototype. Section 4 demonstrates the capabilities of our framework, including formal reasoning, using several example TopHat programs. We highlight related work in Section 5. Section 6 reflects on the goals and research questions outlined above. Section 7 concludes.

2 Task-Oriented Programming

This paper builds upon previous TOP research [20, 1, 26]. In this section we describe the basic idea of TOP and two TOP implementations: iTasks and TopHat.

2.1 Task-Oriented Programming

TOP is centred around the concept of *tasks*, which specify the work a user or system has to perform with a high level of abstraction. Tasks can be combined using combinators, allowing complex programs to be constructed from small building blocks [20].

A TOP language provides a description of the work that has to be performed. It is left to a TOP framework to implement technical details such as event handling or creating a User Interface (UI). *iTasks* [1] is such a framework, implemented in the functional programming language Clean [7]. An example of a basic task in iTasks is presented in Listing 1.1. Developers only have to specify that they want the user to enter some information. Passing this task description to iTasks generates an application that prompts the user for their name.

```
1 enterName :: Task String
2 enterName = Hint "What is your name?" @>> enterInformation []
```

Listing 1.1: A simple task prompting the user for their name (Clean)

The TOP paradigm provides an abstraction over workflow software. Instead of having to write a server, database, user interfaces, etc, programmers just define what needs to be done. The complete application is then derived from this specification. TOP is usually embedded in pure functional programming. TOP is made up of four core concepts [20]:

Tasks that describe the work that has to be performed, providing an abstraction that separates the *what* from the *how* [1].
Shared data sources that allow the sharing of data between tasks.

Generics to generate user interfaces based on data types.
Composition of tasks through combinators, allowing the creation of arbitrary large tasks.

Tasks lie at the heart of TOP. A task models the work that has to be done by the system or a user. Combining small tasks allows creating large and complex applications using simple building blocks. Tasks can be combined using combinators: they can be executed sequentially, in parallel, or conditionally. These combinators closely resemble how collaboration happens in real life.

TOP aims to facilitate collaborating with multiple people towards a shared goal, over the internet. Creating complex applications is further facilitated because tasks are first-class citizens: they can be used as input of functions, they can be returned from them, and tasks can contain other tasks as value.

Tasks are interactive and input-driven. When a task receives input it is reevaluated and results in a new task. A task's value can be observed at all times. Tasks can share information with each other, either directly through shared data stores, or by passing task values to continuations.

TOP itself focuses on the domain logic, with tasks providing merely a description of the work that has to be performed. It is left up to a TOP framework to do the heavy lifting, such as generating the user interface, storing and handling data, setting up a web server, and authenticating users.

2.2 iTasks

iTasks [19] is a TOP framework that uses Clean [5] as its host language. It supplements Clean with a set of combinators, model types, and algorithms that allow the construction of task-oriented programs.

An example of a basic task was given in Listing 1.1. iTasks will automatically generate an entire application for this task. It uses generics to deduce that a task of type `String` requires a text input field. In Listing 1.2 we combine the task with a view task using a sequential step combinator. A user has to enter their name and is greeted by the program after stepping to the next task. Figure 1 shows how these steps would look in iTasks.

```
1 greet :: Task String
2 greet = enterName >>!
3           \result -> viewInformation [] ("Hello " +++ result)
```

Listing 1.2: Combining two tasks with a step combinator (Clean)

iTasks is a work in progress, receiving constant updates and improvements. For example, a recent addition is the usage of a distributed, dynamic infrastructure [18]. iTasks has formed the basis of further research as well. Tonic [28] facilitates the subject for non-technical people by providing graphical blueprints of iTasks specifications. It also provides a way to monitor the process while end users are interacting with the application [27]. iTasks acted as the starting point for research into declarative user interfaces, first for SVG images [2] and later as a generalized solution [3].

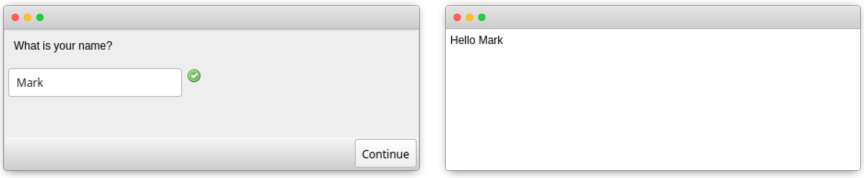


Fig. 1: Entering your name (left) and the result after pressing continue (right)

2.3 TopHat

When software is used in critical applications, it is important that its behavior can be verified and formally reasoned about. iTasks is primarily focused on practical applicability, and therefore lacks this formalization. Testing an iTasks application is time consuming and often incomplete because of the many different execution paths.

TopHat [26] distills TOP's core features to provide a way to reason about task-oriented programs. By employing symbolic execution it is possible to formally verify TopHat programs [17]. Symbolic execution has also been used to provide end-users of tasks with additional feedback [16].

Our work is based on TopHat's Haskell implementation. Listing 1.3 gives the TopHat implementation of the example introduced in Section 2.2. Similar to the iTasks code, this task uses a step combinator to ask a user their name and subsequently greet them.

```
1 greet :: Task h String
2 greet = enter >>? \result -> view ("Hello " ++ result)
```

Listing 1.3: A TopHat task that greets the user (Haskell)

TopHat contains the following set of tasks and combinators:

Editors model user interaction. They are typed containers that are either empty or hold a value. TopHat contains different kinds of editors:
- **Update** contains a predefined value.
- **View** is an editor with a view-only value.
- **Enter** is an editor that is initially empty. Filling it transforms it into an Update editor.
- **Watch** displays the value of a shared data store.
- **Change** is an editor that allows to change the value of a shared data store.

Done and Fail are success and failure end tasks.
Pair combines two tasks (parallel-and).
Choose makes a choice between two tasks (parallel-or).
Step sequentially moves from one task to another.
Share creates a shared data store.
Assign assigns a value to a reference in a shared data store.

2.4 Formal reasoning

iTasks defines tasks as a "state transforming function that reacts to an event, rewrites itself to a reduct and accumulates responses to users" [20]. For combinators, iTasks takes the swiss-army-knife-approach. It defines two combinators that perform a multitude of actions. From these combinators, more simple ones can be constructed. For example, the »* combinator performs sequention, allows the user to choose from a list of tasks, allows automatic progressing tasks, guarded tasks, and stepping on exception. Its definition in the latest version of iTasks is about 100 lines of Clean code, relying on many custom functions [4]. While iTasks is certainly an impressive engineering accomplishment, it is unfit for formal reasoning.

TopHat on the other hand defines tasks as a simple datatype, with three base cases and a small number of simple combinators [26]. The TopHat framework takes care of handling events, rewriting and task rendering. The formal TopHat semantics fits on a single page, and is largely straightforward.

To demonstrate the formal reasoning capabilities of TopHat, a symbolic execution semantics has been developed [17]. For space reasons, we will refrain from repeating syntax and semantics here, but will revisit an example, to use thoughout this paper.

```
let today = 25 Sept 2020 in                                                    1
let provideDocuments = ⊠ Amount ⋈ ⊠ Date in                                    2
let companyConfirm = □ True ◊ □ False in                                       3
let officerApprove = λ invoiceDate. λ date. λ confirmed.                        4
  □ False ◊ if (date − invoiceDate < 365 ∧ confirmed)                          5
    then □ True                                                                6
    else ↯ in                                                                  7
provideDocuments ⋈ companyConfirm ▶                                            8
  λ ⟨⟨invoiceAmount, invoiceDate⟩ , confirmed⟩ .                               9
officerApprove invoiceDate today confirmed ▶ λ approved.                       10
let subsidyAmount = if approved                                                11
  then min 600 (invoiceAmount / 10) else 0 in                                  12
□ ⟨subsidyAmount, approved, confirmed, invoiceDate, today⟩                     13
```

Listing 1.4: Subsidy request and approval workflow at the Dutch tax office.

Listing 1.4 provides the code for a small example task, implementing the process of applying for a tax subsidy. This example was inspired by a collaboration with the Dutch Tax office. The user gets asked to provide documents to back up his or her tax subsidy request for solar panel installation (line 2). The installation company has to confirm that they installed them (line 3), this can be done in parallel (line 8). Finally, a tax officer can either approve or deny the

[4] `https://gitlab.com/clean-and-itasks/itasks-sdk/-/blob/master/Libraries/iTasks/WF/Combinators/Core.icl`

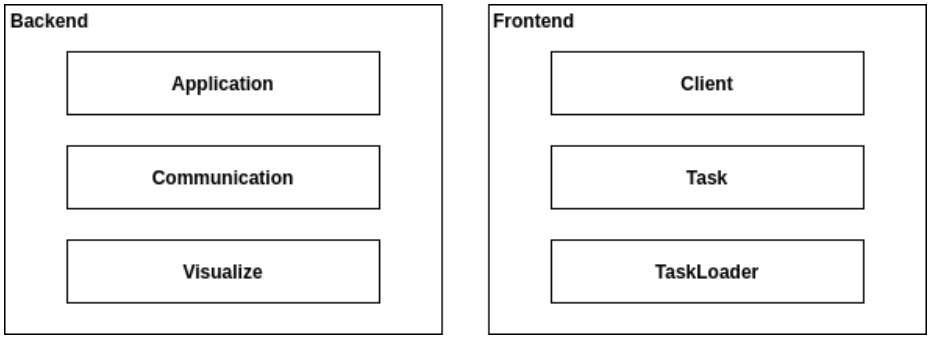


Fig. 2: Architecture. Each box represents a main module.

request (line 4), depending on certain conditions (line 5). After the task has been completed, the subsidy amount is being calculated (line 12), and the details are returned in a view (line 13).

For this task, symbolic execution allowed the authors to prove correctness properties over the code. In Section 4.4 we will take a look at generating a UI using the framework presented in the coming section.

3 TopHat User Interface Framework

In this section we describe our prototype TOP UI framework, which is a proof-of-concept and not a fully fledged TOP framework. Our application supports TopHat tasks as mentioned in Section 2.3. We limit ourselves to a select number of datatypes: only integers, booleans, and strings are supported. Advanced framework features such as multi-user support are out of scope as well. We will reflect on this in Section 6 The framework is published on GitHub[5], along with the examples described below.

Key to our approach is that we leave the task specification of TopHat untouched. This preserves the nice formal properties for which TopHat has been developed in the first place. The prototype UI framework completely relies on the TopHat semantics for handling input and rewriting tasks. The responsibility of the UI framework is to render the task in a web browser, and hand off input that comes in from the user to the TopHat semantics.

The prototype framework is architecturally separated in two parts: the backend and the frontend. Figure 2 shows the main modules of each part. The backend is responsible for initializing tasks and handling communication with TopHat. The frontend renders tasks and allows the user to interact with them. After a comparative study of existing web server and UI frameworks [11], we have selected Servant [21] as our webserver and Halogen [6] for the UI. Other options are discussed in the Section 5. Section 3.1 illustrates the communication between

[5] `https://github.com/mark-gerarts/ou-afstuderen-artefact`

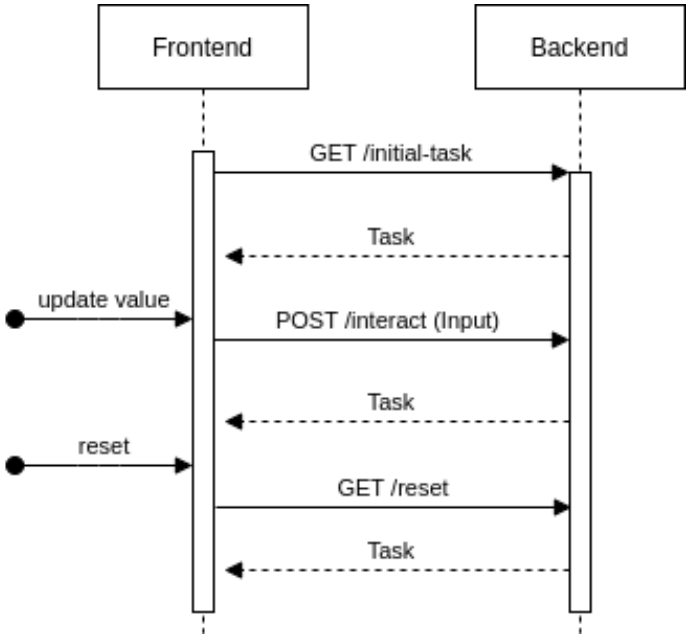


Fig. 3: Communication between frontend and backend. Sequence diagram that displays requests (solid arrows) and responses (dashed arrows). `update value` and `reset` are user actions. Task and Input are JSON objects.

frontend and backend. Section 3.2 explains the working of the backend and the frontend is discussed in Section 3.3.

3.1 Communication between backend and frontend

Figure 3 shows the communication between frontend and backend. The frontend first requests the initial task, which the backend returns using a JSON representation of this task. A user can now interact with the system. In this example, the user updates a value. The frontend sends the input as JSON to the backend, and the backend responds with the updated task. This step can be repeated as necessary. In this case, the user resets the application, which results in the backend resetting back to the initial task.

The frontend is written in PureScript and the backend in Haskell. We choose JSON as data interchange format, because JSON allows custom data structures, it is easy to use, and both backend and frontend support JSON out-of-the-box.

3.2 Backend

The backend is written in Haskell, using Servant [21] as the web server. It has three main responsibilities, which is reflected in its module structure, shown in Figure 2:

1. The Application module loads the application, defines the web server and configures the handlers.
2. The Communication module handles JSON conversion, both encoding tasks to their JSON representation and decoding user input.
3. The Visualize module is intended for the end user. It exposes functions to start the framework, which is demonstrated in Listing 1.5.

```
import Task (Task, enter, view, (>>?))
import Visualize (visualizeTask)

main :: IO ()
main = visualizeTask greet

greet :: Task h String
greet = enter >>? \result -> view ("Hello " ++ result)
```

Listing 1.5: Starting the framework (Haskell)

Application module We create an abstract web application (WAI-application) in the Application module (see the `application` function in Listing 1.6). We define the endpoints, the request and the response formats. For example, see the `TaskAPI` in Listing 1.6. The `server` function provides handlers to serve the initial task, to handle interaction with the frontend and to perform a reset. The remainder of the module consists of functions that expose functionality of TopHat: initializing tasks, deconstructing tasks in a representation that can be sent to the frontend, and interacting with tasks. We have only added key signatures to Listing 1.6.

```
module Application (application, State (..)) where

data State h t = State
    { currentTask :: TVar (Task RealWorld t),
      initialised :: Bool,
      originalTask :: Task RealWorld t
    }

type TaskAPI =
    "initial-task" :> Get '[JSON] TaskDescription
    :<|> "interact"
        :> ReqBody '[JSON] JsonInput :> Post '[JSON] TaskDescription
    :<|> "reset" :> Get '[JSON] TaskDescription

type StaticAPI = Get '[HTML] RawHtml :<|> Raw
type API = TaskAPI :<|> StaticAPI

interactIO :: Input Concrete -> Task RealWorld a -> IO (Task RealWorld a)
initialiseIO :: Task RealWorld a -> IO (Task RealWorld a)
describeIO :: Task RealWorld a -> IO TaskDescription

server :: ToJSON t => State h t -> ServerT API (AppM h t)

application :: ToJSON t => State h t -> Application
```

Listing 1.6: Application module (Haskell)

Communication module In Listing 1.7 we show the core of the communication module. We introduce a new datatype, `TaskDescription`, that holds all data we need to render a task: the task itself (`JsonTask`) and its possible inputs (`InputDescription`), along with the `describe` function that extracts this data from a TopHat task. User input, which is sent back and forth from the client to the server, is defined in `JsonInput`.

```
module Communication (JsonTask (..), TaskDescription (..), describe) where

type JsonTask = Value

type InputDescriptions = List (Input Abstract)

data TaskDescription where
  TaskDescription :: JsonTask -> InputDescriptions -> TaskDescription

instance ToJSON JsonTask

describe :: Members '[Alloc h, Read h] r => Task h t -> Sem r TaskDescription

data JsonInput where
    JsonInput :: Input Concrete -> JsonInput

instance FromJSON JsonInput
```

Listing 1.7: Communication module (Haskell)

Visualize module In Listing 1.8 we show the signatures of the visualize module. We use this module to run the web server in production (`visualizeTask`) or development (`visualizeTaskDevel`) mode. We differentiate between these modes because we implemented live code reloading for development, which requires a bit of additional setup. Both `visualizeTask` and `visualizeTaskDevel` use the `initApp` function. `InitApp` on its turn invokes the application-function of the Application Module.

```
module Visualize (visualizeTask, visualizeTaskDevel) where

initApp :: ToJSON t => Task RealWorld t -> IO Application

visualizeTaskDevel :: ToJSON t => Task RealWorld t -> IO ()

visualizeTask :: ToJSON t => Task RealWorld t -> IO ()
```

Listing 1.8: Visualize module (Haskell)

3.3 Frontend

The frontend renders the UI and provides a way for the user to interact with the it. The code is written in PureScript using the Halogen framework. The frontend consists of three main modules and some auxiliary modules. We explain the main modules:

1. The Client module is the communication layer with the backend. It defines functions which send requests to the backend and handles the responses.
2. The Task module handles JSON encoding and decoding of our domain's datatypes (tasks and user input).
3. The TaskLoader module is the starting point of Halogen and is responsible for rendering the UI.

Client module The client module is responsible for the communication between frontend and backend. The backend sends a response in JSON that consists of two parts: a `Task` and a description of possible inputs. We decode this JSON object into a `TaskResponse`. See Listing 1.9.

```
module App.Client (ApiError, TaskResponse(..), getInitialTask, interact,
    reset) where

data TaskResponse
    = TaskResponse Task (Array InputDescription)

instance decodeJsonTaskResponse :: DecodeJson TaskResponse

getInitialTask :: Aff (Either ApiError TaskResponse)

interact :: Input -> Aff (Either ApiError TaskResponse)

reset :: Aff (Either ApiError TaskResponse)
```

Listing 1.9: Client module (PureScript)

Task module In the Client module we defined a `TaskResponse`. This `TaskResponse` consists of two parts: a `Task` and an array of `InputDescription`. In the Task module we define the decoding process of `Task` and `InputDescription`. See Listing 1.10.

```
module App.Task where

data Task
    = Edit Name Editor
    | Select Name Task Labels
    | Pair Task Task
    | Choose Task Task
    | Step Task
    | Trans Task
    | Done
    | Fail

instance showTask :: Show Task

instance decodeJsonTask :: DecodeJson Task

data Input
    = Insert Int Value
    | Decide Int String

instance showInput :: Show Input

instance encodeInput :: EncodeJson Input

```

```
data InputDescription
    = InsertDescription Int String
    | OptionDescription Int String

instance showInputDescription :: Show InputDescription

instance decodeJsonInputDescription :: DecodeJson InputDescription

```

Listing 1.10: Task module (PureScript)

TaskLoader module The TaskLoader module renders the user interface (the `render` function in Listing 1.11). The module also contains logic to handle events (`handleAction`), for example when a user modifies a value. Finally, the `taskLoader` function (see Listing 1.11) initializes the component.

```
module Component.TaskLoader (taskLoader) where

taskLoader :: forall query input output m. MonadAff m => H.Component query
    input output m

handleAction :: forall output m. MonadAff m => Action -> H.HalogenM State
    Action Slots output m Unit

render :: forall m. MonadAff m => State -> HH.ComponentHTML Action Slots m
```

Listing 1.11: TaskLoader module (PureScript)

4 Examples

We present a few examples to demonstrate how our framework handles TopHat programs. We use a simple multiplication-by-seven machine to demonstrate the Step task and the Edit task (with View, Enter, and Update editors) (Section **??**). The candy vending machine combines the Select and View editor, the Step Task, and the Pair Task to construct a candy machine (Section 4.1). The calorie calculator demonstrates a real-world application of our framework (Section 4.2). The chat sessions demonstrates the use of shared data stores (Section 4.3), and finally Section 4.4 describes UI generation for the tax example from Section 2.4

4.1 Candy vending machine

The candy machine allows a user to choose a chocolate bar and, after the bill is paid, the candy machine returns the bar. The candy machine combines the Edit, Pair and Step task. We have defined different Edit tasks with View and Select editors. The implementation of the initial task is given in Listing 1.12. The Pair combinator is denoted with the operator ><.

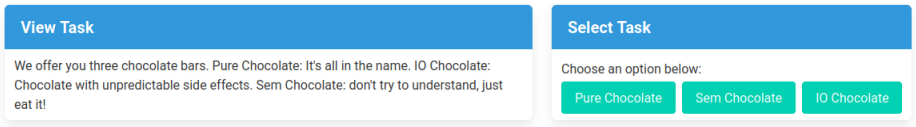
1. After the candy machine is started, the machine displays some introductory text and a selection of chocolate bars (See Figure 4a). This is done using a

Pair Task that consists of two Edit tasks: an Edit task with a View editor and an Edit Task with a Select editor.

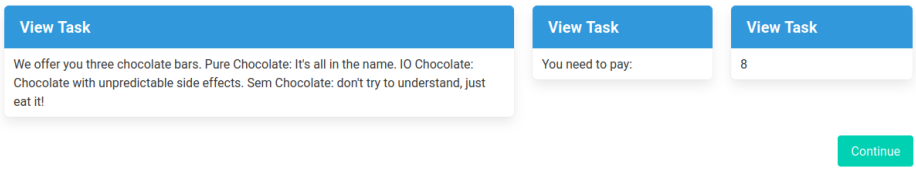
2. Select a chocolate bar. After choosing a bar, the candy machine displays the price of the bar (see Figure 4b). This is done using another Pair Task that consists of an Edit task with a View editor (*"you need to pay:"*) and a Step Task. The Step task consists of two tasks: first a view editor is shown (with the price) and after the step, a select editor is rendered (see Figure 4c).
3. Press the continue button.
4. Insert coins until you have paid the bill (see Figure 4c). The application alternates a view and a select editor.
5. The application shows a view editor to indicate to the user that the bill is paid (see Figure 4d).

```haskell
data CandyMachineMood = Fair | Evil

startCandyMachine :: (Task h (Text, (Text, Text)))
startCandyMachine = view "We offer you three chocolate
    bars. Pure Chocolate: It's all in the name. IO
    Chocolate: Chocolate with unpredictable side effects.
    Sem Chocolate: don't try to understand, just eat
    it!" >< select candyOptions

candyOptions :: HashMap Label (Task h (Text, Text))
candyOptions =
  [ entry "Pure Chocolate" 8,
    entry "IO Chocolate" 7,
    entry "Sem Chocolate" 9
  ]
  where
    entry :: Text -> Int -> (Label, Task h (Text, Text))
    entry name price =
      (name, view "You need to pay:" >< (view price >>? payCandy))

payCandy :: Int -> Task h Text
payCandy bill =
    select (payCoin bill) >>? \billLeft ->
        case compare billLeft 0 of
        EQ -> dispenseCandy Fair
        LT -> dispenseCandy Evil
        GT -> payCandy billLeft

payCoin :: Int -> HashMap Label (Task h Int)
payCoin bill =
  [ coinSize 5,
    coinSize 2,
    coinSize 1
  ]
  where
    coinSize :: Int -> (Label, Task h Int)
    coinSize size = (display size, view (bill - size))

dispenseCandy :: CandyMachineMood -> Task h Text
dispenseCandy Fair =
    view "You have paid. Here is your candy. Enjoy it!"
dispenseCandy Evil =
    view "You have paid too much! Sorry, no change, but here is your candy."
```

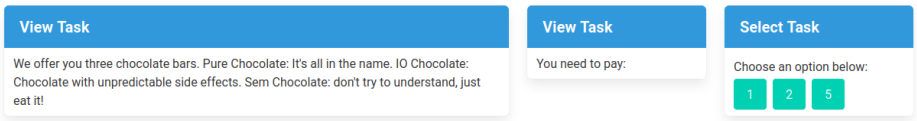
Listing 1.12: Initial Task of the candy vending machine (Haskell)



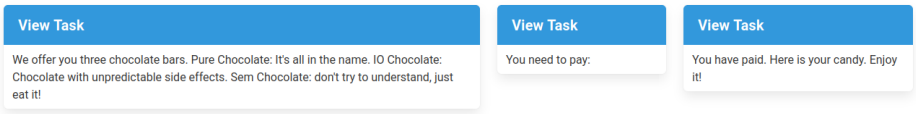
(a) Step 1: Select a chocolate bar



(b) Step 2: Price of the selected candy is shown to the user



(c) Step 3: Insert a coin



(d) Step 4: You have paid the bill

Fig. 4: Different stages of the candy vending machine

4.2 Calorie calculator

To demonstrate a more real-world application that incorporates most task types, we created a calorie calculator. This application calculates how many calories a person should eat per day in order to maintain their weight. The calculation depends on several factors, such as age, weight, and activity level. The application can be broken down in several steps to prompt the user for input, and finally calculating the result. The implementation of the task is given in Listing 1.13.

1. When started, the application presents the user with some information about the calculation using a View editor.
2. After pressing continue, the user is prompted to enter the required data in different steps: height, weight, and age using Enter editors, and gender and activity level using Select editors. Each prompt is wrapped in a Pair task with a View editor on the left side to act as the label. Such a prompt is shown in Figure 5.
3. In the last step the result is displayed using a View editor.

```haskell
data Gender = Male | Female

data ActivityLevel = Sedentary | Low | Active | VeryActive

type Height = Int

type Weight = Int

type Age = Int

calculateCaloriesTask :: Task h Text
calculateCaloriesTask =
    introduction >>? \_ -> do
        (_, height) <- promptHeight
        (_, weight) <- promptWeight
        (_, age) <- promptAge
        (_, gender) <- promptGender
        (_, activityLevel) <- promptActivityLevel
        let calories = calculateCalories gender activityLevel height weight age
        view
        ( "Your resting metabolic rate is: "
            <> display calories
            <> " calories per day."
        )

introduction :: Task h Text
introduction = view <| unlines
    [ "This tool estimates your resting metabolic rate,",
      "i.e. the number of  calories you have to consume",
      "per day to maintain your weight.",
      "Press \"Continue\" to start"
    ]

promptGender :: Task h (Text, Gender)
promptGender =
    view "Select your gender:"
        >< select
            [ "Male" ~> Done Male,
              "Female" ~> Done Female
            ]

promptHeight :: Task h (Text, Height)
promptHeight = view "Enter your height in cm:" >< enter

promptWeight :: Task h (Text, Weight)
promptWeight = view "Enter your weight in kg:" >< enter

promptAge :: Task h (Text, Age)
promptAge = view "Enter your age:" >< enter

promptActivityLevel :: Task h (Text, ActivityLevel)
promptActivityLevel =
    view "What is your activity level?"
        >< select
            [ "Sedentary" ~> Done Sedentary,
              "Low active" ~> Done Low,
              "Active" ~> Done Active,
              "Very Active" ~> Done VeryActive
            ]

-- We omit the actual calculation here since it is a bit lengthy.
calculateCalories :: Gender -> ActivityLevel -> Height -> Weight -> Age -> Int
calculateCalories gender al h w age = ...
```

Listing 1.13: Task of the calorie calculator (Haskell)

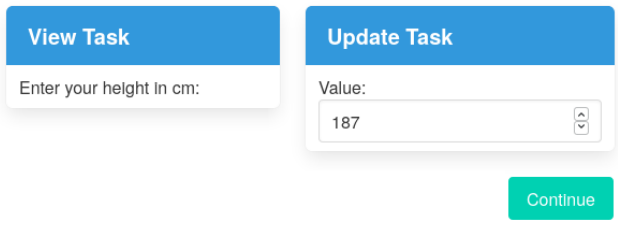


Fig. 5: Prompting the user to enter his/her height

4.3 Chat session

This example uses shared data stores to model a chat session between two users, as displayed in Figure 6. Each user can write messages to the chat history on the left hand side using their respective inputs on the right hand side.

The implementation for this example is given in Listing 1.14. The function `share` creates a data store that can be accessed by multiple tasks, in this case the two `chat` tasks. The `<<=` operator is used to transform the contents of the shared data store.

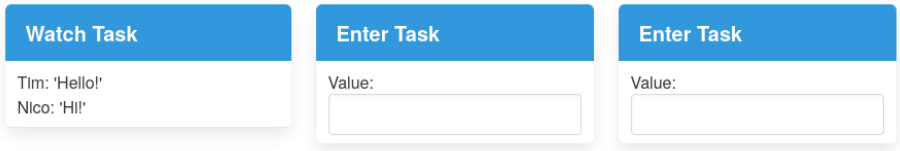


Fig. 6: A chat session using shared data stores.

```
chatSession :: Reflect h => Task h (Text, ((), ()))
chatSession = do
    history <- share ""
    watch history ><
        (chat "Tim" history >< chat "Nico" history)
    where
    chat :: Text -> Store h Text -> Task h ()
    chat name history = repeat <|
        enter >>* ["Send" ~> append history name]

    append :: Store h Text -> Text -> Text -> Task h ()
    append history name msg = do
        history <<= \h ->
        (if h == "" then h else h ++ "\n")
            ++ name ++ ": '"
            ++ msg ++ "'"
```

Listing 1.14: A chat Session using shared data stores (Haskell)

4.4 Tax example

For our final example, we revisit the tax program from Section 2.4.

```
tax :: Task h ((((Amount, Bool), Bool), Date), Date)
tax =
  let today :: Date
      today = 100

      provideDocuments :: Task h (Amount, Date)
      provideDocuments = enter >< enter

      companyConfirm :: Task h Bool
      companyConfirm = enter

      officerApprove :: Date -> Date -> Bool -> Task h Bool
      officerApprove invoiceDate date confirmed =
        view (date - invoiceDate < 365 && confirmed)
   in (provideDocuments >< companyConfirm)
        >>? \((invoiceAmount, invoiceDate), confirmed) ->
          officerApprove invoiceDate today confirmed
            >>? \approved ->
              let subsidyAmount =
                    if approved
                       then min 600 (invoiceAmount `div` 10)
                       else 0
                       in view
             <| unlines
               [ "Subsidy amount: " ++ display subsidyAmount,
                 "Approved: " ++ display approved,
                 "Confirmed: " ++ display confirmed,
                 "Invoice date: " ++ display invoiceDate,
                 "Today: " ++ display today
               ]
```

Listing 1.15: Tax example in Haskell

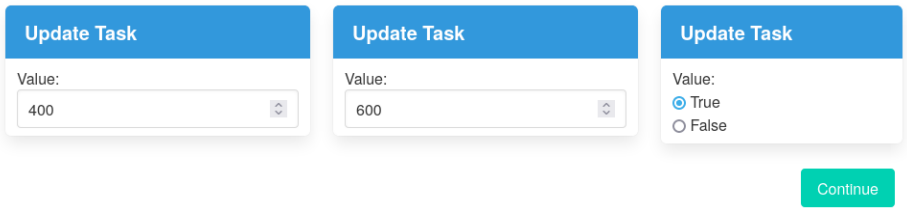
Listing 1.15 gives the Haskell code that implements the task. Compared to the original definition as given in Listing 1.4, the task is nearly identical. The only change made is to the final line, where we have opted for a different presentation of the final result, for simplicity sake.

Figure 7 lists the different stages of the UI for the tax subsidy task. First, the user requesting the subsidy can enter in information (first two tasks), while the company can confirm or deny. Then, the tax officer can verify if the conditions are met, and approve the request. Finally, the outcome is shown.

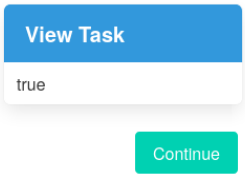
Since we did not have to modify the task at all, besides a minor presentation detail, this task can still be proven correct using symbolic execution. This example clearly illustrates the advantage of TopHat with a UI over the current state-of-the-art in the form of iTasks.

5 Related work

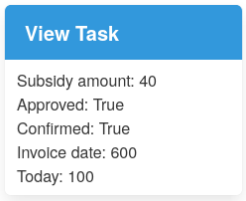
Section 2 presentend related work on TOP and iTasks. In this section, we will briefly discuss Functional reactive programming as an alternative to TOP, as well as alternatives for the UI framework and web server we have used during the development of the UI for TopHat.



(a) Step 1: The citizen enters the request info on the left, the installation company confirms on the right



(b) Step 2: The tax office confirms or denies the request



(c) Step 3: The final outcome of the request is displayed

Fig. 7: Different stages of the tax subsidy application

5.1 Functional Reactive Programming

Functional Reactive programming (FRP) is another approach to UI development using functional programming. FRP is a programming paradigm centered around interactive event-based applications. It has implementations in multiple programming languages, such as Haskell and JavaScript [4].

FRP consists of two main concepts: *behaviors* and *events*. A behavior consists of a value and can be mapped to output, for example a label. Behaviors can depend on other behaviors, so a change in a behavior can propagate through a network of dependent behaviors. An event only occurs at a certain point in time and contains a value. Input is mapped to events, for example the pressing of a key or the position of the mouse cursor. Events can trigger changes in behaviors.

It is worth noting that, while they share some similarities, FRP and TOP are conceptually different. FRP is a paradigm for reactive programming, whereas TOP is a way to model collaboration between users.

5.2 User Interface frameworks

We build upon the Halogen framework to create our prototype, but many other UI frameworks exist in the domain of functional programming. We discuss three of these briefly below.

Elm [9] refers to both Elm, a functional programming language that compiles to JavasScript [8], and TEA [10], a programming pattern that emerged from it.

Elm's ecosystem consists of a large number of available libraries that help in creating web applications.

Miso [12] is a Haskell front-end framework inspired by Elm and Redux. It relies on GHCJS [15], a Haskell-to-JavaScript compiler based on GHC.

Reflex [29] is an FRP framework written in Haskell with support for a variety of platforms, including the web, desktop, and mobile. Reflex applications are modular, which makes growing and refactoring an application efficient and swift.

We have selected PureScript and Halogen because it is a powerful functional programming language that fits our problem domain. Halogen provides an excellent developer experience, has a component based architecture and builds upon PureScript's power and expressiveness.

5.3 Web servers

We have opted for Servant as our web server. Servant provides combinators to implement our features, which makes coding less error prone and time-consuming. Servant is up-to-date, well-maintained, well documented and it is easy to get a working prototype. Below we discuss Yesod and Warp as possible alternatives for the server used in our implementation.

The Yesod Web Framework [23] is a Haskell web framework that allows for rapid development of type-safe, RESTful and high performance web applications [24]. The Yesod Web Framework adds the strengths of Haskell (like type safety) to the web. Especially on the boundaries of Yesod and the world, for example a user enters input or persistent data is loaded, Yesod adds mechanisms to define the expected types [22]. We found that developing a prototype based on Yesod is more difficult than developing a prototype based Servant. We also found that the Yesod Web Framework is too extensive for our purposes [11].

The Warp web server is a light-weight web server that supports the Web Application Interface (WAI) [25]. It is meant to be easy to use and provide easy composition of web services. Because of the design choices to achieve this, the code of a Warp prototype is low-level. This means that implementing all features in this way will be error prone and time-consuming. Therefore, we have chosen Servant. However, Servant also uses Warp as its web server [11].

6 Discussion

We set out this paper with two goals in mind, to answer academic research questions, and to develop an interactive TOP system in Haskell.

Our first questions was, are the advanced Clean features used by iTasks essential for TOP? While these features definitely contribute to the quality of the implementation of iTasks, it is evident that they are not essential.

We were able to develop the UI framework for TopHat in such a way that the original TopHat semantics did not need to be altered. The task specification and its semantics are leveraged by the framework just so it can display the task and pass along input that is entered into the UI.

This answers our second question, do we need to compromise the formal properties of TopHat to build a UI for it. Here the answer is clearly no. The UI framework is completely separate. This is possible due to the fact that TopHat is a deeply embedded DSL, compared to iTasks, which is mostly a shallowly embedded DSL.

The development has been largely a straight forward process. Section 5 lists some details on how we selected the components that make up the UI framework. The implementation has been validated by running several example applications.

As mentioned in Section 3, TOP features such as multi-user support and richer datatypes are out of scope for this publication. We see no technical or formal reason prohibiting them from being included in future versions of the UI framework. As with iTasks, the rendering of values, and editors of values, is generic in the type of the value. Adding support for more complex datatypes would just mean making instances for them for viewing and editing them, similar to how this is done in iTasks. As for multi-user support, this is a limitation in the current version of TopHat. Its developers are already working on adding multi-user support. Once this feature is released, we see no fundamental limitations in supporting this in the UI. The server framework used in the current implementation, Servant, already has extensive support for user authentication, which could be leveraged [6].

7 Conclusion

We conclude that it is indeed possible to create an interactive web UI for TopHat programs without resorting to Clean or iTasks. Even though our implementation does not have the full capabilities of the iTasks framework, we show that all the basic requirements for a TOP framework can be implemented. We support tasks, shared data stores, combinators and generics. This means that we can really have the best of both worlds, a formal semantics for TOP, as well as an interactive UI that can be used to build realistic applications. The source code for our framework is available online, and can thus be leveraged by developers and researchers to advance the field of Task-Oriented Programming.

References

1. Achten, P., Koopman, P., Plasmeijer, R.: An Introduction to Task Oriented Programming, pp. 187–245. Springer International Publishing, Cham (2015)
2. Achten, P., Stutterheim, J., Domoszlai, L., Plasmeijer, R.: Task Oriented Programming with Purely Compositional Interactive Scalable Vector Graphics. pp. 1–13. ACM (2014)
3. Achten, P., Stutterheim, J., Lijnse, B., Plasmeijer, R.: Towards the Layout of Things. pp. 1–13. ACM (2016), https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/pdf/10.1145/3064899.3064905

[6] https://docs.servant.dev/en/stable/tutorial/Authentication.html

4. Bainomugisha, E., Carreton, A., Cutsem, T., Mostinckx, S., Meuter, W.: A survey on Reactive Programming. ACM computing surveys **45**(4), 1–34 (2013)
5. Brus, T., van Eekelen, M.C., Van Leer, M., Plasmeijer, M.J.: Clean—A Language for Functional Graph Rewriting. In: Conference on Functional Programming Languages and Computer Architecture. pp. 364–384. Springer (1987)
6. Burgess and Honeyman et al.: Halogen. `https://github.com/purescript-halogen/purescript-halogen` (2021), version 6.1.2
7. Clean. `https://clean.cs.ru.nl` (2021), version 3.0
8. Czaplicki: Elm. `https://elm-lang.org` (2012), version 0.19.1
9. Czaplicki: Elm: Concurrent frp for functional guis. Senior thesis, Harvard University **30** (2012)
10. The Elm Architecture. `https://guide.elm-lang.org/architecture` (2021), accessed at 2021-07-01
11. Gerarts, M., de Hoog, M.: Creating interactive visualizations of tophat programs (2021)
12. Johnson: Miso. `https://haskell-miso.org` (2020), version 1.7.1
13. Koopman, P.W.M., Plasmeijer, R., Achten, P.: An executable and testable semantics for itasks. In: Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers. pp. 212–232 (2008). https://doi.org/10.1007/978-3-642-24452-0_12, `https://doi.org/10.1007/978-3-642-24452-0\_12`
14. Lijnse, B., Jansen, J.M., Plasmeijer, R., et al.: Incidone: A Task-Oriented Incident Coordination Tool. In: Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM. vol. 12 (2012)
15. Mackenzie et al.: GHCJS. `https://github.com/ghcjs/ghcjs` (2021), version 8.6
16. Naus, N., Steenvoorden, T.: Generating next step hints for task oriented programs using symbolic execution. In: Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers. pp. 47–68 (2020). https://doi.org/10.1007/978-3-030-57761-2_3, `https://doi.org/10.1007/978-3-030-57761-2\_3`
17. Naus, N., Steenvoorden, T., Klinik, M.: A symbolic execution semantics for tophat. In: IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019. pp. 1:1–1:11 (2019). https://doi.org/10.1145/3412932.3412933, `https://doi.org/10.1145/3412932.3412933`
18. Oortgiese, A., van Groningen, J., Achten, P., Plasmeijer, R.: A Distributed Dynamic Architecture for Task Oriented Programming. pp. 1–12. ACM (2017), `https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/pdf/10.1145/3205368.3205375`
19. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. ACM SIGPLAN Notices **42**(9), 141–152 (2007)
20. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-Oriented Programming in a Pure Functional Language. pp. 195–206. ACM (2012)
21. Servant Contributors: Servant – A Type-Level Web DSL. `https://docs.servant.dev/en/stable/index.html` (2021), version 0.18.3
22. Snoyman: Developing Web Applications with Haskell and Yesod. O'Reilly Media, Inc. (2012)
23. Snoyman: The Abominable Snoyman). `https://www.snoyman.com/` (2020)
24. Snoyman: Yesod Web Framework. `https://www.yesodweb.com/` (2020)

25. Snoyman, M.: Warp: A Haskell Web Server. IEEE internet computing **15**(3), 81–85 (2011)
26. Steenvoorden, T., Naus, N., Klinik, M.: TopHat: A formal foundation for task-oriented programming. pp. 1–13. ACM (2019), https://dl-acm-org.ezproxy.elib11.ub.unimaas.nl/doi/10.1145/3354166.3354182
27. Stutterheim, J., Achten, P., Plasmeijer, R., Zsók, V., Porkoláb, Z., Horváth, Z.: Static and Dynamic Visualisations of Monadic Programs. Lecture notes in computer science pp. 341–379 (2019)
28. Stutterheim, J., Plasmeijer, R., Achten, P.: Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In: International Symposium on Trends in Functional Programming. pp. 122–141. Springer (2014)
29. Trinkle et al.: Reflex. https://reflex-frp.org/ (2020), version 0.8.0.0
30. Vervoort, M., Plasmeijer, M.J.: Lazy dynamic input/output in the lazy functional language clean. In: Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers. pp. 101–117 (2002). https://doi.org/10.1007/3-540-44854-3_7, https://doi.org/10.1007/3-540-44854-3_7

Sig-adLib

A Compilable Embedded Language for Synchronous Data-Flow Programming on the Java Virtual Machine [Draft Research Paper]

Baltasar Trancón y Widemann[1,2] and Markus Lepper[2]

[1] Nordakademie, Elmshorn, DE
[2] semantics GmbH, Berlin, DE

Abstract. This paper presents Sig-adLib, an embedded domain-specific language for complex realtime data stream processing tasks on the JVM. It distinguishes a declarative data-flow and an imperative control-flow aspect. Sig-adLib programs can be interpreted, or compiled transparently to JVM bytecode and eventually jit-compiled. Both the interpreter and the compiler are completely modular and extensible. The compiler is fully embedded in the host program. Interpreted and compiled code both run indefinitely on fixed space. Benchmarks indicate a roughly 50-fold speedup by compilation, comparable with hand-coded, statically compiled reimplementations.

1 Introduction

We report on the design and implementation of an embedded domain-specific language (EDSL), Sig-adLib, that adds declarative support for synchronous data-flow computations to the Java platform.

The design of the language covers a middle ground between several related approaches, and has an unusual combination of technical properties: Sig-adLib is a managed language hosted on the Java Virtual Machine (JVM), but its programs can operate indefinitely on a very tight resource budget. It is a dynamic, modular and extensible language, but cooperates at runtime with the just-in-time (jit) compiler of the host environment to generate high-performance code, in the double sense of high throughput and ultra-low latency. It is purely functional in one principal aspect, but procedural in another. It embodies the abstract declarative paradigm of clocked synchronous data-flow programming, but also age-old folklore techniques of low-level imperative stream processing.[3] It has been conceived initially as a backend for compilation of the standalone high-level language Sig [22], but is also productive, educational and fun to use directly.

[3] Data stream programming in this sense can be considered a restricted form of array programming, without random access.

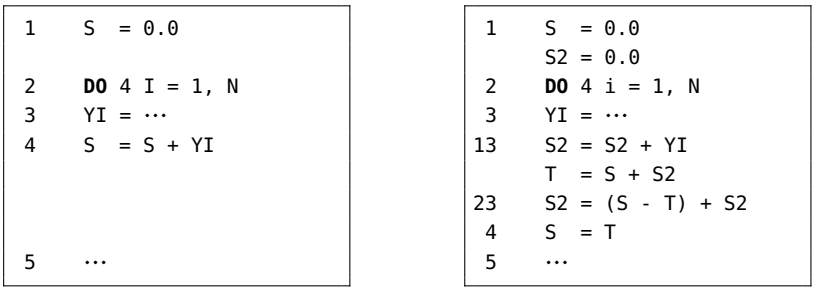


Fig. 1. Naïve (left) vs. Kahan's compensated summation (right, [10]) – imperative style

1.1 Motivating Example

It is a well-known fact that the summation of a stream of floating-point values should not be performed in the naïve way, by simply reducing the stream with the binary addition operations. [10] Except for rare special cases, one operand (the cumulative sum) is bound to outgrow the other (the next value element), and hence the overlap in significant bits and ultimately the precision of the result decrease progressively. A compensating algorithm has been proposed in 1965 [10] already. In Fig. 1 the original Fortran formulation is depicted, juxtaposed with a simplified variant that encodes the naïve summation in the same style.

Arguably, notation has come a long way since then. The depicted code can be considered unnecessarily convoluted by modern standards. In particular, the style is as far removed from *referential transparency* as possible in such a short code fragment: Variables are freely used both *before* and *after* being updated in each loop iteration. A transformation of the code to static single-assignment form [16] is able to reveal that for the variable `S2` alone, there are no less than five distinct regional meanings with different sets of relevant definitions. As a result, reasoning about algorithms is extremely hard and non-scalable in this style. In particular, any perturbation of the assignment statements is quite likely to corrupt the semantics in complicated ways.

It is therefore no coincidence that in areas where this class of algorithms is of practical relevance, such as physical modeling or signal processing, visual approaches that display the algorithmic content as a data-flow network enjoy great popularity (see section 2.3). Fig. 2 depicts the same two algorithms in the graphical style that we have been using in the Sig(-adLib) context. In a data-flow network, most operators are understood as operating repeatedly, once per element of all connected data streams.

The "secret weapon" of the style is the special *delay* operator δ,[4] which delays a data stream by exactly one element. Semantically, an extra initial element is prepended to the stream. Its value must of course be specified somewhere, but is usually omitted from the diagram for visual hygiene reasons. Delay lines allow data-flow networks to be specified without the need to explicitly name stream

[4] variously also written z^{-1}; an abuse of notation from filter theory

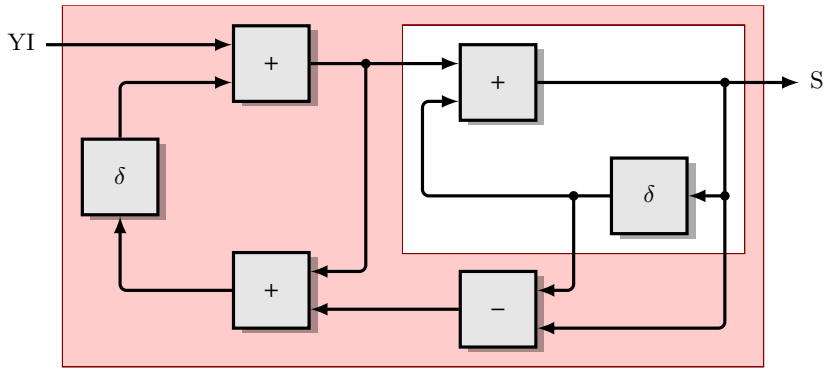


Fig. 2. Naïve (white) vs. Kahan's compensated summation (colored) – data-flow style

variables and distinguish pre-update and post-update access, by sampling the stream after and before the delay operator, respectively.

Furthermore, delayed feedback loops are semantically relatively harmless, but contribute greatly to the expressive power of the approach by supporting *stateful* computations. In [19] we have demonstrated how to translate a data-flow network with arbitrary delayed feedback to a countably infinite system of equations that has both full referential transparency and well-defined operational semantics.

2 Related Work

In this section, we give an overview and comparison of related approaches, in order to clarify and justify the particular position in the solution space taken up by the design of Sig-adLib.

2.1 Functional Reactive Programming

The paradigm of functional reactive programming (FRP) has been developed for computation with time-dependent values in a general sense that subsumes the one discussed above. FRP programs abstract from the nature of time and change (continuous signals, synchronous streams, asynchronous events, etc.) [7] with the help of high-level algebraic structures such as monads or arrows [8].

Fig. 3 depicts an implementation of Kahan's algorithm in Rhine [1], a recent FRP EDSL in Haskell. Note that, of the Rhine language proper, only the `mealy` wrapper is used; the circuit itself is described as a recursive `let` construct. The dense graph structure of this particular data-flow network makes the expression in an arrow-based combinatorial notation exceedingly difficult; cf. Fig. 4.

The abstraction level of FRP is convincingly elegant at the level of denotational semantics, but makes reasoning about resources rather hard [8], and is far removed from traditional programming models for data stream processing. By

```
ksum :: (Monad m, Floating a) ⇒ MSF m a a
ksum = mealy step (0, 0)
  where step y (s, s2) = (t, (t, s2b))
          where s2a = s2 + y
                t   = s + s2a
                s2b = (s - t) + s2a
```

Fig. 3. Kahan's compensated summation – simple (Mealy) FRP style in Rhine

```
ksum :: (Monad m, Floating a) ⇒ MSF m a a
ksum = feedback 0 $
       binop (+) >>> first (sum >>> second (binop (-))) &&& arr id
       >>> arr assoc >>> second (binop (+))
  where sum = feedback 0 $
              binop (+) &&& arr snd >>> arr shuffle
        binop = arr ∘ uncurry
        assoc   ((a, b), c) = (a, (b, c))
        shuffle (a, b)      = ((a, (b, a)), a)
```

Fig. 4. Kahan's compensated summation – arrow-oriented FRP style in Rhine

contrast, SIG-ADLIB is founded on a coalgebraic semantics [19] that connects infinite Mealy machines to causal stream functions. That semantic model comes with a significantly less abstract representation of time, but is highly compatible with the intuitive analogy to digital circuits on one hand, and with traditional imperative programming patterns in the stream processing domain on the other.

2.2 Synchronous Languages

Time-oriented programming, with special emphasis on low-level and safety-critical aspects of the synchronous paradigm, has also been studied extensively in the "French" school of synchronous languages such as Esterel [3], Signal [6] and Lustre [4].

Fig. 5 depicts an implementation of Kahan's summation algorithm in LustreV6 [18]. The operator → delays its right operand by prepending the initial

```
node ksum (y : real) returns (t : real);
let
    s2a = s2 + y
    t   = s + s2a
    s2  = 0 → pre((s - t) + s2a)
    s   = 0 → pre(t)
tel
```

Fig. 5. Kahan's compensated summation – synchronous style in Lustre

```
shuffle = route(2,3,1,1,1,3,2,2);
sum     = (shuffle : +,_)~_;
ksum    = (+ <: (_,(sum : shuffle : -,_))~+) : !,!,_;
```

Fig. 6. Kahan's compensated summation – combinator style in Faust

value of its left operand. Note how the language elegantly avoids the tradeoff between referential transparency and naming parsimony, by virtue of the compositional **pre** operator that distinguishes pre-update from post-update values.

In analogy to hardware description languages [2], it useful in operational semantics of the synchronous paradigm to conceptually distinguish *macro-time* and *micro-time*. Macro-time progresses discretely at global clock tick-like events. Every signal is assigned a constant value per macro-time slice. Micro-time progresses as the updating of the data-flow network propagates by actual value-level computation operations. A synchronous execution model guarantees the absence of macro-time inconsistencies, i.e., the observation of values that have been outdated or prematurely overwritten in micro-time.

A typical resource-efficient implementation strategy achieves synchronicity by data-flow dependency analysis: On one hand, signals are realized as mutable variables, such that updates take effect globally and irreversibly. But on the other hand, computations are sorted in a *causal* micro-time firing order, i.e., each operation may execute and update its result only after its operands have been updated. This strategy implicitly rules out macro-time instantaneous feedback loops, but not delayed ones. For an explication of the latter, see the elimination technique for delay operations proposed in [19].

Faust Fig. 6 depicts an implementation of Kahan's summation algorithm in Faust[12], a functional DSL for synchronous data-flow programming, specifically for the audio domain. Its most distinguishing feature is a very terse syntax that provides a basis of combinators not unlike the algebra of arrows, but with a unique flavor.

2.3 Visual Data-Flow Languages

In several application domains, visual programming tools enjoy great popularity, being considered more accessible and appealing to domain experts. We shall mention just a few very popular examples. They have in common that synchronous stream processing and event-based flow are mixed in a pragmatic fashion that is not grounded in unifying precise semantics.

Max, PD Max/MSP enjoys great popularity in the digital musical community, and has even been hailed as the lingua franca for live performance [14]; PD is its near-identical open-source twin [15]. Fig. 7 depicts an example PD data-flow network (*patch*), where the flow direction and micro-time precedence are indicated by vertical and horizontal alignment, respectively.

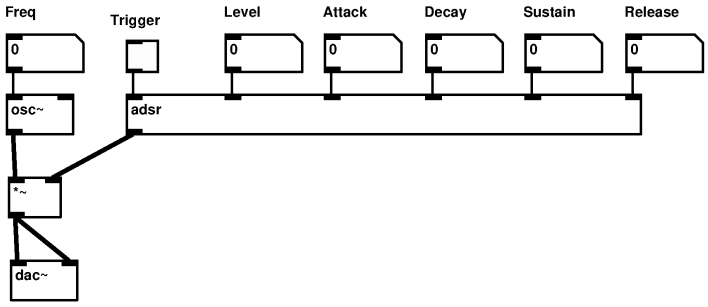


Fig. 7. Example patch (data-flow network) in PD

```
public interface Stream<A> {
    Spliterator<A> spliterator();
}
public interface Spliterator<A> {
    boolean tryAdvance(Consumer<A> action);
}
```

Fig. 8. Java Streams, Low-Level Control API [9]

Matlab/Simulink Simulink is a signal processing system which serves as front-end for Matlab and is widely used in industrial prototyping.[17]

2.4 Java Streams

With Java Version 8, a stream-processing framework API has been introduced into the language. Stream computations are programmed in data-flow style, by setting up *pipelines* with the help of well-known higher-order functions such as `map` and `filter`. Evaluation of actual data elements then takes place on demand. Low-level explicit control is supported with a glorified variant of the `Iterator` pattern, see Fig. 8. By contrast, usual applications use high-level implicit control by means of terminal `reduce`-like operations, see Fig. 9, thus also benefiting from potential transparent parallelization.

The resulting declarative style of usage raises the abstraction level considerably, compared to the procedural, loop-oriented approach of traditional Java patterns. But the abstraction comes at a steep price: Because the *observation* of the current element and the *transition* to the next one are fused in an atomic *consumption* event, data-flow networks are necessarily limited to linear pipelines; *it is not possible to pass the same data element to two simultaneous consumers* (`unzip`). This restriction of expressive power rules out many interesting algorithmic applications, in particular all that rely on feedback.

```
int total = shoppingCart.stream()
                .filter(Item::isAvailable)
                .limit(maxOrderSize)
                .map(Item::getPrice)
                .sum();
```

Fig. 9. Java Streams, High-Level Usage Example

As a result, algorithms such as Kahan's summation cannot be decomposed into Java stream combinators and expressed in the stream EDSL for fundamental reasons. Ironically, the API documentation for the stream operation `sum`[5] suggests that compensated summation may be used, but it has to be implemented under the hood by escaping into the host language. *The language design of* Sig-adLib *can be seen as a variation on the stream framework which trades (mildly) more explicit imperative control at evaluation time for (significantly) enhanced declarative expressive power at construction time.*

3 Design

Sig-adLib is an embedded domain-specific language, i.e., it does not come with a textual syntax or execution environment of its own. Instead, programs are represented as program object graphs (POGs) and executed as method calls by a meta-program in the host language, Java, and share its platform, the JVM. The structure of an embedded program can be written down statically, thus inheriting the syntax of host language, or constructed dynamically by a meta-programming algorithm. Every program object (PO) is at least equipped with an implementation of its own operational semantics, such that the POG as a whole constitutes its own modular, even decentral, interpreter.

Several key features of the design hinge on the characteristic property of Sig-adLib, namely the separation of the concerns of data and control flow. The following subsections describe the respective APIs.

3.1 Core Interfaces and Constructs

Sig-adLib does not require any facilities beyond a vanilla JVM to run. Thus the implementation is a pure Java library. Its API is organized around a small number of simple interfaces for modularity and extensibility, but also provides a large number of implementing classes and factory operations as predefined data-flow network constructs.

[5] https://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.html#sum-- [9]

```
@FunctionalInterface
interface IntSignalSource extends IntSupplier {
    @Override public int getAsInt();
}
```

Fig. 10. SIG-ADLIB Data Flow API (Excerpt)

3.2 Data Flow

The basic unit of SIG-ADLIB data-flow networks is a *signal source*, an abstract consumer-perspective view on a signal, i.e., a strongly typed time-dependent value. Fig. 10 depicts the specialized variant for the unboxed primitive data type `int`;[6] analogous variants exist for other primitive types of the JVM, as well as a generic variant for reference types.

The interface is a specialization of the standard Java functional interface `Supplier`, and as such a valid target type for a lambda expression. Its additional contract stipulates that the observation of the current value (by calling the getter method) does *not* constitute an event for the observed signal, i.e., it must not be the cause of upstream state transitions. This allows multiple observers to share the signal without interference, in marked contrast to the Java Stream API. In general though, values may change arbitrarily between observations due to concurrent activity.

Data-flow networks are constructed from atomic signal sources by various constructs, implemented by constructors and factory operations. Most of these involve the lifting of elementwise operations to signals by memoryless repetition. SIG-ADLIB provides generic constructs for lifting operations of various arities, such as `constant`, `map` and `zip`, as well as specializations for frequently used operations, such as `add` or `equal`. For example, a network that computes the instantaneous average of three input signals x, y, z could be denoted as `x.add(y).add(z).divide(constant(3))`.

3.3 Control Flow

Orthogonally to the data aspect of a SIG-ADLIB computation, its control aspect is described in terms of the `Process` interface, depicted in Fig. 11. A process in the SIG-ADLIB sense is understood as a non-spontaneous provider of signals. A process can be started (or restarted), and commanded to perform a single transition step in macro-time, by calling the `init` and `step` methods, respectively. A process is not entitled to terminate spontaneously; computation ceases implicitly when `step` is called no more. Confer the programming model of the Arduino microcontroller architecture [11], in particular the structure functions `setup` and `loop`[7], respectively.

[6] The redundant method name `getAsInt` is due to the lack of result-type overloading.

[7] Actually, `loop` does not specify a loop but a loop *body*, and hence is a striking misnomer.

```
interface Process {
    public void init();
    public void step(RealtimeContext context);
}
```

Fig. 11. SIG-ADLIB Control Flow API (Excerpt)

```
XSignalSource out1 = ...;
...
ZSignalSource outn = ...;
Process main = ...;

void run(RealtimeContext rc) {
    main.init();
    while (needMoreData()) {
        main.step(rc);
        processData(out1.getAsX(), ..., outn.getAsZ());
    }
}
```

Fig. 12. SIG-ADLIB Driver Loop, General Pattern

Thus the basic usage of a SIG-ADLIB program follows the pattern depicted in Fig. 12. At construction time, POs are allocated and wired, and typed references to all observable data outputs and the control flow entry point retained. The POG can then be run by calling methods in the regular pattern `init (step get*)*`. Reuse ist supported sequentially by simply restarting the pattern, but not concurrently, since POs may have allocated mutable internal state.

SIG-ADLIB processes are compositional in micro-time. The most basic operation combines two processes sequentially, such that `r = p.andThen(q)` results in a process where `r.init()` is equivalent to `{p.init(); q.init();}` and `r.step(c)` is equivalent to `{p.step(c); q.step(c);}`. Note that this operation is very different from sequential composition of (finite) streams in macro-time. Sequential combinators can, and must, be used to construct a causal firing order for a stateful data-flow network.

Besides micro-time sequential composition, various other operations on processes exists. Most notably, rate-changing operations can be used to construct data-flow networks with subsystems operating at different rates. For example, `q = p.every(128)` constructs a process where only every 128th call of `q.step` results in a call of `p.step`.

3.4 Synchronization

While it is often convenient and elegant to separate the data and control aspects of a program, in analogy to digital circuits considerable additional expressive

```
interface IntClockedSignalSource extends IntSignalSource, Process {}

abstract class IntStoredSignalSource implements IntClockedSignalSource {
    protected int value;    // to be written by init & step
    @Override public final int getAsInt() { return value; }
}
```

Fig. 13. SIG-ADLIB Signal Synchronization (Excerpt)

power is gained by connecting them at particular points, thus adding stateful features such as buffering, delay and feedback to the picture.

In the simplest case, one interface for each aspect is attached by multiple inheritance. We call a signal source that is also a process *clocked*. A signal source can be bundled with an arbitrary process (that should of course update its value) using the factory operation `clock`. Fig. 13 (top) depicts the resulting intersection interface for the **`int`** data type. Its additional contract stipulates that the observed value may only change at events caused by the process method `step` of the same PO. In the absence of such calls, a clocked signal source must retain a (temporarily) constant value. Thus multiple observers may never observe inconsistent values for the same macro-time instant. Clocked signal sources can be understood as true data streams.

The simplest implementation of a clocked signal source is a buffering component that provides a field to store and indefinitely retain a data element. Fig. 13 (bottom) depicts the corresponding abstract base class. Subclasses need to implement the methods `init` and `step` to perform the actual computation and write operations. Storing the value of a signal source has numerous uses, most notably the caching of intermediate results for efficient retrieval by multiple observers, as opposed to redundant recalculation in multiple call contexts. For convenience, every signal source supports the factory operation `stored` that constructs such a cache.

Stateful components in general, and stored signal sources in particular, need to be included in the main process of the program, in a causally consistent firing sequence, in order to work correctly. By nature of being an EDSL with local and compositional programming support only, SIG-ADLIB provides no automatic checks on this correspondence requirement on data and control flow, leaving the task to the user. Errors in the control flow of the program manifest as data races and unexpected latency of signals, analogously to wrongly timed clocked digital circuits.

3.5 Delay

Besides value caches, the most important primitive synchronized operation is delay. Delay components are stateful, requiring buffer storage of fixed size and type to retain each value between the macro-time steps where it figures as input and output, respectively. The basic case is a single-step delay as required for

```
delay :: Monad m => a -> MSF m a a
delay = mealy step
  where step (x, p) = (y, q)
          where y = p         -- load
                q = x         -- store
```

Fig. 14. Single-step Delay – FRP style in Rhine

```
interface Register extends Process {
    public Process getLoadPhase();
    public Process getStorePhase();
}
```

Fig. 15. Sig-adLib Load/Store-phased Control Flow (Excerpt)

Kahan's algorithm. Longer delays can be constructed by "shift register" cascading; only for significantly more than a few steps it is beneficial to use optimized implementation strategies such as ring buffers.

A single-step delay requires a buffer of size one. In each step, it forwards the current buffer state to the output (*load*) and the current input to the buffer (*store*). Fig. 14 depicts an implementation in FRP style. The delay elimination technique proposed in [19] hinges on the observation that the load and store phases are separable in the causal firing order, and can accomodate arbitrary feedback loops in between.

Sig-adLib provides a generic interface `Register` for components with separable phases, depicted in Fig. 15. Instances act as `load.andThen(store)` when used as atomic processes, but they also provide a sequential control-flow operation `andMeanwhile` that sandwiches another register or process between the phases.

Fig. 16 depicts the Sig-adLib implementation of single-step delay for the primitive data type **`int`**. Note that each method of each phase performs just a single assignment, and that the double-buffering strategy with `next` and `value` is quite analogous to the design of D-flipflops in hardware.

3.6 Motivating Example, Revisited

The Sig-adLib implementation of Kahan's algorithm is depicted in Fig. 17. Since the host language Java does not support mutually recursive value definitions, cycles are introduced imperatively using the `setInput` method. Otherwise, the program structure is strikingly analogous to the Lustre variant (Fig. 5): The description of data flow is identical; owing to its low-level compositional nature Sig-adLib adds explicit caching of shared values (corresponding to "fan-out solder blobs" in Fig. 2) and a causal firing order.

For ease of reference, colors are used to indicate the distinct aspects of Sig-adLib programming, namely *`control`*, *`data`* and *`synchronization`*.

```
class IntDelay extends IntStoredSignalSource {
    private final int initial;
    private final IntSignalSource input;

    private int next;

    Process load = new Process() {
        @Override public void init() {}
        @Override public void step(RealtimeContext rc) {
            value = next;
        }
    };

    Process store = new Process() {
        @Override public void init() {
            next = initial;
        }
        @Override public void step(RealtimeContext rc) {
            next = input.getAsInt();
        }
    };
}
```

Fig. 16. SIG-ADLIB Single-step Delay

```
FloatClockedSignalSource ksum(FloatSignalSource y) {
    FloatDelay               s    = new FloatDelay(0),
                             s2   = new FloatDelay(0);
    FloatClockedSignalSource s2a  = s2.add(y).stored();
                             t    = s.add(s2a).stored();
    FloatSignalSource        s2b  = s.subtract(t).add(s2a);
    s2.setInput(s2b);
    s.setInput(t);
    return clock(t, s2.andMeanwhile(s).andMeanwhile(s2a.andThen(t)));
}
```

Fig. 17. Kahan's compensated summation – SIG-ADLIB style. (For color legend see text section 3.6)

Note that SIG-ADLIB is not a textual language; thus not the depicted Java code is the actual program, but the resulting POG. *Hence the data-flow aspect of the program is purely declarative, in spite of the use of* `setInput` *in its construction.* While the library API provides many notational constructs to express such directly hosted programs (i.e., as construction statements with trivial linear control flow), other means of production, such as component abstraction and reuse, algorithmic meta-programming, or translation from a different input format, are all effectively equivalent.

4 Execution Environment

Each Sig-adLib PO carries a modular fragment of operational semantics, in the form of its implementations of the interface methods detailed above. Thus the interfaces constitute entry points for a decentral interpreter that is distributed over the abstract syntax POG. The external interface of this interpreter follows an inversion-of-control (IoC) architectural pattern: References to (sub-)processes and signal sources serve as clock inputs and data outputs of the data-flow network, respectively. See Fig. 12 and its discussion given above.

The clock inputs can be driven in various manners: as fast as possible for offline processing, by hardware timers for realtime, or by consumer speed (such as a display frame rate) for modeled time. Iterations of `step` can be performed in a tight loop until terminated externally, in small batches to fill a buffer, or individually in interrupt-handler style for ultra-low-latency computation.

4.1 Memory Usage

The Java language and the JVM have been criticized for not being fully object-oriented, but maintaining the distinction between primitive data types and object reference types.[13] However, the distinction works to our advantage for the efficiency and realtime behavior of Sig-adLib program execution.

The predefined implementations of signal sources of primitive value type have been designed carefully to avoid any dynamic use of objects at runtime; in particular, they do not box or otherwise wrap values, or hold them in collections. By virtue of the distjoint subdomains in both the type system and the instruction set of the JVM, this property is easy to specify, realize and check. Furthermore, typical data-flow algorithms that do require stateful constructs can be limited to fixed amounts of memory that are allocated at construction time of the POG.

As a result of these two properties, Sig-adLib programs, unless explicitly constructed to allocate objects, run idefinitely on constant memory, and do not cause any garbage collector load. Experiments have shown that this eliminates the crucial obstacle to the use of a vanilla JVM, or any managed language environment that is not realtime-hardened for that matter, in low-latency soft-realtime applications such as online audio synthesis.

4.2 Compilation

While a modular, extensible, dynamically meta-programmed embedded language is very convenient for the user, it can be quite challenging to execute efficiently. The fine-grained use of strongly encapsulating interfaces creates numerous abstraction barriers that hinder non-local optimization. Thus the jit compiler of the JVM can not be expected to generate particularly short or fast machine code for Sig-adLib programs.

For this reason, Sig-adLib is also equipped with a dedicated compiler that dynamiclly creates specialized JVM bytecode for a particular whole POG, which

obliterates abstraction barriers and dynamic bindings and is far more suitable for both bytecode interpretation and jit compilation.

In this situation, the apparent curse of modularity turns into a blessing: Because all implementations are hidden behind interfaces, interpreted Sig-adLib POGs can be transparently replaced, wholesale or in parts of arbitrary granularity, by compiled counterparts. For example, the abstract type `IntClockedSignalSource` comes equipped with a default method `IntClockedSignalSource compile()` that generates, loads and instantiates a freshly specialized implementation class on the fly.

The implementation of the bytecode generator is modular and distributed alongside the interpreter. Compilation support is optional for extension classes; since the interpreter is context-free, specialized code can call back into interpretation at any time, if an embedded PO does not come with a specific code generator fragment. At compile time, a context object is passed around, which is responsible for the upper structure of the generated class and acts as a central sink for the bytecode instructions emitted by the per-PO code generator fragments.

Despite the fact that the modular bytecode generator is not able to perform complex non-local optimizations, vast performance gains are obtained by the well-known combination of early binding, control-flow unfolding, constant propagation and aggressive inlining. The latter is aided in particular by the non-recursive nature of micro-time computations. The resulting mostly sequential instruction sequences can then be attacked effectively by the jit compiler with SSA-based non-local optimizations.

The only non-local optimization currently supported by the Sig-adLib compiler is the localization of data cache variables: Any such variables that are introduced by the `stored()` construct and not accessible at the interface of the compiled network may be demoted from heap to stack allocation. This is possible because, with full inlining, the writer and reader code end up in the same method body.

The Sig-adLib compiler is based on the LLJava-live bytecode generator framework, which is an experimental implementation of the staged metaprogramming paradigm for the JVM, and has been applied to educational examples [21] and a real-world Java-hosted EDSL for nondeterministic pattern matching [20]. The basic idea is to pair each interpreter fragment with a corresponding inlining bytecode generator fragment. The *heteroiconic staged metaprogramming* style ensures that the two are reasonably similar in appearance. Fig. 18 depicts an example, namely the inner structure of the class that implements the construct `input.map(op)` for element type type **`int`**. Note how the compiler uses `andThen`, the sequential combinator of a pair of code generators, to realize implicit data flow via the JVM operand stack: the former produces (pushes) a value, and the latter consumes (pops) it. The code generator primitive `storeOutput` replaces the interprocedural **`return`** of the interpreter by a local assignment, in order to behave ideally in an inlining context.

```
final IntSignalSource input;
final CompilableIntUnaryOperator op;

@Override public int getAsInt() {
    return op.applyAsInt(input.getAsInt());
}

@Override public Consumer<CompilationContext> compileGetAsInt() {
    return storeOutput(input.compileGetAsInt()
                       .andThen(op.compileApplyAsInt()));
}
```

Fig. 18. Compilation of SIG-ADLIB construct `input.map(op)` (Excerpt)

5 Case Study: Zero-Crossing Detection

As an example of a realistic problem of nontrivial but manageable algorithmic complexity, we investigate *zero-crossing detection*. The idealized continuous version of this problem is deceptively simple: Given a real-valued time-dependent signal $x : \mathbb{R} \to \mathbb{R}$, detect the points t_0 in time where some $\epsilon > 0$ exists such that

$$\big(\operatorname{sgn} x(t_0 - \delta)\big)\big(\operatorname{sgn} x(t_0 + \delta)\big) = -1 \quad \text{for all } \; 0 < \delta < \epsilon \; .$$

For actual synchronous data-flow implementations, things become more difficult because of four key differences:

1. *Signals are discretized in time by sampling.* Zero-crossing detection must work by comparing the signs of successive sampled values, but regardless whether the actual zero value happens precisely at a sampling point, or in between two adjacent ones.
2. *Signals can have zero intervals.* For many analytic functions, zeroes occur as isolated points. By contrast, arbitrary signals are prone to retaining zero values for arbitrarily long periods. (E.g., consider the output of a sensor that has been switched off to reduce energy consumption.) Zero-crossing detection must not raise false alarms during these periods.
3. *Floating-point numbers come with additional semantics that extend the reals.* Zero-crossing detection must work robustly in the presence of signed infinities and, notoriously, missing values (not-a-number, NaN).
4. *Signals do not continue an infinite past, but start at some particular point in time.* Zero-crossing detection must work right away, with arbitrary initial values.

5.1 Algorithm Design

The most elegant formulation, at least to our knowledge, of an algorithm that decisively settles all of these issues is depicted in Fig. 19. The leftmost column of operations classifies the input signal x into three cases, namely

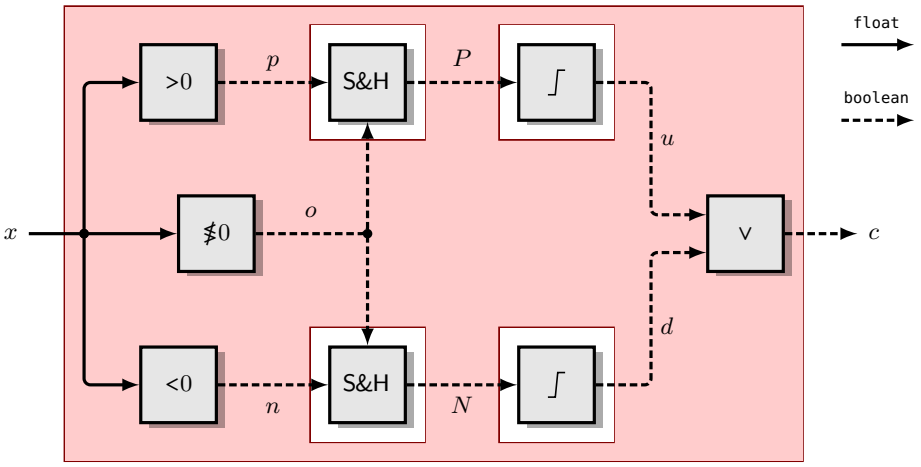


Fig. 19. Data-Flow Network for Zero-Crossing Detector

- positive numbers including infinity (p),
- negative numbers including infinity (n),
- signless numbers, i.e. positive and negative zero (sic!) and NaN (o).[8]

The signals p/n encode the observation that the input signal x is currently definitely in the positive/negative half-space, respectively.

To prevent jitter in case of signless values, in the second column each is fed into a *sample-and-hold* (S&H) component, to the effect that the previous hypothesis value is retained if the current input is signless. Thus the signals P/N encode the observation that the input signal x has last been in the positive/negative half-space respectively.

The third column of operations detect *rising flanks* ($\int$) in the preceding signals, i.e., u/d encode the observation that the input signal has newly moved into the positive/negative half-space, respectively, in the current sampling interval, thereby having crossed zero. Assuming that the direction of crossing is irrelevant, both can be or-ed together, resulting in the output signal c.

By initializing all stateful components, namely the sample-and-hold and rising-flank components highlighted with white background, with *true* for the virtual preceding values, the danger of false positives is averted for signals that start with an indefinitely long sequence of signless values.

The straightforward SIG-ADLIB implementation, together with those of its subcomponents, is depicted in Fig. 20. Note how the different aspects can be segregated into easily discernible clusters in the code; we have found this likely to hold for any well-designed algorithm.

[8] The test for this final condition can of course be replaced by a NOR of the preceding two, but apparent improvement in empirical performance is insignificant.

```
public BooleanClockedSignalSource zeroCrossing() {
    final FloatClockedSignalSource copy = this.stored();
    final BooleanClockedSignalSource
        neut = copy.guard(zero.or(notANumber)).stored(),
        pos  = copy.guard(positive).sampleAndHold(neut, true),
        neg  = copy.guard(negative).sampleAndHold(neut, true),
        up   = pos.rising(true),
        down = neg.rising(true);
    return up.or(down).stored().after(copy, neut, pos, neg, up, down);
}

public BooleanClockedSignalSource sampleAndHold(BooleanSignalSource hold,
                                                boolean initValue) {
    return delayedFeedback(initValue, prev -> hold.choose(prev, this));
}

public BooleanClockedSignalSource rising(boolean initialValue) {
    final BooleanClockedSignalSource prev = delayed(initialValue);
    return zipWith((now, before) -> now & !before, prev)
             .stored().after(prev);
}
```

Fig. 20. SIG-ADLIB Implementation of Zero-Crossing Detector

5.2 Evaluation

The SIG-ADLIB POG as implemented above has been evaluated empirically by a number-crunching benchmark. In this setup, the input stream consists of $K = 10^3$ iterations over a pre-computed array of $M = 10^6$ values generated by an autoregressive random process, such that zero crossings are abundant but occur irregularly. All measurements have been carried out on a dual Core i5-10210U CPU at 1.6 GHz with 8 GiB of RAM, running Ubuntu 20.04LTS and the OpenJDK 11.0.10 64-bit Server VM. Computation times have been measured as whole-loop wallclock times with `System.nanoTime` precision. Every run has been immediately preceded by an identical dry run to allow for jit compiler warmup.

The Java test harness for interpreted/compiled execution of the same POG differs only dynamically, precisely by the occurrence of a call to the factory method `FloatClockedSignalSource.compile()`. Compilation itself runs in few milliseconds, including bytecode generation, loading, verification and instantiation, and requires no external resources besides the LLJAVA-LIVE library. [21]

On average, the interpreted and compiled variant have been measured to take 197.1 ns and 4.2 ns per element, respectively. This translates to a speedup of 47, which indicates that significant abstraction barriers have been removed by compilation. For comparison, a simple baseline experiment with a hand-written monolithic C function, statically compiled with `gcc` 9.3.0 with options `-O3 -fno-`

```
0x7935d141:  vmovss    0x10(%r11,%r9,4),%xmm2      ; load x from array
             ; ...
0x7935d17a:  vxorps    %xmm1,%xmm1,%xmm1
0x7935d17e:  vucomiss  %xmm2,%xmm1                 ; x = 0?
0x7935d182:  jp        0x7935d18a
0x7935d184:  je        0x7935d211                  ; goto side path (0)
0x7935d18a:  vucomiss  %xmm2,%xmm2                 ; x NaN?
0x7935d18e:  jp        0x7935d249                  ; goto side path (NaN)
0x7935d194:  jne       0x7935d249                  ; goto side path (NaN)
0x7935d19a:  movzbl    0x13(%rsi),%r11d            ; load P.prev
0x7935d19f:  movzbl    0x12(%rsi),%r10d            ; load N.prev
0x7935d1a4:  xor       $0x1,%r11d                  ; !P.prev
0x7935d1a8:  xor       $0x1,%r10d                  ; !N.prev
0x7935d1ac:  xor       %r9d,%r9d
0x7935d1af:  mov       $0x1,%ecx
0x7935d1b4:  vucomiss  %xmm2,%xmm1                 ; x > 0?
0x7935d1b8:  mov       $0x1,%ebx
0x7935d1bd:  cmovbe    %r9d,%ebx                   ; p = (x > 0)
0x7935d1c1:  mov       %bl,0x11(%rsi)              ; store p
0x7935d1c4:  mov       %bl,0x13(%rsi)              ; store P
0x7935d1c7:  vucomiss  0xffffff11(%rip),%xmm2      ; x < 0?
0x7935d1cf:  cmovbe    %r9d,%ecx                   ; n = (x < 0)
0x7935d1d3:  mov       %cl,0x10(%rsi)              ; store n
0x7935d1d6:  mov       %cl,0x12(%rsi)              ; store N
0x7935d1d9:  and       %ebx,%r11d                  ; u = P & !P.prev
0x7935d1dc:  and       %ecx,%r10d                  ; d = N & !N.prev
0x7935d1df:  or        %r11d,%r10d                 ; c = u | d
0x7935d1e2:  and       $0x1,%r10d
0x7935d1e6:  mov       %r10l,0x14(%rsi)            ; store c
             ; ...
             ; (side paths)
```

Fig. 21. Disassembly of JIT-Compiled Zero-Crossing Detector

`inline`[9] on the same machine has yielded 4.2 ns as well; the apparent difference is less than the precision of measurement. This finding can be confirmed by disassembly of the resulting jit-compiled machine code. Fig. 21 depicts the relevant excerpt for the hottest path. It can be clearly seen that each Sig-adLib PO translates into a small number of instructions, selected from the most adequate extension layer supported by the host CPU, in this case MMX registers and the AVX instruction set.

[9] Inlining into the main loop must be prevented to preserve the IoC architecture; otherwise benefits from loop unrolling enter the picture.

6 Conclusion

We have demonstrated how data stream processing can be implemented as an EDSL in Java, or for that matter in any mainstream managed language. The SigadLib approach relies on an extensible framework of PO classes and POG constructs. Every node type is equipped at least with decentral self-interpretation, and optionally with a bytecode generator. Programs are constructed by metaprogramming in the host language, and executed by a straightforward IoC API with neglectible overhead.

In contrast to purely declarative language designs, Sig-adLib makes the control flow aspects of programs, namely the causal firing order of POs, visible to the client. While this is perfectly acceptable for use in a compiler backend, it places some unwelcome burden on the human programmer. Since it is well-known that causality is a non-local property [5], a fully automatic solution that is also compatible with the compositional and extensible nature of the language is not easily conceivable. We foresee the possibility of automation by reflection on the completed POG structure, assuming cooperation of all POs, but leave the topic for future research.

The transparent support for compilation for Sig-adLib POGs promises the best of both worlds: Interpreted programs, which can be constructed dynamically (and debugged with vanilla host language IDEs) for rapid prototyping on the one hand, and their compiled counterparts with full participation in JVM code acceleration techniques and competitive real-world performance on the other, just a method call apart. Future experience with realistic projects will tell if there are also substantial downsides, but so far the results are consistent with optimism.

Acknowledgments

Thanks to G. Voutsinos and K. Hufenbach for illuminating discussions.

References

[1] M. Bärenz and I. Perez. "Rhine: FRP with Type-Level Clocks". In: *SIGPLAN Not.* 53.7 (2018), pp. 145–157. DOI: 10.1145/3299711.3242757.

[2] M. Belhadj, R. McConnell, and P. L. Guernic. "A framework for macro- and micro-time to model VHDL attributes". In: *Proc. EURO-VHDL 1993*. IEEE, 1993, pp. 520–525. DOI: 10.1109/EURDAC.1993.410686.

[3] G. Berry and G. Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. DOI: 10.1016/0167-6423(92)90005-V.

[4] P. Caspi et al. "LUSTRE: A Declarative Language for Real-Time Programming". In: *Proc. POPL 1987*. ACM, 1987, pp. 178–188. DOI: 10.1145/41625.41641.

[5] P. Cuoq and M. Pouzet. “Modular Causality in a Synchronous Stream Language”. In: vol. 2028. LNCS. Springer, 2001, pp. 237–251. DOI: 10.1007/3-540-45309-1_16.

[6] T. Gautier and P. Le Guernic. “SIGNAL: A declarative language for synchronous programming of real-time systems”. In: *Proc. FPCA*. Vol. 274. LNCS. Springer, 1987, pp. 257–277. DOI: 10.1007/3-540-18317-5_15.

[7] P. Hudak. “Principles of Functional Reactive Programming”. In: *SIGSOFT Softw. Eng. Notes* 25.1 (2000), p. 59. DOI: 10.1145/340855.340961.

[8] P. Hudak et al. “Arrows, Robots, and Functional Reactive Programming”. In: *Lect. AFP 2002*. Vol. 2638. LNCS. Springer, 2002, pp. 159–187. DOI: 10.1007/978-3-540-44833-4_6.

[9] *Java Platform, Standard Edition 8: API Specification*. Oracle Corporation. 2014–2022. URL: https://docs.oracle.com/javase/8/docs/api/.

[10] W. Kahan. “Further remarks on reducing truncation errors”. In: *Comm. ACM* 8.1 (1965), p. 40. DOI: 10.1145/363707.363723.

[11] *Language Reference*. Arduino. 2019. URL: https://www.arduino.cc/reference/en/.

[12] Y. Orlarey, D. Fober, and S. Letz. “Syntactical and semantical aspects of Faust”. In: *Soft Comput.* 8.9 (2004), pp. 623–632. DOI: 10.1007/s00500-004-0388-1.

[13] N. Ourusoff. “Primitive Types in Java Considered Harmful”. In: *Commun. ACM* 45.8 (2002), pp. 105–106. ISSN: 0001-0782. DOI: 10.1145/545151.545182.

[14] T. Place and T. Lossius. “Jamoma: A modular standard for structuring patches in Max”. In: *Proc. of the International Computer Music Conference 2006*. 2006, pp. 143–146.

[15] *Pure Data Homepage*. 2011. URL: http://puredata.info/docs.

[16] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Global Value Numbers and Redundant Computations”. In: *Proc. POPL 1988*. ACM, 1988, pp. 12–27. DOI: 10.1145/73560.73562.

[17] *Simulink, Dynamic System Simulation for Matlab — Using Simulink*. The MathWorks. 2000. URL: http://www.mathworks.com.

[18] *The Lustre V6 Reference Manual*. Verimag. 2022. URL: https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf.

[19] B. Trancón y Widemann and M. Lepper. “Foundations of Total Functional Data-Flow Programming”. In: *Proc. MSFP 2014*. Vol. 154. EPTCS. 2014, pp. 143–167. DOI: 10.4204/EPTCS.153.10.

[20] B. Trancón y Widemann and M. Lepper. “Improving the Performance of the Paisley Pattern-Matching EDSL by Staged Combinatorial Compilation”. In: *Declarative Programming and Knowledge Management*. Vol. 12057. LNAI. Springer, 2019. DOI: 10.1007/978-3-030-46714-2.

[21] B. Trancón y Widemann and M. Lepper. “LLJava live at the loop: a case for heteroiconic staged meta-programming”. In: *Proc. MPLR 2021*. ACM, 2021, pp. 113–126. DOI: 10.1145/3475738.3480942.

[22] B. Trancón y Widemann and M. Lepper. “On-Line Synchronous Total Purely Functional Data-Flow Programming on the Java Virtual Machine with Sig”. In: *Proc. PPPJ 2015*. ACM, 2015, pp. 37–50. DOI: 10.1145/2807426.2807430.

Appendix

```
final float[] data = new float[N];
// prepare data ...
final FloatClockedSignalSource input = cycle(data);
BooleanClockedSignalSource cross = input.zeroCrossing().after(input);
if (COMPILE)
    cross = cross.compile();
final ConstantRealtimeContext rc = rate(1);

// warm-up
cross.init();
for (int i = 0; i < K * N; i++) {
    cross.step(rc);
    cross.getAsBoolean();
}

// race
cross.init();
final long tstart = System.nanoTime();
for (int i = 0; i < K * N; i++) {
    cross.step(rc);
    cross.getAsBoolean();
}
final long tend = System.nanoTime();
```

Fig. 22. Zero-crossing detection – SIG-ADLIB benchmark harness

```c
#include <stdbool.h>

#define K 1000
#define M 1000000

static float data[M];
static int i;
static bool P, N, Pprev, Nprev;
static volatile bool cross;

void zero_cross_init()
{
  P = true;
  N = true;
  Pprev = true;
  Nprev = true;
}

void zero_cross_step()
{
  float x = data[i];
  i = (i + 1) % M;
  bool p = x > 0;
  bool n = x < 0;
  bool o = (x == 0) | (x != x);
  P = o ? P : p;
  N = o ? N : n;
  bool up = P & !Pprev;
  bool down = N & !Nprev;
  Pprev = P;
  Nprev = N;
  cross = up | down;
}

int main()
{
  zero_cross_init();
  i = 0;
  for (int t = 0; t < K; t++)
    for (int j = 0; j < M; j++)
      zero_cross_step();
}
```

Fig. 23. Zero-crossing detection – C99 baseline implementation

Understanding Algebraic Effect Handlers via Delimited Control Operators

Youyou Cong[1][0000−0003−2315−6182] and Kenichi Asai[2]

[1] Tokyo Institute of Technology
[2] Ochanomizu University
cong@c.titech.ac.jp
asai@is.ocha.ac.jp
https://prg.is.titech.ac.jp/people/cong/
http://pllab.is.ocha.ac.jp/~asai/

Abstract. Algebraic effects and handlers are a powerful and convenient abstraction for user-defined effects. In this paper, we present three results from our ongoing work on enhancing the understanding of effect handlers via control operators. Specifically, we establish two program transformations and a type system for effect handlers, all by reusing the existing results about control operators and their relationship to effect handlers.

Keywords: Algebraic effects and handlers · Delimited control operators · Macro translation · CPS translation · Type systems.

1 Introduction

Algebraic effects [33] and handlers [34] have become an essential element of a programmer's toolbox. Effect handlers provide a convenient interface for defining and composing effects. They also enable concise implementation of sophisticated behavior by giving the programmer access to continuations.

Over the past decade, researchers have been actively studying the theory of effect handlers. As an outcome of these studies, we have obtained various program transformations for effect handlers, which can be used to compile effect handlers into plain λ-terms [16,36,38]. There are also a variety of type systems for effect handlers, in which effects are represented as sets [2], rows [13], or capabilities [7].

We continue the study of effect handlers, but from a different point of view. Instead of directly developing the theory of effect handlers, we *derive* it from the theory of delimited control operators [9,17,27,10]. Control operators have a longer history than effect handlers, and their theory is closely connected to that of effect handlers [12,31]. We aim to enhance the understanding of effect handlers by using the existing results about control operators, as well as the connection between effect handlers and control operators.

In this paper, we discuss a variant of control operators known as `shift0` and dollar [28], and a variant of effect handlers that are called *deep* handlers [18]. Our goal is to answer the following research questions.

- The dollar operator extends the more traditional `reset0` operator with a return clause, and this extension is known to cause no change in the expressiveness [28]. Does this result apply to deep effect handlers as well?
- The `shift0` and dollar operators are associated with a CPS translation that can be viewed as a definitional interpreter. What CPS translation can we derive for deep effect handlers from the CPS translation for `shift0` and dollar?
- The typing of `shift0` and dollar is directed by their CPS translation [8,17,27], rather than their direct-style form. What type system can we derive for deep effect handlers using the CPS approach?

We begin by defining a calculus of `shift0`/dollar (Section 2) and another calculus of deep effect handlers (Section 3). We next answer the three questions one by one (Sections 4 to 6). We then discuss related work (Section 7) and conclude with future perspectives (Section 8).

2 λ_{S_0}: A Calculus of Shift0 and Dollar

As a calculus of control operators, we consider a minor variation of Forster et al.'s calculus of `shift0` and dollar [12], which we call λ_{S_0}. In Figure 1, we present the syntax and reduction rules of λ_{S_0}. The calculus differs from that of Forster et al. in that it is formalized as a fine-grain call-by-value calculus [24] instead of call-by-push-value [23]. This means (i) functions are classified as values; and (ii) computations must be explicitly sequenced using the `let` expression. The fine-grain syntax simplifies the CPS translation and type system developed in later sections.

Among the control constructs, $\mathcal{S}_0 k.\, M$ (pronounced "shift") captures a continuation surrounding itself. The other construct $\langle M \mid x.N \rangle$ (pronounced "dollar") computes the main computation M in a delimited context that ends with the continuation N[3].

There are two reduction rules for the control constructs. If the main computation of dollar evaluates to a value V, the whole expression evaluates to the ending continuation N with V bound to x (rule ($\beta_{\mathcal{S}_0}$)). If the main computation evaluates to $F[\mathcal{S}_0 k.\, M]$, where F is a pure evaluation context that has no dollar surrounding a hole, the whole expression evaluates to M with k being the captured continuation $\lambda y.\, \langle F[\texttt{return}\ y] \mid x.N \rangle$ (rule ($\beta_{\$}$)). Notice that the continuation includes the dollar construct that was originally surrounding the `shift0` operator. This design is shared with the `shift` operator of Danvy and Filinski [9]. Notice next that the body of `shift0` is evaluated without being surrounded by the original dollar. This differentiates `shift0` from `shift`, and allows `shift0` to capture a *meta-context*, i.e., a context that resides outside of the lexically closest dollar.

[3] The original dollar operator proposed by Materzok and Biernacki [28] takes the form $N\,\$\,M$, where M is the main computation and N is an arbitrary expression representing an ending continuation. We are in essence restricting N to be an abstraction $\lambda x.\, N$.

Syntax

$$
\begin{array}{lll}
V, W ::= x \mid \lambda x.\, M & & \text{Values} \\
M, N ::= \texttt{return}\ V \mid V\ V \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \mathcal{S}_0 k.\, M \mid \langle M \mid x.M \rangle & & \text{Computations}
\end{array}
$$

Pure Evaluation Contexts

$$F ::= [] \mid \texttt{let}\ x = F\ \texttt{in}\ M$$

Reduction

$$
\begin{array}{rll}
(\lambda x.\, M)\ V \rightsquigarrow M[V/x] & & (\beta_v) \\
\texttt{let}\ x = \texttt{return}\ V\ \texttt{in}\ M \rightsquigarrow M[V/x] & & (\zeta_v) \\
\langle \texttt{return}\ V \mid x.M \rangle \rightsquigarrow M[V/x] & & (\beta_{\$}) \\
\langle F[\mathcal{S}_0 k.\, M] \mid x.N \rangle \rightsquigarrow M[\lambda y.\, \langle F[\texttt{return}\ y] \mid x.N \rangle / k] & & (\beta_{\mathcal{S}_0})
\end{array}
$$

Fig. 1. Syntax and Reduction Rules of λ_{S_0}

3 λ_h: A Calculus of Effect Handlers

As a calculus of effect handlers, we consider a restricted variant of Hillerström et al.'s calculus of deep handlers [16], which we call λ_h. In Figure 2, we present the syntax and reduction rules of λ_h. The calculus differs from Hillerström et al.'s in that it features unlabeled operations. This means handlers in λ_h can only handle a single operation. The restriction helps us concentrate on the connection to the λ_{S_0} calculus.

Among the effect constructs, $\texttt{do}\ V$ performs an operation with argument V. The other construct $\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}$ computes the main computation M in a delimited context, and handles the result of M using the return clause M_r and the operation clause M_h.

There are again two reduction rules for the effect constructs. If the main computation of a handler evaluates to a value V, the whole expression evaluates to the return clause N with V bound to x (rule (β_h)). If the main computation evaluates to $F[\texttt{do}\ V]$, where F is a pure evaluation context that has no handler surrounding a hole, the whole expression evaluates to the operation clause M_h, with x being V and k being the captured continuation (often called "resumption" in the effect handlers literature) $\lambda y.\, \texttt{handle}\ F[\texttt{return}\ y]\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}$ (rule (β_{do})). Notice that the continuation includes the handler that was originally surrounding the operation. This design is shared with `shift0`, and characterizes handlers in λ_h as deep ones [18]. Notice next that the operation clause is evaluated without being surrounded by the original handler. This is another similarity to `shift0`, and allows handlers to capture a metacontext.

Syntax

$$
\begin{array}{lll}
V, W ::= x \mid \lambda x.\, M & & \text{Values} \\
M, N ::= \texttt{return}\ V \mid V\ V \mid \texttt{let}\ x = M\ \texttt{in}\ M & & \text{Computations} \\
\quad \mid \texttt{do}\ V \mid \texttt{handle}\ M\ \texttt{with}\ \{x.\, M;\ x, k.\, M\} & &
\end{array}
$$

Pure Evaluation Contexts

$$F ::= [] \mid \texttt{let}\ x = F\ \texttt{in}\ M$$

Reduction

$$
\begin{array}{rll}
(\lambda x.\, M)\ V \rightsquigarrow M[V/x] & & (\beta_v) \\
\texttt{let}\ x = \texttt{return}\ V\ \texttt{in}\ M \rightsquigarrow M[V/x] & & (\zeta_v) \\
\texttt{handle}\ \texttt{return}\ V\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\} \rightsquigarrow M_r[V/x] & & (\beta_h) \\
\texttt{handle}\ F[\texttt{do}\ V]\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\} \rightsquigarrow M_h[V/x, f/k] & & (\beta_{do}) \\
\quad \text{where } f = \lambda y.\, \texttt{handle}\ F[\texttt{return}\ y] & & \\
\quad \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\} & &
\end{array}
$$

Fig. 2. Syntax and Reduction Rules of λ_h

4 Adding and Removing the Return Clause

The dollar construct $\langle M \mid x.N \rangle$ in λ_{S_0} is a generalization of the "reset" construct $\langle M \rangle$, which is more commonly found in the continuations literature. The reset construct does not have an ending continuation; it simply evaluates the body M in an empty context. As shown by Materzok and Biernacki [26], the dollar and reset constructs can macro-express [11] each other. That is, there is a pair of local translations, called *macro translations*, that add and remove the ending continuation while preserving the meaning of the program.

It is easy to see that the return clause of an effect handler plays a similar role to the ending continuation of dollar. Thus, we can naturally consider a variant of effect handlers without the return clause. Such handlers are not uncommon in formalizations [36,38,40] as they simplify the reduction and typing rules. Now the reader might wonder: Does the existence of the return clause affects the expressiveness of an effect handler calculus?

In this section, we define macro translations between handlers with and without the return clause. We show that, to remove the return clause, we need to equip the target language with facilities for distinguishing between different kinds of operations. In what follows, we review the macro translations between dollar and reset (Section 4.1), then adapt the translations to effect handlers (Section 4.2), and lastly prove the correctness of the translations (Section 4.3).

4.1 Translating Between Dollar and Reset0

Materzok and Biernacki [28] define the macro translations $[\![\_]\!]_m$ between dollar and `reset0` as follows[4].

From dollar to reset

$$[\![\langle M \mid x.N\rangle]\!]_m = \langle \texttt{let}\ y = [\![M]\!]_m\ \texttt{in}\ \mathcal{S}_0 z.\,(\lambda x.\,[\![N]\!]_m)\ y\rangle$$

From reset to dollar

$$[\![\langle M\rangle]\!]_m = \langle [\![M]\!]_m \mid x.\texttt{return}\ x\rangle$$

The translation from reset to dollar is straightforward: it simply adds a trivial ending continuation. The translation from dollar to reset is more involved: it wraps the computation $[\![M]\!]_m$ around a reset, and inserts a `shift0` to remove the surrounding reset. The removal of reset is necessary for preservation of meaning (the return clause should not be evaluated under the original handler), and is realized by discarding the captured continuation z.

Note that the translation of other constructs is defined homomorphically. For instance, we have $[\![\lambda x.\,M]\!]_m = \lambda x.\,[\![M]\!]_m$.

4.2 Translating Between Handlers with and without Return Clause

Guided by the translations between dollar and reset, we define macro translations between handlers with and without the return clause. We can easily imagine that adding the return clause is simple. To remove the return clause, we must somehow implement the removal of a handler. In a calculus of effect handlers, any non-local control is triggered by an operation call. This means we need to make an operation call when we wish to remove a handler. Unlike `shift0`, however, an operation call does not have an interpretation on its own. This means we need to implement the removing behavior in the surrounding handler, while distinguishing the "return operation" from regular operations.

In Figure 3, we define a calculus of effect handlers without the return clause, which we call λ_h^-. The calculus has labels l and pairs $\langle l, V\rangle$, allowing us to represent the return and regular operations as $\texttt{do}\ \langle \texttt{ret}, V\rangle$ and $\texttt{do}\ \langle \texttt{op}, V\rangle$, respectively. The calculus also has pattern matching constructs, with which we can interpret the two kinds of operations differently in a handler. Note that these facilities are introduced only for the translation purpose. That is, we assume that the user can only program with unlabeled operations; they do not have access to the shaded constructs in Figure 3.

We now define the macro translations between λ_h and λ_h^-[5].

[4] The macro translation is originally defined as:

$$(\lambda k.\,\langle(\lambda x.\,\mathcal{S}_0 z.\,k\ x)\ [\![M]\!]_m\rangle)\ [\![N]\!]_m$$

We adapted the translation by sequencing the application, and by incorporating the fact that N is always a λ-abstraction.

[5] Note that these translations are not designed for a typed setting. Specifically, the translation of regular handlers yields a single pattern variable x representing the arguments of the `ret` and `op` operations, which may be of different types.

Syntax

$$
\begin{array}{rcll}
V, W & ::= & x \mid \lambda x.\, M \mid \langle l, V\rangle & \text{Values} \\
M, N & ::= & \mathtt{return}\ V \mid V\ V \mid \mathtt{let}\ x = M\ \mathtt{in}\ M \mid \mathtt{do}\ V & \text{Computations} \\
 & & \mid \mathtt{case}_p\ V\ \mathtt{of}\ \{\langle l, x\rangle \to M\} \mid \mathtt{case}_l\ l\ \mathtt{of}\ \{\mathtt{ret} \to M; \mathtt{op} \to M\} & \\
 & & \mid \mathtt{handle}\ M\ \mathtt{with}\ \{x, k.\, M\} & \\
l & ::= & \mathtt{op} \mid \mathtt{ret} & \text{Labels}
\end{array}
$$

Reduction

$$
\begin{array}{rcll}
(\lambda x.\, M)\ V & \rightsquigarrow & M[V/x] & (\beta_v) \\
\mathtt{let}\ x = \mathtt{return}\ V\ \mathtt{in}\ M & \rightsquigarrow & M[V/x] & (\zeta_v) \\
\mathtt{case}_p\ \langle l, V\rangle\ \mathtt{of}\ \{\langle l', x\rangle \to M_r\} & \rightsquigarrow & M_r[l/l', V/x] & (\iota_p) \\
\mathtt{case}_l\ \mathtt{ret}\ \mathtt{of}\ \{\mathtt{ret} \to M_r; \mathtt{op} \to M_h\} & \rightsquigarrow & M_r & (\iota_{ret}) \\
\mathtt{case}_l\ \mathtt{op}\ \mathtt{of}\ \{\mathtt{ret} \to M_r; \mathtt{op} \to M_h\} & \rightsquigarrow & M_h & (\iota_{op}) \\
\mathtt{handle}\ \mathtt{return}\ V\ \mathtt{with}\ \{x, k.\, M_h\} & \rightsquigarrow & \mathtt{return}\ V & (\beta_h) \\
\mathtt{handle}\ F[\mathtt{do}\ V]\ \mathtt{with}\ \{x, k.\, M_h\} & \rightsquigarrow & M[V/x, f/k] & (\beta_{do}) \\
 & & \text{where } f = \lambda y.\, \mathtt{handle}\ F[y]\ \mathtt{with}\ \{x, k.\, M_h\} &
\end{array}
$$

Fig. 3. λ_h^-: A Calculus of Effect Handlers without the Return Clause

From λ_h to λ_h^-

$$
\begin{array}{rcl}
[\![\mathtt{do}\ V]\!]_m & = & \mathtt{do}\ \langle \mathtt{op}, [\![V]\!]_m\rangle \\
[\![\mathtt{handle}\ M\ \mathtt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m & = & \mathtt{handle}\ (\mathtt{let}\ y = [\![M]\!]_m\ \mathtt{in}\ \mathtt{do}\ \langle \mathtt{ret}, y\rangle)\ \mathtt{with} \\
 & & \{p, k \to \mathtt{case}_p\ p\ \mathtt{of}\ \langle l, x\rangle \to \\
 & & \ \{\mathtt{case}_l\ l\ \mathtt{of}\ \{\mathtt{ret} \to [\![M_r]\!]_m; \mathtt{op} \to [\![M_h]\!]_m\}\}\}
\end{array}
$$

From user fragment of λ_h^- to λ_h

$$
\begin{array}{rcl}
[\![\mathtt{do}\ V]\!]_m & = & \mathtt{do}\ [\![V]\!]_m \\
[\![\mathtt{handle}\ M\ \mathtt{with}\ \{x, k.\, M_h\}]\!]_m & = & \mathtt{handle}\ [\![M]\!]_m\ \mathtt{with}\ \{x.\, \mathtt{return}\ x;\ x, k.\, [\![M_h]\!]_m\}
\end{array}
$$

The first translation attaches a label `op` to regular operations and simulates the return clause by performing a `ret` operation, which removes the surrounding handler by discarding the continuation k. The second translation is fairly easy.

4.3 Correctness

The macro translations defined above preserve the meaning of programs. We state this property as the following theorem.

Theorem 1 (Correctness of Macro Translations). *Let $=$ be the least congruence relation that includes the reduction $\rightsquigarrow$ in λ_h and λ_h^-.*

1. *If $M = N$ in λ_h, then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h^-.*
2. *If $M = N$ in the user fragment of λ_h^- (i.e., the fragment consisting of non-shaded constructs), then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h.*

Proof. By cases on the reduction relation $M \rightsquigarrow N$. We provide the proof of interesting cases in Appendix A.

5 Deriving a CPS Translation

The classical way of specifying the semantics of control operators is to give a translation into continuation-passing style (CPS) [32], which converts control operators into plain lambda terms. In the case of `shift0` and dollar (or reset), there exist several variants of CPS translation, differing in the representation of continuations [3,10,37,27,28].

Compared to control operators, the semantics of effect handlers seems to be less tightly tied to the CPS translation. In fact, the CPS translation of effect handlers has not been formally studied until recently [16,15]. This gives rise to a question: What would we obtain if we derive the CPS translation of effect handlers from that of control operators, which is considered definitional?

In this section, we derive the CPS translation of effect handlers by composing the following translations.

1. The macro translation from effect handlers to `shift0`/dollar [12]
2. The CPS translation of `shift0`/dollar [28]

We show that, by not introducing "administrative" constructs in the macro translation, we obtain the same CPS translation as the unoptimized[6] translation given by Hillerström et al. [16]. In what follows, we review the existing CPS translations of `shift0`/dollar and effect handlers (Sections 5.1 and 5.2), as well as the macro translation from effect handlers to `shift0`/dollar (Section 5.3). We then compose the first and third translations to obtain the second translation (Section 5.4), thus formally relating the CPS translations of `shift0`/dollar and effect handlers.

5.1 CPS Translation of Shift0 and Dollar

In Figure 4, we present the CPS translation from λ_{S_0} to the plain λ-calculus. The definition is adopted from Hillerström et al. [16] (for the λ-terms fragment) and Materzok and Biernacki [28] (for `shift0`/dollar). We have two mutually-defined translations $[\![\_]\!]_{c'}$ and $[\![\_]\!]_c$, taking care of values and computations, respectively.

[6] By "unoptimized translation", we mean the first-order translation in Figure 5 of Hillerström et al. [16].

$$
\begin{aligned}
[\![x]\!]_{c'} &= x \\
[\![\lambda x.\, M]\!]_{c'} &= \lambda x.\, [\![M]\!]_c \\
[\![\texttt{return}\ V]\!]_c &= \lambda k.\, k\ [\![V]\!]_{c'} \\
[\![V\ W]\!]_c &= [\![V]\!]_{c'}\ [\![W]\!]_{c'} \\
[\![\texttt{let}\ x = M\ \texttt{in}\ N]\!]_c &= \lambda k.\, [\![M]\!]_c\ (\lambda x.\, [\![N]\!]_c\ k) \\
[\![\mathcal{S}_0 k.\, M]\!]_c &= \lambda k.\, [\![M]\!]_c \\
[\![\langle M \mid x.N \rangle]\!]_c &= [\![M]\!]_c\ (\lambda x.\, [\![N]\!]_c)
\end{aligned}
$$

Fig. 4. CPS Translation of λ_{S_0} [16,27]

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_c &= \lambda k.\, \lambda h.\, h\ [\![V]\!]_{c'}\ (\lambda x.\, k\ x\ h) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_c &= [\![M]\!]_c\ (\lambda x.\, \lambda h.\, [\![M_r]\!]_c)\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_c)
\end{aligned}
$$

Fig. 5. CPS Translation of λ_h [16] (cases for λ-terms are the same as Figure 4)

The former yields a direct-style value in the λ-calculus, whereas the latter yields a continuation-taking function. To go through the cases of the control operators, the translation turns a `shift0` construct into a λ-abstraction, and a dollar construct into an application of the main computation $[\![M]\!]_c$ to an ending continuation $\lambda x.\, [\![N]\!]_c$[7].

5.2 CPS Translation of Effect Handlers

In Figure 5, we present the CPS translation from λ_h to the plain λ-calculus. The definition is adopted from Hillerström et al. [16]. We again have separate translations for values and computations. Among them, the computation translation yields a function that takes in two continuations: a pure continuation k for returning a value, and an effect continuation h for handling an operation. To go through the interesting cases, the translation of an operation calls the effect continuation h representing the interpretation of that operation[8]. Notice that the resumption $\lambda x.\, k\ x\ h$ passed to h includes h itself, reflecting the deep nature of handlers. The translation of a handler sets the two continuations for the handled computation $[\![M]\!]_c$. Other cases are the same as the translation of λ_{S_0}. In particular, the translation of a return clause requires only one continuation argument as the effect continuation can be removed by η-reduction.

[7] The CPS translation of the original dollar operator $N\,\$\,M$ is defined as $\lambda k.\, [\![N]\!]_c\ (\lambda f.\, [\![M]\!]_c\ f\ k)$.

[8] The original translation of Hillerström et al. [16] packages the two arguments to h into a tuple and associates the tuple with an operation label.

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \mathcal{S}_0 k.\, \texttt{return}\ (\lambda h.\, \texttt{let}\ v_1 = h\ [\![V]\!]_m \\
&\qquad\qquad \texttt{in}\ v_1\ (\lambda x.\, \texttt{let}\ v_2 = k\ x\ \texttt{in}\ v_2\ h)) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m &= \texttt{let}\ v = \langle [\![M]\!]_m \mid x.\texttt{return}\ \lambda h.\, [\![M_r]\!]_m \rangle \\
&\qquad\qquad \texttt{in}\ v\ (\lambda x.\, \texttt{return}\ (\lambda k.\, [\![M_h]\!]_m))
\end{aligned}
$$

Fig. 6. Macro Translation from λ_h to λ_{S_0}

$$
\begin{aligned}
[\![\texttt{do}\ V]\!]_m &= \mathcal{S}_0 k.\, (\lambda h.\, h\ [\![V]\!]_m\ (\lambda x.\, k\ x\ h)) \\
[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m &= \langle [\![M]\!]_m \mid x.\lambda h.\, [\![M_r]\!]_m \rangle\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_m)
\end{aligned}
$$

Fig. 7. Simpler Macro Translation from λ_h to λ_{S_0} [12]

5.3 Macro Translation from λ_h to λ_{S_0}

Having seen the CPS translations, we look at the macro translation from λ_h to λ_{S_0} (Figure 6), which is adapted from Forster et al. [12]. Roughly speaking, the translation converts operations into `shift0` and handlers into dollar. Technically, it introduces an effect-continuation-passing mechanism to move the handling code from the delimiter to the control operator. The result of the translation is somewhat verbose due to the presence of the `return` and `let` expressions; these are necessary for ensuring that the translation produces a well-formed expression in fine-grain call-by-value. The original translation of Forster et al. (Figure 7) is simpler, as it is defined on call-by-push-value (where the operator of an application may be a computation). Note that, in both versions, the translation of λ-terms is defined homomorphically.

5.4 Composing Macro and CPS Translations

Now, let us derive a CPS translation of λ_h by composing the macro translation on λ_h and the CPS translation on λ_{S_0}.

Operations We begin with the case of operations. Below is the naïve composition of the macro and CPS translations.

$$
\begin{aligned}
&[\![[\![\texttt{do}\ V]\!]_m]\!]_c \\
&= [\![\mathcal{S}_0 k.\, \texttt{return}\ (\lambda h.\, \texttt{let}\ v_1 = h\ [\![V]\!]_m\ \texttt{in}\ v_1\ (\lambda x.\, \texttt{let}\ v_2 = k\ x\ \texttt{in}\ v_2\ h)]\!]_c \\
&= \lambda k.\, \lambda k_1.\, k_1\ (\lambda h.\, \lambda k_2.\, h\ [\![[\![V]\!]_m]\!]_{c'}\ (\lambda v_1.\, v_1\ (\lambda x.\, \lambda k_3.\, k\ x\ (\lambda v_2.\, v_2\ h\ k_3))\ k_2))
\end{aligned}
$$

The result has four continuation variables: k, k_1, k_2, and k_3. Among them, the latter three continuations come from the `return` and `let` expressions introduced by the macro translation. As we mentioned before, these `return` and `let` are

only necessary for the well-formedness of macro-translated expressions. In other words, they do not have any computational content. What this implies is that, when our ultimate goal is to convert effect handlers into plain λ-terms, we can use a simpler macro translation that does not yield these administrative `return` and `let`. This simpler translation is exactly the original translation by Forster et al. [12], and it allows us to avoid the three continuation variables as shown below.

$$\begin{aligned}
&[\![[\![\texttt{do}\ V]\!]_m]\!]_c \\
&= [\![\mathcal{S}_0 k.\, \lambda h.\, h\ [\![V]\!]_m\ (\lambda x.\, k\ x\ h)]\!]_c \\
&= \lambda k.\, \lambda h.\, h\ [\![[\![V]\!]_m]\!]_{c'}\ (\lambda x.\, k\ x\ h)
\end{aligned}$$

The result of composition is identical to the existing translation of operations, which we saw in Figure 5.

Handlers We next look at the handler case. Below is the naïve composition of the two translations.

$$\begin{aligned}
&[\![[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m]\!]_c \\
&= [\![\texttt{let}\ v = \langle [\![M]\!]_m \mid x.\texttt{return}\ \lambda h.\, [\![M_r]\!]_m \rangle\ \texttt{in}\ v\ (\lambda x.\, \texttt{return}\ \lambda k.\, [\![M_h]\!]_m)]\!]_c \\
&= \lambda k_1.\, ([\![[\![M]\!]_m]\!]_c\ (\lambda x.\, \lambda k_2.\, k_2\ (\lambda h.\, [\![[\![M_r]\!]_m]\!]_c)))\ (\lambda v.\, v\ (\lambda x.\, \lambda k_3.\, k_3\ (\lambda k.\, [\![[\![M_h]\!]_m]\!]_c))\ k_1)
\end{aligned}$$

As in the case of operations, the result has three continuation variables k_1, k_2, and k_3 that originate from the macro translation. To avoid these continuation variables, we use the simpler macro translation of Forster et al. This makes the composition more concise, as shown below.

$$\begin{aligned}
&[\![[\![\texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m]\!]_c \\
&= [\![\langle [\![M]\!]_m \mid x.\lambda h.\, [\![M_r]\!]_m \rangle\ (\lambda x.\, \lambda k.\, [\![M_h]\!]_m)]\!]_c \\
&= [\![[\![M]\!]_m]\!]_c (\lambda x.\, \lambda h.\, [\![[\![M_r]\!]_m]\!]_c)\ (\lambda x.\, \lambda k.\, [\![[\![M_h]\!]_m]\!]_c)
\end{aligned}$$

The result is again the same as the existing translation of handlers we saw in Figure 5.

6 Deriving a Type System from the CPS Translation

The traditional approach to designing a type system for control operators is to analyze their CPS translation [8], which exposes the typing constraints of each syntactic construct. In the case of `shift0` and dollar, the CPS translation gives rise to a stack-like representation of effects, reflecting the ability of `shift0` to capture metacontexts [27,28].

Unlike control operators, the type system of effect handlers has been designed independently of the CPS translation. Indeed, the first type system of effect handlers [2] was proposed before the first CPS translation [16]. This brings up a question: What would we obtain if we derive the type system of effect handlers from their CPS translation?

In this section, we derive the type system of effect handlers following the recipe of Danvy and Filinski [8], which consists of two steps:

1. Annotate the CPS image of each construct in the most general way
2. Encode the identified typing constraints into the typing rules

We show that, when guided by the CPS translation, we obtain a type system with explicit answer types and a restricted form of answer-type modification [8]. In what follows, we review the existing type systems of `shift0`/dollar and effect handlers (Sections 6.1 and 6.2), design a new type system of effect handlers by applying the CPS approach (Section 6.3), and prove the soundness of the type system (Section 6.4).

6.1 Type System of Shift0 and Dollar

In Figure 8, we present the type system of λ_{S_0}. The type system is adopted from Hillerström et al [16] (for λ-terms) and Forster et al. [12] (for `shift0` and dollar). There are two classes of types: value types (A, B) and computation types (C, D). The latter take the form $A\,!\,\epsilon$, which reads: the computation returns a value of type A while possibly performing an effect represented by ϵ[9]. An effect is a list of *answer types*, representing what kind of context is needed to evaluate a computation[10]. The list is extended by (Shift0) (which requires a specific kind of context) and shrunk by (Dollar) (which supplies the required context).

6.2 Type System of Effect Handlers

In Figure 9, we present the type system of λ_h. The type system is adopted from Hillerström et al. [16]. We again have value and computation types. An effect is either the pure effect ι or an operation type $A \Rightarrow B$; the latter tells us what kind of handler is needed to evaluate a computation. Operation types are introduced by (Do) (which requires a specific kind of operation clause) and eliminated by (Handle) (which supplies the required operation clause). The typing rules for λ-terms stay the same as those of λ_{S_0} because they do not modify effects.

[9] In the original type system of Forster et al. [12], effects are attached to the typing judgment instead of being part of a computation type.

[10] The effect representation does not allow *answer-type modification*. A more general (and involved) type system supporting answer-type modification is given by Materzok and Biernacki [28].

Syntax of Types and Effects

$$\begin{array}{rll} A, B ::= b \mid A \to C & & \text{Value Types} \\ C, D, E ::= A \,!\, \epsilon & & \text{Computation Types} \\ \epsilon ::= [] \mid A :: \epsilon & & \text{Effects} \end{array}$$

Typing Rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.\, M : A \to C}\ (\textsc{Abs}) \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{return}\ V : A \,!\, \epsilon}\ (\textsc{Return})$$

$$\frac{\Gamma \vdash V : A \to C \quad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ (\textsc{App}) \qquad \frac{\Gamma \vdash M : A \,!\, \epsilon \quad \Gamma, x : A \vdash N : B \,!\, \epsilon}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : B \,!\, \epsilon}\ (\textsc{Let})$$

$$\frac{\Gamma, k : A \to B \,!\, \epsilon \vdash M : B \,!\, \epsilon}{\Gamma \vdash \mathcal{S}_0 k.\, M : A \,!\, (B :: \epsilon)}\ (\textsc{Shift0}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \,!\, (B :: \epsilon) \\ \Gamma, x : A \vdash N : B \,!\, \epsilon\end{array}}{\Gamma \vdash \langle M \mid x.N \rangle : B \,!\, \epsilon}\ (\textsc{Dollar})$$

Fig. 8. Type System of λ_{S_0}

Syntax of Types and Effects

$$\begin{array}{rll} A, B ::= b \mid A \to C & & \text{Value Types} \\ C ::= A \,!\, \epsilon & & \text{Computation Types} \\ \epsilon ::= \iota \mid A \Rightarrow B & & \text{Effects} \end{array}$$

Typing Rules

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{do}\ V : B \,!\, A \Rightarrow B}\ (\textsc{Do}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A \,!\, A' \Rightarrow B' \\ \Gamma, x : A \vdash M_r : C \\ \Gamma, x : A', k : B' \to C \vdash M_h : C\end{array}}{\Gamma \vdash \texttt{handle}\ M\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\} : C}\ (\textsc{Handle})$$

Fig. 9. Type System of λ_h (rules for λ-terms are the same as Figure 8)

6.3 Applying the CPS Approach

The type system presented in Figure 9 is defined on the direct-style expressions in λ_h. We now design a new type system based on the CPS translation. As we saw in Section 5, a CPS-translated λ_h computation is a function that receives a return continuation and an effect continuation. Among the two arguments, a return continuation takes in a value and a handler, whereas an effect continuation takes in an operation argument and a resumption (a continuation whose handler is already given). Hence, if we annotate the CPS image of a computation in the most general way, we obtain something like this:

$$\lambda k^{A \to (A_1 \to (B_1 \to C_1) \to D_1) \to E_1}.\, \lambda h^{A_2 \to (B_2 \to C_2) \to D_2}.\, e^{E_2}$$

The annotations are however too general. The semantics of deep handlers naturally gives rise to the following invariants.

1. The two handler types $A_1 \to (B_1 \to C_1) \to D_1$ and $A_2 \to (B_2 \to C_2) \to D_2$ are the same. This is because a computation and its continuation are evaluated under the same handler.
2. The return type E_1 of the return continuation and the return type C_2 of the resumption are the same. This is because a resumption is constructed using a return continuation.
3. The type E_2 of the eventual answer must be either C_2 (the type of the return clause) or D_2 (the type of the operation clause).

These invariants lead to the following refined annotations.

$$\lambda k^{A \to (A' \to (B' \to C) \to D) \to C}.\, \lambda h^{A' \to (B' \to C) \to D}.\, e^{E} \qquad \text{where } E = C \text{ or } D$$

The annotations have six different types. Among them, A', B', C, D, and E specify the kind of the handler required by the computation. By including these types in non-empty effects, we arrive at the following effect representation.

$$\epsilon ::= \iota \mid \langle A \Rightarrow B, C, D, E \rangle \qquad \text{where } E = C \text{ or } D$$

Here, $\langle A \Rightarrow B, C, D, E \rangle$ carries the following information.

- The computation may perform an operation of type $A \Rightarrow B$.
- When evaluated under a handler whose return clause has type C and whose operation clause has type D, the computation returns an answer of type E (which must be either C or D).

Using this representation of effects, we design the typing rules for λ_h computations and values. We do this by annotating the CPS image of individual syntactic constructs and converting annotated expressions into typing derivations.

Return Expressions We begin by annotating the CPS image of a `return` expression. To make the calculation of types easier to follow, we explicitly write the effect continuation h.

$$\lambda k^{A\to(A'\to(B'\to C)\to D)\to C}.\, \lambda h^{A'\to(B'\to C)\to D}.\, (k\ [\![V]\!]^A_{c'}\ h)^C$$

The trivial use of the continuation forces the type of the eventual answer to be C (corresponding to invariant 3 mentioned above). From this annotated expression, we obtain the following typing rule.

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \texttt{return}\ V : A\,!\,\langle A' \Rightarrow B', C, D, C\rangle}\ (\textsc{Return})$$

Application We next annotate the CPS image of an application.

$$([\![V]\!]^{A\to C}_c\ [\![W]\!]^A_c)^C$$

The annotations simply tell us that the effect of an application comes from the body of the function. This corresponds to the following typing rule.

$$\frac{\Gamma \vdash V : A \to C \quad \Gamma \vdash W : A}{\Gamma \vdash V\ W : C}\ (\textsc{App})$$

Let Expressions Having analyzed application, we consider the `let` expression.

$$\begin{aligned}&\lambda k^{B\to(A'\to(B'\to C)\to D)\to C}.\\ &\quad [\![M]\!]^{(A\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to E}_c\\ &\quad (\lambda x^A.\, [\![N]\!]^{(B\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to C}_c\ k)\end{aligned}$$

The sequencing of M and N forces the eventual answer type of N to be C. The corresponding typing rule is as follows.

$$\frac{\Gamma \vdash M : A\,!\,\langle A' \Rightarrow B', C, D, E\rangle \quad \Gamma, x : A \vdash N : B\,!\,\langle A' \Rightarrow B', C, D, C\rangle}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : B\,!\,\langle A' \Rightarrow B', C, D, E\rangle}\ (\textsc{Let})$$

Operations We now move on to the case of an operation call.

$$\lambda k^{B'\to(A'\to(B'\to C)\to D)\to C}.\, \lambda h^{A'\to(B'\to C)\to D}.\, (h\ [\![V]\!]_{c'}\ (\lambda x^A.\, k\ x\ h))^D$$

The application of h forces the eventual answer type to be D (invariant 3). The application $k\ x\ h$ restricts the annotations in two ways: (i) the handler type of the whole expression and that of k must coincide (invariant 1); and (2) the return type of k and that of the resumption must coincide (invariant 2). Below is the derived typing rule.

$$\frac{\Gamma \vdash V : A'}{\Gamma \vdash \texttt{do}\ V : B'\,!\,\langle A' \Rightarrow B', C, D, D\rangle}\ (\textsc{Do})$$

Handlers Lastly, we consider the case of a handler.

$$\begin{array}{l}[\![M]\!]_c^{(A\to(A'\to(B'\to C)\to D)\to C)\to(A'\to(B'\to C)\to D)\to E}\\ (\lambda x^{A}.\,\lambda h^{A'\to(B'\to C)\to D}.\,[\![e_r]\!]_c^{C})\\ (\lambda x^{A'}.\,\lambda k^{B'\to C}.\,[\![e_h]\!]_c^{D}))^{E}\end{array}$$

The construction requires the effect of the handled expression M and the type of the two clauses of the handler to be consistent. Thus we obtain the following typing rule.

$$\frac{\begin{array}{c}\Gamma \vdash M : A\,!\,\langle A' \Rightarrow B', C, D, E\rangle\\ \Gamma, x : A \vdash M_r : C\\ \Gamma, x : A', k : B' \to C \vdash M_h : D\end{array}}{\Gamma \vdash \texttt{handle}\ M\ \texttt{with}\ \{x.\,M_r;\ x, k.\,M_h\} : E}\ (\textsc{Handle})$$

Values Let us complete the development by giving the typing rules for variables and abstractions. These are derived straightforwardly from the CPS translation on values. The only thing that is non-standard is the well-formedness condition $\vdash A$ in the (Var) rule. Here, well-formedness means every function type that appears in a type takes the form $A \to B\,!\,\langle A' \Rightarrow B', C, D, C\rangle$ or $A \to B\,!\,\langle A' \Rightarrow B', C, D, D\rangle$.

$$\frac{x : A \in \Gamma \quad \vdash A}{\Gamma \vdash x : A}\ (\textsc{Var}) \qquad \frac{\Gamma, x : A \vdash M : C}{\Gamma \vdash \lambda x.\,M : A \to C}\ (\textsc{Abs})$$

Through this exercise, we obtained a type system that is different from the one presented in Figure 9. First, the type system explicitly keeps track of answer types. Second, the type system allows a form of answer-type modification [8,1,20]. Answer-type modification is the ability of an effectful computation to change the return type of its delimited context. In our setting, this ability is gained by allowing the return and operation clauses of a handler to have different types.

The answer-type modification supported in this type system is however very limited. Specifically, it does not allow answer-type modification in a captured continuation. In fact, we have already seen where this restriction comes from: the `let` expression. Recall that, in the typing rule (Let), the body N of `let` (which corresponds to the continuation of M) has a type of the form $B\,!\,\langle A' \Rightarrow B', C, D, C\rangle$. This means N must be a pure computation (which is eventually handled by the return clause of type C) or it must be handled by a handler whose return and operation clauses have the same type (i.e., $C = D$).

6.4 Soundness

The type system we derived from the CPS translation is sound. We state this property as the following theorem.

Theorem 2 (Soundness of Type System). *If $\Gamma \vdash M : A\,!\,\iota$ and $M \rightsquigarrow \texttt{return}\ V$, then $\Gamma \vdash V : A$.*

Proof. The statement is witnessed by a CPS interpreter[11] written in the Agda programming language [30]. The interpreter takes in a well-typed λ_h computation and returns a fully-evaluated Agda value. The signature of the interpreter corresponds exactly to the statement of soundness, and the well-typedness of the interpreter implies the validity of the statement.

7 Related Work

Variations of Effect Handler Calculi There is a discussion of the relationship between effect operations with and without a label, which is similar to our discussion of handlers with and without the return clause. The handling of a labeled operation may involve skipping of handlers that do not take care of the operation in question. To simulate labeled operations in a calculus with unlabeled operations, Biernacki et al. [4] introduce an operator $\lfloor \_ \rfloor$ called "lift", which allows an operation to skip the innermost handler and be handled by an outer handler. This enables programming with multiple effects without using labels, although it requires the programmer to know what handlers are surrounding each operation in what order.

CPS Translations of Effect Handlers The CPS translations of effect handlers have been developed as a technique for compiling effect handlers into common runtime platforms. For this reason, existing CPS translations are either directed by the specific typing discipline of the source language [22,6] or tuned for the particular implementation strategy of the compiler [16,15]. We derived a CPS translation of effect handlers from the definitional CPS translation of `shift0` and dollar. As a result, we obtained a translation that is identical to Hillerström's [16] unoptimized translation, which we regard as the definitional translation of general (deep) effect handlers.

Type Systems of Effect Handlers There are different flavors of type systems for effect handlers. In the most traditional type systems [2,18], effects are represented as a set of operations, similar to the effect systems for side-effect analysis [29]. In some research languages such as Koka [21], Frank [25], and Links [13], effects are treated as row types, which were originally developed for type inference with records [35]. More recently, several languages [39,7] adopt the notion of capabilities from object-oriented programming as an approach to safe handling of effects. Unlike the type system we developed in Section 6, these type systems are all defined independent of the CPS translation, and they do not explicitly carry answer types.

Effect Handlers and Control Operators The relationship between deep effect handlers and the `shift0`/dollar operators was first established by Forster et al. [12] in an untyped setting. The result was later extended by Piróg et al. [31] to a typed relation, through a special form of polymorphism that fills the gap

[11] The interpreter is available at `https://github.com/YouyouCong/tfp22`.

between effect handlers and control operators. Such relations have been used for implementation of effect handlers [18,19], but not for theoretical purposes as in our work.

8 Conclusion and Future Work

In this paper, we presented three results from our study on understanding deep effect handlers through the lens of the `shift0` and dollar operators. Specifically, we defined a pair of macro translations that add and remove the return clause, derived a CPS translation for effect handlers from the CPS translation for `shift0`/dollar, and designed a type system for effect handlers from the CPS translation.

As a continuation of this work, we intend to visit those challenges that have been solved for `shift0` and dollar (or reset) but not for effect handlers. We are particularly interested in exploring equational axiomatization [26] and reflection [5], which could be useful for optimization purposes.

As an extension of our approach, we plan to study shallow effect handlers [14] by leveraging their connection to the `control0` and `prompt0` operators [31]. These operators have a more complex semantics due to the absence of the delimiter surrounding captured continuations, but we conjecture that it is possible to establish similar results to what we have presented in this paper.

References

1. Asai, K.: On typing delimited continuations: three new solutions to the printf problem. Higher-Order and Symbolic Computation **22**(3), 275–291 (2009)
2. Bauer, A., Pretnar, M.: An effect system for algebraic effects and handlers. In: Heckel, R., Milius, S. (eds.) Algebra and Coalgebra in Computer Science. pp. 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
3. Biernacki, D., Danvy, O., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. ACM Trans. Program. Lang. Syst. **38**(1) (Oct 2015). https://doi.org/10.1145/2794078
4. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Handle with care: Relational interpretation of algebraic effects and handlers. Proc. ACM Program. Lang. **2**(POPL), 8:1–8:30 (Dec 2017). https://doi.org/10.1145/3158096
5. Biernacki, D., Pyzik, M., Sieczkowski, F.: Reflecting Stacked Continuations in a Fine-Grained Direct-Style Reduction Theory. PPDP '21, Association for Computing Machinery, New York, NY, USA (2021), `https://doi.org/10.1145/3479394.3479399`
6. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effect handlers for the masses. Proc. ACM Program. Lang. **2**(OOPSLA), 111:1–111:27 (Nov 2018)
7. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effects as capabilities: Effect handlers and lightweight effect polymorphism. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428194, `https://doi.org/10.1145/3428194`
8. Danvy, O., Filinski, A.: A functional abstraction of typed contexts. BRICS 89/12 (Aug 1989)

9. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM conference on LISP and functional programming. pp. 151–160. ACM (1990)
10. Dyvbig, R.K., Peyton Jones, S., Sabry, A.: A monadic framework for delimited continuations. J. Funct. Program. **17**(6), 687–730 (Nov 2007). https://doi.org/10.1017/S0956796807006259
11. Felleisen, M.: On the expressive power of programming languages. In: Selected Papers from the Symposium on 3rd European Symposium on Programming. pp. 35–75. ESOP '90, Elsevier North-Holland, Inc., New York, NY, USA (1991)
12. Forster, Y., Kammar, O., Lindley, S., Pretnar, M.: On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. Proc. ACM Program. Lang. **1**(ICFP), 13:1–13:29 (Aug 2017). https://doi.org/10.1145/3110257
13. Hillerström, D., Lindley, S.: Liberating effects with rows and handlers. In: Proceedings of the 1st International Workshop on Type-Driven Development. pp. 15–27. TyDe 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976022.2976033
14. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Asian Symposium on Programming Languages and Systems. pp. 415–435. APLAS '18, Springer (2018)
15. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. Journal of Functional Programming **30** (2020). https://doi.org/10.1017/S0956796820000040
16. Hillerström, D., Lindley, S., Atkey, R., Sivaramakrishnan, K.: Continuation passing style for effect handlers. In: Proceedings of 2nd International Conference on Formal Structures for Computation and Deduction. pp. 18:1–18:19. FSCD '17, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017)
17. Kameyama, Y., Yonezawa, T.: Typed dynamic control operators for delimited continuations. In: International Symposium on Functional and Logic Programming. pp. 239–254. FLOPS '08, Springer (2008)
18. Kammar, O., Lindley, S., Oury, N.: Handlers in action. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 145–158. ICFP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2500365.2500590
19. Kiselyov, O., Sivaramakrishnan, K.C.: Eff directly in ocaml. In: ML Workshop (2016)
20. Kobori, I., Kameyama, Y., Kiselyov, O.: Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. In: Electronic Proceedings in Theoretical Computer Science EPTCS 212 (Post-Proceedings of the Workshop on Continuations 2015). pp. 36–52 (June 2016). https://doi.org/10.4204/EPTCS.212.3
21. Leijen, D.: Koka: Programming with row polymorphic effect types. In: 5th Workshop on Mathematically Structured Functional Programming. MSFP '14 (2014). https://doi.org/10.4204/EPTCS.153.8
22. Leijen, D.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 486–499. POPL '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009872
23. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis. Springer, Dordrecht (2003)
24. Levy, P.B., Power, J., Thielecke, H.: Modelling environments in call-by-value programming languages. Information and computation **185**(2), 182–210 (2003)

25. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 500–514. POPL '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009897
26. Materzok, M.: Axiomatizing subtyped delimited continuations. In: Computer Science Logic 2013. CSL 2013, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2013)
27. Materzok, M., Biernacki, D.: Subtyping delimited continuations. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. pp. 81–93. ICFP '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2034773.2034786
28. Materzok, M., Biernacki, D.: A dynamic interpretation of the CPS hierarchy. In: Asian Symposium on Programming Languages and Systems. pp. 296–311. APLAS '12, Springer (2012)
29. Nielson, F., Nielson, H.R.: Type and effect systems. In: Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel). pp. 114–136. Springer-Verlag, Berlin, Heidelberg (1999)
30. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)
31. Piróg, M., Polesiuk, P., Sieczkowski, F.: Typed equivalence of effect handlers and delimited control. In: 4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
32. Plotkin, G.: Call-by-name, call-by-value and the λ-calculus. Theoretical computer science **1**(2), 125–159 (1975)
33. Plotkin, G., Power, J.: Algebraic operations and generic effects. Applied categorical structures **11**(1), 69–94 (2003)
34. Plotkin, G., Pretnar, M.: Handlers of algebraic effects. In: European Symposium on Programming. pp. 80–94. ESOP '09, Springer (2009)
35. Rémy, D.: Type Inference for Records in Natural Extension of ML, p. 67–95. MIT Press, Cambridge, MA, USA (1994)
36. Schuster, P., Brachthäuser, J.I., Ostermann, K.: Compiling effect handlers in capability-passing style. Proc. ACM Program. Lang. **4**(ICFP) (aug 2020). https://doi.org/10.1145/3408975, `https://doi.org/10.1145/3408975`
37. Shan, C.c.: A static simulation of dynamic delimited control. Higher-Order and Symbolic Computation **20**(4), 371–401 (2007)
38. Xie, N., Brachthäuser, J.I., Hillerström, D., Schuster, P., Leijen, D.: Effect handlers, evidently. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (2020). https://doi.org/10.1145/3408981
39. Zhang, Y., Myers, A.C.: Abstraction-safe effect handlers via tunneling. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290318, `https://doi.org/10.1145/3290318`
40. Zhang, Y., Salvaneschi, G., Myers, A.C.: Handling bidirectional control flow **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428207, `https://doi.org/10.1145/3428207`

A Correctness of Macro Translations between λ_h and λ_h^-

Lemma 1 (Commutativity of Translation and Substitution). *Let $\equiv$ be syntactic equality.*

- *For all V and V' in λ_h, we have $[\![V[V'/x]]\!]_{c'} \equiv [\![V]\!]_{c'}[[\![V']\!]_{c'}/x]$.*
- *For all M and V in λ_h, we have $[\![M[V/x]]\!]_c \equiv [\![M]\!]_c[[\![V]\!]_{c'}/x]$.*

Proof. By mutual induction on the structure of V and M.

Theorem 1 (Correctness of Macro Translations). *Let $=$ be the least congruence relation that includes the reduction $\rightsquigarrow$ in λ_h and λ_h^-.*

1. *If $M = N$ in λ_h, then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h^-.*
2. *If $M = N$ in the user fragment of λ_h^-, then $[\![M]\!]_m = [\![N]\!]_m$ in λ_h.*

Proof. We first prove the cases where $M \rightsquigarrow N$. We then derive the goal by congruence. Here we show the interesting cases for the first part.

Part 1

Case 1 (β_{do}).

$$
\begin{aligned}
&[\![\texttt{handle}\ F[\texttt{do}\ V]\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m \\
\equiv\ &\texttt{handle}\ (\texttt{let}\ y = [\![F]\!]_m[\texttt{do}\ \langle \texttt{op}, [\![V]\!]_m\rangle]\ \texttt{in do}\ \langle \texttt{ret}, y\rangle)\ \texttt{with} \\
&\{p, k \to \texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\} \\
\rightsquigarrow\ &\texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\}[\langle \texttt{op}, [\![V]\!]_m\rangle/p, f/k] \\
&\quad \text{where } f = \lambda z.\, \texttt{handle}\ (\texttt{let}\ y = [\![F]\!]_m[z]\ \texttt{in do}\ \langle \texttt{ret}, y\rangle)\ \texttt{with} \\
&\qquad\qquad \{p, k \to \texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\}\} \\
\rightsquigarrow\ &\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}[\texttt{op}/l, [\![V]\!]_m/x, f/k] \\
\rightsquigarrow\ &[\![M_h]\!]_m[[\![V]\!]_m/x, f/k] \\
\equiv\ &[\![M_h[V/x, \lambda y.\, \texttt{handle}\ F[y]\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}/k]]\!]_m
\end{aligned}
$$

Case 2 (β_h).

$$
\begin{aligned}
&[\![\texttt{handle return}\ V\ \texttt{with}\ \{x.\, M_r;\ x, k.\, M_h\}]\!]_m \\
\equiv\ &\texttt{handle}\ (\texttt{let}\ y = \texttt{return}\ [\![V]\!]_m\ \texttt{in do}\ \langle \texttt{ret}, y\rangle)\ \texttt{with} \\
&\{p, k \to \texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\} \\
\rightsquigarrow\ &\texttt{handle do}\ \langle \texttt{ret}, [\![V]\!]_m\rangle\ \texttt{with} \\
&\{p, k \to \texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\} \\
\rightsquigarrow\ &\texttt{case}\ p\ \texttt{of}\ \langle l, x\rangle \to \{\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}\}[\langle \texttt{ret}, [\![V]\!]_m\rangle/p] \\
\rightsquigarrow\ &\texttt{case}_l\ l\ \texttt{of}\ \{\texttt{ret} \to [\![M_r]\!]_m; \texttt{op} \to [\![M_h]\!]_m\}[\texttt{ret}/l, [\![V]\!]_m/x] \\
\rightsquigarrow\ &[\![M_r]\!]_m[[\![V]\!]_m/x] \\
\equiv\ &[\![M_r[V/x]]\!]_m
\end{aligned}
$$

Part 2

Case 1 (β_{do}).

$$
\begin{aligned}
&[\![\texttt{handle}\ F[\texttt{do}\ V]\ \texttt{with}\ \{x, k.\, M_h\}]\!]_m \\
\equiv\ &\texttt{handle}\ [\![F]\!]_m[\texttt{do}\ [\![V]\!]_m]\ \texttt{with}\ \{x.\,\texttt{return}\ x;\ x, k.\, [\![M_h]\!]_m\} \\
\rightsquigarrow\ &[\![M_h]\!]_m[[\![V]\!]_m/x, f/k] \\
&\quad \text{where } f = \lambda y.\,\texttt{handle}\ [\![F]\!]_m[y]\ \texttt{with}\ \{x.\,\texttt{return}\ x;\ x, k.\, [\![M_h]\!]_m\} \\
\equiv\ &[\![M_h[V/x, \lambda y.\,\texttt{handle}\ F[y]\ \texttt{with}\ \{x, k.\, M_h\}/k]]\!]_m
\end{aligned}
$$

Case 2 (β_h).

$$
\begin{aligned}
&[\![\texttt{handle}\ \texttt{return}\ V\ \texttt{with}\ \{x, k.\, M_h\}]\!]_m \\
\equiv\ &\texttt{handle}\ \texttt{return}\ [\![V]\!]_m\ \texttt{with}\ \{x.\,\texttt{return}\ x;\ x, k.\, [\![M_h]\!]_m\} \\
\rightsquigarrow\ &\texttt{return}\ x[[\![V]\!]_m/x] \\
\equiv\ &[\![\texttt{return}\ V]\!]_m
\end{aligned}
$$

Reducing the Power Consumption of IoT with Task-Oriented Programming

Sjoerd Crooijmans, Mart Lubbers[0000−0002−4015−4878], and Pieter Koopman[0000−0002−3688−0957]

Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands sjoerd@scrooijmans.nl {mart,pieter}@cs.ru.nl

– Extended Abstract –

Abstract Limiting the energy consumption of IoT nodes is a hot topic for green computing. For battery-powered devices this necessity is obvious. The enormous growth of the number of IoT nodes makes energy efficiency important for every node in the IoT. In this paper, we show how we can automatically compute execution intervals for our Task Oriented Programs for the IoT. We do allow an arbitrary number of tasks on the IoT nodes and achieve significant reductions of the energy consumption by bringing the microprocessor in sleep mode as much as possible. We have seen energy reductions of an order of magnitude without imposing any constraints on the tasks to be executed on the IoT nodes.

Keywords: Sustainable IoT · Green Computing · Task Oriented Programming.

1 Introduction

The Internet of Things (IoT) is omnipresent and powered by software. Depending on who you ask, the estimated number of connected IoT endpoint devices reaches between 25 and 100 billion in 2021. IoT systems are traditionally designed according to multi-layered or tiered architectures. As a consequence, discrete programs written in distinct languages with different abstraction levels power the individual layers, forming a heterogeneous system. The variation in components makes programming IoT systems complicated, error-prone and expensive.

The edge layer of IoT contains the small devices that sense and interact with the world and it is crucial to lowering their energy consumption. While individual devices consume little energy, the sheer number of devices in total amounts to a lot. Furthermore, many IoT devices operate on batteries and higher energy consumption increases the amount of e-waste [5]. Moreover, IoT devices may be hard to reach, so replacing or recharging batteries is often difficult.

To reduce the power consumption of an IoT device, the specialised low-power sleep modes of the microprocessors can be leveraged. Different sleep modes achieve different power reductions because of their different run time characteristics. These specifics range from disabling or suspending WiFi; stopping powering (parts) of the RAM; disabling peripherals or even completely turning off the

processor, requiring an external signal to wake up again. Determining when exactly and for how long it is possible to sleep is expensive in the general case and often requires annotations in the source code, a Real Time Operating System (RTOS) or a handcrafted scheduler.

Task-Oriented Programming (TOP) is a novel declarative programming paradigm with the potential to solve many of the aforementioned problems. In this paradigm, tasks are the basic building blocks and they can be combined using combinators to describe workflows [7]. This declarative specification of the program only describes the *what* and not the *how* of execution. The system executing the tasks takes care of the gritty details such as the user interface, data storage and communication [8]. An example of a TOP language is the iTask system, a general-purpose framework for specifying multi-user distributed web applications for workflows [6]. iTask is implemented as an Embedded Domain Specific Language (EDSL) in the functional programming language Clean [1].

mTask lies on the other side of the spectrum and aims to solve semantic friction in IoT. It is a domain-specific TOP language and system specifically for IoT devices, implemented as an EDSL in iTask. Where iTask abstracts away from details such as user interfaces, data storage and persistent workflows, mTask offers abstractions for edge layer-specific details such as the heterogeneity of architectures, platforms and frameworks; peripheral access and multi-tasking. Yet, it lacks abstractions for energy consumption and scheduling. In mTask, tasks are implemented as a rewrite system, where the work is automatically segmented in small atomic bits and stored as a task tree. Each cycle, a single rewrite step is performed on all task trees, during rewriting, tasks do a bit of their work and progress steadily, allowing interleaved and seemingly parallel operation. After a loop, the Run Time System (RTS) knows which task is waiting on which triggers and is thus able to determine the next execution time for each task automatically. Utilising this information, the RTS can determine when it is possible and safe to sleep and choose the optimal sleep mode according to the sleeping time. For example, the RTS never attempt to sleep during an I^2C communication because I/O is always contained *within* a rewrite step.

1.1 Research contribution

This paper shows that with minor changes to the mTask language from the perspective of the TOP programmer, the energy consumption of the program's execution can be significantly reduced. We show that with an intensional analysis of the task trees at run time, the mTask scheduler can automatically determine the optimal sleep time and sleep mode. Not all tasks have a default rewrite rate that works in all situations so variants of tasks are added in which the programmer can fine-tune the polling rate. Furthermore, we add an interface to (hardware) interrupts to the mTask language, allowing the program to be notified in case of an external event, resulting in more reactive programs.

2 Task-Oriented Programming

TOP is a high-level declarative programming paradigm to specify distributed interactive multi-user systems [6,7]. The developers describe in TOP the tasks to be done by systems or users in the form of abstract tasks. Implementation details, like the representation of date during communication, are handled by the system rather than by the TOP programmer. Tasks describe a unit of work ranging from reading a single sensor to an entire IoT system. A task is a rewrite system that produces a result after each step. Possible task results are:

NoValue if the task has no observable value for other tasks. For example, a web-editor that is empty or incomplete is a task with a `NoValue` result.

Unstable if the task has an intermediate observable value. This value is by construction properly typed and can change in the future. Examples are a properly filled out web-editor and a sensor reading.

Stable if the task has a final observable value. This value is by construction properly typed and fixed. Examples are a properly filled out web-editor after pressing the `Continue` button and a task that determines that a temperature sensor has passed a given threshold.

Basic tasks are the primitive building blocks of a TOP program. Typical examples are web-editors, reading sensors, waiting some time and controlling peripherals. Tasks can be composed into bigger tasks by combinators. There are combinators for the parallel and sequential composition of tasks. The *step* combinator is used to specify actions, selecting a new task, based on the current value of some task.

Apart from task results, tasks can also communicate via Shared Data Sources (SDSs). There are basic tasks to read, write and update such a typed SDS.

2.1 mTask

TOP is also very suited to program nodes in the IoT. However, typical nodes in the IoT are cheap and small microprocessors with very limited amounts of processing power and memory. Such a system cannot run a web-server nor a browser as a client to execute an iTask program. This is also not necessary, typical IoT nodes just control a few sensors and actuators. To enable TOP on small microprocessors we have created mTask [3,4]. The Domain Specific Language (DSL) mTask is also embedded in Clean. In contrast to iTask, it is not a shallowly embedded DSL but it is a tagless DSL [2]. The target language of iTask is equal to the host language Clean, this is called a homogeneous system [9]. The target code of mTask is byte code which is shipped to a microprocessor in the IoT and interpreted by the mTask runtime system running there. Hence, mTask is a heterogeneous EDSL. The big advantage of this approach is that we can carefully control which parts of the application is mTask code and needs to be shipped to the IOT node. This prevents that all Clean language machinery ends up in such a small system. Moreover, we can dynamically ship tasks to the

IoT nodes without the need to reprogram the flash memory of those systems. Reprogramming this memory is slow and can be done only a few thousand times.

The archetypal example of programs running on microprocessors is the blink example that blinks the single bit *display*, just a LED, of such systems. In Listing 1.1 we give a slightly more complex example in mTask that blinks two LEDs at their own slowly changing speed.

```
blinkTask :: Main (MTask v Bool) | mtask v
blinkTask =
    declarePin BuiltinLEDPin PMOutput λled1 →
    declarePin d0 PMOutput λled2 →
    fun λblink = (λ(led,wait,state) →
      delay wait
      ≫|. writeD led state
      ≫|. blink (led, time +. (lit 1), Not state)
    In {main = blink (led1, lit 6174, true) .||. blink (led2, lit 73, true)}
```

Listing 1.1. An mTask to blink the LED

The Clean function `blinkTask` contains the mTask code for the blink task. It starts by declaring two `led` objects to represent the output General Purpose Input/Output (GPIO) pin to control the LED as output. Next, it declares a mTask function called `blink`. This function has the `led`, delay `time` and new `state` of the LED as arguments. The task in this function is composed of three subtasks. First, it waits `time` millisecond by `delay`. Second, it writes the give `state` to the declared output `led`. Finally, it calls itself recursively with the inverted `state` as the argument. In the `main` expression two of these `blink` task are composed in parallel with the `.||.` combinator. The `lit` in the arguments of the subtasks lifts the given constant from the host language to the EDSL.

There are some important differences to the usual C-programs controlling microprocessors. First, tail-recursive functions are perfectly save in mTask. Next, the `delay` task is not blocking the entire program, but just producing a `NoValue` result until the given `time` has passed.

3 Scheduling Tasks Efficiently

The mTask code is dynamically transformed to byte code that is shipped to a microprocessor that is selected at run time by an iTask program. The byte code is interpreted by a preloaded program on the microprocessor. This interpreter performs a series of rewrites of the mTask code. In each rewrite, all leaves of the task expression are evaluated. Whenever enabled by the current value of its arguments a mTask combinator evaluates to its results. When a reduction is not yet possible the mTask expression is recreated with the new arguments. This explains for instance why the `delay` in Listing 1.1 is not blocking. This is very similar to the implementation of iTask. In iTask, the task expression is evaluated again at an event, like an edit event form a web-editor. In mTask, we do not have editor events. Most sensors work with polling rather than on an event basis. Hence, the mTask interpreter evaluates the task expression in a tight loop. Since

no other programs are running on the microprocessor and its cycles have to be used by executing some instructions, such a frequent evaluation of the task expression is perfectly fine.

Microprocessors offer various ways to reduce their power consumption. These range from switching off parts of the system that are temporarily not needed, such as the WiFi radio, to putting the entire system into some sleep mode. Different branches of microprocessors offer diverse sleeping modes. In this paper, we will distinguish light sleep where the memory stays powered and hence its contents will be unaffected by sleeping and deep sleep where also the memory is switched off and its content will be lost. Some microprocessors have parts of the memory that will be saved in deep sleep. Since that is very brand specific we will ignore that here.

In this paper, we introduce a way to reduce the power consumption of a microprocessor implementing mTask by sleeping as much as possible. When there is no mTask to be executed the system goes to deep sleep mode. It just wakes up every now and then to keep the WiFi connection active and checks for a new mTask to be executed. When there are one or more mTasks to be executed it tries to predict when it is useful to evaluate the mTask expression again. For instance, evaluating a `delay(time)` expression can be delayed `time` milliseconds without a problem. No progress will be made before the delay time has passed. Also for many sensors, it is possible to reduce the evaluation rate. In most applications, a temperature measured by a sensor will not change significantly within a couple of seconds. We can safely delay the reading of such a sensor.

3.1 Evaluation Interval

To implement the delayed evaluation of tasks we associate dynamically an evaluation interval with each task. The interval $\langle low, high \rangle$ indicates that the evaluation can be safely delayed by any number of milliseconds in that range. Such an interval is just a hint for the runtime system. It is not a guarantee that the evaluation will take place in the given interval. When other parts of the task expression desire an earlier evaluation also this part of the task can be evaluated before the lower bound. When the system is very busy with other work, the task might be executed after the upper bound of the interval.

The `delay(time)` primitive guarantees that the task will be delayed at least `time` milliseconds. Since mTask is not a hard real-time system also this delay might be higher than the given time since the microprocessor might be busy with other things, like handling WiFi communication.

3.2 Deriving Refresh Rates

The refresh rates are calculated automatically from the current task expression. This has as advantage that the programmer does not have to deal with them and that they are available in each and every mTask program. We start by assigning default refresh rates to basic tasks.

task	default interval
reading Shared Data Source	$\langle 0, 2000 \rangle$
slow sensor, like temperature	$\langle 0, 2000 \rangle$
gesture sensor	$\langle 0, 1000 \rangle$
fast sensor, like sound or light	$\langle 0, 100 \rangle$
reading GPIO pins	$\langle 0, 100 \rangle$

Table 1. Default refresh rates of basic tasks.

Based on these refresh rates the system can be automatically derived refresh rates for composing mTask expressions.

$$\mathcal{R} :: (MTask\ v\ a) \to \langle Int, Int \rangle$$

$$\mathcal{R}(t_1\ .||.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{1}$$

$$\mathcal{R}(t_1\ .\&\&.\ t_2) = \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) \tag{2}$$

$$\mathcal{R}(t_1\ >>\ |.\ t_2) = \mathcal{R}(t) \tag{3}$$

$$\mathcal{R}(t\ >>=\ .\ f) = \mathcal{R}(t) \tag{4}$$

$$\mathcal{R}(t\ >>\ *.\ [a_1 \ldots a_n]) = \mathcal{R}(t) \tag{5}$$

$$\mathcal{R}(rpeat\ t) = \mathcal{R}(t) \tag{6}$$

$$\mathcal{R}(rpeatEvery\ d\ t) = \begin{cases} \mathcal{R}(t) & \text{if } t \text{ is UnStable} \\ \langle d, d \rangle & \text{otherwise} \end{cases} \tag{7}$$

$$\mathcal{R}(delay\ d) = \langle d, d \rangle \tag{8}$$

$$\mathcal{R}(t) = \begin{cases} \langle \infty, \infty \rangle & \text{if } t \text{ is Stable} \\ \langle r_l, r_u \rangle & \text{otherwise} \end{cases} \tag{9}$$

We use the operator $\cap_{safe}$ to compose refresh ranges. When the ranges overlap the result is the overlapping range. Otherwise, the range with the lowest numbers is produced. The rationale is that subtasks should not be delayed longer than their refresh range. Evaluating a task earlier should not change its result but can consume more energy.

$$\cap_{safe} :: \langle Int, Int \rangle\ \langle Int, Int \rangle \to \langle Int, Int \rangle$$

$$R_1 \cap_{safe} R_2 = R_1 \cap R_2 \qquad \text{if } R_1 \cap R_2 \neq \emptyset \tag{10}$$

$$\langle l_1, h_1 \rangle \cap_{safe} \langle l_2, h_2 \rangle = \langle l_2, h_2 \rangle \qquad \text{if } h_2 < l_1 \tag{11}$$

$$R_1 \cap_{safe} R_2 = R_1 \qquad \text{otherwise} \tag{12}$$

We will briefly discuss the various cases of deriving refresh rates.

Parallel Combinators For the parallel composition of tasks we compute the intersection of the refresh intervals of the components as outlined in the definition of $\cap_{safe}$. The operator $.||.$ in Equation 1 is the *or*-combinator; the first

subtask that produces a stable value determines the result of the composition. The operator `.&&.` in Equation 2 is the *and*-operator. The result is the tuple containing both results when both subtasks have a stable value.

Sequential Combinators For the sequential composition of tasks we only have to look at the refresh rate of the current task on the left. The opertor `≫|.` in Equation 3 is sequential composition similar to the monadic sequence operator `≫|`. The operator `≫=.` in Equation 4 provides the stable task result to the function on the right-hand side, similar to the monadic bind. The operator `>>~.` steps on unstable value and is otherwise equal to `≫=.`. The step combinator `>>*.` in Equation 5 contains a list of actions that specify a condition and a new task.

Repeat Combinators The repeat combinators repeats their argument indefinitely. The combinator `rpeatEvery` adds the given delay between those repetitions. The refresh rate is equal to the refresh rate of the current argument task. Only when `rpeatEvery` waits between the iterations of the argument the refresh interval is equal to the remaining delay time.

Other Combinators The refresh rate of the `delay` in Equation 8 is equal to the remaining delay. Refreshing stable tasks can be delayed indefinitely, their value will never change. For other basic tasks, the values from Table 3.2 apply. The values r_l and r_u in Equation 9 are the lower and upper-bound of the rate mentioned there.

As example we define a mTask that updates the SDS `tempSDS` in iTask once per minute.

```
delayTime =: lit (60 * 1000) // 1 minute in milliseconds

devTask :: Main (MTask v Real) | mtask, dht, liftsds v
devTask =
  DHT (DHT_pin DHT11) λdht =
  liftsds λlocalSds = tempSDS
  In {main = rpeatEvery delayTime (temperature dht >>~. setSds localSds)}
```

Listing 1.2. Updating a SDS in iTask once per minute.

3.3 User Defined Refresh Rates

In some applications, it is necessary to read sensors at a different rate than the default rate given in Table 3.2. This is achieved by defining such an object with a custom refresh rate. The required rate is an additional argument of its definition.

```
class dht v where
    temperature‘ :: (TimingInterval v) (v DHT) → MTask v Real
    temperature  ::                    (v DHT) → MTask v Real
    humidity‘    :: (TimingInterval v) (v DHT) → MTask v Real
```

```
    humidity     ::                    (v DHT) → MTask v Real

class dio p v | pin p where
    readD‘ :: (TimingInterval v) (v p) → MTask v Bool | pin p
    readD  ::                    (v p) → MTask v Bool | pin p
```

Listing 1.3. Definition for DHT sensors and reading digital values from GPIO pins with a custom timing interval.

The timing intervals are defined by a tailor-made algebraic data type.

```
:: TimingInterval v
  = Default
  | BeforeMs (v Int)            // yields ⟨0, x⟩
  | BeforeS  (v Int)            // yields ⟨0, x × 1000⟩
  | ExactMs  (v Int)            // yields ⟨x, x⟩
  | ExactS   (v Int)            // yields ⟨0 × 1000, x × 1000⟩
  | RangeMs  (v Int) (v Int)    // yields ⟨x, y⟩
  | RangeS   (v Int) (v Int)    // yields ⟨x × 1000, y × 1000⟩
```

Listing 1.4. The ADT for timing intervals in mTask

As example we define a mTask that updates the SDS `tempSDS` in iTask in a tight loop. The `temperature‘` reading requires that this happens at least once per minute. Without other tasks on the IoT node, the temperature SDS will be updated once per minute. Other tasks can cause a slightly more frequent update.

```
delayTime =: BeforeSec (lit (60 * 1000)) // 1 minute in milliseconds

devTask :: Main (MTask v Real) | mtask, dht, liftsds v
devTask =
  DHT (DHT_pin DHT11) λdht =
  liftsds λlocalSds = tempSDS
  In {main = rpeat (temperature‘ delayTime dht >>~. setSds localSds)}
```

Listing 1.5. Updating a SDS in iTask at least once per minute.

4 Implementing Refresh Rates

The refresh rates from the previous section tell us how much the next evaluation of the task can be delayed. An IoT node executes multiple tasks interleaved. In addition, it has to communicate with a server to collect new tasks and updates of SDSs. Hence, we cannot use those refresh intervals directly to let the microprocessor sleep. Our scheduler has the following objectives.

- Meet the deadline whenever possible. Only too much work on the device might cause an overflow of the deadline.
- Achieve long sleep times. Waking up from sleep consumes some energy and takes some time. Hence, we prefer a single long sleep over splitting this interval into several smaller pieces.
- The scheduler tries to avoid unnecessary evaluations of tasks as much as possible. A task should not be evaluated now whenever its execution can also be delayed until the next time that the device is active.

– The optimal power state should be selected. Although a system uses less power in a deep sleep mode, it also takes more time and energy to wake up from deep sleep. When the system knows that it can sleep only a short time it is better to go to light sleep mode since waking up from light sleep is faster and consumes less energy.

For the implementation, it is important to note that the evaluation of a task takes time. Some tasks are extremely fast, but other tasks require long computations and time-consuming communication with peripherals as well as with the server.

A naive implementation of `rpeatEvery` could just execute the given task once, wait the given delay and call itself recursively.

```
rpeatEvery :: (v Int) (MTask v a) → MTask v a
rpeatEvery d t = t ≫|. delay d ≫|. rpeatEvery d t
```

Listing 1.6. A naive implementation of rpeatEvery.

However, such an implementation will cause a considerable time drift. When it takes τ seconds to execute `t` the repeat time is actually $\tau+\tilde{\mathrm{d}}$. The solution is to determine a start time and to use a `waitUntil` primitive instead of the `delay`.

```
rpeatEvery :: (v Int) (MTask v a) → MTask v a
rpeatEvery d t = getTime ≫=. λtime.t ≫|. waitUnitl (time +. d) ≫|. rpeatEvery d t
```

Listing 1.7. An implementation of rpeatEvery without drift.

These start times are used at many places in our scheduler. The algorithm $\mathcal{R}$ computes evaluation rates of Individual tasks. This is a relative interval that should start at the time when that task is created. For the scheduler we transform these intervals to absolute evaluation intervals; the lower and upper bound of the start time of that task measured in the time of the IoT node. We obtain those bounds by adding the start time of the evaluation to the evaluation rate computed by $\mathcal{R}$.

4.1 Evaluation Rounds

To reduce the energy consumption of an IoT device as much as possible we put the microprocessor in its deepest sleep mode as long as possible. Since the current state of the task to be reduced has to be remembered most microprocessors can only go to a light sleep mode. It is fine to switch off the processor and the WiFi radio, but the memory should stay active. There are various special ways to store the current state during deep sleep, like adding FRAM memory and using the real-time clock memory of an ESP32. Currently, we do not use such an advanced method and limit the system to light sleep.

The algorithm $\mathcal{R}$ computes the evaluation rate of the current tasks. For the scheduler, we transform this interval to an absolute evaluation interval; the lower and upper bound of the start time of that task measured in the time of the IoT node. We obtain those bounds by adding the start time of the evaluation to the evaluation rate computed by $\mathcal{R}$.

Apart from the task to execute, the IoT device has to maintain the connection with the server and check there for new tasks and updates of Shared Data Sources. When the microprocessor is active it checks the connection and updates from the server and executes the task if it is in its execution window. Next, the microprocessor goes to light sleep for the minimum of server-interval and the task delay.

5 Scheduling Tasks

In general, the microprocessor node executes multiple mTasks. When there are no tasks to be executed the microprocessor can go to deep sleep since there is no task-specific data to be archived. The microprocessor wakes up from time to time to maintain the connection with the server and to check for new tasks allocated to this microprocessor.

In general, there are multiple tasks to be executed. We compute an absolute timing interval for each of these tasks as outlined above. We maintain a priority queue with the mTasks ordered at their absolute earliest start time. Execution of all tasks in the queue is delayed as much as possible. That is until the earliest last start time of all tasks in the queue or until the system has to become active for communication with the server. When the execution of a task has to be started, all tasks with their earliest start time before the current time are executed.

```
evalutionRound :: (Queue MTask) → (Queue MTask)
evalutionRound queue = round queue emptyQueue
where
    round queue seen =
        if (isEmpty queue)
            seen
            let (task, queue2) = pop queue in
            if (currentTime < earliest task)
                (insert seen queue)
                let task2 = evaluate task in
                round queue2 (if (isStable task2) seen (insert task2 seen))
```

Listing 1.8. Pseudo code to execute mTasks

It makes sense to execute tasks before their latest start time since waking up the entire system takes additional energy. A new execution interval is computed after the execution of a task for a single step. Based on this interval the new task is inserted in the priority queue. After stepping all tasks with an execution interval that contains the current time, the new sleeping time is determined. Note that this sleeping time can also be zero if one or more of these tasks require immediate execution. When the energy consumption of the computed sleeping time is higher than the energy required to wake up the system after the sleep, the microprocessor is brought into sleeping mode. When it is not worthwhile to go to sleeping mode the system executes tasks just as before the introduction of this energy optimisation.

6 Running Tasks on Interrupts

In this chapter, we show that we can also execute tasks based on interrupts. The interrupts we consider are generated from changes in the value of GPIOs triggered by sensors. These interrupts can wake up the microprocessor themselves. This is more energy-efficient than waking up regularly and polling the GPIOs. Moreover, it prevents missing input from sensors while the microprocessor is sleeping. The mTask system adds the task associated with the interrupt to the priority queue for immediate execution, i.e. execution interval $\langle 0, 0 \rangle$.

```
lightSwitch :: Main (MTask v Bool) | mtask v
lightSwitch
  = declarePin ButtonPin PMInput λbutton→
    declarePin BuiltinLEDPin PMOutput λled→
      fun λswitch=(λx→
        writeD led x
        ≫|. delay (lit 50) // Debounce
        ≫|. interrupt Falling button
        ≫|. switch (Not x)
    )
    In {main=switch (lit False)}
```

Listing 1.9. Example of a toggle switch with interrupts

7 Resulting Power Reductions

Measurements of simple examples show a drop in energy consumption from 27 up to 88%.

8 Conclusion

In this paper, we show how we can automatically associate execution intervals to tasks. Based on these intervals we can delay the executions of those tasks. When all task executions can be delayed, the microprocessor executing those tasks can go to sleep mode to reduce its energy consumption. This is a rather hard problem that must be solved dynamically since we make no assumptions on the number and nature of the tasks that will be allocated to an IoT node. The actual reduction of the energy is of course highly dependent on the number and nature of the task shipped to the IoT node. Our examples show a reduction in energy consumption of two orders of magnitude. Those reductions are a necessity for IoT nodes with battery power Given the exploding number of IoT nodes, such savings are also mandatory for other nodes to limit the total power consumption of the IoT.

References

1. Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean — A language for functional graph rewriting. In: Kahn, G. (ed.) Functional Programming

Languages and Computer Architecture. pp. 364–384. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
2. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009). https://doi.org/10.1017/S0956796809007205, https://doi.org/10.1017/S0956796809007205
3. Koopman, P., Lubbers, M., Plasmeijer, R.: A Task-Based DSL for Microcomputers. In: Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018. pp. 1–11. ACM Press, Vienna, Austria (2018). https://doi.org/10.1145/3183895.3183902, http://dl.acm.org/citation.cfm?doid=3183895.3183902
4. Lubbers, M., Koopman, P., Ramsingh, A., Singer, J., Trinder, P.: Tiered versus tierless iot stacks: Comparing smart campus software architectures. In: Proceedings of the 10th International Conference on the Internet of Things. IoT '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3410992.3411002, https://doi.org/10.1145/3410992.3411002
5. Nižetić, S., Šolić, P., López-de-Ipiña González-de-Artaza, D., Patrono, L.: Internet of things (iot): Opportunities, issues and challenges towards a smart and sustainable future. Journal of Cleaner Production **274**, 122877 (2020). https://doi.org/https://doi.org/10.1016/j.jclepro.2020.122877, https://www.sciencedirect.com/science/article/pii/S095965262032922X
6. Plasmeijer, R., Achten, P., Koopman, P.: iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007). pp. 141–152. ACM, Freiburg, Germany (Oct 1–3 2007)
7. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. p. 195–206. PPDP '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2370776.2370801, https://doi.org/10.1145/2370776.2370801
8. Stutterheim, J., Achten, P., Plasmeijer, R.: Maintaining Separation of Concerns Through Task Oriented Software Development. In: Wang, M., Owens, S. (eds.) Trends in Functional Programming, vol. 10788, pp. 19–38. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89719-6, http://link.springer.com/10.1007/978-3-319-89719-6_2
9. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. **30**(6) (oct 2008). https://doi.org/10.1145/1391956.1391958, https://doi.org/10.1145/1391956.1391958

Semantic equivalence of task-oriented programs in TopHat

Tosca Klijnsma[1] and Tim Steenvoorden[2]

[1] Radboud University, Nijmegen, The Netherlands. tosca.klijnsma@gmail.com
[2] Open University, Heerlen, The Netherlands. tim.steenvoorden@ou.nl

Abstract. Task-oriented programming (TOP) is a new programming paradigm for specifying multi-user workflows. The iTasks framework is an implementation of TOP in the functional programming language Clean. To reason formally about TOP programs, a formal language called TopHat has been designed, together with its operational semantics. For proving properties about task-oriented programs, it is desirable to be able to say when two TopHat-programs are semantically equivalent. This paper aims to answer this question. We show that a task can be in either one of five conditions after fixation, and for every two tasks in the same condition, we define what it means for them to be semantically equivalent. Using this definition, we study a number of transformation laws for TopHat-programs. Amongst those, we show that the Task operation on types in TopHat cannot be a monad.

Keywords: Task-oriented programming · Program equivalence · Formal semantics.

1 Introduction

Task-oriented programming (TOP) is a relatively new programming paradigm designed for developing distributed interactive multi-user workflow systems. In this programming paradigm, the concept of a *task* plays a central role. A task is a unit of work assigned to some user, and consists of two parts: a description of the work that should be done, and a typed interface that defines the type of the task value that it returns. Tasks are described in an abstract, declarative manner, and from this abstract description, a TOP engine automatically generates a GUI. It also takes care of the client-server communication and user authentication that is needed for users to work together on tasks. TOP allows programmers to define workflows which describe what tasks should be executed by its users, without having to worry about how this is achieved.

User collaboration is a central concept in TOP. The different ways in which users can collaborate are captured by *task combinators*. By using these combinators, TOP-programmers can construct larger tasks from smaller ones in several ways. There is *sequential composition*, which allows tasks to be executed one after the other. And there is *parallel composition*, which allows tasks to be executed in parallel at the same time. For parallel composition, it is possible to either combine the results, or to conditionally continue with either one of two tasks.

In order to collaborate, users also need to be able to communicate with each other, and with the system. Using task combinators, it is possible to pass along *data* from one task onto the next. For communication with the outside world, there are *editors*. They provide interaction with the environment via *input* events. Editors are typed containers which remember the last value that has been sent to them, and users can communicate with the system through these editors. Furthermore, editors allow users to view and edit *shared data*, which are mutable references whose changes are immediately visible to all other tasks watching them.

Another important element of TOP is that task are *typed*. This is important to determine the type of the values that are communicated to other tasks and to the environment. Not all tasks have a value, and a task does not just produce a value when it is completed. Instead, a task's value is continually updated while the work takes place, and can be observed at any point during execution. Moreover, it may be possible that a task is never completed, and that its value never reaches a stable condition. A task's value reflects a task's current progress. They can be inspected by other tasks to base decisions on, which in turn can impact the things users can see or do.

Finally, the TOP language is *modular*: tasks are composed of smaller tasks, and can be arguments to or results of functions. This allows programmers to re-use tasks, and to model their own collaboration patterns using higher-order tasks [Plasmeijer et al., 2012].

Related work

TOP describes in an abstract way *what* work should be done by the system and its users. It does not describe *how* this should be done, this question should be answered by a *TOP engine*. The iTASKS engine [Achten et al., 2013] is an implementation of TOP, written in the pure and lazy functional programming language CLEAN [Plasmeijer et al., 2002]. It is implemented as a shallowly embedded domain-specific language, which means that it inherits features from its host language CLEAN. Amongst these features is a strong typing system, and because CLEAN is a functional language, it allows task combinators to be expressed as functions. From the high level description of the tasks, iTASKS generates a web application that is able to execute the described tasks. It takes care of generating a GUI, and of coordinating the tasks in a distributed manner by using a client-server architecture. The server side of an iTASKS application runs a web service to which users on a wide range of different devices can connect, and the client side realises the front-end components. This way, programmers of iTasks-programs need not be concerned by lower-level implementation details. iTASKS has shown itself effective in the past for the implementation of interactive, distributed, workflow applications [Lijnse et al., 2012, Stutterheim et al., 2017].

Because iTASKS has been designed for developing real-world applications, reasoning formally about iTASKS programs is hard [Koopman et al., 2008]. Steenvoorden et al. [2019] introduces TOPHAT, a formal language plus operational semantics for reasoning about task-oriented programs. The semantics of TOPHAT have been machine verified[3]

[3] https://github.com/timjs/tophat-proofs

using the dependently typed programming language Idris [Brady, 2013]. These formal semantics have been used to prove user defined properties of tasks using symbolic execution [Naus et al., 2019]. An example of this are the circumstances when a a tax subsidy request for solar panels should be granted. Symbolic execution has also been applied to generate next step hints for end users [Naus and Steenvoorden, 2020]. For example, does the given data during a subsidy request indeed lead to subsidy greater then zero?

Motivation

When two tasks are equivalent is a long standing open research question. The formal semantics of TopHat are a good starting point to investigate this question. TopHat is a domain specific language embedded in the simply typed λ-calculus. The semantic equivalence in context of the (simply typed) λ-calculus has been extensively studied [Pitts, 2000, Sewell, 2017]. As TopHat is build on top of the simply typed λ-calculus, we can relay on this work for equivalence in the host language. However, this knowledge is not immediately transferable to the world of tasks. There, need to take the semantics of task constructs into account.

Example 1 (Task equivalence). The next two tasks both ask end users to enter an integer, and thereafter show the absolute value of the given integer:

$$t_1 := \square\mathsf{Int} \blacktriangleright \lambda x : \mathsf{Int}.\ \textbf{if}\ x < 0\ \textbf{then}\ \boxtimes(-x)\ \textbf{else}\ \boxtimes x$$
$$t_2 := \square\mathsf{Int} \blacktriangleright \lambda y : \mathsf{Int}.\ \textbf{if}\ y \geq 0\ \textbf{then}\ \boxtimes y\ \textbf{else}\ \boxtimes(-y)$$

It should be clear that these two tasks are equivalent.

If we can define such a notion of semantic equivalence for task-oriented programs, what interesting properties can we prove (or disprove)? If we know that one task-oriented program is semantically equivalent to another, then we know that we can substitute one for the other, without changing the meaning of the program. Which in turn could be useful for doing compiler optimizations. An example of such optimisations is whether the monad laws hold for the step combinator ($\blacktriangleright$) used in above example. Showing that certain equalities hold for task-oriented programs could also prove useful for the iTasks system in the future.

Structure

The remainder of this paper is structured as follows. Section 2 will explain the components of the TopHat language, together with some examples. We will follow the updated semantics presented in Steenvoorden [2022]. In the following section, Section 3, we give a short overview of what contextual equivalence means. Then, in Section 4, we give a definition for the semantic equivalence of two expressions in our host language, and in Section 5 we show that a task can be in either one of five conditions. For every two tasks in the same condition, we define what it means for them to be semantically equivalent in Section 6. Additionally, Section 7 presents a set of laws that we believe hold true or false for TopHat-programs according to our definition. Finally, Section 8 will conclude this paper.

2 The TopHat language

TopHat is a formal language for reasoning about task-oriented programs. It is described by a layered operational semantics, consisting of multiple big-step semantic functions for reducing expressions, and two labelled transition systems for handling user inputs. Its main layers are *evaluation*, *fixation*, and *interaction*. To make clear which features come from TOP and which features come from functional programming, TopHat is separated into a task language and an underlying host language [Steenvoorden et al., 2019, Steenvoorden, 2022].

The host language of TopHat is a simply typed λ-calculus, extended with with basic types β, references h, and the task type constructor Task. Basic types are the unit type; primitive types for booleans, integers, and strings; and pairs of other basic types. We will present the main components of the task language, which is embedded into the host language, in the next subsections.

2.1 Editors

Editors allow end users to interact with the system by entering and changing information. When a user sends an input event to an editor, the editor will update its current value to reflect the change. There are no output events. Instead, the current value of an editor can be observed and used in subsequent tasks. All editors and input events are *keyed* by a unique key k. This is to identify which editor needs to handle which user input. There are three types of editors in TopHat:

- *Empty editors* or *unvalued editors* ($\square^k \beta$) are editors that currently hold no value. They can be seen as an input prompt to the user to enter data. Empty editors are annotated with a basic type β, which means that only basic values of type β are accepted by the editor. Once an empty editor receives a valid input event, it becomes a filled editor containing the new data.
- *Filled editors* or *valued editors* ($\boxminus^k b$) are editors that currently hold the basic value b. Filled editors can be seen as either outputting a value, or as an input prompt that comes with a default value. They can never be cleared, only updated with new values of the same type.
- *Shared editors* ($\boxplus^k h$) watch heap locations h. They allow the user to view and change shared values. Whenever a shared editor is updated, all shared editors watching the same heap location h will be updated as well.

Filled editors and shared editors both have read-only variants ($\boxdot^k b$, $\boxdot^h$). End users can see the value in a read-only editor, but not change it. As editors can only handle basic values of type β, there is also a lift combinator ($\blacksquare v$) which lifts any value v into the task world.

2.2 Sequential composition

The step combinator ($\blacktriangleright$) allows the result from one task to determine the next task. We call this *sequential composition*. The step combinator expects a task t of type Task τ_1 on

the left hand side, and a continuation e on the right hand side, which is a function from from τ_1 to a successor task of type Task τ_2. Steps are guarded. A step can only be taken if two conditions are met: (1) the task on the left-hand side has an observable value; and (2) the evaluation of the continuation on the right-hand side with this value does not fail. A failing task ($\lightning$) stands for an impossible task. A task that is failing never has a value and never accepts input.

Example 2 (Coffee machine). Consider the following task-program:

let coffeeMachine : Task Drink = □Int ▶ λx.
  if $x \equiv 1$ **then** ■Coffee **else if** $x \equiv 2$ **then** ■Tea **else** $\lightning$

This program describes a coffee machine that can either serve coffee or tea. Coffee is served when one coin is inserted, and tea is served when two coins are inserted. For any other number of coins, the step remains guarded by the failing task ($\lightning$). This means that the coffee machine returns nothing and waits until a correct number of coins is inserted.

The transform combinator (•) maps a function over a task. It takes a function of type $\tau_1 \to \tau_2$ on the left-hand side, and a task of type Task τ_1 on the right-hand side, resulting in a task of type Task τ_2. If we want to apply the function and use the result, we would need to use the transform combinator in combination with the step combinator, as we can see in the example below.

Example 3 (Traffic light). Let us consider a simple example:

let trafficLight : Task Light =
  ((λx. **if** x **then** Green **else** Red) • □Bool) ▶ λy. ⧈y

This program describes a traffic light whose light is initially turned off, but given the right input, it can either become red or green. So long as no input is given, the transform task on the left-hand side of the step combinator has no value, and the step remains guarded. Once an input is entered, transform returns the value Green if True was entered, and Red if False was entered, upon which the step proceeds and displays the result.

2.3 Parallel composition

Pairing (⋈) combines the result of two tasks, but only if both branches have a value. If the left task is of type Task τ_1, and the right task is of type Task τ_2, then their pairing is of type Task $(\tau_1 \times \tau_2)$. However, if one or both branches have no value, then the resulting task also has no value.

Choosing (◀▶) chooses one of two branches. Therefore, both branches should be of the same type Task τ. This combinator is left-biased: it returns the leftmost task that has a value. If neither task has a value, then the resulting task also has no value. See also Fig. 2 for the definition of the value observation for pairing and choice.

These combinators allow user to work on two tasks in parallel, but unlike the name suggests, parallel does not mean that there is non-determinism. The order of execution is determined by the order of user inputs send. Instead, parallel here means that the order in which we execute the tasks, and their subtasks, does not matter.

Example 4 (Breakfast). Let us consider the following task, which makes use of both parallel combinators:

```
let make : τ → Task τ = λx. □Unit ▶ λy. ⧈x in
let makeBreakfast : Task (Drink × Food) =
  ((make Tea ◆ make Coffee) ⋈ make Egg) ▶ eatBreakfast
```

This program describes a simple workflow for making breakfast. Breakfast consists of something to drink (tea or coffee), and something to eat (eggs). The drink and the food are prepared in parallel (⋈), which means that the order in which they are made does not matter. For the drink, users have a choice (◆) whether they want tea or coffee with their breakfast. For the food, users will always make an egg. We will use the function make to simulate that the user must first perform an action (i.e. send user input) before an item is prepared and the task has a value. Only when both the drink and food are ready, can the step be taken and can we enjoy our breakfast.

2.4 Observations

Not just $\lightning$ can fail, tasks with failing subtasks can also fail. For example, the pairing $\lightning \bowtie \lightning$ is also a failing task, and is equivalent to $\lightning$. To capture what tasks are failing we introduce the failing observation $\mathcal{F}$, which is defined in Fig. 1. Failing is especially useful when used in combination with the step combinator, as we have seen in Example 2.

$\mathcal{F}$: Task → Boolean

$$
\begin{aligned}
\mathcal{F}(\lightning) &= \text{True} \\
\mathcal{F}(e_1 \bullet t_2) &= \mathcal{F}(t_2) \\
\mathcal{F}(t_1 \blacktriangleright e_2) &= \mathcal{F}(t_1) \\
\mathcal{F}(t_1 \bowtie t_2) &= \mathcal{F}(t_1) \wedge \mathcal{F}(t_2) \\
\mathcal{F}(t_1 \blacklozenge t_2) &= \mathcal{F}(t_1) \wedge \mathcal{F}(t_2) \\
\mathcal{F}(\_) &= \text{False}
\end{aligned}
$$

$\mathcal{I}$: Normalised task → $\mathcal{P}$(Input)

$$
\begin{aligned}
\mathcal{I}(\square^k \beta) &= \{k!b' \mid b' : \beta\} \\
\mathcal{I}(\boxminus^k b) &= \{k!b' \mid b' : \beta\} \quad \text{where } \boxminus b : \text{Task}\beta \\
\mathcal{I}(\boxplus^k h) &= \{k!b' \mid b' : \beta\} \quad \text{where } \boxplus l : \text{Task}\beta \\
\mathcal{I}(e_1 \bullet t_2) &= \mathcal{I}(t_2) \\
\mathcal{I}(t_1 \blacktriangleright e_2) &= \mathcal{I}(t_1) \\
\mathcal{I}(t_1 \bowtie t_2) &= \mathcal{I}(t_1) \cup \mathcal{I}(t_2) \\
\mathcal{I}(t_1 \blacklozenge t_2) &= \mathcal{I}(t_1) \cup \mathcal{I}(t_2) \\
\mathcal{I}(\_) &= \varnothing
\end{aligned}
$$

Fig. 1. Failing and inputs observations on tasks

Once a TopHat program is normalised, it is possible to send input to it. The input event $k!b$ indicates that the input b should be entered into the editor with key k. To do this, it is useful to know what inputs a given task accepts. The observation function $\mathcal{I}$ returns the set of input events that are currently possible for a given task. Its definition is given in Fig. 1. For the three read-write editors, $\mathcal{I}$ returns all input events $k!b$ where b is of the correct type. For task combinators, $\mathcal{I}$ is defined recursively. For all other tasks, $\mathcal{I}$ returns the empty set. We consider an input event ι a *valid* input event for the task t iff $\iota \in \mathcal{I}(t)$.

To be able to normalise tasks, we need to have a way to determine the value of tasks. For this, we introduce the task observation $\mathcal{V}$. Given a task t : Task τ and its current state σ, this function returns the task's value v of type τ. It is also possible that a task's value is undefined, in which case we write $\mathcal{V}(t, \sigma) = \bot$. The definition of $\mathcal{V}$ is given in Fig. 2.

$\mathcal{V}$: Normalised task $\times$ State $\rightharpoonup$ Value

$$
\begin{array}{lcl}
\mathcal{V}(\square^k \beta, \sigma) & = & \bot \\
\mathcal{V}(\boxminus^k b, \sigma) & = & b \\
\mathcal{V}(\boxdot^k b, \sigma) & = & b \\
\mathcal{V}(\boxplus^k h, \sigma) & = & \sigma(h) \\
\mathcal{V}(\boxtimes^k h, \sigma) & = & \sigma(h) \\
\mathcal{V}(\blacksquare v, \sigma) & = & v \\
\mathcal{V}(\lightning, \sigma) & = & \bot
\end{array}
\qquad
\begin{array}{lcl}
\mathcal{V}(t_1 \blacktriangleright e_2, \sigma) & = & \bot \\
\mathcal{V}(e_1 \bullet t_2, \sigma) & = & \begin{cases} v_2' & \text{when } \mathcal{V}(t_2, \sigma) = v_2 \wedge e_1\ v_2 \downarrow v_2' \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{V}(t_1 \bowtie t_2, \sigma) & = & \begin{cases} \{v_1, v_2\} & \text{when } \mathcal{V}(t_1, \sigma) = v_1 \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \text{otherwise} \end{cases} \\
\mathcal{V}(t_1 \blacklozenge t_2, \sigma) & = & \begin{cases} v_1 & \text{when } \mathcal{V}(t_1, \sigma) = v_1 \\ v_2 & \text{when } \mathcal{V}(t_1, \sigma) = \bot \wedge \mathcal{V}(t_2, \sigma) = v_2 \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 2. Value observation on tasks

2.5 Semantics

Evaluating terms in the host language to values is handled by the big-step *evaluation* semantics, where a value is an expression in the host language that cannot be reduced further. Values v can be lambda functions, pairs of values, unit, constants, locations, or tasks. Basic values b are a subset of values v that are of basic type β. We denote the evaluation of expression e to the value v by $e \downarrow v$.

After evaluation is done, a task is ready to be *fixated*. Fixation is a big-step semantics that is responsible for reducing tasks until they are ready to accept input. We write $t, \sigma, \delta \Downarrow t', \sigma'$ to denote the fixation of task t in state σ given the dirty set δ, results into task t' in state σ'. The set δ' contains all heap locations whose value has been changed while fixation took place. The state σ is a mapping from heap locations to basic values. It keeps track of all references created so far, and what value they currently hold. Fixation makes use of *normalisation*, which is a helper semantics.

Input events are handled by the *interaction* semantics. For the interaction semantics, we write $t, \sigma \overset{\iota}{\Longrightarrow} t', \sigma'$ to denote that task t in state σ transitions to task t' in state σ' after the user input ι. The interaction semantics makes use of a helper *handling* semantics, and the previously mentioned fixation semantics to make sure that, after user interaction, a task is fully reduced and ready to accept the next input. For further details we refer to Steenvoorden [2022].

3 Contextual equivalence

Our goal is to examine when two TopHat-programs are semantically equivalent. However, let us first consider what it means in general for two programs e_1 and e_2 to be

semantically equivalent, denoted by $e_1 \simeq e_2$. According to Sewell [2017], a good definition of semantic equivalence should satisfy the following properties:

1. Programs that result in values that are observably different should not be equivalent.
2. Programs that terminate should not be equivalent to programs that do not terminate.
3. The relation $\simeq$ should be an *equivalence relation*: it is reflexive, symmetric and transitive.
4. The relation $\simeq$ should be a *congruence*. That is if $e_1 \simeq e_2$, then for all program contexts $E[\cdot]$, we like to have $E[e_1] \simeq E[e_2]$. Here we fill the hole $\cdot$ with any well typed expression,
5. The relation $\simeq$ should contain as many programs as possible subject to the above properties.

It should be obvious that the first three properties are essential. The fourth property about congruence states that if two programs e_1 and e_2 are semantically equivalent, then we should be able to use e_1 and e_2 interchangeably within any program without changing its meaning. Finally, the last property ensures that $\simeq$ is not just the empty relation. We will keep these properties in mind when giving our definitions of semantic equivalence. Throughout this text, we use the symbol $\simeq$ for the semantic equivalence of two expressions in the host language, and the symbol $\cong$ for the semantic equivalence of two tasks.

Next, in Section 4, we give a definition for the semantic equivalence of two expressions in our host language, the simply typed λ-calculus. Then, in Section 5, we introduce five conditions tasks can be in. Using these conditions, we define the semantic equivalence of two tasks in Section 6. Finally, Section 7 presents a set of laws that we believe hold true for TopHat-programs with our definition.

4 Expression equivalence

Before we consider semantic equivalence of tasks, we first look at semantic equivalence of expressions in the host language. Let us start with an example.

Example 5 (Expression equivalence). Consider the following two expressions:

$$e_1 := \lambda x : \textsc{Int}.\ \textbf{if}\ x < 0\ \textbf{then}\ -x\ \textbf{else}\ x$$
$$e_2 := \lambda y : \textsc{Int}.\ \textbf{if}\ y \geq 0\ \textbf{then}\ y\ \textbf{else}\ -y$$

It should be obvious that these two functions are equivalent: they both return the absolute value of their argument. Therefore, we should be able to use them interchangeably within any TopHat-program without changing its behavior. So even though the functions e_1 and e_2 are different, we will never detect a difference between them when they are being used within a TopHat-program, because for all possible arguments, e_1 and e_2 evaluate to the same result.

When deciding if two expressions in the host language are equivalent, it is not enough to just look at the resulting value after evaluation. We need to consider all *contexts* that an expression can be used in. This leads to the definition of *contextual equivalence*. Pitts [2000] defines contextual equivalence informally as follows:

> Two phrases of a programming language are *contextually equivalent* if any occurrences of the first phrase in a complete program can be replaced by the second phrase without affecting the observable results of executing the program.

This kind of equivalence is also called *operational*, or *observational* equivalence. To formally define such a notion of contextual equivalence for a given programming language, we must answer two questions: 1. What is a *complete program*? 2. What are the *observable results*? Depending on the answers to these two questions, this can result in different definitions of semantic equivalence for the same programming language [Pitts, 2000]. For expressions in TopHat, we answer these two questions as follows:

1. We will consider an expression in the host-language a complete program if it does not contain any free variables.
2. The only observation we are interested in is the resulting value after evaluation.

We need a way to substitute an expression in a program by another. For this, we use the notion of an expression context. An *expression context* $E[\cdot]$ is a complete program that can contain holes, denoted by the symbol $\cdot$, which can be filled. We write $E[e]$ for the expression that results from replacing all occurrences of $\cdot$ in E by e. So, we will replace all holes with the same expression e. In the most common case, there will be just one hole in E to be filled. Fig. 3 gives the grammars for expression- and task-contexts.

$E ::=$	Contexts
$\cdot$	hole
$\lambda m : \tau.\ E$	abstraction
$E_1\ E_2$	application
x	variable
h	heap location
if E_1 **then** E_2 **else** E_3	conditional
$\{\}$	unit
$\{E_1, E_2\}$	tuple
c	constant
P	pre-task context

$P ::=$	Pre-task contexts
$\square^{\nu}\beta$	unvalued editor
$\boxminus^{\nu}E$	valued editor
$\boxplus^{\nu}E$	shared editor
$\boxdot^{\nu}E$	valued read-only editor
$\boxtimes^{\nu}E$	shared read-only editor
$E_1 \bullet E_2$	transform
$E_1 \bowtie E_2$	pair
$\blacksquare E$	lift
$E_1 \blacklozenge E_2$	choose
$\lightning$	fail
$E_1 \blacktriangleright E_2$	step
share E	share
$E_1 := E_2$	assign

Fig. 3. Context grammar

If we would allow contexts E to be of every available type, we would have the same problem as introduced in Example 5. Then, E can also be a lambda function and we can only determine that two lambda functions are equivalent by considering all contexts that they can be used in. To avoid a circular definition, our definition of expression equivalence will only quantify over contexts of basic types β. Indeed, we can only directly observe the equivalence of two basic values.

Taking this together, this leads to the following definition of expression equivalence:

Definition 1 (Expression equivalence ($\simeq$)). *Given two expressions $e_1, e_2 : \tau$ where $\tau \neq \textsf{Task}\ \tau'$ for some τ', we say that e_1 and e_2 are semantically equivalent, written $e_1 \simeq e_2$ if for all contexts $E[\cdot] : \beta$ where $\cdot : \tau$, and for all values $b : \beta$ we have that $E[e_1] \downarrow b$ if and only if $E[e_2] \downarrow b$*

Actually proving that two expressions are contextually equivalent is hard, as we would need to quantify over all contexts. That is, we would need to consider all possible ways that a program can use an expression. However, showing that two expressions e_1 and e_2 are contextually *in*equivalent is straightforward. All we have to do is find one context $E[\cdot] : \beta$ such that $E[e_1] \downarrow b_1$ and $E[e_2] \downarrow b_2$ with $b_1 \neq b_2$.

Example 6 (Expression inequivalence). The expressions

$$e_1 := \lambda x : \textsf{Int}.\ \textbf{if}\ x < 0\ \textbf{then}\ 2\ \textbf{else}\ 3$$
$$e_2 := \lambda x : \textsf{Int}.\ \textbf{if}\ x > 0\ \textbf{then}\ 3\ \textbf{else}\ 2$$

are not contextually equivalent. Take the context $E[\cdot] : \textsf{Int}$ with $E[\cdot] = \ \cdot\ 0$, then:

$$E[e_1] = (\lambda x : \textsf{Int}.\ \textbf{if}\ x < 0\ \textbf{then}\ 2\ \textbf{else}\ 3)\ 0 \downarrow 3$$
$$E[e_2] = (\lambda x : \textsf{Int}.\ \textbf{if}\ x > 0\ \textbf{then}\ 3\ \textbf{else}\ 2)\ 0 \downarrow 2$$

So we found a context for which both expressions evaluate to a different (basic) value.

5 Task conditions

For expression equivalence, we needed contexts to determine the equivalence of two lambda functions, whose results can only be observed after evaluation. For tasks however, we do not need contexts to view their results. A task's value can be determined at any point during execution by using the $\mathcal{V}$ observation, whereupon it either has a value, or it is undefined. Note that, we will restrict ourselves to tasks which result in a basic value, as these are results which are directly comparable.

On the other hand, tasks do allow user interaction, and depending on what inputs are sent, the resulting task may be different. So while lambda functions can produce different results depending on their arguments, so can tasks produce different results depending on what inputs are send to them. So, in a sense, for tasks the contexts are user input.

To satisfy the first property at the beginning of **??**, that programs resulting in values which are observably different must not be equivalent, we will use the observations

on tasks, as introduced in Section 2.4. Before we observe a task however, we should first fully fixate the tasks whose equivalence we want to determine. We use the fixing semantics ($\Downarrow$) to ensure that tasks are fully fixated. Fixation keeps track of a state σ. For semantic equivalence, we do not want that fixation results in two different states. So, given two tasks $t_1, t_2 : \textsc{Task}\ \beta$, we want that for all states σ, fixation of t_1 and t_2 end in the same state σ':

$$t_1, \sigma, \varnothing \Downarrow t_1', \sigma' \iff t_2, \sigma, \varnothing \Downarrow t_2', \sigma'$$

After fixation, we need to decide what observations t_1' and t_2' must have in common for them to be considered equivalent. Let us recall what observations can be made on tasks. The value function $\mathcal{V}$ returns the value v of a task, or $\bot$ if it is undefined; there is the failing function $\mathcal{F}$ which returns whether a task is failing; and there is the inputs function $\mathcal{I}$ which returns the set of all possible input events that a task accepts, that is, their *input space*. The value and failing functions are used in the normalisation rules of TopHat, and different observations for these functions can result in different derivation rules being triggered. Therefore, we can say that tasks for which the value or failing function return a different result must not be semantically equivalent.

Similarly, tasks whose inputs function $\mathcal{I}$ return a different set of input events can also not be semantically equivalent, because that would mean that the types of interaction that can be done with the tasks are different. Recall that input events should be named by the same key as the editor it is meant for. So if we require that $\mathcal{I}(t_1') = \mathcal{I}(t_2')$, then this also implies that t_1' and t_2' must have the same keys for all their editors. If this is not the case, we can never have that $\mathcal{I}(t_1') = \mathcal{I}(t_2')$, and the tasks cannot be semantically equivalent.

Given these impressions, we say that at least the following property must hold for t to be equivalent to t':

$$\mathcal{F}(t) = \mathcal{F}(t') \text{ and } \mathcal{V}(t, \sigma) = \mathcal{V}(t', \sigma) \text{ and } \mathcal{I}(t) = \mathcal{I}(t')$$

For any of these task observations, we can distinguish two cases: either a task fails or does not fail, it either has a value or its value is undefined, and it either accepts input or it accepts no more input. Based on these case distinctions, we say that a task is in either one of five conditions after fixation. These task conditions, and some examples, are shown in Table 1. In the next subsections, we will discuss each condition in more detail and give some examples.

5.1 Failing tasks

A *failing* task t is a task for which the failing function $\mathcal{F}(t)$ yields true.

In the original TopHat paper [Steenvoorden et al., 2019], Theorem 6.5 states that a task fails if and only if it accepts no more user input. However, with the introduction of the lift combinator (■) in Steenvoorden [2022], this is no longer the case. For example, ■e does not fail, and neither does it accept user input. The same holds for read-only editors (⧈, ⊞). What we still can say however, is that if a task is failing, we know that it has no value and accepts no more user input. We have proven this property in Proposition 12 in the appendix.

$\mathcal{F}$	$\mathcal{V}$	$\mathcal{I}$	Condition	Examples
✓	–	–	Failing	$\lightning$, $\lightning \bowtie \lightning$, $\lightning \blacklozenge \lightning$, $(\lambda x.x) \bullet \lightning$, $\lightning \blacktriangleright \lambda x.\, \boxdot\, x$
–	✓	–	Finished steady	$\boxdot 2$, $\boxdot 2 \bowtie \boxdot 3$, $\boxdot\{2,3\}$, $(\lambda x.x+1) \bullet \boxdot 2$
–	✓	✓	Finished unsteady	$\boxminus 2$, $\boxminus 2 \bowtie \boxminus 3$, $\boxminus\{2,3\}$
–	–	✓	Running	$\square\textsc{Int} \bowtie \lightning$, $\boxminus 2 \blacktriangleright \lambda x.\lightning$
–	–	–	Stuck	$\boxdot 2 \blacktriangleright \lambda x.\lightning$, $\boxdot 2 \bowtie \lightning$, $\lightning \bowtie \boxdot 2$

Checkmarks for $\mathcal{F}$, $\mathcal{V}$ and $\mathcal{I}$ for a task t and a state σ indicate that $\mathcal{F}(t) = \text{True}$, $\mathcal{V}(t,\sigma) = v$ for $v \neq \bot$, and $\mathcal{I}(t) \neq \varnothing$ respectively.

Table 1. Conditions for fixated tasks

Proposition 1 (Failing tasks stay failing). *If t is a failing task, then it stays failing. Or, more formally: for all fixated tasks t : $\textsc{Task}\ \tau$ and states σ, we have that if $\mathcal{F}(t)$, there is no input ι such that $t, \sigma \xRightarrow{\iota} t', \sigma''$.*

Proof. Once a task fails, it will always remain failing, because by Corollary 3 from the appendix no more user interaction is possible, and by assumption, the task is already fixated.

Failing can thus be regarded as one type of termination, and we will consider all tasks that fail to be equivalent. Hence, we will say that the tasks $\lightning$, $\lightning \bowtie \lightning$, $\lightning \blacklozenge \lightning$, $(\lambda x.\, x) \bullet \lightning$, $\lightning \blacktriangleright \lambda x.\, \boxdot\, x$, and all other failing tasks, are semantically equivalent to each other.

5.2 Finished tasks

A *finished* task t is a task which yields a value $\mathcal{V}(t,\sigma) = v$ for $v \neq \bot$. This value can either be *steady* when no more user input is possible, or *unsteady* when the task still accepts user input. An example of a finished task with a steady value is the task $\boxdot^k 42$. Because this task accepts no more user input, its value will always remain equal to 42. An example of a finished task with an unsteady value is the task $\boxminus^k 42$. This task still accepts user input, and thus its value can keep on changing. Even though both tasks yield the same value, they should not be equivalent, since one's value can be changed and the other one's value cannot.

Definition 2 (Finished, steady, and unsteady task). *We call a fixated task t : $\textsc{Task}\ \tau$ in state σ* finished *if and only if $\mathcal{V}(t,\sigma) = v$, for $v \neq \bot$. Furthermore, we call a finished task* steady *if and only if $\mathcal{I}(t) = \varnothing$, and* unsteady *in the other case.*

A steady task can thus never be semantically equivalent to an unsteady task, even if their values are (initially) the same. But just looking at the resulting values, and whether the resulting value is steady or not, is not enough to determine semantic equivalence of finished tasks. Consider for example the following tasks:

$$t_1 := \boxdot\{2,3\} \qquad t_3 := \boxminus^{k_1}\ \{2,3\}$$
$$t_2 := \boxdot 2 \bowtie \boxdot 3 \qquad t_4 := \boxminus^{k_1}\ 2 \bowtie \boxminus^{k_2} 3$$

These are all finished tasks with value $\{2, 3\}$. Tasks t_1 and t_2 are both steady, and thus $t_1 \cong t_2$. We have already concluded that steady tasks cannot be equivalent to unsteady tasks, so $t_1 \ncong t_3$, $t_1 \ncong t_4$, $t_2 \ncong t_3$, and $t_2 \ncong t_4$. But we will also say that $t_3 \ncong t_4$, because we have that:

$$\mathcal{I}(t_3) = \{k_1!b_1 \mid b_1 : \textsf{INT} \times \textsf{INT}\}$$
$$\mathcal{I}(t_4) = \{k_1!b_1 \mid b_1 : \textsf{INT}\} \cup \{k_2!b_2 \mid b_2 : \textsf{INT}\}$$

Meaning that the types of interaction that can be done with t_3 differ from the types of interaction that can be done with t_4. In the case of t_3, the user can only alter the pair in one go, whereas for t_4, the user can partially update it. So for finished tasks, we also need to require that $\mathcal{I}(t) = \mathcal{I}(t')$ if we want to conclude that $t \cong t'$.

And yet this is still not enough... Consider the following two tasks:

$$t_5 := (\lambda x.\ \textbf{if}\, x < 0\, \textbf{then}\, -x\, \textbf{else}\, x) \bullet \boxminus^k 42$$
$$t_6 := \boxminus^k 42$$

Both tasks have the value 42 and accept the same input. However for negative values, the resulting values diverge, because the transform function in task t_5 ensures that its output is always positive. So if a task still accepts user input, not only do we need to look at a task's current value, but we also need to consider all values that a task can have after user interaction.

We will regard finished tasks as another type of condition a task can be in after fixation. Of course, if the value is unsteady, the task is not terminated in the true sense of the word, since there is still user interaction possible. In fact, the task will never terminate: a finished task will always remain a finished task with the same input space, only its value may change.

Proposition 2 (Finished tasks stay finished). *If t is a finished task, then for all inputs $\iota \in \mathcal{I}(t)$: if $t, \sigma \xRightarrow{\iota} t', \sigma'$, then t' is again a finished task. Moreover, we also have that $\mathcal{I}(t') = \mathcal{I}(t)$.*

Proof. Because t is finished, it is fixated and has a value by Definition 2. Therefore, by Proposition 14 from the appendix, we know that t is a *static* task. That is, it is a task which does not change the combinators it utilises. Using the properties of static tasks, as proved in the appendix, we know t stays static by Corollary 4, and its inputs will stay the same over interactions by Corollary 5.

5.3 Stuck tasks

A *stuck* task t is a task which does not fail, does not have a value, and does not accept user input. Examples of stuck tasks are $\boxdot 2 \blacktriangleright \lambda x.\ \lightning$, $\boxdot 2 \bowtie \lightning$, and $\lightning \bowtie \boxdot 2$. The first example is stuck because the right-hand side always fails, and thus the step can never be taken. For pairing, we have that both sides must fail before $\bowtie$ fails, and both sides must have a value before $\bowtie$ has a value. So, if one side fails and the other side has a value, then neither observation is true, and the task is stuck.

Definition 3 (Stuck task). *We call a fixated task* $t : \textsc{Task}\ \tau$ *in state* σ stuck *if and only if* $\neg\mathcal{F}(T), \mathcal{V}(t,\sigma) = \bot$, *and* $\mathcal{I}(t) = \varnothing$.

A stuck task will always remain stuck, because no more user interaction is possible, and by assumption it is already fully fixated.

Proposition 3 (Stuck tasks stay stuck). *If* t *is a stuck task, then it stays stuck. Or, more formally: for all fixated tasks* $t : \textsc{Task}\ \tau$ *and states* σ, *we have that if* $\mathcal{I}(t) = \varnothing$, *there is no input* ι *such that* $t, \sigma \overset{\iota}{\Longrightarrow} t', \sigma'$.

Similar to failing, we will consider stuck tasks as another type of termination, and we will say that all stuck tasks are semantically equivalent to each other.

5.4 Running tasks

A *running* task t is a task which does not fail, does not have a value, but still accepts user input. Because there is still user interaction possible, it may be the case that with the right input, it transitions to one of the previously described task conditions. The simplest example of a running task is the empty editor $\square^k \beta$, which becomes a finished (unsteady) task once it receives a valid input event. There also exist running tasks that only transition to another task condition for some inputs, or for no inputs at all. For example, the tasks $\square^k \textsc{Int} \blacktriangleright \lambda x.\ \lightning$, $\square^k \bowtie \lightning$, and $\lightning \bowtie \boxminus^k 2$ are all running tasks which will forever remain running; and the task $\square^k \textsc{Int} \blacktriangleright \lambda x.\ \textbf{if}\ x \leq 2\ \textbf{then}\ \boxdot x\ \textbf{else}\ \lightning$ will only transition to another task condition for some inputs, but not for others.

Definition 4 (Running task). *We call a fixated task* $t : \textsc{Task}\ \tau$ *in state* σ running *if and only if* $\neg\mathcal{F}(T), \mathcal{V}(t,\sigma) = \bot$, *and* $\mathcal{I}(t) \neq \varnothing$.

To determine the equivalence of two running tasks, we therefore need to look at all possible user interactions, and check that they affect the two tasks in the same way. To do this, we need a way to talk about sequences of input events, instead of just single input events. We give the following definition for this:

Definition 5 (Input sequences). *An input sequence* $I = \iota_1 \cdot \ldots \cdot \iota_j$ *is a finite sequence of input events. Given a task* $t_0 : \textsc{Task}\ \tau$ *and a state* σ_0, *we say that* I *is a* valid *input sequence for* t_0 *in* σ_0 *if and only if:*

$$t_0, \sigma_0 \overset{\iota_1}{\Longrightarrow} t_1, \sigma_1 \overset{\iota_2}{\Longrightarrow} \ldots \overset{\iota_j}{\Longrightarrow} t_j, \sigma_j$$

with $\iota_i \in \mathcal{I}(t_{i-1})$, *for all* $\iota \in \{1, \ldots, j\}$.

We will use the shorthand notation $t_0, \sigma_0 \overset{I}{\Longrightarrow}{}^* t_j, \sigma_j$ *to denote the above derivation. We also consider the empty input sequence, denoted by* ϵ, *to be a valid input sequence, and for all tasks* t *and states* σ *we have that:* $t, \sigma \overset{\epsilon}{\Longrightarrow}{}^* t, \sigma$.

We will make a distinction between running tasks that forever remain running, no matter what inputs you send to it; and running tasks for which there exists at least one input sequence which escapes, i.e. which transitions to another task condition. We will call the former class *looping* tasks, and the latter class *branching* tasks, formally:

Definition 6 (Looping and branching). *We call a running task* $t : \textsc{Task}\ \tau$ *in state* σ looping *if and only if for all valid input sequences* I *we have that* $t, \sigma \overset{I}{\Longrightarrow}^* t', \sigma'$ *and* t' *is again a running task. If there exists at least one valid input sequence for which* t' *is not a running task, then we call* t branching.

For branching tasks, it is possible to transition to either a finished or a stuck task condition. Take for example the task $\Box^k \textsc{Int} \blacktriangleright \lambda x.\ t$, which is a running task that transitions to t after a valid input event. So long as t is not failing, the step can be taken, and so t can be any non-failing task. It is also possible that for some input events, it transitions to one task condition, and for others, that it transitions to a different task condition. For example the task $\Box^k \textsc{Int} \blacktriangleright \lambda x.\ \textbf{if}\ x \leq 2\ \textbf{then}\ t\ \textbf{else}\ t'$ transitions to t for some inputs, and to t' for other inputs. However, a running task can never transition to a failing task.

Proposition 4 (Running tasks will not fail). *If* $t : \textsc{Task}\ \tau$ *is a running task, then for all states* σ *and for all valid input sequences* I*, we have that if* $t, \sigma \overset{I}{\Longrightarrow}^* t', \sigma'$ *then* $\neg \mathcal{F}(t')$.

Proof. By assumption t is running, and therefore itself not failing, and by the definition of the normalisation rules, t cannot make a transition to a failing task.

We say that two running tasks t_1 and t_2 in state σ are semantically equivalent if for all valid input sequences I we have that $t_1, \sigma \overset{I}{\Longrightarrow} t_1', \sigma' \iff t_2, \sigma \overset{I}{\Longrightarrow} t_2', \sigma'$, with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$, and $\mathcal{I}(t_1') = \mathcal{I}(t_2')$. That is, for all possible user interactions, t_1 and t_2 are not observably different. Because of Proposition 4, we do not need to check whether the tasks fail or not.

6 Task equivalence

Fig. 4 shows all possible task states and their transitions. In this diagram, a looping task is a task which never leaves the running state, that is, which always takes the transition back to the running state, no matter what input is given. A branching task is a running task for which there exists at least one input sequence which will transition to either stuck, steady, or unsteady. We will argue that this state diagram is complete.

First, we show that the five conditions discussed so far are mutually exclusive and exhaustive.

Proposition 5 (Condition exclusivity). *For all well typed fixated tasks* t *and states* σ*, we have that* t *is in one of the five conditions show in Fig. 4.*

Proof. By Proposition 12, we know that if a task is failing, it has no observable value and no possible inputs. This makes *Failing* the only condition a task can be in when $\mathcal{F}(t) = \text{True}$. Otherwise, one of the four remaining conditions (*Steady*, *Unsteady*, *Running*, and *Stuck*) should hold. These four conditions are mutual exclusive by definition.

Now that we know the conditions in Fig. 4 are the only conditions for any fixated task to be in, we show that the transitions in the state diagram are also complete.

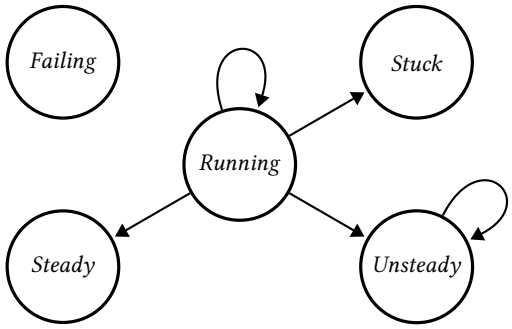


Fig. 4. Possible task conditions and their transitions, where a transition is caused by user interaction ($\overset{\iota}{\Longrightarrow}$) for some input ι.

Proposition 6 (Condition completeness). *The state diagram in Fig. 4 is complete. That is, for any two task conditions C and C', there is a transition from C to C' if and only if there exist two tasks $t \in C$ and $t' \in C'$ such that $t, \sigma \overset{\iota}{\Longrightarrow} t', \sigma'$ for some input $\iota \in \mathcal{I}(t)$ and states σ and σ'.*

Proof. For the conditions in the state diagram of Fig. 4 we have:

- Running tasks can *branch* to all other conditions, except for the failing condition, as stated in Proposition 4, or *loop* to itself.
- Failing tasks stay failing, by Proposition 1.
- Stuck tasks stay stuck, by Proposition 3.
- Finished tasks stay finished, by Proposition 2. Moreover, they keep their input space. So steady tasks stay steady, because their input space stays empty, and unsteady tasks stay unsteady, keeping their non-empty input space.

Hereby, we covered every possible transition in Fig. 4.

In the previous section, we mentioned that with the addition of internal editors (■) and read-only editors (⧈, ⊞) in this paper, it is no longer the case that a task t is failing if and only if it accepts no more user input, as is shown in Theorem 6.5 in the original TopHat paper [Steenvoorden et al., 2019]. Were this theorem still true, we could not have the stuck and steady states, because they contain tasks that do not fail and do not accept user input. We will therefore claim that, if we remove ■, ⧈, and ⊞ from the TopHat language as presented here, we no longer have the stuck and steady conditions.

Corollary 1 (Read-only tasks). *If we remove the editors ■, ⧈, and ⊞ from TopHat, we are left with only three possible task conditions after fixation: failing, running and finished unsteady.*

Based on the five task conditions, we give the following definition for the semantic equivalence of two tasks:

Definition 7 (Task equivalence). *Given two fixated tasks $t_1, t_2 : \textsc{Task}\ \beta$ of basic inner type, and given a state σ, we say that t_1 and t_2 are semantically equivalent if one of the following holds:*

1. *both tasks are* failing*;*
2. *both tasks are* finished *with $\mathcal{V}(t_1, \sigma) = \mathcal{V}(t_2, \sigma)$, and this value is* steady*;*
3. *both tasks are* finished *with $\mathcal{V}(t_1, \sigma) = \mathcal{V}(t_2, \sigma)$, and this value is* unsteady*, and for all valid input sequences I: $t_1, \sigma \overset{I}{\Longrightarrow} t_1', \sigma' \iff t_2, \sigma \overset{I}{\Longrightarrow} t_2', \sigma'$ with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$;*
4. *both tasks are* stuck*;*
5. *both tasks are* running*, and for all valid input sequences I: $t_1, \sigma \overset{I}{\Longrightarrow} t_1', \sigma' \iff t_2, \sigma \overset{I}{\Longrightarrow} t_2', \sigma'$ with $\mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma')$ and $\mathcal{I}(t_1') = \mathcal{I}(t_2')$.*

We write $t_1 \cong t_2$.

To determine a task's condition, we use the task observation functions $\mathcal{F}$, $\mathcal{V}$, and $\mathcal{I}$. For some task conditions, we also need to take the interactive setting of TopHat into account. Namely, if the set of inputs given by $\mathcal{I}$ is not empty, as is the case for the unsteady and running task states, we also need to check the task observations after every possible input sequence.

Because we also allow input sequences to be empty, we can generalise the above definition as follows:

Definition 8 (Generalised task equivalence). *Given two fixated tasks $t_1, t_2 : \textsc{Task}\ \beta$ of basic inner type, and given a state σ, we say that t_1 and t_2 are semantically equivalent ($t_1 \cong t_2$) if for all valid input sequences I we have that $t_1, \sigma \overset{I}{\Longrightarrow} t_1', \sigma' \iff t_2, \sigma \overset{I}{\Longrightarrow} t_2', \sigma'$ with $\mathcal{F}(t_1') = \mathcal{F}(t_2') \wedge \mathcal{V}(t_1', \sigma') = \mathcal{V}(t_2', \sigma') \wedge \mathcal{I}(t_1') = \mathcal{I}(t_2')$.*

7 Laws on tasks

Now that we have a definition of semantic equivalence for TopHat-programs, we can look at some interesting properties. This section gives some of the properties we expect to hold for equivalence. We will give some reasons of why we think a certain equality holds, or provide a counterexample to show the inequality of two expressions.

In TopHat, we can express functor laws over the type constructor Task as follows.

Proposition 7 (Laws on transform (•)).

$$\mathsf{id} \bullet t \cong t \qquad \textit{identity}$$
$$(e_1 \circ e_2) \bullet t \cong e_1 \bullet (e_2 \bullet t) \qquad \textit{composition}$$

Proof. Given any fixated task t and state σ, we have that

- $\mathcal{F}(\mathsf{id} \bullet t) = \mathcal{F}(t)$ by definition (see Fig. 1);
- $\mathcal{I}(\mathsf{id} \bullet t) = \mathcal{I}(t)$ by definition (see Fig. 1); and
- $\mathcal{V}(\mathsf{id} \bullet t, \sigma) = \mathcal{V}(t, \sigma)$, because if t has no value, then neither does $\mathsf{id} \bullet t$, and if $\mathcal{V}(t, \sigma) = v$ for $v \neq \bot$, then $\mathcal{V}(\mathsf{id} \bullet t, \sigma) = \mathsf{id}\ v \downarrow v$.

A similar argumentation can be made for the composition law. We therefore believe that the Task operator on types is a functor.

For the TopHat pairing construct (⋈), we can formulate laws which are inspired by an alternative formulation of the laws of Haskell's Applicative type class.

Proposition 8 (Laws on pair (⋈)).

$$\blacksquare\{\} \bowtie t \not\cong t \qquad \textit{left identity}$$
$$t \bowtie \blacksquare\{\} \not\cong t \qquad \textit{right identity}$$
$$\text{assoc} \bullet (t_1 \bowtie (t_2 \bowtie t_3)) \cong (t_1 \bowtie t_2) \bowtie t_3 \qquad \textit{associativity}$$
$$\{e_1\ \_, e_2\ \_\} \bullet (t_1 \bowtie t_2) \cong (e_1 \bullet t_1) \bowtie (e_2 \bullet t_2) \qquad \textit{naturality}$$

Proof. If we take $t = \lightning$ for the left and right identity laws, then in both cases we have that the right-hand side is a failing task, but the left-hand side is not. Pairing requires that both sides fail before it fails itself. Therefore, we will say that the Task constructor on types does not form a monoidal functor.

However, the other two laws hold using a similar argumentation as in the proof of Proposition 7. For associativity, we need the function $\text{assoc} := \lambda\{a, \{b, c\}\}.\{\{a, b\}, c\}$ to make sure that both sides have the same type. Also, the naturality law states that first pairing two tasks and then mapping two functions is the same as first mapping the functions separately and then pairing them.

For choosing (◆) we formulate the following properties, which are loosely based on the laws of the Alternative type class in Haskell.

Proposition 9 (Laws on choose (◆)).

$$t_1 \blacklozenge (t_2 \blacklozenge t_3) \cong (t_1 \blacklozenge t_2) \blacklozenge t_3 \qquad \textit{associativity}$$
$$\blacksquare e \blacklozenge t \cong \blacksquare e \qquad \textit{left catch}$$
$$t \blacklozenge \blacksquare e \not\cong \blacksquare e \qquad \textit{right catch}$$
$$e \bullet (t_1 \blacklozenge t_2) \cong (e \bullet t_1) \blacklozenge (e \bullet t_2) \qquad \textit{distributivity}$$
$$t_0 \bowtie (t_1 \blacklozenge t_2) \not\cong (t_0 \blacklozenge t_1) \bowtie (t_0 \blacklozenge t_2) \qquad \textit{left pair}$$
$$(t_1 \blacklozenge t_2) \bowtie t_0 \not\cong (t_1 \blacklozenge t_0) \bowtie (t_2 \blacklozenge t_0) \qquad \textit{right pair}$$
$$t_0 \blacktriangleright \lambda x.(t_1 \blacklozenge t_2) \cong (t_0 \blacktriangleright \lambda x.t_1) \blacklozenge (t_0 \blacktriangleright \lambda x.t_2) \qquad \textit{left step}$$
$$(t_1 \blacklozenge t_2) \blacktriangleright e \cong (t_1 \blacktriangleright e) \blacklozenge (t_2 \blacktriangleright e) \qquad \textit{right step}$$

Proof. Associativity, left catch, and a number of distributivity laws can be proved in a similar way as in Proposition 7. Catch only holds for the left side, because the choice combinator is left-biased. The first distributivity law shows that mapping a function over choice should be equivalent to mapping it separately over its component subtasks. The last two distributivity laws for step (▶) show that stepping to or from a choice is equivalent to choosing between steps.

Left and right distributivity for pairing (⋈) do not hold. Suppose $\mathcal{V}(t_0, \sigma) = \bot$, $\mathcal{V}(t_1, \sigma) = v_1$ and $\mathcal{V}(t_2, \sigma) = v_2$. Then in both cases, we have that the left-hand side of the inequality has no value, because pairing requires that both sides have a value

before it has a value itself. However, the right-hand side does have a value in both cases, namely $\{v_1, v_2\}$, because the choice combinator normalises t_0 away, and we are left with $t_1 \bowtie t_2$ in both cases.

When considering the failing task ($\lightning$), we can formulate the following laws:

Proposition 10 (Laws on fail ($\lightning$)).

$$\begin{aligned}
\lightning \blacklozenge t &\cong t && \textit{left identity}\\
t \blacklozenge \lightning &\cong t && \textit{right identity}\\
e \bullet \lightning &\cong \lightning && \textit{annihilation}\\
\lightning \bowtie t &\ncong \lightning && \textit{left zero}\\
t \bowtie \lightning &\ncong \lightning && \textit{right zero}\\
\lightning \blacktriangleright e &\cong \lightning && \textit{left annihilation}\\
t \blacktriangleright \lambda x.\lightning &\ncong \lightning && \textit{right annihilation}
\end{aligned}$$

Proof. The failure task ($\lightning$) acts as the left and right identity for the choice combinator ($\blacklozenge$). That means that$\lightning$ can be cancelled out from the left- and right-hand side. For pairing, left and right pair annihilation do not hold, because $\mathcal{F}(t_1 \bowtie t_2) = \mathcal{F}(t_1) \wedge \mathcal{F}(t_2)$. Pairing therefore requires that both the left-hand and right-hand side fail before their pairing fails, and thus if only one side fails, it cannot be equivalent to the failing task $\lightning$.

We defined all failing tasks to be semantically equivalent, thus annihilation for $\bullet$, and left step annihilation for $\blacktriangleright$ trivially hold. Right step annihilation does not however. Suppose that t is not a failing task, then we also have that $t \blacktriangleright \lambda x.\lightning$ is not a failing task, because $\mathcal{F}(t \blacktriangleright \lambda x.\lightning) = \mathcal{F}(t)$ by definition (see Fig. 1). Neither can we normalise $t \blacktriangleright \lambda x.\ \lightning$, because the right-hand side fails . So if t does not fail, then neither does $t \blacktriangleright \lambda x.\lightning$, and we cannot conclude that a non-failing task is semantically equivalent to the failing task $\lightning$.

Steps ($\blacktriangleright$) in TopHat have a monadic flavour to them, and we can wonder whether the step combinator is a bind operation. So, if we consider Task to be the monadic constructor, $\blacksquare$ the return function, and $\blacktriangleright$ the bind function, then we can express the three monadic laws in TopHat as follows.

Proposition 11 (Laws on step ($\blacktriangleright$)).

$$\begin{aligned}
\blacksquare x \blacktriangleright g &\ncong g\ x && \textit{left identity}\\
t \blacktriangleright (\lambda y.\blacksquare y) &\ncong t && \textit{right identity}\\
(t \blacktriangleright g) \blacktriangleright h &\cong t \blacktriangleright (\lambda y.g\ y \blacktriangleright h) && \textit{associativity}
\end{aligned}$$

Proof. With our definition of program equivalence, we can show that neither the left nor the right identity laws hold. For the left identity, take for example $g := \lambda x.\lightning$, then we have that $g\ x$ is a failing task, because $g\ x = (\lambda x.\lightning)\ x \downarrow \lightning$. However, the left-hand side $\blacksquare x \blacktriangleright g = \blacksquare x \blacktriangleright \lambda x.\lightning$ is a stuck task, because it does not fail, and the step can never be taken.

For the right identity law we can take for example $t := \square^k \textsc{Int}$. If we send the input event $k!42$ to both sides, then the left-hand side normalises to $\blacksquare 42$, while the right-hand side becomes $\boxminus^v 42$. As we have already seen, these two tasks are not semantically equivalent, because the value of $\blacksquare 42$ is steady and cannot be changed any more, while the value of $\boxminus^v 42$ is unsteady and can be updated through user input.

However, the associativity law holds. Given any fixated task t and state σ, for $\mathcal{F}$ and $\mathcal{I}$ we have that:

$$\begin{aligned} \mathcal{F}\big((t \blacktriangleright g) \blacktriangleright h\big) = \mathcal{F}(t \blacktriangleright g) = \mathcal{F}(t) \quad &= \mathcal{F}\big(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)\big) \\ \mathcal{I}\big((t \blacktriangleright g) \blacktriangleright h\big) = \mathcal{I}(t \blacktriangleright g) = \mathcal{I}(t) \quad &= \mathcal{I}\big(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)\big) \end{aligned}$$

by definition (see Fig. 1). Furthermore, supposing that $\mathcal{V}(t, \sigma) = v$, and $g\ v \downarrow t'$ with $\neg\mathcal{F}(t')$, then both sides normalise to $t' \blacktriangleright h$ in some state σ'. On the other hand, if either $\mathcal{V}(t, \sigma) = \bot$ or $\mathcal{F}(t')$, then the step cannot be taken, and we have that $\mathcal{V}\big((t \blacktriangleright g) \blacktriangleright h\big) = \mathcal{V}\big(t \blacktriangleright (\lambda y.g\ y \blacktriangleright h)\big) = \bot$.

8 Conclusions

In this paper, we took TopHat as a foundation to reasoning about task-oriented programs. We gave a definition for the semantic equivalence of two TopHat-programs. We split this definition into two classes: expression equivalence, and task equivalence. For task equivalence, we showed that a task can be in either one of five conditions after fixation, and for every two tasks in the same condition, we defined what it means for them to be semantically equivalent. We also noted that for task conditions that still accept user input, it is important to take the interactive setting of TopHat into account, and compare how both tasks react to user input. We presented a set of transformation laws that we hold true for TopHat-programs. Especially, we showed that the Task type constructor in TopHat is not a monad but it is a functor. Using these laws, developers and compilers can do safe transformations on task-oriented programs, while being sure the semantic meaning of the program stays the same.

Future work

In future work, we would like to implement the transformation laws presented in Section 7 in TopHat's symbolic execution engine [Naus et al., 2019]. This could speed up symbolic evaluation and give faster results on next step hints for end users. Also, we would like to investigate the implications of our findings on the iTasks system, which could benefit from implementing the transformation laws in the same way.

Acknowledgements

The authors like to thank Herman Geuvers for supervising the bachelor research on which this article is based.

Bibliography

Peter Achten, Pieter W. M. Koopman, and Rinus Plasmeijer. An introduction to task oriented programming. In Viktória Zsók, Zoltán Horváth, and Lehel Csató, editors, *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, volume 8606 of *Lecture Notes in Computer Science*, pages 187–245. Springer, 2013. ISBN 978-3-319-15939-3.

Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

Pieter W. M. Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for itasks. In Sven-Bodo Scholz and Olaf Chitil, editors, *Implementation and Application of Functional Languages - 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers*, volume 5836 of *Lecture Notes in Computer Science*, pages 212–232. Springer, 2008. ISBN 978-3-642-24451-3.

Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. Incidone: A task-oriented incident coordination tool. In *Proceedings of ISCRAM*, 2012.

Nico Naus and Tim Steenvoorden. Generating next step hints for task oriented programs using symbolic execution. In Aleksander Byrski and John Hughes, editors, *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers*, volume 12222 of *Lecture Notes in Computer Science*, pages 47–68. Springer, 2020. ISBN 978-3-030-57760-5.

Nico Naus, Tim Steenvoorden, and Markus Klinik. A symbolic execution semantics for tophat. In Jurriën Stutterheim and Wei-Ngan Chin, editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25-27, 2019*, pages 1:1–1:11. ACM, 2019. ISBN 978-1-4503-7562-7.

Andrew M. Pitts. Operational semantics and program equivalence. In Gilles Barthe, Peter Dybjer, Luís Pinto, and Jo ao Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 378–412. Springer, 2000. ISBN 3-540-44044-5.

Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. Clean language report version 2.1. Technical report, 2002.

Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pages 195–206, 2012.

Peter Sewell. Semantics of programming languages, computer science tripos, part 1b, 2017. Accessed 29-November-2019.

Tim Steenvoorden. *TopHat: Task-oriented programming with style.* PhD thesis, 2022.

Tim Steenvoorden, Nico Naus, and Markus Klinik. Tophat: A formal foundation for task-oriented programming. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 17:1–17:13, 2019.

Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. Maintaining separation of concerns through task oriented software development. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*, pages 19–38, 2017.

A Interplay

In this section we prove some properties of our observations functions introduced in Section 2.4 and show that play well together.

A.1 Failing

First, we prove that our failing observation $\mathcal{F}$ indeed observes a task that does *not* have a value, and does *not* have possible inputs. That is, the value observation $\mathcal{V}$ is undefined, and the inputs observation $\mathcal{I}$ gives the empty set.

Proposition 12 (Failing tasks cannot have interaction). *For all well typed normalised tasks n and state σ, we have that if $\mathcal{F}(n) = \mathsf{True}$, then $\mathcal{V}(n, \sigma) = \bot$ and $\mathcal{I}(n) = \emptyset$.*

Proof. This proposition is mechanically proved by induction over n in Idris, see `Task.Proofs.Failing.failingMeansNoInteraction`.

Note that the converse statement, is not true: a task which does not have a value and does not have possible inputs is not necessarily failing. The task ⧈42 ⋈ ↯ does not have a value, because the right hand side of the pair combinator does not have a value; and does not have possible inputs, because the left hand side is a read-only editor. However, $\mathcal{F}$(⧈42 ⋈ ↯) = False, because the failing observation needs both sides of the pair to be failing, but the left hand side clearly is not. This deviates from the properties form earlier TopHat versions [Steenvoorden et al., 2019, Naus et al., 2019].

A.2 Inputs

Next, we validate our inputs observation $\mathcal{I}$. It calculates all possible inputs for a given task. We need to show that the set of possible inputs it produces is both sound and complete with respect to the handling ($\longrightarrow$) and interaction ($\Longrightarrow$) semantics. By sound we mean that all inputs in the set of possible inputs can actually be handled by the semantics, and by complete we mean that the set of possible inputs contains all inputs that can be handled by the semantics. Proposition 13 expresses exactly this property for the handling semantics.

Proposition 13 (Inputs is sound and complete (wrt handle)). *For all well typed normalised tasks n, states σ, and inputs ι, we have that $i \in \mathcal{I}(n)$ if and only if there exists task t', state σ', and dirty set δ' such that $n, \sigma \xrightarrow{\iota} t', \sigma', \delta'$.*

Proof. This theorem is mechanically proved by induction on n in Idris, see `Task.Proofs.Inputs.inputIsHandled` and `Task.Proofs.Inputs.handleIsPossible`.

Because of the structure of the interaction semantics ($\Longrightarrow$), Proposition 13 directly gives us soundness and completeness of inputs with respect to interact.

Corollary 2 (Inputs is sound and complete (wrt normalise)). *For all well typed normalised tasks n, states σ, and inputs ι, we have that $i \in \mathcal{I}(n)$ if and only if there exists normalised task n', and state σ' such that $n, \sigma \overset{\iota}{\Longrightarrow} n', \sigma'$.*

By combining Proposition 12 and Corollary 2, we know that failing tasks cannot handle input.

Corollary 3 (Failing tasks cannot handle input). *For all well typed normalised tasks n and states σ, we have that if $\mathcal{F}(n)$, there is no input ι such that $n, \sigma \overset{\iota}{\longrightarrow} t', \sigma', \delta'$.*

A.3 Value

Last, we prove that, when we can observe a value from a task, this task is a *static task*. By static tasks, we mean that the shape of the tree of task nodes and leaves are static and cannot be dynamically altered at runtime.

s ::=		Static tasks
|	$\boxminus^k b$	valued editor
|	$\boxplus^k h$	shared editor
|	$\boxdot^k b$	valued read-only editor
|	$\boxtimes^k h$	shared read-only editor
|	$v_1 \bullet s_2$	transform
|	$s_1 \bowtie s_2$	pair
|	$\blacksquare v$	lift

Fig. 5. Static tasks

Definition 9 (Static task). *We call tasks* static *when they consist of a valued editor, pairs of static tasks, or transforms of static tasks. That is, they confirm to the grammar presented in Fig. 5.*

Note that static tasks are a subset of normalised tasks.

Proposition 14 (Valued tasks are static). *For all well typed normalised tasks n and states σ, we have that if $\mathcal{V}(n, \sigma) = v$, n is static.*

Proof. We have that $\mathcal{V}(n, \sigma) = v : \beta$. When taking the definition of $\mathcal{V}$ into account (as given in Fig. 2), we see that n can only be in one of four shapes:

- a valued editor d (that is $d \neq \square\beta$);
- a pair $n_1 \bowtie n_2$, for which $\mathcal{V}(n_1, \sigma) = v_1$ and $\mathcal{V}(n_2, \sigma) = v_2$;
- a transform $e_1 \bullet n_2$, for which $\mathcal{V}(n_2, \sigma) = v_2$;
- a lift $\blacksquare v$.

Especially, step ($n_1 \blacktriangleright e_2$) and fail ($\lightning$) are ruled out, because they have no value. Choice ($n_1 \blacklozenge n_2$) is ruled out, because by rules N-Choose[Left,Right,None], the only way that this combinator can survive normalisation is for both n_1 and n_2 to not have a value, which should be the case here. Now, by induction, we have that n is static and meets the grammar in Fig. 5.

See also `Task.Proofs.Static.valued_means_static` in Idris.

As static tasks do not contain steps ($n_1 \blacktriangleright e_2$) or choices ($n_1 \blacklozenge n_2$), their form cannot be altered at runtime. That is, there is no way for end users to create new (sub)tasks.

Corollary 4 (Static tasks stay static). *For all well typed normalised tasks n and states σ, we have that if n is static, its shape cannot be altered at runtime.*

In particular, we cannot create or delete editors at runtime. So the input events a static task can process will always be the same.

Corollary 5 (Static tasks keep input observation). *For all well typed normalised tasks n and states σ, we have that if n is static, and $n, \sigma \xRightarrow{\iota} n', \sigma'$ for some $\iota \in \mathcal{I}(n)$ then $\mathcal{I}(t) = \mathcal{I}(t')$.*

Algorithm Design with the Selection Monad

Johannes Hartmann and Jeremy Gibbons

Department of Computer Science, University of Oxford, UK
firstname.lastname@cs.ox.ac.uk

Abstract. The selection monad has proven useful for modelling exhaustive search algorithms. It is well studied in the area of game theory as an elegant way of expressing algorithms that calculate optimal plays for sequential games with perfect information; composition of moves is modeled as a 'product' of selection functions. This paper aims to expand the application of the selection monad to other classes of algorithms. The structure used to describe exhaustive search problems can easily be applied to greedy algorithms; with some changes to the product function, the behaviour of the selection monad can be changed from an exhaustive search behaviour to a greedy one. This enables an algorithm design framework in which the behaviour of the algorithm can be exchanged modularly by using different product functions.

Keywords: Selection monad · Functional programming · Algorithm design · Greedy algorithms · Monads.

1 Introduction

In 2010 Martín Escardó and Paulo Oliva first describe the selection monad in their paper *Selection Functions, Bar Recursion, and Backward Induction* [2], where they explain bar recursion in terms of sequential games. They use the selection monad to model bar recursion and further show how sequential games can be solved using the selection monad. In subsequent work [3], they relate the selection monad to several different applications, such as Double-Negation Shift in the field of logic and proof theory and the Tychonoff Theorem in the field of topology.

Selecting a candidate for a solution out of a collection of options, based on some property function that tells us how good a candidate is, is a recurring pattern in computer science. We can use selection functions of type `(A -> R) -> A` to model this decision process. For example, when playing a sequential game, making a move means deciding for a particular move out of all possible moves. We can imagine a property function that somehow is able to compute how good each move is, and we then select the best one. The general idea when using selection functions is to model a problem in terms of a list of selection functions `[(A -> R) -> A]`. These selection functions then can be combined into a single selection function `([A] -> R) -> [A]` with the help of the product for selection functions. Further, it turns out that these selection functions form a monad, and

that the product for selection functions is given by the monadic `sequence` function. The product for selection functions hereby models an exhaustive search algorithm, that trying out every possible solution to the problem and then selecting the overall best one. This can be applied to find optimal strategies for sequential games, where each individual selection function is modeling the choice of which move to play next, and the sequential composition of all this individual choices computes an optimal strategy for this game.

In this paper we describe alternative product implementations for selection functions, providing different computational behaviour. With these different product implementations we are able to model both greedy algorithms and limited-lookahead search algorithms. This enables us to extend the application of selection functions to a wider range of problems.

In Section 2 we introduce selection functions in greater detail and provide an intuition for and examples of the product for selection functions. Then in Section 3 we provide an alternative product implementation that models a greedy behaviour, and in Section 4 we will have a look at some examples of greedy algorithms that are modeled with the greedy product for selection functions. In Section 6 we introduce another variant of the product of selection function that limits the number of steps it looks into the future. We conclude this paper and discuss potential future work in Section 7.

2 Selection functions

Selection functions summarise the process of selecting an individual object out of a collection of objects. Selecting something means making a choice, deciding in favour of one object over all the other objects of the collection we are choosing from. So in order to make a choice we need to be able to judge these objects, and based on that judgement, decide on a particular object out of a set of all possibilities.

As an example, let's say we want to buy a car. Therefore we have a collection of all available cars on the market, and we want to buy the best one. "Best" in this case depends heavily on our personal perspective on cars, so let's define "best" as "fastest". Then our personal selection process would be: look up top speeds of every car in our collection of all available cars, and select the one with the highest top speed. We can model this in the functional programming language Haskell as follows:

```
myChoice :: Car
myChoice = maxWith allCars topSpeed

maxWith :: Ord b => [a] -> (a -> b) -> a
topSpeed :: Car -> Int
allCars :: [Car]
```

Here, `maxWith` is the function that selects an element from the given list to maximise the given property function. Of course, top speed is not the only property

one might take into account when buying a car; `maxWith` abstracts from the property used for judging.

Given elements of type `A` and properties of type `R`, a selection function takes a property function `p :: A -> R` for judging elements, and selects some element of type `A` that is optimal according to the given property function. A good example of such a selection function is the `maxWith` from above, partially applied to a collection of objects, or the complementary `minWith` function. Escardó and Oliva studied these selection functions and connected them to different research areas, including game theory and proof theory [3].

2.1 Pairs of selection functions

As a next step, we want to look at how to combine two selection functions into a new bigger selection function. We will call this *pairing* of selection functions. To be consistent with the notation of previous literature we first define a type synonym for selection functions:

```
type J r a = (a -> r) -> a
```

One way of thinking of a function of type `J R A` is as embodying a nonempty collection of `A` elements: when told how to judge individual elements by `R`-values, the function can deliver an optimal element by that measure.

Now we define the `pair` operator combining two selection functions to make a new one [3]:

```
pair :: J r a -> J r b -> J r (a,b)
pair f g p = (a,b)
    where
        a = f (\x -> p(x,g(\y -> p(x,y))))
        b = g (\y -> p(a,y))
```

This new selection function selects `(A,B)` pairs, and therefore awaits a property function `p :: (A,B) -> R` that judges pairs. Suppose we are given such property function; then an `(A,B)` pair needs to be produced as output. To do so, we need to extract an `A` out of the first selection function `f`; we do so by constructing a new property function that judges `A`s. However, we only have something that judges `(A,B)` pairs. The trick is, for every `x :: A` we would like to judge, we extract a corresponding `y :: B` out of the second selection function and the pair `(x,y)` with the given property function `p`. Once we have found our optimal `a`, we can use it in the same way to select a corresponding `b` out of the selection function `g` and returning both as a pair.

So intuitively, this `pair` operator for selection functions combines two given selection functions in a way that the resulting selection function produces a pair that is optimal for a property function that judges these pairs. The two elements of this pair are extracted from the given selection functions in a way that they are always judged in the context of a full pair. From the perspective of selection functions as embodying a collection of objects, the pair for two selection functions embodies the cartesian product of the two collections.

2.2 Password example

Let's consider the following example, where we use the product of selection functions to crack a secret password. This secret password consists of two different tokens, a secret number between 1–9 and a secret character between a–z:

```
type Password = (Int, Char)
```

We are also given a property function that tells us if a given password is correct. We will treat this property function as a black box and not depend on its implementation details:

```
p :: Password -> Bool
p (a,b) = a == 7 && b == 'p'
```

In order to crack this password we will now define two individual selection functions for the two different parts of the password:

```
selectInt :: Ord r => J r Int
selectInt p = maxWith p [1..9]

selectChar :: Ord r => J r Char
selectChar p = maxWith p ['a'..'z']
```

Note here that both selection functions require a property function `p` to judge `Int`s or `Char`s individually, which we don't have at this stage. However, building the pair with the above defined `pair` function will result in a combined selection function that we can apply our property function to, and it indeed calculates the correct solution:

```
> pair selectInt selectChar p
---> (7,'p')
```

(Recall that Haskell defines `False < True`, so this expression returns a pair that satisfies `p`, provided that any of the given pairs does so.)

Because the only way to judge each individual component is by the given property function that judges complete password pairs, the `pair` function needs to create new property functions for its selection functions that are able to judge individual objects in a broader context.

This example fits nicely with the intuition that each selection function already embodies a set of objects, and that the `pair` function builds the cartesian product of these two sets and judges each pair individually and returns an optimal pair.

2.3 Iterated product of selection functions

The next logical step is to expand this from pairs to n-ary products. Unfortunately the standard Haskell type system is too restrictive to allow arbitrarily typed products of selection functions, and therefore the closest we can get is combining a list of selection functions for a common element type into a single selection function [3]:

```
product :: [J r a] -> J r [a]
product []       _ = []
product (e : es) p = a : as
   where a  = e(\x -> p(x : product es (p . (x:))))
         as = product es (p . (a:))
```

This particular implementation of `product` behaves similarly to the previous `pair` function. Given a property function `p :: [A] -> R` that judges lists, this function iterates through the list of selection functions in order to extract a concrete object out of each of them and therefore building a property function that considers both all previous decisions and all future decisions.

Note that for each recursive call a new property function is created that prepends the current choice of object via `(p . (a:))`. Further, to extract an object out of the current selection function `e`, a property function is created that judges each element of the underlying set in context of all possible future choices by recursively calling `product` within the property function.

2.4 Dependently typed version

Note that in this implementation we are not using the more restrictive type to our advantage. In contrast to the tuples of the above defined `pair` operator, the choice of lists comes with two drawbacks. First, the elements of the list must all be of the same type and second, lists in Haskell can be of arbitrary length while tuples are always of a fixed size.

The above presented `product` implementation does not take advantage of the first drawback, and assumes the second restriction. It constructs property functions that always judge elements in context with previous choices and potential future choices, making the restriction that every element needs to be of the same type irrelevant and assumes that the given property function is able to judge lists of the exact same length as the given list of selection functions.

In a dependently typed language we would be able to type this particular `product` implementation with a more expressive type that allows heterogeneous lists that can contain elements of any type and also ensures that each list we are dealing with has the correct length. An example implementation with this more expressive type in the programming language Idris is not difficult to construct [6].

We will later use this less expressive type to our advantage when we have a look at greedy algorithms, where we omit lookahead into the future.

2.5 Extended password example

The previous password example can now easily be extended to make use of the `product` function. We now want to crack a password of type `String`. We know only that it contains characters from a–z and that its length is 8:

```
type Password = String
```

and we are given a property function as a black box again:

```
p :: String -> Bool
p x@[_,_,_,_,_,_,_,_] = x == "password"
p _                   = undefined
```

Note that our property function is only defined for `String`s of length 8.

We can now utilise the previous `selectChar` and construct a list that contains this selection function exactly 8 times, once for each character of the password:

```
es :: Ord r => [J r Char]
es = replicate 8 selectChar
```

Utilising the previous `product` function will calculate the correct solution:

```
> product es p
---> "password"
```

This example again fits nicely the previously described intuition, where each component of the solution can only be judged in context with the previous selections as well as all possible future selections. This is underlined by the fact that by design, the property function is only able to judge solutions of the correct length, while being undefined for inputs of different lengths. The absence of any exceptions strengthens our intuition, that individual objects are always judged in full context.

2.6 Selection functions form a monad

Further, Escardó and Oliva show [3] that the type of selection functions forms a strong monad. This strengthens the intuition that a selection function embodies a collection of elements.

Selection functions with a fixed property type can be made instance of the monad class with this bind function:

```
(>>=) :: J r a -> (a -> J r b) -> J r b
(>>=) e f = \p -> f (e (p . flip f p)) p
```

and this `return` function:

```
return :: a -> J r a
return a = \_ -> a
```

Intuitively, the monad instance takes care that the underlying set elements are always judged in context. For `e >>= f`, we first want to extract an object of type `A` out of the given `e`, and then use `f` to transform it into a `J R B`. To extract an object of type `A` out of `e` we need to build a new property function that judges each element by how well it produces objects of type `b` by incorporating `f` into the new property function.

With `J R A` being a monad, we can utilise the prelude's monad functions to our advantage. Escardó and Oliva claim [3] that their `product` implementation is equivalent to the monadic `sequence` function from the Haskell prelude. A proof of this is claim is given in the appendix.

2.7 History-dependent product of selection functions

As an extension to the previous product for selection functions, Escardó and Oliva introduced [3] a history-dependent version of the `product` function. It keeps track of previous decisions and allows therefore for more dynamic selection functions, that can base the set from which they select on previous decisions:

```
hProduct :: [a] -> [[a] -> J r a] -> J r [a]
hProduct h []       p = []
hProduct h (e : es) p = a : as
   where a  = (e h) (\x -> p(x : hProduct (h++[x]) es (p . (x:))))
         as = hProduct (h++[a]) es (p . (a:))
```

This history-dependent version proves useful when modeling sequential games, in which the moves available at a given stage of the game typically depend on the previously played moves. The application of the selection monad to sequential games is already widely studied [1–4, 7, 8].

2.8 Efficiency drawbacks of this implementation

The `pair`, `product`, and `hProduct` implementations introduced above combine individual selection functions into a single product selection function. In order to extract the necessary elements out of the individual selection functions, the required property functions are created so as to always consider previous choices as well as potential future choices. This models an exhaustive search behaviour, where every possible combination of objects is judged by a given property function and the overall favorable combination of objects is then chosen. This results in an exponential runtime in respect to the number of selection functions to be combined. This exponential runtime makes the application of the selection monad infeasible for computing solutions for many problems.

Several methods to cope with this runtime have been explored in the past. There are several approaches to avoid unnecessary computations. Firstly, the individual selection functions can be neatly constructed such that they prune the inspection of their elements in unnecessary cases. Concretely, if the cost value `R` by which each object is judged has an upper or lower bound, the search can be stopped once a solution reaches one of these bounds [5]. Additionally, one should try to make as little as possible use of the property function, as it always constructs all possible future objects to judge the current element in context. One concrete example for this can be to avoid the use of the property function completely if there is only one element to choose from.

Secondly, the different `product` variants themselves leave room for improvement. For example, for minimax algorithms [3], a different `hProduct` function could implement alpha-beta pruning, which is used in game theory to reduce the search space when calculating perfect plays for sequential games.

One final flaw with the current implementations of `product` and `hProduct` variants is that they perform a lot of redundant calculations. In particular, each starts with the first selection function in the list and calculates all possible

future choices for each internal element. Based on this information it chooses the object that leads to the best possible future outcome. However, continuing the recursion, it forgets that it already explored all possibilities from there and starts the computation all over again, leading to a lot of redundant computations.

For the remainder of this paper, we will focus on different implementations for the iterated product of selection functions, which will have different computational behaviour. This will enable us to efficiently express greedy algorithms and other algorithm classes as products of selection function, and further enable us to abstract the behaviour of the different algorithms into the `product` functions.

3 Greedy algorithms

The product of selection functions as we have seen so far models an exhaustive search algorithm, exploring all possible combination of objects in the underlying sets of the selection functions. In this section, we explore a different product for selection functions. By using the less expressive type of the `product` function to our advantage, we are able to modify it in a way that allows us to model a greedy algorithm behaviour as a product of selection functions.

We previously identified that by choosing lists as container type for the selection functions, we restrict all the selection functions in the container to be of the same type. Further, the Haskell list type does not impose any length restrictions on the input list of property function. In order to model a greedy behaviour we can use this more general type to our advantage.

Once we require the property function to be defined for partial solutions as well, we can define a new `greedyProduct` function that judges the underlying objects of the selection functions only in the context of previous choices without caring about potential future choices:

```
greedyProduct :: [J r a] -> J r [a]
greedyProduct []       p = []
greedyProduct (e : es) p = a : as
   where a  = e (p . ([]:))
         as = greedyProduct es (p . (a:))
```

So iterating through the list of selection functions, we extract a value out of each selection function by building a new property function of type `A -> R` by converting each `A` into a singleton list and then applying our property function to it. In the recursive call we build a new property function that keeps track of the previous choices. Note that this definition differs from the `product` definition by omitting the recursive call inside the new property function that calculates all possible future choices. Further, we can also define the corresponding history-dependent version:

```
greedyHProduct :: [a] -> [[a] -> J r a] -> J r [a]
greedyHProduct h []       p = []
greedyHProduct h (e : es) p = a : as
```

```
  where a  = (e h) (p . ([]:))
        as = greedyHProduct (h++[a]) es (p . (a:))
```

Judging the individual objects outside of a global context locally captures the essence of greedy algorithms. In general, algorithms are called "greedy" when they perform only a local optimisation at each step. We can now utilise these new products for selection functions to implement greedy algorithms. The general idea is to define each individual local step of the greedy algorithm as a selection function; then the greedy algorithm arises from building the greedy product of these selection functions. The greedy behaviour is thereby abstracted away into the product function, enabling us to describe greedy algorithms in a similar way form to exhaustive search algorithms.

4 Examples of greedy algorithms

In this section we look at some examples applying the new `greedyProduct` variants to solve problems in a greedy way.

4.1 Password example

To continue the password example from above in a greedy way, we require a more sophisticated property function, that is able to judge partial solutions. So we are now given the following property function, that returns the number of correctly guessed characters of the password, instead of simply a boolean:

```
p :: String -> Int
p = length . filter id . zipWith (==)
```

We can reuse our previously defined selection functions:

```
es :: Ord r => [J r Char]
es = replicate 8 selectChar
```

And with the new property function we can utilise the `greedyProduct` which will calculate the correct solution:

```
> greedyProduct es p
--> "password"
```

With the new property function, we don't need to know all future possibilities when determining the correct character at the next position of the password. We just need to be aware of our previous choices and can then decide greedily for the character that maximises the property function.

4.2 Prim's algorithm

A textbook example for a greedy algorithm is Prim's algorithm for finding a minimal spanning tree for a graph [9]. Given a weighted, undirected graph, a minimal spanning tree a subset of edges of the given graph that forms a tree and is minimal in its weight. The general idea of Prim's algorithm is to grow a tree by repeatedly greedily selecting the lightest edge extending the tree (that is, the lightest edge connected to the tree and not creating a cycle).

To implement Prim's algorithm in terms of selection functions, we first define a graph as a list of edges, and an edge as a triple of integers with the first two elements being the two nodes of the edge and the third element the weight of the edge:

```
type Node = Int
type Weight = Int
type Edge   = (Node, Node, Weight)
type Graph  = [Edge]
```

Further, we then define a helper function that calculates if a node is part of a given graph:

```
nodeOf :: Graph -> Node -> Bool
nodeOf [] _           = False
nodeOf ((x,y,_):xs) n = n == x || n == y || nodeOf xs n
```

We can then define a function calculating the total weight of a given collection of edges:

```
p :: [Edge] -> Weight
p [] = 0
p ((_,_,x):xs) = x + p xs
```

Further, we then need a function that calculates all edges that that can be added to the tree such that it stays a tree. An edge of the graph is a candidate exactly if one of its ends is already in the tree, and the other is not. In the initial case, where there is no tree existing, all edges are potential candidates.

```
getCandidates :: Graph -> [Edge] -> [Edge]
getCandidates g [] = g
getCandidates g h  = filter f g
  where
    f (x,y,_) = nodeOf g x && not (nodeOf g y) ||
                not (nodeOf g x) && nodeOf g y
```

To build our list of selection functions for a given graph, we want to select the lightest edge of all possible edges according to our property function. We also need exactly one fewer selection functions than there are nodes in the original graph.

```
selectEdge :: Ord r => Graph -> [[Edge] -> J r Edge]
selectEdge g = replicate (length (nodes g) - 1) f
 where
  f x p = minWith p (getCandidates g x)
```

Now considering the following example graph:

```
exampleGraph :: Graph
exampleGraph = [(1,2,1),(2,3,5),(2,4,9),(4,5,20),(3,5,1)]
```

we can form the product of our selection functions with the `greedyProduct` function. Applying this to our property function we get a minimal spanning tree as a result:

```
greedyHProduct [] (selectEdge exampleGraph) p
---> [(1,2,1),(2,3,5),(3,5,1),(2,4,9)]
```

4.3 Greedy graph walking

In this example, we are given a directed weighted graph and a start node, and we want to walk a given number of steps in the graph and thereby minimise the cost of the path we walked. This example will illustrate the different computational behaviour of the greedy product in comparison to the normal product. We use the same representation of graphs and edges from the previous example, together with the property function that can also be used to calculate the cost of a path. We now define an `getCandidates` function that, given a graph and the path we already walked, calculates the possible next edges:

```
getCandidates :: Graph -> [Edge] -> [Edge]
getCandidates g []        = undefined
getCandidates g [(_,n,_)] = filter (\(x,_,_) -> x == n) g
getCandidates g (_:xs)    = getCandidates g xs
```

Note, that the `getCandidates` function is undefined for the empty path. Therefore we are required to have the start edge in the initial path. Next, we define the list of selection functions. Each selection function takes a history describing the path that was already walked on the graph, calculating all possible next edges and selecting the edge with the minimum cost according to the property function. For our example, we want to walk 3 steps on the graph and therefore replicate the selection function 3 times.

```
es :: Ord r => Graph -> [[Edge] -> J r Edge]
es = replicate 3 f
    where f x p = minWith p (getCandidates g x)
```

Now consider the following graph, as shown in Figure 1.

```
exampleGraph :: Graph
exampleGraph = [(1,2,1),(2,3,1),(2,4,5),(3,5,10),
                (4,6,1), (6,7,2), (5,7,1)]
```

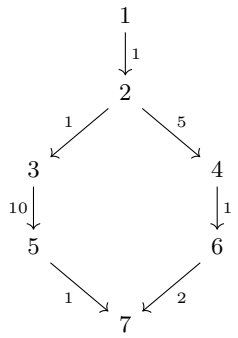


Fig. 1. Example Graph

To greedily walk a path on this graph, we can utilise the `greedyHProduct` with the initial edge `(1,2,1)` in the history and our selection functions applied to the `exampleGraph`. However, applying a greedy algorithm on this graph, locally choosing the best available edge at each stage, we won't get an optimal result:

```
greedyHProduct [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,3,1),(3,4,10),(5,7,1)]
```

In contrast, when using the normal product, we do achieve the optimal result (total cost 9 rather than 13):

```
hProduct [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,4,5),(4,6,1),(6,7,2)]
```

This example illustrates that for a graph walking algorithm to work, some insight about future edges is needed in order to calculate the correct result. When choosing the edge going out of node 2, we need to look further than the current edge to detect that going from node 2 to node 3 would lead to an overall worse outcome. That is, a greedy algorithm doesn't work.

5 Correctness

While the idea of a greedy algorithm is easy to grasp, proving that a greedy algorithm solves a given problem turns out to be quite difficult. If we view greedy algorithms in terms of selection functions, we can state that a greedy algorithm works if both the `greedyProduct` and the `product` functions calculate a result with the same cost:

```
p (greedyProduct selectFunc p) = p (product selectFunc p)    (1)
```

and further with the history dependent version with a initial history `h0`:

```
p (greedyHProduct h0 selectFunc p) = p (hProduct h0 selectFunc p)
                                                              (2)
```

In the case of the graph walking example, we can construct a counter example for which this property does not hold:

```
p (greedyHProduct [(1,2,1)] (es exampleGraph) p) ==
p (hProduct [(1,2,1)] (es exampleGraph) p)
---> False
p (hProduct [(1,2,1)] (es exampleGraph) p)
---> 9
p (greedyHProduct [(1,2,1)] (es exampleGraph) p)
---> 13
```

6 Limited lookahead

While greedy algorithms base their decision on the currently available options without considering the future, there are use cases where a limited lookahead into the future improves the result of an algorithm, without needing to go as far as exhaustive search. We can alter the product further to represent such a limited lookahead. To do so we introduce a limiting parameter to the product and distinguish the behaviour of the product depending on whether we reached the maximum lookahead depth. When building the property function for judging the individual underlying elements of a selection function, we decrement the lookahead depth in the recursive call.

```
limProduct :: Int -> [J r a]  -> J r [a]
limProduct i [] p              = []
limProduct i (e:es) p | i >  0 = a : as
                      | i <= 0 = [a']
    where
        a   = e (\x -> p (x : limProduct (i-1) es (p . (x:))))
        as  = limProduct i es (p . (a:))
        a'  = e (p . ([]:))
```

Further, we can also introduce a history-dependent version with limited lookahead:

```
limHProduct :: Int -> [a] -> [[a] -> J r a] -> J r [a]
limHProduct i h [] p              = []
limHProduct i h (e:es) p | i >  0 = a : as
                         | i <= 0 = [a']
   where
       a  = e h (\x -> p (x : limHProduct (i-1) (h++[x]) es (p . (x:))))
       as = limHProduct i (h++[a]) es (p . (a:))
       a' = e h (p . ([]:))
```

6.1 Graph Example

Recalling the greedy graph walking example from Section 4.3, the greedy algorithm is not able to produce optimal results for the given example graph in

Figure 1. The greedy algorithm is limited to local decisions, and therefore unable to detect an upcoming costly edge. Utilising the limited lookahead product `limHProduct` now with a lookahead of 1, we are able to detect the upcoming costly edge $3 \rightarrow 5$ and are able to calculate an optimal solution (total cost of 9) for this example graph:

```
shortestPathLimited = limHProduct 1 [(1,2,1)] (es exampleGraph) p
---> [(1,2,1),(2,4,5),(4,6,1),(6,7,2)]
```

This might work for this particular graph, there is no guaranty that limited lookahead can be utilised for arbitrary complex graphs. However, considering graphs where every split eventually converges again into a single node after at most n steps, a n-step lookahead is sufficient for our graph walking example. Such graph might look similar to Figure 2.

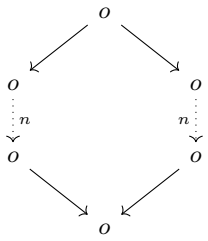


Fig. 2. Example limited lookahead graph

This example shows that if you have deeper insights about your problem can enable programmers to utilise limited lookahead algorithms. Another possible application of limited lookahead algorithms can be in the area of game theory. When, for example, implementing an AI opponent for chess, it is not computationally feasible to calculate a perfect game of chess. At some point, there needs to be a cutoff of the search space to ensure reasonably runtimes.

7 Conclusion and future work

We have seen that in addition to the already known monadic product for selection functions, there are other product implementations for selection functions, each capturing different computational behaviour. In particular with the above presented variations, the idea of describing problems as collections of selection functions can now be applied to problems that are solvable by greedy algorithms and limited lookahead algorithms. Moreover, the computational behaviour of an algorithm can be changed modularly by using different products for selection functions, while the problem description stays the same.

In addition to greedy products and limited lookahead products, there is the potential for more product implementations that behave differently. One example

for this might be a product that is able to perform Alpha-Beta pruning minimax algorithms. Further, a common pattern of the above presented examples is that we are replicating the same selection function n-times. An `iterateNTimes` function would probably feel more natural. Further, the presented examples only work if the length of the result is known. Another useful extension to the collection of product functions might therefore be a `iterateUntil` function that keeps iterating a selecting function until a given predicate is satisfied.

References

1. Bolt, J., Hedges, J., Zahn, P.: Sequential games and nondeterministic selection functions. arXiv preprint arXiv:1811.06810 (2018)
2. Escardó, M., Oliva, P.: Selection functions, bar recursion and backward induction. Math. Struct. Comput. Sci. **20**(2), 127–168 (2010)
3. Escardó, M., Oliva, P.: What sequential games, the tychonoff theorem and the double-negation shift have in common. In: Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming. pp. 21–32 (2010)
4. Escardó, M., Oliva, P.: Sequential games and optimal strategies. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences **467**(2130), 1519–1545 (2011)
5. Hartmann, J.: Finding optimal strategies in sequential games with the novel selection monad. arXiv preprint arXiv:2105.12514 (2018)
6. Hartmann, J.: Dependently Typed Selection Monad (2022), https://github.com/IncredibleHannes/DependentlyTypedSelectionMonad
7. Hedges, J.: The selection monad as a cps transformation. arXiv preprint arXiv:1503.06061 (2015)
8. Hedges, J.M.: Towards compositional game theory. Ph.D. thesis, Queen Mary University of London (2016)
9. Prim, R.C.: Shortest connection networks and some generalizations. The Bell System Technical Journal **36**(6), 1389–1401 (1957)

Appendix

A Proof that product equals sequence

```
xm >>= \x -> sequence' xms >>= \xs -> return (x : xs)
-- {{ expand >>= definition }}
xm >>= \x -> (\p -> (\xs -> return (x : xs)) ((sequence' xms)
(p . flip (\xs -> return (x : xs)) p)) p)
-- {{ apply lambda }}
xm >>= \x -> (\p -> (return (x : ((sequence' xms)
(p . flip (\xs -> return (x : xs)) p))) ) p)
-- {{ resolve return }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . flip (\xs -> return (x : xs)) p))))
-- {{ apply flip }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . (\_ xs -> (x : xs)) p))))
```

```
-- {{ apply lambda }}
xm >>= \x -> (\p -> (x : ((sequence' xms) (p . (\xs -> (x : xs))))) )
-- {{ resolve function composition }}
xm >>= \x p -> (x : ((sequence' xms) (\xs -> p (x : xs))))
-- {{expand >>= definition  }}
\p' -> ( (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))
((xm) (p' . flip (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))  p')) p' )
-- {{ apply lambda }}
\p' -> ( (\x p -> (x : ((sequence' xms) (\xs -> p (x : xs)))))
(xm (\x -> p' (x : ((sequence' xms) (\xs -> p' (x : xs)))))) p' )
-- {{ apply lambda }}
\p' -> (\x -> (x : ((sequence' xms) (\xs -> p' (x : xs)))))
(xm (\x -> p' (x : ((sequence' xms) (\xs -> p' (x : xs))))))
-- {{ rewrite with where }}
\p' -> z : sequence' xms (\xs -> p' (z : xs))
           where z = xm (\x -> p' (x : ((sequence' xms)
           (\xs -> p' (x : xs)))))
-- {{ rewrite with where }}
\p' -> z : zs
           where z  = xm (\x -> p' (x : sequence' xms (p' . (x:))))
                 zs = sequence' xms (p' . (z:))
```

Towards a more perfect union type

Michał J. Gajda
Migamake Pte Ltd
mjgajda@migamake.com

Mikhail Lazarev
MLabs
m282021@gmail.com

We present a principled theoretical framework for inferring and checking the union types, and show its work in practice on JSON data structures.

The framework poses a union type inference as a learning problem from multiple examples. The framework is generic, based on equational properties and, easily extensible.

1 Introduction

Typing dynamic languages has been long considered a challenge [5]. The importance of the task has grown with the ubiquity of cloud application programming interfaces (APIs) utilizing JavaScript object notation (JSON), where one needs to infer the structure having only a limited number of sample documents available. Previous research has suggested it is possible to infer adequate type mappings from sample data [6–9].

In the present study, we expand on these results. We propose a modular framework for defining type systems of programming languages as learning algorithms subject to a set of equations, and evaluate its performance on inference of Haskell data types from JSON API examples.

1.1 Related work

1.1.1 Union type providers

The earliest practical effort to apply union types to JSON inference was made to generate Haskell types [6]. It uses union type theory, but it also lacks an extensible theoretical framework. F# type providers for JSON facilitate deriving a schema automatically; however, the type system does not support union of alternatives and is given as a shape inference algorithm, instead of the design being driven by desired properties [8]. The other attempt to automatically infer schemas has been introduced in the PADS project [9, 10]. Nevertheless, it has not specified a generalized design methodology for type systems. One approach uses Markov chains to derive JSON types [7]. This approach requires considerable engineering time due to the implementation of unit tests in a case-by-case mode, instead of formulating laws applying to all types. Moreover, this approach lacks a sound underlying theory. Regular expression types were also used to type XML documents [11], which does not allow for selecting alternative representations. In the present study, we generalize previously introduced approaches and enable a systematic addition of not only value sets, but also inference sub-algorithms, to the union type system.

1.1.2 Frameworks for describing type systems

Type systems are commonly expressed with a partial relation of *typing*. Their properties, such as subject reduction are also expressed relatively to the relation of *reduction* within a term rewriting system. General formulations have been introduced for the Damas-Milner type systems parameterized by constraints

Submitted to:
TFP 2022

[12, 13]. It is also worth noting that traditional Damas-Milner type disciplines enjoy decidability, and embrace the laws of soundness, and subject-reduction. However these laws often prove too strict during type system development and extension. The resulting type systems are fragile and often lack the subject-reduction property [14–23]. Given the publication bias [24], many more unpublished examples are expected. Because of this fragility, the type systems of widely used programming languages are either undecidable [25], or even unsound [26].

Early approaches used lattice structure on the types [27], which is more stringent than ours since it requires idempotence of two information fusion operations, which correspond to the meet and join operations on a lattice. These are used for fusing the information in covariant and contravariant positions of the types. Semantic subtyping provides a set-based characterization of union, intersection, and complement types [28, 29], which allows for modelling subtype containment on first-order types and functions. This model relies on building a model using infinite sets in set theory, but fails to generalize to non-idempotent learning, such as the use of machine learning techniques like Markov chains to infer optimal type representations from the frequency of value occurrence [7].

We are also not aware of a type inference framework that consistently and completely preserves information in the face of inconsistencies nor errors. Most frameworks simply use $\bot$ or the *infamous undefined behaviour* [30–32].

We propose a constructive framework that preserves the soundness in inference, while allowing for consistent approximations. It is based on equational laws that need to be satisfied by type-like structures. Indeed, our demonstration shows that most parts of the implementation of a type system may be generic.

2 Motivation

Here, we consider several examples similar to JSON API descriptions. We provide these examples in the form of a few JSON objects, along with desired representation as Haskell data declaration.

1. Subsets of data within a single constructor:
   a. *API argument is an email* – it is a subset of valid STRING values that can be validated on the client-side.
   b. *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (INT) between 10, and 10,000
   c. *The date field contains ISO8601 date* – a record field represented as a STRING that contains a calendar date in the format "2019-03-03"
2. Optional fields: *The page size is equal to 100 by default* – it means we expect to see the record like {"page_size": 50} or an empty record {} that should be interpreted in the same way as {"page_size": 100}
3. Variant fields: *Answer to a query is either a number of registered objects, or String "unavailable"* – this is ether an integer value (INT) or a STRING value (a union of two types: INT :|: STRING, behaves like EITHER INT STRING but is untagged in JSON)
4. Variant records: *Answer contains either a text message with a user identifier or an error.* – That can be represented as one of following options:

```
{ "MESSAGE": "Where can I submit proposal?", "UID"  : 1014 }
{ "MESSAGE": "Submit it to HotCRP",          "UID"  : 317  }
{ "ERROR"  : "Authorization failed" ,        "CODE" : 401  }
{ "ERROR"  : "User not found",               "CODE" : 404  }
```

```
data EXAMPLE4 = MESSAGE { message :: STRING
                        , uid     :: INT }
              | ERROR   { error   :: STRING
                        , code    :: INT }
```

5. Arrays corresponding to records:

```
[ [1, "Nick", null              ]
, [2, "George", "2019-04-11"]
, [3, "Olivia" , "1984-05-03"] ]
```

6. Maps of identical objects (example from [7]):

```
{ "6408F5": { "SIZE"       : 969709     , "HEIGHT"   : 510599
            , "DIFFICULTY" : 866429.732 , "PREVIOUS" : "54fced" },
  "54FCED": { "SIZE"       : 991394     , "HEIGHT"   : 510598
            , "DIFFICULTY" : 866429.823 , "PREVIOUS" : "6c9589" },
  "6C9589": { "SIZE"       : 990527     , "HEIGHT"   : 510597
            , "DIFFICULTY" : 866429.931 , "PREVIOUS" : "51a0cb" } }
```

It should be noted that the last example presented above requires Haskell representation inference to be non-monotonic, as an example of object with only a single key would be best represented by a record type:

```
data EXAMPLE = EXAMPLE { f6408f5      :: O6408F5
                       , f54fced      :: O6408F5
                       , f6c9589      :: O6408F5 }
data O6408F5 = O6408F5 { size, height :: INT
                       , difficulty   :: DOUBLE
                       , previous     :: STRING }
```

However, when this object has multiple keys with values of the same structure, the best representation is that of a mapping shown below. This is also an example of when user may decide to explicitly add evidence for one of the alternative representations in the case when input samples are insufficient. (like when input samples only contain a single element dictionary.)

```
::: { #sec:nonmonotonic-inference }
data EXAMPLEMAP = EXAMPLEMAP (MAP HEX EXAMPLEELT)
data EXAMPLEELT = EXAMPLEELT { size       :: INT,    height   :: INT
                             , difficulty :: DOUBLE, previous :: STRING }
```

3 Problem definition

As we focus on JSON, we utilize Haskell encoding of the JSON term for convenient reading(from Aeson package [33]); specified as follows:

```
data VALUE = OBJECT (MAP STRING VALUE)
           | ARRAY  [           VALUE]
           | NULL
           | NUMBER SCIENTIFIC
           | STRING TEXT
           | BOOL   BOOL
```

3.1 Goal of inference

Given an undocumented (or incorrectly labelled) JSON API, we may need to read the input as encoded in Haskell and avoid checking for the presence of *unexpected* format deviations. At the same time, we may decide to accept all known valid inputs outright so that we can use types to ensure that the input is processed exhaustively (by relying on the compiler's ability to check for unmatched cases).

Accordingly, we assume the *minimal containing set principle*: the smallest non-singleton set is a better approximation type than a singleton set.

Second, the *information content principle*: we aim to minimize the number of *degrees of freedom* of a type, while conforming to a common structure.

Given these principles, and examples of frequently occurring patterns, we can infer a reasonable *world of types* that approximate sets of possible values. In this way, we can implement *type system engineering* that allows deriving type system design directly from the information about data structures and the likelihood of their occurrence.

4 Type inference

If an inference fails, it is always possible to correct it by introducing an additional observation (example). To denote unification operation, or **information fusion** between two type descriptions, we use a SEMIGROUP interface operation $\diamond$ to merge types inferred from different observations.[1] We use a neutral element of the MONOID to indicate a type corresponding to no observations.

class SEMIGROUP *ty* **where**
  $(\diamond) :: ty \to ty \to ty$
class SEMIGROUP *ty*
  $\Rightarrow$ MONOID *ty* **where**
  mempty :: *ty*

In other words, we can say that *mempty* (or $\bot$) element corresponds to situation where **no information was received** about a possible value (no term was seen, not even a null). It is a neutral element of TYPELIKE, since it bring no additional information. For example, an empty array [ ] can be referred to as an array type with *mempty* as an element type. This represents the view that $\diamond$ always **gathers more information** about the type, as opposed to the traditional unification that always **narrows down** possible solutions.

4.0.1 Beyond set

In the domain of permissive union types, a *beyond* set represents the case of **everything permitted** or a fully dynamic value when we gather the information that permits every possible value inside a type. At the first reading, it may be deemed that a *beyond* set should comprise of only one single element – the $\top$ one (arriving at a complete bounded semilattice), but this is too narrow for our purpose of *monotonically gathering information*.

However, since we defined the **generalization** operator $\diamond$ as **information fusion** (corresponding to unification in the dual case of strict type systems), we may encounter difficulties in assuring that no

[1] If the semigroup is idempotent, that is when $\alpha \diamond \alpha == \alpha$, then $\diamond$ is a join operation (least upper bound on a semilattice). Note that the approach presented here is dual to traditional unification that *narrows down* solutions; we remove the requirement for idempotence in order to accommodate non-idempotent learning, and will not discuss semilattices any further.

$t_1 \diamond (t_2 \diamond t_3)$	$=$	$t_1 \diamond (t_2 \diamond t_3)$	*(semigroup associativity)*
$\text{mempty} \diamond t$	$=$	t	*(left identity of the monoid)*
$t \diamond \text{mempty}$	$=$	t	*(right identity of the monoid)*
beyond t_1	$\Rightarrow$	*beyond* $(t_1 \diamond t_2)$	*(beyond is closed to the right)*
beyond t_2	$\Rightarrow$	*beyond* $(t_1 \diamond t_2)$	*(beyond is closed to the left)*

Figure 1: Laws of Typelike

information has been lost during the generalization[2]. Moreover, strict type systems usually specify more than one error value, as it should contain information about error messages and keep track from where an error has been originated[3].

This observation lets us go well beyond typing statement of gradual type inference as a discovery problem from incomplete information [34]. Here we consider type inference as a **learning problem** and, furthermore, find common ground between the dynamic and the static typing discipline. The languages relying on the static type discipline usually consider *beyond* as a set of error messages, as a value should correspond to a statically assigned **narrow** type. In this setting, *mempty* would be the fully polymorphic type $\forall a.a$. Languages with dynamic type discipline will treat *beyond* as an untyped, dynamic value and *mempty* will again be an entirely unknown, polymorphic value (like a type of an element of an empty array).

```
class   MONOID  t
   ⇒  TYPELIKE t where
  beyond ::        t → BOOL
```

Besides the standard laws for a **commutative** MONOID, we state the new law for the *beyond* set: it is always **closed to information addition** by $(\diamond\alpha)$ or $(\alpha\diamond)$ for any value of α. In other words, the *beyond* set forms a **submonoid** , and an **attractor** of $\diamond$ on both sides. Crucially, we describe the typing laws in Fig.1 as QuickCheck [35] properties so that automated testing via random sample generation can be implemented to detect apparent violations.

However, we do not require *idempotence* of $\diamond$, which is uniformly present in union type frameworks based on lattices [36] and sets [28]. That is because this requirement is valid only for strict type inference, not for a more general type inference as a learning problem. As we saw on EXAMPLEMAP in sec. 2, we need non-monotonic inference when dealing with alternative representations.

When a specific instance of TYPELIKE is not a semilattice (an idempotent semigroup), we will explicitly indicate that is the case. This seems to be particularly useful to validate when testing a recursive structure of the type.

In this way, we can specify other elements of *beyond* set instead of a single $\top$. When under strict type discipline, like that of Haskell [25], we seek to enable each element of the *beyond* set to contain at least one error message.

Time to present the **typing relation** and its laws. Instead of the classical *val* : *ty*, we read and denote typing as *ty* ‘*Types*‘ *val*. Specifying the laws of typing is important, since we may need to separately consider the validity of a domain of types/type constraints, and that of the sound typing of the terms by these valid types.

[2]Examples will be provided later.

[3]In this case: *beyond* (ERROR _) = TRUE | *otherwise* = FALSE.

				check	mempty	v	=	**False**	*(mempty contains no terms)*
beyond	t		$\Rightarrow$	check	t	v	=	**True**	*(beyond contains all terms)*
check	t_1	v	$\Rightarrow$	check	$(t_1 \diamond t_2)$	v	=	**True**	*(left fusion keeps terms)*
check	t_2	v	$\Rightarrow$	check	$(t_1 \diamond t_2)$	v	=	**True**	*(right fusion keeps terms)*
				check	(infer v)	v	=	**True**	*(inferred type contains the source term)*

Figure 2: Laws for typing

class TYPELIKE *ty* $\Rightarrow$ *ty* ‘TYPES‘ *val* **where**
infer :: *val* $\rightarrow$ *ty*
check :: *ty* $\rightarrow$ *val* $\rightarrow$ BOOL

The minimal definition of typing inference relation and type checking relation is formulated as consistency between these two operations, as shown in Fig.2.

First, we note that to describe *no information*, *mempty* cannot correctly type any term. A second important rule of typing is that all terms are typed successfully by any value in the *beyond* set. The next couple of fusion rules make sure that typing is preserved when adding information to the left or to the right. (For a classical type inference relation, it would be described as *principal type property*.) Finally we state the most intuitive rule for typing: a type inferred from a term must always be valid for that particular term.

The minimal TYPELIKE instance is the one that contains only *mempty* corresponding to the case of *no sample data received*, and a single *beyond* element for *all values permitted*. We will define it below as PRESENCECONSTRAINT in sec. 4.2.3. These laws are also compatible with the strict, static type discipline: namely, the *beyond* set corresponds to a set of constraints with at least one type error, and a task of a compiler to prevent any program with the terms that type only to *beyond* as a least upper bound.

4.1 Type engineering principles

Considering that we aim to infer a type from a finite number of samples, we encounter a *learning problem*, so we need to use *prior* knowledge about the domain for inferring types. Observing that field $a =$ FALSE we can expect that in particular cases, we may obtain that another example where $a =$ TRUE. After observing a $b = 123$, we expect that $b = 100$ would also be accepted. It means that we need to consider a typing system to *learn a reasonable general class from few instances*. This observation motivates formulating the type system as an inference problem. As the purpose is to deliver the most descriptive[4] types, we assume that we need to obtain a broader view rather than focusing on a *free type* and applying it to larger sets whenever it is deemed justified.

The other principle corresponds to **correct operation**. It implies that having operations regarded on types, we can find a minimal set of types that assure correct operation in the case of unexpected errors. Indeed we want to apply this theory to infer a type definition from a finite set of examples. We also seek to generalize it to infinite types. We endeavour rules to be as short as possible. For the inference to be compositional, it should be also be a **contravariant functor** with regards to the data constructors. For example, if ATYPE *x y* types the value{"a": X, "b": Y}, then *x* must type the value X, and *y* must type the value Y. This behaviour corresponds to that of a **tensor product**.

[4]The shortest one according to the information complexity principle. This is formalized below as *typeCost* metric.

4.2 Constraint definition

4.2.1 Flat type constraints

Let us first consider typing of flat type: STRING (similar treatment should be given to the NUMBER type.)

```
data StringConstraint = SCDate | SCEmail
  | SCEnum (Set.Set Text) – non-empty set of observed values
  | SCNever – mempty
  | SCAny – beyond
instance Semigroup StringConstraint where
  SCNever     ◇ α           = α
  α           ◇ SCNever     = α
  SCAny       ◇ _           = SCAny
  _           ◇ SCAny       = SCAny
  SCDate      ◇ SCDate      = SCDate
  SCEmail     ◇ SCEmail     = SCEmail
  (SCEnum α) ◇ (SCEnum β)
    | isValidStringConstSet (α ◇ β) = SCEnum (α ◇ β)
  _ ◇ _                     = SCAny

instance Monoid StringConstraint where
  mempty = SCNever

instance Typelike StringConstraint where
  beyond = (==SCAny)
```

Strings can either be dates, emails or one of some user-specified fixed values. By including a bottom (SCNEVER) and top (SCANY) element, we easily see how STRINGCONSTRAINT forms a typelike structure. Now we can define the typing rules for strings, that is inferring the constraint a given string adheres to and also checking against a given type:

```
instance StringConstraint `Types` Text  where
  infer (isValidDate  → True) = SCDate
  infer (isValidEmail → True) = SCEmail
  infer ""                    = SCAny
  infer value                 = SCEnum (Set.singleton value)
  infer _                     = SCAny

  check SCDate  σ = isValidDate  σ
  check SCEmail σ = isValidEmail σ
  check (SCEnum vs) σ = σ `Set.member` vs
  check SCNever _ = False
  check SCAny   _ = True
```

4.2.2 Free union type

Before we endeavour on finding type constraints for compound values (arrays and objects), it might be instructive to find a notion of *free type*, that is a type with no additional laws but the ones stated above. Given a term with arbitrary constructors we can infer a *free type* for every term set T as follows: For any

T value type *Set T* satisfies our notion of *free type* specified as follows:

```
data FREETYPE α = FREETYPE { captured :: SET α } | FULL
instance  (ORD α, EQ  α)
      ⇒ SEMIGROUP ( FREETYPE α) where
  FULL ◇ _       = FULL
  _    ◇ FULL    = FULL
  α    ◇ β       = FREETYPE
                   $ (Set.union      `on` captured) α β
instance  (ORD α,  EQ α, SHOW  α)
      ⇒ TYPELIKE ( FREETYPE  α) where
  beyond = (==FULL)

instance  (ORD α,  EQ  α, SHOW  α)
      ⇒ FREETYPE α  `TYPES`  α  where

  infer                       = FREETYPE . Set.singleton
  check FULL             term = TRUE
  check (FREETYPE σ)     term = term `Set.member` σ
```

This definition is deemed sound and applicable to finite sets of terms or values. For a set of values: ["yes", "no", "error"], we may reasonably consider that type is an appropriate approximation of C-style enumeration, or Haskell-style ADT without constructor arguments. However, the deficiency of this notion of *free type* is that it does not allow generalizing in infinite and recursive domains! It only allows to utilize objects from the sample.

4.2.3 Presence and absence constraint

We call the degenerate case of TYPELIKE a *presence or absence constraint*. It just checks that the type contains at least one observation of the input value or no observations at all. It is vital as it can be used to specify an element type of an empty array. After seeing *true* value, we also expect *false*, so we can say that it is also a primary constraint for pragmatically indivisible like the set of boolean values. The same observation is valid for *null* values, as there is only one *null* value ever to observe.

```
type BOOLCONSTRAINT = PRESENCECONSTRAINT BOOL
type NULLCONSTRAINT = PRESENCECONSTRAINT ()
data PRESENCECONSTRAINT α = PRESENT | ABSENT
instance PRESENCECONSTRAINT α `TYPES` α where
  infer  _          = PRESENT
  check  PRESENT _  = TRUE
  check  ABSENT  _  = FALSE
```

Variants It is simple to represent a variant of two *mutually exclusive* types. They can be implemented with a type related to EITHER type that assumes these types are exclusive, we denote the tagged union by :|:. In other words for INT :|: STRING type, we first control whether the value is an INT, and if this check fails, we attempt to check it as a STRING. Variant records are slightly more complicated, as it may be unclear which typing is better to use:

```
{ "MESSAGE" :  "Where can I submit my proposal?",  "UID"   :  1014 }
{ "ERROR"   :  "Authorization failed",             "CODE"  :  401  }
```

```
data OurRecord = OurRecord { message, error :: Maybe String
                           , code,    uid   :: Maybe Int }

data OurRecord2 = Message { message :: String, uid  :: Int }
                | Error   { error   :: String, code :: Int }
```

The best attempt here is to rely on the available examples being reasonably exhaustive. That is, we can estimate how many examples we have for each, and how many of them match. Then, we compare this number with type complexity (with options being more complex to process because they need additional **case** expression.) In such cases, the latter definition has only one MAYBE field (the *optionality* on the toplevel is one), while the former definition has four MAYBE fields (*optionality* is four). When we obtain more samples, the pattern emerges:

```
{ "ERROR"   : "Authorization failed",              "CODE" : 401 }
{ "MESSAGE" : "Where can I submit my proposal?", "UID"  : 1014 }
{ "MESSAGE" : "Sent it to HotCRP",               "UID"  : 93  }
{ "MESSAGE" : "Thanks!",                         "UID"  : 1014 }
{ "ERROR"   : "Missing user",                    "CODE" : 404 }
```

Type cost function Since we are interested in types with less complexity and less optionality, we will define cost function as follows:

```
class  ( TypeLike ty
       , Eq       ty)
  ⇒    TypeCost ty where
  typeCost :: ty → TyCost

  typeCost α = if    α == mempty then 0 else 1
instance Semigroup TyCost where (◇) = (+)
instance Monoid    TyCost where mempty = 0

newtype TyCost = TyCost Int
```

When presented with several alternate representations from the same set of observations, we will use this function to select the least complex representation of the type. For flat constraints as above, we infer that they offer no optionality when no observations occurred (cost of *mempty* is 0), otherwise, the cost is 1. Type cost should be non-negative, and non-decreasing when we add new observations to the type.

4.2.4 Object constraint

To avoid information loss, a constraint for JSON's object type is introduced in such a way to **simultaneously gather information** about representing it either as a MAP, or a record. The typing of MAP would be specified as follows, with the optionality cost being a sum of optionalities in its fields:

```
data MappingConstraint = MappingNever -- empty
  | MappingConstraint {
      keyConstraint   :: StringConstraint
    , valueConstraint :: UnionType }
instance TypeCost MappingConstraint where
  typeCost MappingNever            =  0
  typeCost MappingConstraint {..} =
      typeCost keyConstraint
    + typeCost valueConstraint
```

Here, UnionType stands for any JSON type, and will be composed later in this section as the union of all the different constraints that we present.

Separately, we acquire the information about a possible typing of a JSON object as a record of values. Note that RCTop never actually occurs during inference. That is, we could have represented the RecordConstraint as a Typelike with an empty *beyond* set. The merging of constraints would be simply merging of all column constraints.

```
data RecordConstraint = RCTop {- beyond -}    | RCBottom {- empty -}
                      | RecordConstraint { fields :: HashMap Text UnionType }
instance RecordConstraint `Types` Object where
  infer = RecordConstraint . Map.fromList
        . fmap (second infer) . Map.toList
  check RecordConstraint {fields} obj =
        all (`elem` Map.keys fields) (Map.keys obj)
    && and (Map.elems $ Map.intersectionWith check fields obj)
    && all isNullable (Map.elems $ fields `Map.difference` obj)
```

Observing that the two abstract domains considered above are independent, we can store the information about both options separately in a record[5]. It should be noted that this representation is similar to *intersection type*: any value that satisfies ObjectConstraint must conform to both *mappingCase*, and *recordCase*. Thus the constraint contains parallel information, it is the *tensor product* of component constraints, which allows us to address the problem of enforcing the *principal type property* in alternative union type representations, meaning that this product is the principal type that serves to acquire the information corresponding to different representations and handle them separately. Since we plan to choose only one representation for the object, we can say that the minimum cost of this type is the minimum of component costs.

```
data ObjectConstraint = ObjectNever -- empty
                      | ObjectConstraint { mappingCase :: MappingConstraint
                                         , recordCase  :: RecordConstraint }
instance TypeCost ObjectConstraint where
  typeCost ObjectConstraint {..} =  typeCost mappingCase
                                 `min` typeCost recordCase
```

4.2.5 Array constraint

Similarly to the object type, ArrayConstraint is uses a *parallel tensor product* to simultaneously obtain information about all possible representations of an array, differentiating between an array of the same elements, and a row with the type depending on a column. We need to acquire the information for

[5]The choice of representation will be explained later. Here we only consider acquiring information about possible values.

both alternatives separately, since we need to measure a relative likelihood of either case before mapping the union type to a specific Haskell declaration. Since we will eventually pick one possible representation, the cost of optionality is again the lesser of the costs of the representation-specific constraints. Semigroup operation just merges information on the components, and the same is done when inferring types or checking them.

```
data ARRAYCONSTRAINT = ARRAYNEVER – mempty
  | ARRAYCONSTRAINT { rowCase :: ROWCONSTRAINT, arrayCase :: UNIONTYPE }
data ROWCONSTRAINT = ROWTOP {- beyond -} | ROWNEVER {- mempty -} | ROW [UNIONTYPE]
instance ARRAYCONSTRAINT 'TYPES' ARRAY where
  infer vs = ARRAYCONSTRAINT { rowCase   = infer      vs
                             , arrayCase = mconcat ( infer <$>
                                                     Foldable.toList vs) }
  check ARRAYNEVER           vs = FALSE
  check ARRAYCONSTRAINT {..} vs =          check rowCase vs
                                  && and ( check arrayCase <$>
                                           Foldable.toList  vs)
```

A row constraint is valid only if there is the same number of entries in all rows, which is represented by escaping the *beyond* set whenever there is an uneven number of columns. Row constraint remains valid only if both constraint describe the record of the same length; otherwise, we yield ROWTOP to indicate that it is no longer valid. In other words, ROWCONSTRAINT is a *levitated*[6] *semilattice* [37] with a neutral element over the content type that is a list of UNIONTYPE objects.

```
instance ROWCONSTRAINT 'TYPES' ARRAY where
  infer  = ROW
         . Foldable.toList
         . fmap infer
  check ROWTOP    _ = TRUE
  check ROWNEVER  _ = FALSE
  check (ROW rs)  vs
    | length rs == length vs =
      and $
        zipWith check                rs
                      ( Foldable.toList vs)
  check _ _ =           FALSE
```

4.2.6 Combining the union type

It should note that given the constraints for the different type constructors, the union type can be considered as mostly a generic MONOID instance [38]. Merging information with $\diamond$ and *mempty* follow the pattern above, by just lifting operations on the component.

```
data UNIONTYPE = UNIONTYPE {
    unionNull :: NULLCONSTRAINT,   unionBool :: BOOLCONSTRAINT
  , unionNum  :: NUMBERCONSTRAINT, unionStr  :: STRINGCONSTRAINT
  , unionArr  :: ARRAYCONSTRAINT,  unionObj  :: OBJECTCONSTRAINT }
```

The generic structure of union type can be explained by the fact that the information contained in each record field is *independent* from the information contained in other fields. We call it *disjoint*

[6] *Levitated lattice* is created by appending distinct $\bot$ and $\top$ to a set that does not possess them by itself.

tensor product, and it is commonly broken down by value constructors. It means that we generalize independently over disjoint dimensions.[7]

Type checking and inference are dispatches to the appropriate record fields, according to the constructor of the input JSON value. It enables implementing a clear and efficient treatment of different alternatives separately[8].

```
instance UNIONTYPE `TYPES` VALUE where
  infer (BOOL β)     = mempty { unionBool = infer β }
  infer  NULL        = mempty { unionNull = infer () }
  infer (NUMBER ν)   = mempty { unionNum = infer ν }
  infer (STRING σ)   = mempty { unionStr = infer σ }
  infer (OBJECT ω)   = mempty { unionObj = infer ω }
  infer (ARRAY α)    = mempty { unionArr = infer α }

  check UNIONTYPE { unionBool } (BOOL β) =
        check       unionBool       β
  check UNIONTYPE { unionNull } NULL =
        check       unionNull   ()
  check UNIONTYPE { unionNum } (NUMBER ν) =
        check       unionNum       ν
  check UNIONTYPE { unionStr } (STRING σ) =
        check       unionStr       σ
  check UNIONTYPE { unionObj } (OBJECT ω) =
        check       unionObj       ω
  check UNIONTYPE { unionArr } (ARRAY α) =
        check       unionArr       α
```

Since union type is all about optionality, we need to sum all options from different alternatives to obtain its *typeCost*.

4.3 Overlapping alternatives

The essence of union type systems has long been dealing with conflicting types provided in the input. Motivated by the examples in sec. 2, we also aim to address conflicting alternative assignments. It is apparent that examples 4. to 6. hint at more than one assignment: in example 5, a set of lists of values may correspond to INT, STRING, or *null*, or a table that has the same (and predefined) type for each row; in example 6, a record of fixed names, or the mapping from hash to a single object type. It is therefore natural to pick an alternative that manifests itself more frequently in the input samples.

Let us now discuss how to gather information about the number of samples supporting each alternative type constraint. To explain this, the other example can be considered:

[7] In this example, JSON terms can be described by terms without variables, and sets of tuples for dictionaries, so generalization by anti-unification is straightforward.

[8] The question may arise: what is the *union type* without *set union*? When the sets are disjoint, we just put the values in different bins for easier handling.

```
{"HISTORY": [ { "ERROR"   : "Authorization failed",                "CODE" : 401  }
            , { "MESSAGE" : "Where can I submit my proposal?", "UID"  : 1014 }
            , { "MESSAGE" : "Sent it to HotCRP",               "UID"  : 93   }
            , { "MESSAGE" : "Thanks!",                         "UID"  : 1014 }
            , { "ERROR"   : "Authorization failed",            "CODE" : 401  }]}
```

First, we need to identify it as a list of similar elements. Second, there are multiple instances of each record example, hence we consider the most appriopriate model to be that of multisets. To find the best representation, we can a compute their type complexity, and attempt to minimize the term.

Next step is to detect the similarities between type descriptions introduced for different parts of the term:

```
{ "HISTORY"      : [ ...]
, "LAST_MESSAGE" : { "MESSAGE": "Thanks!", "UID" : 1014} }
```

The crucial observation here is that we can augment a type constraint with auxiliary information about the number of samples observed and it will remain a TYPELIKE object. The COUNTED constraint counts the number of samples observed for the constraint inside so that we can decide on which alternative representation is best supported by evidence.

```
data COUNTED α = COUNTED { count::INT, constraint::α }
instance   SEMIGROUP                α
       ⇒ SEMIGROUP ( COUNTED α) where
  α ◇ β  = COUNTED {
             count      = count      α + count      β
           , constraint = constraint α ◇ constraint β
           }
instance              ty 'TYPES' term
       ⇒ COUNTED ty 'TYPES' term where
  infer  term = COUNTED 1 $ infer    term
  check (COUNTED _ ty)  term = check ty term
```

Notice how *infer* provides a single observation, to be summed up by the semigroup fusion operation. It should also be noted that COUNTED constraint is the first example that does not correspond to a semilattice, that is $\alpha \diamond \alpha \neq \alpha$. This is natural for a TYPELIKE object; it is not a type constraint in a conventional sense, just an accumulation of knowledge.

Therefore, at each step, we may need to maintain a **cardinality** of each possible value and, given a sufficient number of samples, we may attempt to detect the most suitable encoding.[9] For the sake of efficiency, we may need to merge alternatives whenever the size of a multiset size crosses a specified threshold.

5 Finishing touches

The final touch would be to perform the post-processing of an assigned type before generating it to make it more resilient to common uncertainties. These assumptions may bypass the defined *information content principle* criterion specified in the initial part of the paper; however, they prove to work well in practice [6, 7].

If we have no observations corresponding to an array type, it can be inconvenient to disallow an array to contain any values at all. Therefore, we introduce a non-monotonic step of converting the *mempty* into

[9] If we detect a pattern too early, we risk to make the types too narrow to work with actual API responses.

a final TYPELIKE object aiming to introduce a representation allowing the occurrence of any VALUE in the input. That still preserves the validity of the typing. We note that the program using our types must not have any assumptions about these values; however, at the same time, it should be able to print them for debugging purposes.

In most JSON documents, we observe that the same object can be simultaneously described in different parts of sample data structures. Due to this reason, we compare the sets of labels assigned to all objects and propose to unify those that have more than 60% of identical labels. For transparency, the identified candidates are logged for each user, and a user can also indicate them explicitly instead of relying on automation. We conclude that this allows considerably decreasing the complexity of types and makes the output less redundant.

6 Framework extensibility

Framework can be easily adopted and extended to infer Haskell data types from other data representations. Consider following example for CSV file, which potentially contains banking related information. For such CSV file with the help of framework we want to infer type for each CSV column.

We represent CSV column as Union type. Each cell of column can contain any of: DATE, NUMBER, STRING, CURRENCY or be empty.

```
data CSVCOLUMN = CSVCOLUMN
  { csvDate     :: DATECONSTRAINT
  , csvNumber   :: NUMBERCONSTRAINT
  , csvString   :: STRINGCONSTRAINT
  , csvCurrency :: CURRENCYCONSTRAINT
  , csvEmpty    :: NULLCONSTRAINT
  }
  deriving (EQ, SHOW, GENERIC)
```

We add CURRENCYCONSTRAINT and required instances to infer currency codes in ISO format.

```
instance CURRENCYCONSTRAINT 'TYPES' TEXT where
  infer v
    | JUST _    ← readAlpha v
    = CCALPHA
    | otherwise = CCBEYOND

  check CCALPHA σ  = isJust $ readAlpha σ
  check CCNEVER  _ = FALSE
  check CCBEYOND _ = TRUE
```

Here we trying to read arbitrary string as ISO currency code with *readAlpha* function, if an attempt fails the data provided as text considered *beyond* the currency type.

We add special DATECONSTRAINT to get more fine grained inference. For each valid DATEFORMAT, we record how many times it was matched.

```
data DATECONSTRAINT
  = DCNEVER
  | DCBEYOND
  | DCDATE
      { formats :: MAP DATEFORMAT TIMESMATCHED
      }
  deriving (EQ, SHOW)

instance  DATECONSTRAINT 'TYPES' TEXT where
  infer  v
    | JUST fmt    ← findFirstMatching knownFormats v =
      DCDATE $ M.singleton fmt 1
    | otherwise =
      DCBEYOND

  check DCNEVER _ = FALSE
  check DCBEYOND _ = TRUE
  check (DCDATE  fmts) v = isJust $ findFirstMatching (M.keys fmts) v
```

Call to "findFirstMatching knownFormats v" provides adjustable logic to infer dates, where result of inference depends from specified date formats. If this fails fails, then again – type of data we tried to read is *beyon* the type of DATE we need.

The union type CSVCOLUMN is constructed as a parallel tensor product:

```
instance SEMIGROUP CSVCOLUMN where
  c1 ◇  c2 =
    CSVCOLUMN
      { csvDate      = ((◇) 'on' csvDate) c1 c2
      , csvNumber    = ((◇) 'on' csvNumber) c1  c2
      , csvString    = ((◇) 'on' csvString)   c1  c2
      , csvCurrency  = ((◇) 'on' csvCurrency) c1   c2
      , csvEmpty     = ((◇) 'on' csvEmpty) c1  c2
      }
```

During inference we determine the type of each cell and then fuse gathered knowledge to get the type of the whole column. We can go further and try to infer arbitrary text as Haskell data type representing CSV entries, e.g.:

```
type COLUMNNAME =  TEXT
type NAMEDCOLUMNS = [(   COLUMNNAME, CSVCOLUMN)]

data CSVRECORD
  = CSVRECORD {namedColumns :: !NAMEDCOLUMNS}
  | RECORDBEYOND
```

Here we use RECORDBEYOND to signal that after inspecting raw data of DECODEDROWS we can conclude that parsed data not good enough for our purpose and should be treated as plain text, e.g..

CSVRECORD *types* collection of DECODEDROWS, which obtained from running CSV parser like *cassava* for target file. We check if decoded data is good enough for our purpose with *checkRows* (if it's not good, consider it *beyond* meaningful type), extract colums from decoded data with *makeColumns*, infer type of each column and gather results together as CSVRECORD:

```
type DecodedRows = Vector (Vector Text)

instance CSVRecord 'Types' DecodedRows where
  infer dr
    | Just checkedRecrods ← checkRows dr
    = let
        inferColumn = foldMap infer
        CSVRecord $ V.toList . fmap inferColumn . makeColumns $ checkedRecrods
    | otherwise
    = RecordBeyond
```

Finally, having this in hand we can specify Haskell data type CSVEntry and build it from CSVRecord:

```
instance ToHType CSVRecord where
  toHTypes RecordNever = []
  toHTypes RecordBeyond = ["Text"]
  toHTypes (CSVRecord cols) =
    [ HADT
        [HCons "CSVEntry" $ fmap toArgs cols]
    ]
    where
      toArgs (title, v) = (HFieldId . unpack $ title, toHType v)
```

7 Future work

In the present paper, we only discuss typing of tree-like values, with no facilities to refer to previous parts. However, it is natural to scale this approach to types being referred to by name and possibly containining each other. To address these cases, we plan to show that the environment of TypeLike objects is also TypeLike, and that constraint generalization (*anti-unification*) can be extended in the same way.

It should be noted that many TypeLike instances for non-simple types usually follow one of two patterns: (1) for a finite sum of disjoint constructors, we bin this information by each constructor during the inference (2) for typing terms with multiple alternative representations, we infer all constraints separately for each alternative representation. In both cases, Generic derivation of the Monoid, TypeLike, and TypeCost instances is possible [38, 39]. This allows us to design a type system by declaring datatypes themselves and leave implementation to the compiler. Manual implementations would only be necessary for special constraints, like StringConstraint and Counted.

Finally, we believe that we can explain the duality of the equation-based framework of TypeLike and use generalization (anti-unification) instead of unification (or narrowing) as a type inference mechanism. The *beyond* set would then correspond to a set of error messages, and a result of the inference would represent a principal type in the Damas-Milner sense [40].

8 Conclusion

In the present study, we aimed to derive the types that were valid with respect to the provided specification, thereby obtaining the information from the input in the most comprehensive way. We defined type inference as representation learning and type system engineering as a meta-learning problem in which the **priors corresponding to the data structure induced typing rules**. We show how type safety can

be quickly tested with equational laws in QuickCheck, which is particularly useful during prototyping, and may be supplemented with a fully formal proof in the future.

We demonstrate how enforcing compositionality by making inference a contravariant functor creates two different tensor products (parallel and disjoint) that are useful for systematically defining compound union types.

We also formulated the **union type discipline** as manipulation of TYPELIKE commutative monoids that represent knowledge about the data structure. In addition, we proposed a methodology for engineering union type systems that was logically justified by theoretical criteria. We demonstrated that it was capable of consistently explaining the decisions made in practice. We followed a strictly constructive, algorithmic procedure, that can be implemented generically.

We hope that this kind of straightforward type system engineering will become more widespread in future practice, replacing less modular approaches of the past. The proposed approach may be used to underlie the way towards formal construction and derivation of type systems based on the specification of value domains and design constraints.

Bibliography

1. Knuth, D.E.: Literate programming. Comput. J. 27, 97–111 (1984). https://doi.org/10.1093/comjnl/27.2.97.
2. Hidding, J.: enTangleD: A bi-directional literate programming tool, `https://blog.escience center.nl/entangled-1744448f4b9f`.
3. McFarlane, J.: Pandoc: A universal document converter, `https://pandoc.org`.
4. Mitchell, N.D.: GHCID - a new GHCi based IDE (ish), `http://neilmitchell.blogspot.com/2014/09/ghcid-new-ghci-based-ide-ish.html`.
5. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005 - object-oriented programming. pp. 428–452. Springer Berlin Heidelberg, Berlin, Heidelberg (2005).
6. Anonymous: JSON autotype: Presentation for Haskell.SG, `https://engineers.sg/video/json-autotype-1-0-haskell-sg--429`.
7. Siegel, D.: A first look at quicktype, `https://blog.quicktype.io/first-look/`.
8. Petricek, T., Guerra, G., Syme, D.: Types from Data: Making Structured Data First-Class Citizens in F#. SIGPLAN Not. 51, 477–490 (2016). https://doi.org/10.1145/2980983.2908115.
9. Fisher, K., Walker, D.: The PADS Project: An Overview. In: Proceedings of the 14th international conference on database theory. pp. 11–17. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1938551.1938556.
10. Fisher, K., Walker, D., Zhu, K.Q.: LearnPADS: Automatic tool generation from ad hoc data. In: SIGMOD conference (2008).
11. Hosoya, H., Pierce, B.: XDuce: A typed XML processing language. (2000).
12. Sulzmann, M., Stuckey, P.j.: HM(X) Type Inference is CLP(X) Solving. J. Funct. Program. 18, 251–283 (2008). https://doi.org/10.1017/S0956796807006569.

13. Jones, M.P.: Simplifying and Improving Qualified Types. In: Proceedings of the seventh international conference on functional programming languages and computer architecture. pp. 160–169. Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/224164.224198.

14. Imai, K., Yuen, S., Agusa, K.: A session type system with subject reduction. IEICE Transactions on Information and Systems. E95.D, 2053–2064 (2012). https://doi.org/10.1587/transinf.E95.D.2053.

15. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. Electron. Notes Theor. Comput. Sci. 171, 73–93 (2007). https://doi.org/10.1016/j.entcs.2007.02.056.

16. Barthe, G., Grégoire, B., Riba, C.: Type-based termination with sized products. In: Kaminski, M. and Martini, S. (eds.) Computer science logic. pp. 493–507. Springer Berlin Heidelberg, Berlin, Heidelberg (2008).

17. (INRIA), F.B., (INPL), C.R.: Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In: Hermann, M. and Voronkov, A. (eds.) Logic for programming, artificial intelligence, and reasoning. pp. 105–119. Springer Berlin Heidelberg, Berlin, Heidelberg (2006).

18. Le Botlan, D., Rémy, D.: MLF made simple. (2007).

19. Rémy, D., Yakobowski, B.: A church-style intermediate language for MLF. In: Proceedings of the 10th international conference on functional and logic programming. pp. 24–39. Springer-Verlag, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12251-4_4.

20. Yang, H., Reddy, U.: Imperative lambda calculus revisited, (1997).

21. Bakel, S. van: Partial intersection type assignment in applicative term rewriting systems. In: Proceedings of the international conference on typed lambda calculi and applications. pp. 29–44. Springer-Verlag, Berlin, Heidelberg (1993).

22. Luo, Z.: Coercive subtyping. J. Log. Comput. 9, 105–130 (1999). https://doi.org/10.1093/logcom/9.1.105.

23. Luo, Z., Soloviev, S., Xue, T.: Coercive subtyping: Theory and implementation. Information and Computation. 223, 18–42 (2013). https://doi.org/https://doi.org/10.1016/j.ic.2012.10.020.

24. Rothstein, H., Sutton, A., Borenstein, M.: Publication bias in meta-analysis: Prevention, assessment and adjustments. Publication Bias in Meta-Analysis. Prevention, Assessment, and Adjustments. (2005). https://doi.org/10.1002/0470870168.

25. Peyton Jones, S.: Type inference as constraint solving: How GHC's type inference engine actually works, https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/, (2019).

26. Henry P. Tung and other contributors: https://github.com/microsoft/TypeScript/issues/9825.

27. Tiuryn, J.: Subtyping over a lattice (abstract). In: Gottlob, G., Leitsch, A., and Mundici, D. (eds.) Computational logic and proof theory. pp. 84–88. Springer Berlin Heidelberg, Berlin, Heidelberg (1997).

28. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping. In: Proceedings 17th annual IEEE symposium on logic in computer science. pp. 137–146 (2002).

29. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM. 55, (2008). https://doi.org/10.1145/1391289.1391293.

30. Yodaiken, V.: C Standard undefined behaviour versus Wittgenstein, `https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/`.

31. Fairhead, H.: C Undefined Behavior - Depressing and Terrifying (Updated), `https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/`.

32. Cuoq, P., Regehr, J.: Undefined behavior in 2017, `https://blog.regehr.org/archives/1520`.

33. O'Sullivan, B.: Aeson: Fast JSON parsing and generation, `https://hackage.haskell.org/package/aeson`.

34. Siek, J., Taha, W.: Gradual typing for objects. In: Proceedings of the 21st european conference on object-oriented programming. pp. 2–27. Springer-Verlag, Berlin, Heidelberg (2007).

35. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on functional programming. pp. 268–279. ACM, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266.

36. Tiuryn, J.: Subtype inequalities. [1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science. 308–315 (1992).

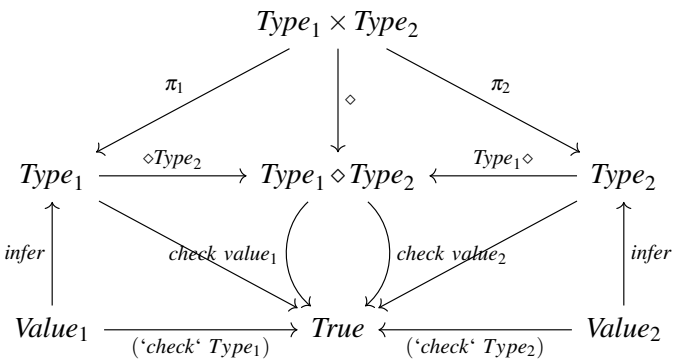
37. Grenrus, O.: Lattices: Fine-grained library for constructing and manipulating lattices, `http://hackage.haskell.org/package/lattices-2.0.2/docs/Algebra-Lattice-Levitated.html`.

38. Snoyman, M.: Generics example: Creating monoid instances, `https://www.yesodweb.com/blog/2012/10/generic-monoid`.

39. Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for haskell. In: Proceedings of the third ACM haskell symposium on haskell. pp. 37–48. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863523.1863529.

40. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: DeMillo, R.A. (ed.) Conference record of the ninth annual ACM symposium on principles of programming languages, albuquerque, new mexico, USA, january 1982. pp. 207–212. ACM Press (1982). https://doi.org/10.1145/582153.582176.

Appendix: Commutative diagram of laws for typing



9 Appendix: Inference rules for JSON union types

$$\frac{i \text{ is integral}}{\vdash_{number} \{i\} \ : \ NCInt}\ number{-}int \qquad \frac{f \text{ is not integral}}{\vdash_{number} \{f\} \ : \ NCFloat}\ number{-}float$$

$$\frac{\vdash :}{}\ number{-}merge{-}int$$

$$\frac{f \text{ is in date format}}{\vdash_{number} f \ : \ SCDate}\ string{-}date \qquad \frac{f \text{ is in email format}}{\vdash_{number} f \ : \ SCEmail}\ string{-}email$$

$$\frac{f \text{ is neither date nor email}}{\vdash_{number} f \ : \ SCEnum(\{f\})}\ string{-}enum$$

Acknowledgments

The author thanks for all tap-on-the-back donations to his past projects. Paul Tarau, Karla Ramirez Pulido, attendees of Munich Haskell Meeting and anonymous reviewers provided helpful comments on the article.

We wrote the article with the great help of bidirectional literate programming [1] tool enTangleD[2], Pandoc [3] markdown publishing system and live feedback from GHCid [4].

Less Arbitrary waiting time

Short paper

Michał J. Gajda[0000−0001−7820−3906]

Migamake Pte Ltd,
mjgajda@migamake.com,
WWW company page: https://migamake.com

Abstract. Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We prove connection between deeply recursive data structures, and epidemic growth rate, and show how to fix the problem, and make Arbitrary instances run in linear time with respect to assumed test size.

1 Introduction

Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We show that connection between deeply recursive data structures, and epidemic growth rate can be easily fixed with a generic implementation. After our intervention the Arbitrary instances run in linear time with respect to assumed test size. We also provide a fully generic implementation, so error-prone coding process is removed.

2 Motivation

Typical arbitrary instance just draws a random constructor from a set, possibly biasing certain outcomes.

Generic arbitrary instance looks like this:

```
data Tree α =
    Leaf α
  | Branch [Tree α]
  deriving (Eq,Show,Generic.Generic)
instance Arbitrary α
      ⇒ Arbitrary (Tree α) where
  arbitrary = oneof [Leaf <$> arbitrary
                    ,Branch <$> arbitrary
                    ]
```

Assuming we run QuickCheck with any size parameter greater than 1, it will fail to terminate!

List instance is a wee bit better, since it tries to limit maximum list length to a constant option:

```
instance Arbitrary α
      ⇒ Arbitrary [α] where
  lessArbitrary = sized $ λsize do
    len ← choose (1,size)
    vectorOf len lessArbitrary
```

Indeed QuickCheck manual[5], suggests an error-prone, manual method of limiting the depth of generated structure by dividing *size* by reproduction factor of the structure[1] :

```
data Tree = Leaf Int | Branch Tree Tree

instance Arbitrary Tree where
  arbitrary = sized tree'
    where tree' 0 = Leaf <$> arbitrary
      tree' ν | ν>0 =
        oneof [Leaf <$> arbitrary
          ,Branch <$> subtree <*> subtree]
        where subtree = tree' (ν `div` 2)
```

Above example uses division of size by maximum branching factor to decrease coverage into relatively deep data structures, whereas dividing by average branching factor of ~2 will generate both deep and very large structures.

This fixes non-termination issue, but still may lead to unpredictable waiting times for nested structures. The depth of the generated structure is linearly limited by dividing the ν by expected branching factor of the recursive data structure. However this does not work very well for mutually recursive data structures occuring in compilers[6], which may have 30 constructors with highly variable[2] branching factor just like GHC's HSExpr data types.

Now we have a choice of manual generation of these data structures, which certainly introduces bias in testing, or abandoning property testing for real-life-sized projects.

Another motivation is that more complex data structures like lambda terms require additional information for their generation. For example a list of free variables. That means that for generating complex structures we need to pass a state through the test case generator. It would be much more convenient to have an explicit state assigned to each generated type, so we may use type classes to generate nested data structures.

[1] We changed *liftM* and *liftM2* operators to <$> and <*> for clarity and consistency.
[2] Due to list parameters.

3 Complexity analysis

We might be tempted to compute average size of the structure. Let's use reproduction rate estimate for a single rewrite of *arbitrary* function written in conventional way.

We compute a number of recursive references for each constructor. Then we take an average number of references among all the constructors. If it is greater than 1, any **non-lazy** property test will certainly fail to terminate. If it is slightly smaller, we still can wait a long time.

What is an issue here is not just non-termination which is fixed by error-prone manual process of writing own instances that use explicit *size* parameter.

The much worse issue is unpredictability of the test runtime. Final issue is the poor coverage for mutually recursive data structure with multitude of constructors.

Given a *maximum size* parameter (as it is now called) to QuickCheck, would we not expect that tests terminate within linear time of this parameter? At least if our computation algorithms are linear with respect to input size?

Currently for any recursive structure like Tree α, we see some exponential function. For example $size^n$, where n is a random variable.

4 Solution

We propose to replace implementation with a simple state monad[7] that actually remembers how many constructors were generated, and thus avoid limiting the depth of generated data structures, and ignoring estimation of branching factor altogether.

```
newtype Cost = Cost Int
  deriving ( Eq , Ord, Enum ,Bounded,Num)
newtype CostGen σ α =
        CostGen {
         runCostGen :: State.StateT (Cost, σ) QC.Gen α
        }
  deriving (Functor, Applicative, Monad, State.MonadFix)

instance State.MonadState σ (CostGen σ) where
  state    :: ∀          σ  α .
              (σ  →     ( α ,  σ)) →  CostGen σ α
  state nestedMod      = CostGen $ State.state mod
    where
      mod :: (Cost     , σ) →  ( α   , (Cost , σ))
      mod (aCost,      aState) =   ( result  , ( aCost, newState))
          where
           (result, newState) = nestedMod aState
```

We track the spending in the usual way:

```
spend :: Cost → CostGen σ ()
spend γ = do
  CostGen $ State.modify (first (-γ+))
  checkBudget
```

To make generation easier, we introduce *budget check* operator:

```
($$$?) :: HasCallStack
       ⇒ CostGen σ α
       → CostGen σ α
       → CostGen σ α
cheapVariants $$$? costlyVariants = do
  budget ← fst ⋄ CostGen State.get
  if | budget > (0 :: Cost) → costlyVariants
     | budget > −10000      → cheapVariants
     | otherwise            → error $
       "Recursive structure with no loop breaker."

checkBudget :: HasCallStack ⇒ CostGen σ ()
checkBudget = do
  budget ← fst ⋄ CostGen State.get
  if budget < −10000
    then error "Recursive structure with no loop breaker."
    else return ()
```

In order to conveniently define our budget generators, we might want to define a class for them:

```
class StartingState σ
   ⇒ LessArbitrary σ α where
  lessArbitrary :: CostGen σ α
```

Note that starting state can default to ():

```
class StartingState σ where
  startingState :: σ

instance StartingState () where
  startingState = ()

  default lessArbitrary :: ( Generic α
                           , GLessArbitrary σ (Rep α))
                        ⇒ CostGen σ α
  lessArbitrary = genericLessArbitrary
```

Then we can use them as implementation of *arbitrary* that should have been always used:

```
fasterArbitrary :: ∀ σ α .
                   LessArbitrary σ α
                ⇒ QC.Gen α
fasterArbitrary = (sizedCost :: CostGen σ α → QC.Gen α) (lessArbitrary :: CostGen σ α)

sizedCost :: LessArbitrary σ α
          ⇒ CostGen σ α
          → QC.Gen α
sizedCost gen = QC.sized (`withCost` gen)
```

Then we can implement Arbitrary instances simply with:

```
instance _
      ⇒ Arbitrary α where
  arbitrary = fasterArbitrary
```

Of course we still need to define LessArbitrary, but after seeing how simple was a Generic defintion Arbitrary we have a hope that our implementation will be:

```
instance LessArbitrary where
```

That is - we hope that the the generic implementation will take over.

5 Introduction to GHC generics

Generics allow us to provide default instance, by encoding any datatype into its generic Representation:

```
instance Generics (Tree α) where
  to   :: Tree α → Rep (Tree α)
  from :: Rep (Tree α) → Tree α
```

The secret to making a generic function is to create a set of **instance** declarations for each type family constructor.

So let's examine Representation of our working example, and see how to declare instances:

1. First we see datatype metadata D1 that shows where our type was defined:

```
type instance Rep (Tree α) =
  D1
   (  'MetaData "Tree"
                "Test.Arbitrary"
                "less-arbitrary" 'False)
```

2. Then we have constructor metadata C1:

```
      (C1
          ('MetaCons "Leaf" 'PrefixI 'False)
```

3. Then we have metadata for each field selector within a constructor:

```
(S1
    (/MetaSel
      / Nothing
      / NoSourceUnpackedness
      / NoSourceStrictness
      / DecidedLazy)
```

4. And reference to another datatype in the record field value:

```
(Rec0 α))
```

5. Different constructors are joined by sum type operator:

```
:+:
```

6. Second constructor has a similar representation:

```
C1
    ( /MetaCons "Branch" /PrefixI /False)
    ( S1
        (/MetaSel
         / Nothing
         / NoSourceUnpackedness
         / NoSourceStrictness
         / DecidedLazy)
        ( Rec0 [Tree α ]))))
          ignored
```

7. Note that REPresentation type constructors have additional parameter that is not relevant for our use case.

For simple datatypes, we are only interested in three constructors:

- :+: encode choice between constructors
- :*: encode a sequence of constructor parameters
- M1 encode metainformation about the named constructors, C1, S1 and D1 are actually shorthands for M1 C, M1 S and M1 D

There are more short cuts to consider: * U1 is the unit type (no fields) * Rec0 is another type in the field

5.1 Example of generics

This generic representation can then be matched by generic instances. Example of Arbitrary instance from [8] serves as a basic example[3]

1. First we convert the type to its generic representation:

[3] We modified class name to simplify.

```
genericArbitrary :: ( Generic           α
                    , Arbitrary (Rep α))
                 ⇒  Gen                 α
genericArbitrary = to <$> arbitrary
```

2. We take care of nullary constructors with:

```
instance Arbitrary G.U1 where
  arbitrary = pure G.U1
```

3. For all fields arguments are recursively calling Arbitrary class method:

```
instance Arbitrary γ ⇒ Arbitrary (G.K1 i γ) where
  gArbitrary = G.K1 <$> arbitrary
```

4. We skip metadata by the same recursive call:

```
instance Arbitrary          f
      ⇒  Arbitrary (G.M1 i γ f) where
  arbitrary = G.M1 <$> arbitrary
```

5. Given that all arguments of each constructor are joined by :*:, we need to recursively delve there too:

```
instance ( Arbitrary α,
         , Arbitrary          β)
      ⇒    Arbitrary ( α G.:*: β) where
  arbitrary = (G.:*:) <$> arbitrary <*> arbitrary
```

6. In order to sample all constructors with the same probability we compute a number of constructor in each representation type with SumLen type family:

```
type family   SumLen α ::   Nat where
  SumLen ( α G.:+: β)   = ( SumLen α) + (SumLen β)
  SumLen α              = 1
```

Now that we have number of constructors computed, we can draw them with equal probability:

```
instance ( Arbitrary α
         , Arbitrary β
         , KnownNat ( SumLen α)
         , KnownNat ( SumLen β)
 )
      ⇒    Arbitrary (α G.:+: β) where
  arbitrary = frequency
    [ ( lfreq , G.L1 <$> arbitrary)
    , ( rfreq , G.R1 <$> arbitrary) ]
    where
      lfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen α))
      rfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen β))
```

Excellent piece of work, but non-terminating for recursive types with average branching factor greater than 1 (and non-lazy tests, like checking Eq reflexivity.)

5.2 Implementing with Generics

It is apparent from our previous considerations, that we can reuse code from the existing generic implementation when the budget is positive. We just need to spend a dollar for each constructor we encounter.

For the Monoid the implementation would be trivial, since we can always use *mempty* and assume it is cheap:

```
genericLessArbitraryMonoid :: ( Generic                    α
                              , GLessArbitrary σ (Rep      α)
                              , Monoid                     α )
                           ⇒    CostGen σ                  α
genericLessArbitraryMonoid =
  pure mempty $$$? genericLessArbitrary
```

However we want to have fully generic implementation that chooses the cheapest constructor even though the datatype does not have monoid instance.

Class for budget-conscious When the budget is low, we need to find the least costly constructor each time.

So to implement it as a type class GLessArbitrary that is implemented for parts of the Generic REPresentation type, we will implement two methods:

1. *gLessArbitrary* is used for normal random data generation
2. *cheapest* is used when we run out of budget

```
class GLessArbitrary σ datatype where
  gLessArbitrary :: CostGen σ ( datatype p)
  cheapest       :: CostGen σ ( datatype p)

genericLessArbitrary :: ( Generic                  α
                        , GLessArbitrary σ (Rep    α))
                     ⇒  CostGen          σ         α
genericLessArbitrary = G.to ⋄ gLessArbitrary
```

Helpful type family First we need to compute minimum cost of the in each branch of the type representation. Instead of calling it *minimum cost*, we call this function Cheapness.

For this we need to implement minimum function at the type level:

```
type family Min μ ν where
  Min μ ν = ChooseSmaller (CmpNat μ ν) μ ν

type family ChooseSmaller ( ω :: Ordering)
                          ( μ :: Nat)
                          ( ν :: Nat) where
  ChooseSmaller 'LT μ ν = μ
  ChooseSmaller 'EQ μ ν = μ
  ChooseSmaller 'GT μ ν = ν
```

so we can choose the cheapest^[We could add instances for :

```
type family Cheapness α :: Nat where
  Cheapness (α :*: β) =
      Cheapness α + Cheapness β
  Cheapness (α :+: β) =
    Min (Cheapness α) ( Cheapness β)
  Cheapness U1 = 0
  ≪ flat-types ≫
  Cheapness ( K1 α other) = 1
  Cheapness ( C1 α other) = 1
```

Since we are only interested in recursive types that can potentially blow out our budget, we can also add cases for flat types since they seem the cheapest:

```
Cheapness ( S1 α ( Rec0 Int ) ) = 0
Cheapness ( S1 α ( Rec0 Scientific) ) = 0
Cheapness ( S1 α ( Rec0 Double ) ) = 0
Cheapness ( S1 α ( Rec0 Bool ) ) = 0
Cheapness ( S1 α ( Rec0 Text.Text ) ) = 1
Cheapness ( S1 α ( Rec0 other ) ) = 1
```

Of course we could also try to narrow generation of data structures by their size, but that would complicate the code here, and also it would only work for regular data structures that are not greatly affected by the passed state.

Base case for each datatype For each datatype, we first write a skeleton code that first spends a coin, and then checks whether we have enough funds to go on expensive path, or we are beyond our allocation and need to generate from among the cheapest possible options.

```
instance GLessArbitrary σ f
      ⇒ GLessArbitrary σ (D1 μ f) where
  gLessArbitrary = do
    spend 1
    M1 <$> ( cheapest $$$? gLessArbitrary)
  cheapest = M1 <$> cheapest
```

Skipping over other metadata First we safely ignore metadata by writing an instance:

```
instance GLessArbitrary σ           f
      ⇒  GLessArbitrary σ (G.C1 γ   f) where
  gLessArbitrary = G.M1  ⍚ gLessArbitrary
  cheapest       = G.M1  ⍚ cheapest

instance GLessArbitrary σ           f
      ⇒  GLessArbitrary σ (G.S1 γ   f) where
  gLessArbitrary = G.M1  ⍚ gLessArbitrary
  cheapest       = G.M1  ⍚ cheapest
```

Counting constructors In order to give equal draw chance for each constructor, we need to count number of constructors in each branch of sum type :+: so we can generate each constructor with the same frequency:

```
type family  SumLen α :: Nat where
  SumLen (α G.:+: β)  = SumLen α + SumLen β
  SumLen  α           = 1
```

Base cases for GLessArbitrary Now we are ready to define the instances of GLessArbitrary class.

We start with base cases GLessArbitrary for types with the same representation as unit type has only one result:

```
instance GLessArbitrary σ G.U1 where
  gLessArbitrary = pure    G.U1
  cheapest       = pure    G.U1
```

For the product of, we descend down the product of to reach each field, and then assemble the result:

```
instance (GLessArbitrary σ α
         ,GLessArbitrary σ          β)
      ⇒   GLessArbitrary σ (α G.:*: β) where
  gLessArbitrary = (G.:*:)    ⍚    gLessArbitrary
                              ⍟    gLessArbitrary
  cheapest       = (G.:*:)    ⍚    cheapest
                              ⍟    cheapest
```

We recursively call instances of LessArbitrary for the types of fields:

```
instance LessArbitrary  σ          γ
      ⇒ GLessArbitrary σ (G.K1 i γ) where
  gLessArbitrary = G.K1  ⍚ lessArbitrary
  cheapest       = G.K1  ⍚ lessArbitrary
```

Selecting the constructor We use code for selecting the constructor that is taken after[8].

```
instance  ( GLessArbitrary σ α
          , GLessArbitrary σ β
          , KnownNat ( SumLen α)
          , KnownNat ( SumLen β)
          , KnownNat ( Cheapness α)
          , KnownNat ( Cheapness β)
)
      ⇒  GLessArbitrary σ (α Generic.:+: β) where
  gLessArbitrary =
    frequency
      [ (  lfreq  , L1   ⋄ gLessArbitrary)
      , (  rfreq  , R1   ⋄ gLessArbitrary) ]
    where
      lfreq  =  fromIntegral
             $  natVal (Proxy :: Proxy (SumLen α))
      rfreq  =  fromIntegral
             $  natVal (Proxy :: Proxy (SumLen β))
  cheapest =
    if   lcheap                ≤ rcheap
         then L1 ⋄ cheapest
         else R1 ⋄ cheapest
    where
      lcheap ,   rcheap :: Int
      lcheap  =  fromIntegral
             $  natVal (Proxy :: Proxy (Cheapness α))
      rcheap  =  fromIntegral
             $  natVal (Proxy :: Proxy (Cheapness β))
```

6 Conclusion

We show how to quickly define terminating test generators using generic programming. This method may be transferred to other generic programming regimes like Featherweight Go or Featherweight Java.

We recommend because it reduces time spent on making test generators and improves user experience when a data structure with no terminating constructors is defined.

7 Bibliography

1. Knuth, D.E.: Literate programming. Comput. J. 27, 97–111 (1984). https://doi.org/10.1093/comjnl/27.2.97.
2. Hidding, J.: enTangleD: A bi-directional literate programming tool, [Link].
3. McFarlane, J.: Pandoc: A universal document converter, [Link].

4. FP Complete: stack 0.1 released.

5. John Hughes: QuickCheck: An Automatic Testing Tool for Haskell, [Link].

6. Day, L.E., Hutton, G.: Compilation à la Carte. In: Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, Nijmegen, The Netherlands (2013).

7. Jones, M.P., Duponcheel, L.: Composing monads. (1993).

8. contributors, Typeable. io: generic-arbitrary: Generic implementation for QuickCheck's Arbitrary, [Link].

9. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of haskell programs. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on functional programming. pp. 268–279. ACM, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266.

10. Kerckhove, T.S.: genvalidity-property: Standard properties for functions on 'Validity' types, [Link].

Appendix: Module headers

```
{-# language DefaultSignatures #-}
{-# language FlexibleInstances #-}
{-# language FlexibleContexts #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language InstanceSigs #-}
{-# language Rank2Types #-}
{-# language PolyKinds #-}
{-# language MultiParamTypeClasses #-}
{-# language MultiWayIf #-}
{-# language ScopedTypeVariables #-}
{-# language TypeApplications #-}
{-# language TypeOperators #-}
{-# language TypeFamilies #-}
{-# language TupleSections #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
{-# language DataKinds #-}
module Test.LessArbitrary(
    LessArbitrary(..)
  , oneof
  , choose
  , budgetChoose
  , CostGen (..)
  , (<$$$>)
  , ($$$?)
  , currentBudget
  , fasterArbitrary
  , genericLessArbitrary
  , genericLessArbitraryMonoid
  , flatLessArbitrary
  , spend
  , withCost
  , elements
  , forAll
  , sizedCost
  , StartingState(..)
) where

import qualified Data.HashMap.Strict              as Map
import qualified Data.Set                         as Set
import qualified Data.Vector                      as Vector
import qualified Data.Text                        as Text
import           Control.Monad (replicateM)
import           Data.Scientific
import           Data.Proxy
import qualified Test.QuickCheck.Gen              as QC
import qualified Control.Monad.State.Strict       as State
import           Control.Arrow (first, second)
import           Control.Monad.Trans.Class
import           System.Random (Random)
import           GHC.Generics     as G
import           GHC.Generics     as Generic
import           GHC.TypeLits
import           GHC.Stack
import qualified Test.QuickCheck as QC
```

Appendix: lifting classic Arbitrary functions

Below are functions and instances that are lightly adjusted variants of original implementations in QuickCheck[9]

```
instance LessArbitrary σ α
      ⇒ LessArbitrary σ [α] where
  lessArbitrary = pure [] $$$? do
    budget ← currentBudget
    len    ← choose (1,fromEnum budget)
    spend $ Cost len
    replicateM len lessArbitrary

instance (QC.Testable α
         ,LessArbitrary σ α)
      ⇒ QC.Testable (CostGen σ α) where
  property = QC.property
           . sizedCost
```

Remaining functions are directly copied from QuickCheck[9], with only adjustment being their types and error messages:

```
forAll :: CostGen σ α → ( α → CostGen σ β) → CostGen σ β
forAll gen prop = gen >>= prop

oneof :: HasCallStack
      ⇒ [CostGen σ α ] → CostGen σ α
oneof [] = error
           "LessArbitrary.oneof used with empty list"
oneof gs = choose (0,length gs - 1) >>= (gs !!)

elements :: [ α ] → CostGen σ α
elements gs = ( gs!!) ⍟ choose (0,length gs - 1)

choose :: Random α
       ⇒ ( α , α)
       → CostGen σ α
choose (α,β) = CostGen $ lift $ QC.choose (α, β)

– | Choose but only up to the budget (for array and list sizes)
budgetChoose :: CostGen σ Int
budgetChoose = do
  Cost β ← currentBudget
  CostGen $ lift $ QC.choose (1, β)

– | Version of ‘suchThat‘ using budget instead of sized generators.
cg ‘suchThat‘ pred = do
  result ← cg
  if pred result
    then return result
    else do
      spend 1
      cg ‘suchThat‘ pred
```

This key function, chooses one of the given generators, with a weighted random distribution. The input list must be non-empty. Based on QuickCheck[9].

```
frequency        ::  HasCallStack
                 ⇒  [(Int, CostGen σ α)]
                 →          CostGen σ α
frequency []  =
  error $  "LessArbitrary.frequency "
        ++ "used with empty list"
frequency xs
  | any (<  0) (map fst xs) =
    error $  "LessArbitrary.frequency: "
          ++ "negative weight"
  | all (== 0) (map fst xs) =
    error $  "LessArbitrary.frequency: "
          ++ "all weights were zero"
frequency xs0 = choose (1, tot) >>= ('pick' xs0)
 where
  tot  = sum (map fst xs0)

  pick ν ( ( κ ,x): xs)
    |  ν ≤ κ    = x
    |  otherwise = pick (ν-κ) xs
  pick _ _ = error
    "LessArbitrary.pick used with empty list"
```

Appendix: test suite

As observed in [10], it is important to check basic properties of Arbitrary instance to guarantee that shrinking terminates:

```
shrinkCheck ::  ∀          term .
             ( Arbitrary  term
             , Eq         term)
            ⇒             term
            →  Bool
shrinkCheck term =
  term `notElem` shrink term

arbitraryLaws ::  ∀          ty .
               (  Arbitrary  ty
               ,  Show       ty
               ,  Eq         ty)
               ⇒  Proxy      ty
               →  Laws
arbitraryLaws (Proxy :: Proxy ty) =
  Laws "arbitrary"
       [( "does not shrink to itself",
          property (shrinkCheck :: ty → Bool))]
```

For LessArbitrary we can also check that empty budget results in choosing a cheapest option, but we need to provide a predicate that confirms what is actually the cheapest:

```
otherLaws :: [Laws]
otherLaws = [lessArbitraryLaws @() isLeaf]
  where
    isLeaf :: Tree Int → Bool
    isLeaf (Leaf _) = True
    isLeaf (Branch _) = False

lessArbitraryLaws :: ∀ σ α .
                     LessArbitrary σ α
                  ⇒ (α → Bool)
                  → Laws
lessArbitraryLaws cheapestPred =
    Laws "LessArbitrary"
         [("always selects cheapest",
           property $
           (prop_alwaysCheapest @σ @α) cheapestPred)]

prop_alwaysCheapest :: ∀ σ α.
                       LessArbitrary σ α
                    ⇒ (α → Bool)
                    → Gen Bool
prop_alwaysCheapest cheapestPred =
  cheapestPred =<< (withCost @σ @α) 0 lessArbitrary
```

Again some module headers:

```
{-# language DataKinds #-}
{-# language FlexibleInstances #-}
{-# language Rank2Types #-}
{-# language MultiParamTypeClasses #-}
{-# language ScopedTypeVariables #-}
{-# language TypeOperators #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
module Test.Arbitrary.Laws(
      arbitraryLaws
)     where

import Data.Proxy
import Test.QuickCheck
import Test.QuickCheck.Classes
import qualified Data.HashMap.Strict as Map
import Data.HashMap.Strict (HashMap)
```

≪ *arbitrary-laws* ≫

```
{-# language DataKinds #-}
{-# language FlexibleInstances #-}
{-# language Rank2Types #-}
{-# language MultiParamTypeClasses #-}
{-# language ScopedTypeVariables #-}
{-# language TypeApplications #-}
{-# language TypeOperators #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
module Test.LessArbitrary.Laws(
      lessArbitraryLaws
 )    where

import  Data.Proxy
import  Test.QuickCheck(Gen, property)
import  Test.QuickCheck.Classes(Laws(..))
import  Test.LessArbitrary
import  qualified Data.HashMap.Strict as Map
import            Data.HashMap.Strict (HashMap)

≪ less-arbitrary-laws ≫
```

And we can compare the tests with LessArbitrary (which terminates fast, linear time):

```
≪ test-file-header ≫
≪ test-less-arbitrary-version ≫

≪ test-file-laws       ≫
≪ less-arbitrary    -check ≫
```

Appendix: non-terminating test suite

Or with a generic Arbitrary (which naturally hangs):

```
≪ test-file   -header ≫
≪ tree-type-typical-   arbitrary ≫
otherLaws =  []
≪ test-file   -laws ≫
```

Here is the code:

```
{-# language FlexibleInstances #-}
{-# language InstanceSigs #-}
{-# language Rank2Types #-}
{-# language MultiParamTypeClasses #-}
{-# language ScopedTypeVariables #-}
{-# language TypeApplications #-}
{-# language TypeOperators #-}
{-# language UndecidableInstances #-}
{-# language AllowAmbiguousTypes #-}
{-# language DeriveGeneric #-}
module Main where

import           Data.Proxy
import           Test.QuickCheck
import qualified Test.QuickCheck.Gen as QC
import qualified GHC.Generics as Generic
import           Test.QuickCheck.Classes

import           Test.LessArbitrary
import           Test.Arbitrary.Laws
import           Test.LessArbitrary.Laws

≪ tree-type ≫

instance ( Arbitrary                 α
         , LessArbitrary σ           α)
      ⇒   LessArbitrary σ (Tree      α) where
  lessArbitrary = genericLessArbitrary

instance ( LessArbitrary () (Tree    α)
         , Arbitrary                 α)
      ⇒  Arbitrary (Tree             α)  where
  arbitrary = fasterArbitrary @()       @(Tree α)
  shrink    = recursivelyShrink


main :: IO ()
main = do
  lawsCheckMany
    [("Tree",
      [  arbitraryLaws (Proxy :: Proxy (Tree Int))
      ,  eqLaws        (Proxy :: Proxy (Tree Int))
      ] ◇ otherLaws)  ]
```

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Test.LessArbitrary.Cost where

≪ cost ≫
```

Appendix: convenience functions provided with the module

Then we limit our choices when budget is tight:

```
currentBudget :: CostGen σ Cost
currentBudget = fst <$> CostGen State.get
– unused: loop breaker message type name
– FIXME: use to make nicer error message
type family   ShowType κ where
  ShowType ( D1 (t MetaData name _ _ _) _)   = name
  ShowType   other                           = "unknown type"

showType ::   ∀                              α .
              ( Generic                      α
              , KnownSymbol (ShowType (Rep α)))
         ⇒    String
showType =    symbolVal (Proxy :: Proxy (ShowType (Rep α)))
```

Towards Incremental Language Definition with Reusable Components

Damian Frölich[1] and L. Thomas van Binsbergen[2][0000−0001−8113−2221]

[1] Informatics Institute, University of Amsterdam, The Netherlands
dfrolich@acm.org
[2] Informatics Institute, University of Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Abstract. This paper introduces a novel method for defining software languages incrementally as the composition of smaller languages, starting from reusable components for the specification of syntax and semantics. The method is enabled by the combined application of several advanced techniques implemented in functional languages: datatypes à la carte for the fine-grained composition of (abstract) syntactic categories and composable micro-interpreters that implement the operational semantics of certain reusable components known as 'funcons'. We demonstrate the method makes it possible to perform incremental language development with prototyping. The generality of the method is demonstrated through a variety of case studies.

Keywords: software language engineering, language composition, syntax, semantics, interpretation

1 Introduction

Incremental programming is a style of programming in which software is built in a step-by-step fashion by submitting code fragments that, for example, declare a single type or execute a single statement with immediate feedback on the validity and effect of the code fragment. This style of programming is naturally supported by read-eval-print-loop (REPL) interpreters (also referred to as interactive shells) such as JShell and IPython and computational notebooks such as Jupyter [9] and Mathematica [7]. In the context of software engineering, a common usage of a REPL is to test a library under development by loading its latest version and interacting with the functions it defines, perhaps in combination with functions from other libraries (under development). In the context of data science, a common usage of a computational notebook is the simultaneous development and testing of a scientific workflow. In the context of language-oriented programming [17], an incremental programming environment should support the definition and testing of language constructs (akin to library functions), also in combination with the constructs of other languages, by extending language definitions and by running test programs written in the language(s)

currently under construction. However, there are various challenges to realising such a system. For example, simultaneously extending the syntax and semantics of a language definition without modifying or recompiling existing parts are requirements of solutions to the well-documented expression problem coined by Wadler [16]. As another example, the composition of two deterministic context-free grammars may produce a non-deterministic context-free grammar to which the parsing technology of choice is not applicable or which results in ambiguities that need to be resolved.

In this paper we experiment with an approach to incremental language development enabled by certain functional techniques for modular language specification: data types à la carte [15] and micro-interpreters implementing the operational semantics of a reusable library of fundamental programming languages constructs known as 'funcons' [2]. The funcons of the Funcons-beta library [13] are used as a common base language on top of which object languages are defined. The practicality of this approach is to be evaluated in this paper.

The full version of this paper is to contribute by:

- Presenting a novel approach for incremental language definition with reusable components
- Presenting an implementation of the approach as a framework consisting of existing Haskell EDSLs resulting from recent advances in modular language specification techniques
- Demonstrating the applicability and generality of the approach through various case studies and in comparison with existing approaches to language extension and unification

2 Background

The initial algebra semantics of Goguen et al. [6], concisely described by Mosses in [11], provides important formal foundations and terminology to our work, as it can be seen to capture the essential elements of many existing semantic specification formalisms such as denotational semantics and attribute grammar semantics. In initial algebra semantics, a multi-sorted signature lays out the operations of a language. Abstract syntax is defined by assigning term constructors to the operations (as an initial algebra) according to their structure, as determined by the signature. Semantics are defined by assigning 'semantic domains' to sorts and functions to operations (within an evaluation algebra) that map the terms (within their respective semantic domains) computed for operands to the value describing the result of applying the operation to these operands.

As a solution to the expression problem [16], *data types à la carte* [15] provides a method for assembling data types and functions from individual components to form signatures and initial algebras, and evaluation algebras respectively. With this technique, signatures of independent languages can be freely composed, enabling mixing of syntactic constructs of different languages. The approach achieves this by implementing signatures as functors, combining via

their coproduct and using the fix-point of functors to tie the recursive knot. Using type-classes, automatic injections into the coproduct can be achieved, which combined with smart constructors provide a natural way to construct values of the coproduct type. To assign semantics, catamorphisms are used with the underlying algebra implemented via type-classes.

The comp-data library [1] provides a comprehensive Haskell library implementing the data types à la carte approach with some extensions, including support for generalised algebraic data types, contexts, automatic deriving of several type-class instances using template Haskell, and more. The library also supports usage of higher-order functors [8] to implement signatures. A consequence of using higher-order functors is that algebras become natural transformations instead of functions, which affects the kind of the algebra.

The component-based approach to operational semantics presented in [12] is centred around reusable definitions of the fundamental constructs of programming – *funcons* for shorts. As explained in [2], 'micro-interpreters' can be generated from funcon definitions. The micro-interpreters are compositional evaluation functions expressing the behaviour of an individual funcon that can be generated and compiled separately. In this paper we leverage the generality of the Funcons-beta [13] library to be able to express the semantics of various languages in a shared base language, applying the micro-interpreters generated for funcons as the constructs of an EDSL.

By combining the described techniques, language extensions and compositions can be written in a highly modular fashion, permitting rapid prototyping. Via the method proposed in [3] and implemented in [5], REPLs for the resulting languages can be obtained with minimal effort.

Erdweg et al. provide a framework for discussing and comparing meta-languages, tools and formalisms that support various form of incremental language development [4]. In particular, the authors define the concepts of (modular) language extension, restriction, and unification which they apply to both the syntax, static semantics, operational semantics and IDE services of languages. In this paper we adopt their terminology and use their framework as the basis for our evaluation.

Template Haskell is a Haskell extension permitting compile-time meta-programming [14]. With Template Haskell users can write programs that transform programs and is useful, for instance, when generating boilerplate code or to perform calculations at compile-time instead of run-time for performance reasons. The extension provides several facilities to inspect and operate on Haskell programs, including a quotation monad that enables reification of Haskell constructs which gives the user access to the internal represenation of the compiler.

3 Incremental definitions

This section describes this paper's approach to language development conceptually, with an implementation strategy (in Haskell) described in Section 4. Essential to the approach is the separation between operator (or language construct) definitions on the one hand and language definitions on the other hand. A lan-

guage definition can freely choose from the available operators and constrains the flexibility with which the chosen operators can be used. The definition of an operator consists of an abstract syntax definition and a denotational semantics, choosing funcon terms as a semantic domain. The separation between operator and language definitions is enabled by an alternative take on abstract syntax definitions.

3.1 Abstract syntax

A common approach to defining the abstract syntax of a language is listing algebraic datatypes (ADTs) of which the operator[3] signatures determine, in a mutually recursive fashion, the set of terms that forms the abstract syntax of the language. For example, the abstract syntax of a lambda calculus can be represented as follows, where Var_O, Abs_O, and App_O are operators (as indiciated by the subscript) and *String* and *Expr* are sorts.

$$\begin{aligned} Var_{\mathcal{O}} &: String \to Expr \\ Abs_{\mathcal{O}} &: String \times Expr \to Expr \\ App_{\mathcal{O}} &: Expr \times Expr \to Expr \end{aligned}$$

In this style, the signature of an operator simultaneously identifies the sort of term constructed by applications of the operator, the arity of the operator, and the sort of term required at each operand position in valid applications of the operator.

A key insight of our approach is to delay the decisions related to sorts (but not arity) until the definition of a language, rather than making these part of operator definitions. This is achieved by (1) using a unique sort at every position in the signature and by (2) introducing separate *sort constraints* to establish the relations between sorts. Following (1), the sorts are effectively naming operand positions. The right-hand side of a signature is made redundant and can be removed, as every operator already has a unique name. With these changes, the operators are defined as follows:

$$\begin{aligned} Var_{\mathcal{O}} &: VarVar \\ Abs_{\mathcal{O}} &: AbsVar \times AbsBody \\ App_{\mathcal{O}} &: AppAbs \times AppArg \end{aligned}$$

In contrast to the conventional approach, the signatures do not share any sorts, and the three operators are completely unrelated. To re-establish the relationships, we introduce sort constraints. Sort constraints are based on the interpretation of sorts as sets of operators. For example, the following sort constraint indicates that strings serve as identifiers in both variable references and abstractions:

$$\begin{aligned} String &\subseteq VarVar \\ String &\subseteq AbsVar \end{aligned}$$

[3] Such as *constructors* in Haskell and *variants* in the ML family of languages

This kind of sort constraint is referred to as a *sub-sort declaration*.

The other kind of sort constraint, referred to as an *operator assignment*, indicates that terms constructed by the $Var_{\mathcal{O}}$ operator can be used as the body of an abstraction:

$$Var_{\mathcal{O}} \in AbsBody$$

To express the same relations between the operators as in the initial example, operator assignments can be written for every pair of an operator and sort taken from the sets $\{Var_{\mathcal{O}}, Abs_{\mathcal{O}}, App_{\mathcal{O}}\}$ and $\{AbsBody, AppAbs, AppArg\}$. Writing down these operator assignments grows increasingly tedious (and error-prone) as more and more operators are added to a language. Therefore, as a convenience, sort constraints can also be used to introduce new sorts that serve as a level of indirection and enable reuse. The following sort constraints (re-)introduce the sort *Expr* as a convenience, stating that all operators assigned to *Expr* are also assigned to *AbsBody*, *AppAbs* and *AppArg*:

$$Expr \subseteq AbsBody$$
$$Expr \subseteq AppAbs$$
$$Expr \subseteq AppArg$$

The relations of the original example are then expressed by assigning the operators to *Expr*:

$$Var_{\mathcal{O}} \in Expr$$
$$App_{\mathcal{O}} \in Expr$$
$$Abs_{\mathcal{O}} \in Expr$$

By separating operator definitions from constraints on where operators can be used, a new operator can be introduced without modifying existing ones. For example, extending the lambda calculus with integer addition can be achieved by defining an *Add* operator and assigning this operator to the sorts where we want to use the *Add* operator.

$$Add_{\mathcal{O}} : AddLeft \times AddRight$$
$$Add_{\mathcal{O}} \in Expr$$

This definition adds $Add_{\mathcal{O}}$ to *Expr*, such that the *Add* operator can be used at the operand positions over which we distributed *Expr* earlier. Interestingly, no operators have been assigned to the operands of the *Add* operator yet. Consider the following sort constraints:

$$Integer \in AddLeft$$
$$Integer \in AddRight$$
$$Integer \subseteq Expr$$
$$Add_{\mathcal{O}} \in AddRight$$

These constraints express that integer literals can appear as operands of *Add* in both positions. However, since the *Add* operator is only added to *AddRight*, the constraints allow only nested occurrences of *Add* on the right side, encoding right-associativity. This example demonstrates the flexibility of sort constraints: integer expressions can be used in lambda-expressions — owing to the constraints $Add_{\mathcal{O}} \in Expr$ and $Integer \subseteq Expr$ — whereas lambda-expessions cannot be used in integer expressions. Such rules of composition can be changed simply by selecting a different set of sort constraints without affecting the definitions of the operators themselves. As discussed in §3.4, finalising a selection of sort constraints is done as part of the definition of a language.

3.2 Incremental semantics

To retain the disjoint property of the operators, their semantics must be defined independently as well. This is achieved by utilising semantic functions. Semantic functions translate an operator into a specific semantic domain. For example, our previous operators defining the lambda calculus can have the following semantic functions, with funcons being our semantic domain[4].

$$\begin{aligned} Var_{\mathcal{F}}(lit) &= \textbf{bound string } lit \\ Abs_{\mathcal{F}}(x, b) &= \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), b) \\ App_{\mathcal{F}}(abs, arg) &= \textbf{apply}(abs, arg) \end{aligned}$$

When a translation occurs, the operands of an operator are already translated by their respective translation function. Hence, an operator only needs to translate itself into the semantic domain while having access to the already translated operands.

3.3 Glueing components together

In certain circumstances it may be necessary to adapt language fragments in order to make them compatible for composition. So-called 'glue code' is often used in these circumstances, being applied at the location where two fragments interact. This glue code is to be written modularly, and such that both fragments can be defined in isolation, without anticipating, or constraining, future interactions. To make these observations more concrete, consider the following example in which the *Abs* operator is combined with commands in order to describe procedures that can terminate abruptly by returning a value. In funcons, a return is implemented via a signal that is to be handled at the call-site. As given previously, the translation function for the *Abs* operator is such that the operator does not handle return signals. To handle return signals, the following definition should have been given instead:

$$Abs_{\mathcal{F}}(x, b) = \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), \textbf{handle-return}(b))$$

[4] In the right-hand side, juxtaposition is the right-associative application of a funcon to a (single) funcon term, i.e. **bound string** *lit* == **bound**(**string**(*lit*)).

However, we would like to be able use the original definition instead, in which this usage of *Abs* was not anticipated.

The solution we apply is to associate glue code with the sort constraint that enables commands (forming the body of a procedure) in the body of abstractions:

$$\begin{aligned} &\mathit{Command} \subseteq \mathit{AbsBody} && \text{(Sort constraint with glue code)} \\ &\quad \hookrightarrow \textsf{handle-return}(\mathit{Command}_{\mathcal{F}}) && \text{(glue code)} \end{aligned}$$

$\mathit{Command}_{\mathcal{F}}$ refers to the result of the translation function associated with the sort *Command*, which is implicitly defined in terms of the translation functions given for the operators contained in the sort *Command*, i.e. the free homomorphism between initial and semantic algebra. The result is that the glue code is applied to all commands whenever they appear at the position where an *AbsBody* is expected, i.e. as the second operand of the *Abs* operator. The original translation for *Abs* is thus correct, as the funcon term bound by b now handles return signals whenever the translation is applied to an abstraction with a command as a body.

The following definitions complete the example by determining that the return operator ($\mathit{Return}_{\mathcal{O}}$) is the only operator in the sort *Command* and that its translation simply applies the **return** funcon to returned value:

$$\begin{aligned} \mathit{Return}_{\mathcal{O}} &: \mathit{ReturnVal} && \text{(Operator declaration)} \\ \mathit{Return}_{\mathcal{F}}(\mathit{val}) &= \textsf{return}\ \mathit{val} && \text{(Semantic function)} \\ \mathit{Return}_{\mathcal{O}} &\in \mathit{Command} && \text{(Sort constraint)} \end{aligned}$$

3.4 Language definition

A language is defined as a structure $\langle O, O_t, S_c, S_s, G_o, G_t \rangle$, with O being the set of operators, O_t the operators at the top-level with $O_t \subseteq O$, S_c the sort-constraints of the language — assigning operators to operand positions — S_s the sub-sort declarations (i.e. $S_c \cup S_s$ contains all sort constraints), G_o the set of operator glue code, and G_t is the glue code for the top-level operators. The G_t component provides an extra glue layer that is applied over the full program and can be used to perform initialisation, handle uncaught exceptions, etc. The top-level operators of a language determine the entry points of the language — i.e., the root of an AST is always a top-level operator.

4 Implementation

In this section we demonstrate an implementation in Haskell of the introduced approach. The section follows the same layout as the previous section.

4.1 Operators

In order to delay sort decisions and describe relations via constraints, operators are implemented as GADTs with two type parameters, u and t, corresponding to the universe and the meta-type of the operator, respectively.

```
data Operator^p [5] u t where
  Operator^p :: Sorts^p ⇒ Arguments^p → Operator^p u Operator^p Type
data Operator^p Type
```

The universe contains all the operators in the language and is needed for automatic injections into the coproduct type. The meta-type is used to differentiate between different operators after injection into the universe and is implemented as an empty data type (with no values and constructors).

The sorts in an operator signature are implemented as type families such that an operator is assigned to the sort when its meta-type is an instance of the family that evaluates to the type-level boolean *True*. The arity of an operator is determined by its arguments. In an argument definition, arguments are linked to a sort defined in the sorts definition. For example, *Abs* can be defined as follows.

```
data Abs u t where
  Abs :: IsTrue (AbsBody t) ⇒ String → u t → Abs u AbsType
data AbsType
type family AbsBody t
```

The sorts definition links the *AbsBody* sort the second argument via the meta-type of the second argument — identified by *t*. To assign an operator to the body of abstractions, we define the meta-type of the operator as an instance of the *AbsBody* type class that evaluates to *True*. To enable adaptibility, we delay the instantation of such an instance via a template Haskell call. Thus, to allow abstractions inside abstractions, we define the *AbsType* as an instance of the *AbsBody* type class:

```
$ (genSortConstraint [('' AbsBody '' AbsType)])
```

which evaluates to

```
type instance AbsBody AbsType = True
```

and is the encoding for $Abs_{\mathcal{O}} \in AbsBody$.

Indirection sorts are also defined as type families:

```
type family Expr t
```

and adding an instance to the *Expr* sort is identical as assigning an instance to a sort that exists in an operator definition. However, to allow elemens of the *Expr* sort to occur inside the *AbsBody*, we need to copy over the instances of the *Expr* sort to the *AbsBody* sort. To achieve this, we again utilise template Haskell such that a user only has to define the relation. For instance, to link the elements of the *Expr* sort to the *AbsBody*, we perform the following template Haskell call.

```
$ (genSubSort [('' Expr, '' AbsBody)])
```

[5] Values with the p superscript denote placeholders that are filled in by an operator definition.

The *genSubSort* function takes as arguments a list of tuples, with the first element the source of the relation and the second element the target of the relation. After this call, the target sort contains all elements also present in the source sort. The arguments to this call thus describe the edges in a directed graph, which must be acyclic.

4.2 Semantic functions

Inheriting from data types à la carte, semantic functions are algebra's implemented via type classes and applied via a fold. Because of the type classes, instances can be defined in isolation, enabling modularity in the definition of semantic functions. The definition for the semantic function with funcons as the semantic domain is as follows.

```
class HFunctor f ⇒ ToFuncons f where
  toFuncons :: Alg f (K Funcons)
```

Since the implementation uses higher-order functors, the second parameter needs to be of kind $* \to *$, which requires wrapping funcons, which have kind $*$, in a wrapper functor — identified by the K constructor.

By defining an instance of the *ToFuncons* type class, an operator defines its translation to the semantic domain of funcons. For example, the following instance encodes the definition of $Abs_{\mathcal{F}}(x, b)$ given in subsection 3.2.

```
instance ToFuncons Abs where
  toFuncons (Abs s (K body)) = K $
    function_ [ closure_ [ scope_ [ bind_ [T.string_ [6] s, given_], body]]]
```

4.3 Glue code

Glue code is implemented as functions of type *operator* $(K\ Funcons)\ t \to$ *operator* $(K\ Funcons)\ t$ and is executed before the translation of the operator but after the translation of the operands. Hence, glue code can not alter the operator but only the operands.

Internally, these functions are generated into type class instances via a template Haskell call. So, the glue definition from section 3 is achieved with the following template Haskell call:

```
$ (genGlue (' Abs, ' absGlueReturn))
absGlueReturn e@(Abs s body) =
  if unK body `G.contains` return_ [G.matchhole_] then Abs s glued_body else e
```

[6] The T module provides helper functions to transform Haskell values into funcon values. Funcon smart constructors — identified by the trailing underscore — take a variable number of arguments, hence the usage of lists.

```
  where
    glued_body = hfkmap ( handle_return_ ∘ (:[])) body
```

which is generated into the following instance declaration.

```
instance GlueFuncon Abs where
  glueFuncon e = absGlueReturn e
class HFunctor f ⇒ PostFuncon f where
  postFuncon :: f (K Funcons) t → f (K Funcons) t
  postFuncon = id
```

When no glue code is specified for an operator, the identity function is used.

The *absGlueReturn* function checks whether the body operand contains a return statement and if so it modifies the body by wrapping it inside a **handle-return**. To modify the body, the *hfkmap* $:: (a \to b) \to K\ a\ t \to K\ b\ t$ function that maps over a higher-order functor without modifying the second paramater is used. This ensures the reconstruction of the operator with modified operands is type correct.

Glue code functions can use a match library provided by us — recognised by the *G* prefix — to apply glue code conditionally. The match library provides functionality to search for (partial) funcon terms inside other funcon terms. Partiallity is achieved by introducing a new funcon (*G*.**matchhole_**) that matches all other funcon terms. Nonetheless, the matching library does not require usage of the new funcon, glue code can also match on specific funcons terms or be applied unconditionally.

4.4 Language definition

The language definition is implemented as a data type defined in terms of Template Haskell constructs.

```
data Language = Language
  { operators :: [Operator]
  , op_constr :: [Constraint]
  , sub_sorts :: [SubSort]
  , glue_code :: [(Constructor, GlueFunction)]
  , top_glue  :: Maybe GlueFunction
  } deriving (Show)
type Constructor = Name
type MetaType = Name
type Sort = Name
type Operator = (Constructor, MetaType)
type Constraint = (MetaType, Sort)
type GlueFunction = Exp
type OperatorAssignment = (MetaType, Sort)
type SubSort = (Sort, Sort)
```

An operator is uniquely identified by its constructor and meta-type; the operator constraints of the language are given as a list of tuples as expected by *genSortConstraint*; sub-sort constraints are given as a list of tuples as expected by *genSubSort*; glue code is defined as a list of tuples as expected by *genGlue*; and a language can define an optional glue code function for the top-level sort.

The language definition does not contain a special entry for top-level operators. Instead, we utilise a special sort — *TopLevel* — that can be used inside the sort constraint and sub-sort definitions.

To instantiate a language, the *genLanguage* Template Haskell function is used. This function is defined in terms of the earlier introduced Template Haskell functions.

$$genLanguage :: Language \rightarrow Q\ [Dec]$$

The *genLanguage* function generates the sort constraint instances, generates smart constructors that automatically inject the operator into the coproduct type, and the glue code definitions are transformed into type class instances.

Our original expression language can thus be generated via the following call.

```
$ (genLanguage expressionLanguage
  where
    expressionLanguage = Language
      { operators = [('' Var, '' VarType), ('' Abs, '' AbsType), ('' App, '' AppType)]
      , op_constr = [(op, '' Expr) | op ← ['' VarType, '' AbsType, '' AppType]]
      , sub_sorts = [('' Expr, t) | t ← ['' AbsBody, '' AppLeft, '' AppRight, '' TopLevel]],
      , glue_code = []
      , top_glue = Nothing
      })
```

5 Evaluation

To evaluate the implementation, we implement the components described by Liang [10] and use the components to construct several languages and demonstrate the different forms of extensibility as described by Erdweg [4].

In the final version of this paper, our evaluation will contain the following:

- construction of several languages with the introduced components via language composition;
- demonstrating the usage of different language operators that determine how languages are composed;
- generation of REPLs for the constructed languages that permit prototyping with the object language and enable evaluation of our incremental and prototyping claims;

5.1 Components

The components presented in this section are an aritmetic component for integer addition; a function component containing variables, call-by-value, and call-by-name abstractions; an assignment component; a lazy evaluation component; a tracing component; a callcc component; and a nondeterminism component. We detail the implementation of the function component.

```
data Abs e l where
  Abs :: IsTrue (AbsBody t) ⇒ String → e t → Abs e AbsType
data AbsType
type family AbsBody t
instance ToFuncons Abs where
  toFuncons (Abs s (K body)) = K $ function_
    [ closure_ [ scope_ [ bind_ [T.string_ s, given_], body]]]
data AppByName e l where
  AppByName :: e t1 → e t2 → AppByName e AppByNameType
data AppByNameType
type family AppByNameAbs t
type family AppByNameArg t
instance ToFuncons AppByName where
  toFuncons (AppByName (K abs) (K arg)) = K $ apply_ [abs, thunk_ [ closure_ [arg]]]
data AppByValue e l where
  AppByValue :: e t1 → e t2 → AppByValue e AppByValueType
data AppByValueType
type family AppByValueAbs t
type family AppByValueArg t
instance ToFuncons AppByValue where
  toFuncons (AppByValue (K abs) (K arg)) = K $ apply_ [abs, arg]
data Var (e :: ∗ → ∗) l where
  Var :: String → Var e VarType
data VarType
instance ToFuncons Var where
  toFuncons (Var s) = K $ T.string_ s
```

For ease of exportation and usage by other modules, we define the component as a language.

```
functionalLanguage = Language
  { operators =
    [('' Abs, '' AbsType), ('' AppByName, '' AppByNameType),
      ('' AppByValue, '' AppByValueType), ('' Var, '' VarType)]
  , op_constr =
    [(op, '' FunctionalExpr)
      | op ← ['' AbsType, '' AppByNameType, '' AppByValueType, '' VarType]]
  , sub_sorts = []
```

```
  , glue_code = [ ]
  , top_glue  = Nothing
  }
type family FunctionalExpr t
```

The defined language makes no choice with respect to the locations of operators. Instead, it only defines a helper sort to group the operators of the component.

6 Related work

7 Discussion

8 Conclusions and future work

In this paper we have shown an approach to incremental language development enabled by functional techniques for modular language specification. With the approach, a language can be incrementally defined by writing new languages, using existing language components, or using existing languages.

Currently, we have given an overview of our approach using a variant of the lambda calculus. In the final paper, we evaluate our approach according to Erdweg's framework by performing several case-studies and compare our approach to existing approaches for incremental language development. We will also demonstrate the generation of a REPL for the constructed languages that enables prototyping with the defined language, and discuss the generation of parsers from our language description.

References

1. Bahr, P., Hvitved, T.: Compositional data types. In: Järvi, J., Mu, S. (eds.) Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 83–94. ACM (2011). https://doi.org/10.1145/2036918.2036930
2. van Binsbergen, L.T., Mosses, P.D., Sculthorpe, N.: Executable component-based semantics. J. Log. Algebraic Methods Program. **103**, 184–212 (2019). https://doi.org/10.1016/j.jlamp.2018.12.004
3. van Binsbergen, L.T., Verano Merino, M., Jeanjean, P., van der Storm, T., Combemale, B., Barais, O.: A Principled Approach to REPL Interpreters, pp. 84–100. ACM (2020). https://doi.org/10.1145/3426428.3426917
4. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. LDTA '12, ACM (2012). https://doi.org/10.1145/2427048.2427055
5. Frölich, D., van Binsbergen, L.T.: A generic back-end for exploratory programming. In: The 22nd International Symposium on Trends in Functional Programming (TFP 2021). LNCS, vol. 12834. Springer (2021). https://doi.org/10.1007/978-3-030-83978-9_2
6. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. Journal of the ACM **24**(1), 68–95 (1977). https://doi.org/10.1145/321992.321997
7. Hayes, B.: Thoughts on Mathematica. Pixel **1**(January/February), 28–34 (1990)
8. Johann, P., Ghani, N.: Foundations for structured programming with gadts. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 297–308. ACM (2008). https://doi.org/10.1145/1328438.1328475
9. Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., development team, J.: Jupyter notebooks - a publishing format for reproducible computational workflows. In: Loizides, F., Scmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas. pp. 87–90. IOS Press, Netherlands (2016). https://doi.org/10.3233/978-1-61499-649-1-87
10. Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. pp. 333–343. ACM Press (1995). https://doi.org/10.1145/199448.199528
11. Mosses, P.D.: Denotational semantics. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 575–631. Elsevier and MIT Press (1990). https://doi.org/10.1016/b978-0-444-88074-1.50016-0
12. Mosses, P.D.: Software meta-language engineering and CBS. Journal of Computer Languages **50**, 39–48 (2019). https://doi.org/10.1016/j.jvlc.2018.11.003
13. Mosses, P.D., Sculthorpe, N., Van Binsbergen, L.T.: Funcons-Beta, Online GitHub repository, `https://plancomps.github.io/CBS-beta/Funcons-beta/`
14. Sheard, T., Jones, S.L.P.: Template meta-programming for haskell. ACM SIGPLAN Notices **37**(12), 60–75 (2002). https://doi.org/10.1145/636517.636528

15. Swierstra, W.: Data types à la carte. J. Funct. Program. **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758
16. Walder, P.: The expression problem, `http://www.ctan.org/pkg/acmart`
17. Ward, M.P.: Language-oriented programming. Softw. Concepts Tools **15**(4), 147–161 (1994). https://doi.org/10.1007/978-1-4302-2390-0_12

Sound and Complete Type Inference for Closed Effect Rows

Kazuki Ikemori, Youyou Cong, Hidehiko Masuhara, and Daan Leijen

[1] Tokyo Institute of Technology, Japan
ikemori.k.aa@prg.is.titech.ac.jp
[2] Tokyo Institute of Technology, Japan
cong@c.titech.ac.jp
[3] Tokyo Institute of Technology, Japan
masuhara@acm.org
[4] Microsoft Research, USA
daan@microsoft.com

Abstract. Koka is a functional programming language that has algebraic effect handlers and a row-based effect system. Koka adopts a type inference algorithm based on the Hindley-Milner type inference, but naive application of this type inference yields many effect-polymorphic functions. Such functions are more difficult to handle for users due to the complex type signature, and lead to less optimal code when using the evidence passing translation implemented in the Koka compiler. Instead, we aim to infer *closed* effect rows when possible, simplifying types and improving performance, and *open* closed effect rows automatically at instantiation to avoid loss of expressiveness. In this paper, we define a type inference algorithm with the open and close constructs. We prove that the inference algorithm is sound and complete, and infers most general types.

1. Introduction

Koka [13, 22] is a functional programming language that has algebraic effects and handlers [7, 16] which are a recently introduced abstraction of computational effects. An important aspect[1] of Koka is that it tracks the (side) effects of functions in their type. For example, the following function:

```
fun sqr( x : int ) : ⟨⟩ int
  x * x
```

has an empty effect row type ⟨⟩ as it has no side effect at all. In contrast, a function like:

```
fun head( xs : list⟨a⟩ ) : ⟨exn⟩ a
  match xs
    Cons(x,xx) → x
```

[1] Koka is the first practical language that tracks the (side) effects of functions in their type.

gets the ⟨exn⟩ effect row type as it may raise an exception at runtime (if we pass an empty list). To track effects, Koka uses row-based effect types [10]. These are quite suitable to combine with standard Hindley-Milner type inference as row equality has decidable unification (unlike subtyping for example).

When doing straightforward Hindley-Milner type inference with row-based effect types many functions will be polymorphic in the tail of the effect row (we call such effect rows *open*). For example, the naively inferred types of the previous two functions would be:

```
fun sqr : int → e int
fun head: list⟨a⟩ → ⟨exn|e⟩ a
```

Observe that both types are polymorphic in the effect tail as effect variable `e`. These are in a way the natural types, signifying for example that we can use `sqr` in any effectful context. However, in practice, we prefer closed types instead, as these are easier to explain, and easier to read and write without needing to always consider the polymorphic tail.

Moreover, it is possible to generate more efficient code for functions with closed effect rows. When executing an effect operation (which is similar to raising an exception), there is generally a dynamic search at runtime for a corresponding handler of that effect. This can be expensive, and Koka uses *evidence passing* [21, 22] to pass handler information as a vector at runtime. When an effect row is closed, the runtime shape of the vector is statically determined, and instead of searching for a handler, we can select the right handler at a fixed offset in the vector. This can be much more efficient.

Therefore, the type inference algorithm in the current Koka compiler generally infers closed effect rows for function bindings, where it relies on two constructs, open and close, for converting between open and closed function types. However, the opening and closing features of the inference algorithm have never been formalized.

In this paper, we make the following contributions:

- We formalize type inference with the open and close constructs precisely (Section 4).
- We prove that the type inference algorithm is sound and complete (Section 5) and infers most general types.

First, we give an introduction to algebraic effects and handlers, and explain what kind of types we would like to infer (Section 2). Next, we present an implicitly typed calculus with algebraic effects and handlers, and give a set of declarative type inference rules (Section 3). Then, we define syntax-directed type inference rules and show they are sound and complete with respect to the declarative rules (Section 4). We also define a type inference algorithm and prove its soundness and completeness (Section 5). Lastly, we discuss related work (Section 6) and conclude with future directions (Section 7).

2. Motivation

2.1. Algebraic Effects and Handlers

Algebraic effects and handlers [7, 16] are a uniform mechanism for representing computational effects. When programming with effect handlers, the user first declares an effect with a series of *operations*. They then define a handler that specifies the meaning of operations, using the continuation (`resume` in Koka) surrounding the operation. As an example, let us implement a reader effect in Koka.

```
effect read2
  ask-int() : int
  ask-bool(): bool
```

The reader effect $\texttt{read}_2$ has two operations `ask-int` and `ask-bool`, which take no argument and return an integer and a boolean, respectively. Below is a program that uses the two operations.

```
handler {
  ask-int()  { resume(12)   }
  ask-bool() { resume(True) }
} {
  if ask-bool() then ask-int() else 42
}
```

In the above program, the conditional expression is surrounded by a handler that specifies the meaning of `ask-int` and `ask-bool`. The evaluation goes as follows. First, `ask-bool()` is interpreted by the second clause of the handler, which says: continue (`resume`) evaluation with the value `True`. Second, the conditional expression reduces to the `then`-clause. Third, `ask-int()` is interpreted by the first clause of the handler, which says: continue evaluation with the value `12`. Thus, the program evaluates to `12`.

2.2. Naive Hindley-Milner Type Inference with Algebraic Effects and Handlers

Koka employs a row-based effect system similar to a record type system [10]. It is also equipped with polymorphic type inference, which is similar to the Hindley-Milner type inference but has an additional mechanism for manipulating effects.

It turns out that the types inferred by a natural extension of the Hindley-Milner inference are not suitable for evidence passing [21, 22]. Consider the following example.

```
fun f(x)
  ask-int()
  x
```

The function `f` is inferred to have type $\texttt{forall}\langle\texttt{a,e}\rangle\ \texttt{a} \to \langle\texttt{read}_2\texttt{|e}\rangle\ \texttt{a}$. The type contains an effect variable `e`, representing the effects of the function body. Since the Koka compiler cannot statically instantiate `e` to a concrete effect at

compile time, it needs to dynamically search for a matching handler when calling `f` at runtime. However, we would like to infer the type `forall⟨a⟩ a → ⟨read₂⟩ a` for `f`. We call this function type *closed*, in the sense that we cannot grow the effect row by instantiating the effect variable. When `f` is given this closed function type, we know that `f` only performs the `read₂` effect. Therefore, we can obtain a corresponding handler in constant time.

The Hindley-Milner type inference also occasionally yields type signatures that are more general than what the user may expect. Consider the identity function.

```
fun id( x )
  x
```

It is likely that the user defines `id` as a function of the following type.

```
fun id( x : a ): ⟨⟩ a
  x
```

Notice that the effect of function body is an empty row ⟨⟩. This is because the body of `id` is variable, which has no effect. However, based on the Hindley-Milner inference, `id` is given type `forall⟨a,e⟩ a → e a`. Here, `e` is an effect variable representing *any* effects. When the user is shown this type, they might be surprised because they did not introduce the effect variable `e`. In contrast, the type `forall⟨a⟩ a → ⟨⟩ a`, which has a total effect ⟨⟩, seems more natural.

2.3. Type Inference with open and close

In order to optimize more functions and display precise types, the Koka compiler manipulates effect types using special constructs open and close. In this paper, we formalize type inference with these constructs. The key principle is to close the effect row of all *named* functions (bound by the `let`/`val` expression), and open the effect row when we encounter variables of a closed function type. Let us illustrate how open and close work through an example.

```
val id = fn(x) x
id( ask-int() )
```

In the type system described in this paper, the function `fn(x) x` is given the most general type `forall⟨a,e⟩ a → e a`. When the function is bound to `id`, the type is *closed* to `forall⟨a⟩ a → ⟨⟩ a`. This ensures that, when the type of a named function is displayed to the user, it must be of the closed form. When `id` is instantiated in the body, the type is *opened* again, first to `forall⟨a,e⟩ a → e a` and then unified to `int → ⟨read₂⟩ int` so that `id( ask-int() )` is well-typed in its context.

In general, open introduces a fresh effect variable μ into a closed effect row $\langle l_1, \ldots, l_n \rangle$, yielding $\langle l_1, .. l_n \mid \mu \rangle$. Dually, close removes an effect variable from an open effect row. Together these allow us to avoid dynamic handler search and display simplified type signatures.

The reader may find opening and closing similar to subtyping. We use these mechanisms to avoid complex constraints over the subtyping relation and un-

Monotypes	MType	$\ni$	τ	$::=$	$\alpha^k \mid \tau \to \epsilon\, \tau \mid c^k\, \tau \ldots \tau$
Types	Type	$\ni$	σ	$::=$	$\tau \mid \forall \overline{\alpha^k}.\tau$
Kinds	Kind	$\ni$	k	$::=$	$* \mid k \to k \mid \mathsf{eff} \mid \mathsf{lab}$
Type Constructors	Con	$\ni$	c^k	$::=$	$\langle\,\rangle \mid \langle \_ \mid \_ \rangle \mid c^{\mathsf{lab}} \mid \mathsf{s}^k$
Effect Rows	Eff	$\ni$	ϵ	$::=$	$\langle\,\rangle \mid \alpha^{\mathsf{eff}} \mid \langle l \mid \epsilon \rangle$
Kind Context	KCtx	$\ni$	Ξ	$::=$	$\emptyset \mid \Xi,\, \alpha^k$
Type Context	TCtx	$\ni$	$\Gamma,\, \Delta$	$::=$	$\emptyset \mid \Gamma,\, x : \sigma$
Substitution	Subst	$\ni$	θ	$::=$	$\emptyset \mid \theta[\alpha^k := \tau]$
Effect signature			sig	$::=$	$\{\, \overline{op_i \;:\; \forall \overline{\alpha_i^k}.\, \tau_1^i \to \epsilon\, \tau_2^i} \,\}$
Effect signatures			Σ	$::=$	$\{\, \overline{l_i \;:\; sig_i} \,\}$
Syntax Convention			$\langle l_1, \ldots, l_n \rangle$	$\doteq$	$\langle l_1 \mid \ldots \mid \langle l_n \mid \langle\,\rangle \rangle \ldots \rangle$
			$\langle l_1, \ldots, l_n \mid \mu \rangle$	$\doteq$	$\langle l_1 \mid \ldots \mid \langle l_n \mid \mu \rangle \ldots \rangle$

$\mu ::= \alpha^{\mathsf{eff}}, \quad l ::= c^{\mathsf{lab}}$

Figure 1. Types, Kinds, Effect Signatures, and Effect Rows of ImpKoka

decidability of the unification algorithm. In the following sections, we will show simple type inference for open and close.

3. Implicitly Typed Calculus for Algebraic Effects and Handlers

In this section, we present ImpKoka, an implicitly typed surface language that has algebraic effects and handlers, as well as polymorphism and higher-order kinds à la System Fω. The structure of this section is as follows. First, we define the syntax of kinds, types, and effect rows (Sections 3.1.1-3.1.3). Next, we define the syntax of expressions (Section 3.1.4). Lastly, we give a set of declarative type inference rules (Section 3.2).

Note that we do not define an operational semantics for ImpKoka, but instead define a translation from ImpKoka to an extension of System F$^\epsilon$ [21]. The definition can be found in the appendix.

3.1. Syntax

3.1.1. Kinds and Types We define the syntax of kinds and types of ImpKoka in Figure 1. Similar to System F$^\epsilon$ of [21], we have kinds for value types ($*$), type constructors ($k \to k$), effect rows (eff), and effect labels (lab). Differently from System F$^\epsilon$, we distinguish between monotypes and type schemes. Monotypes consist of type variables α^k, type constructors $c^k\, \tau \ldots \tau$, and function types $\tau \to \epsilon\, \tau$. In particular, s^k is a special type constructors used in a unfication function for operations, and the detail of s^k will be explained in section 5.1 Note that type variables and constructors have a kind annotation k. The effect ϵ in a function type $\tau \to \epsilon\, \tau$ represents the effect performed by the body of the function. We will use α and β for value type variables and μ for effect type variables, often without kind annotations.

$$\frac{}{\epsilon \equiv \epsilon}\ [\textsc{refl}] \qquad \frac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3}\ [\textsc{eq-trans}]$$

$$\frac{l_1 \not\equiv l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1, l_2 \mid \epsilon_1 \rangle \equiv \langle l_2, l_1 \mid \epsilon_2 \rangle}\ [\textsc{eq-swap}] \qquad \frac{\epsilon_1 \equiv \epsilon_2}{\langle l \mid \epsilon_1 \rangle \equiv \langle l \mid \epsilon_2 \rangle}\ [\textsc{eq-head}]$$

Figure 2. Equivalence of Row Types

3.1.2. Signatures We define operations *op*, effect signatures *sig*, and sets of effect signatures Σ again as in System F^ϵ (Figure 1). An effect signature is a set of pairs of an operation name and a type. A set of effect signatures associates each effect label l_i with a corresponding effect signature sig_i. We assume that operation names and effect labels are all unique, and that Σ is defined at the top level.

3.1.3. Effect Rows In ImpKoka, we use effect rows [5, 12] to represent a collection of effects to be performed by an expression. As in System F^ϵ, an effect row is either an empty row $\langle\,\rangle$, or an effect variable μ, or an extension $\langle l \mid \epsilon \rangle$ of an effect row ϵ with an effect label l, respectively. For example, assuming exn is an effect label representing exceptions, $\langle \mathsf{exn}, \mathsf{read}_2 \rangle$ is an effect row representing a collection of exception and reader effects. The kinds of $\langle\,\rangle$ and l are eff (representing an effect type) and lab (representing an effect label), respectively. The kind of $\langle \_ \mid \_ \rangle$ is $\mathsf{lab} \to \mathsf{eff} \to \mathsf{eff}$. We will use $\langle\,\rangle$ and $\langle \_ \mid \_ \rangle$ without kind annotations.

The equivalence relation for effect rows is also defined in the same way as in System F^ϵ (Figure 2). The [refl] and [eq-trans] rules are the reflexivity and transitivity rules. The [eq-swap] rule says that two effect labels l_1 and l_2 can be swapped if they are distinct. The [eq-head] rule tells us that two effect rows are equivalent when their heads and tails are equivalent.

Note that effect rows can have multiple occurrences of the same effect label. For example, we may have $\langle \mathsf{exn} \rangle$ and $\langle \mathsf{exn}, \mathsf{exn} \rangle$, and they are treated as different effect rows. The advantage of this design is that we can define type inference rules in a simple manner, by using only type equivalence.

3.1.4. Expressions We define the syntax of expressions of ImpKoka in Figure 3. In addition to the standard lambda terms, we have handlers handler h and operation performing perform *op*. Note that we treat both handler h and perform *op* as a value. If we want to handle an expression e with handler h, we write handler $h\,(\lambda\_.e)$ via application. Similarly, if we want to perform an operation *op* with argument e, we write perform *op* e via application.

3.2. Declarative Type Inference rules

We now turn to the declarative type inference rules with open and close (Figure 4). Here, we use a typing judgment of the form $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon$. The

Expressions	Exp	$\ni$	e	$::=$	v	(value)
				$\mid$	$e\,e$	(application)
Values	Val	$\ni$	v	$::=$	x	(variable)
				$\mid$	$\lambda x.\, e$	(function)
				$\mid$	handler h	(effect handler)
				$\mid$	perform op	(operation)
Handlers	Hnd	$\ni$	h	$::=$	$\{\overline{op_i \to v_i}\}$	(operation clauses)

Figure 3. Expressions of ImpKoka

judgment states that, under type variable context Ξ and typing context Γ and Δ, expression e has type σ and performs effect ϵ. Among the two contexts, Δ is a type assignment for named (i.e., `let`-bound) functions, which inhabit a function type with a closed effect row (we will call such types *closed function types*). The other context Γ is a type assignment for all other variables.

The [Var] rule concludes with a type σ that comes from either Δ or Γ, and an arbitrary effect ϵ. Note that, although a variable does not perform any effects, we cannot replace ϵ by $\langle\,\rangle$, because we do not have subeffecting rules in ImpKoka. Note also that the rule applies only to variables whose type is not a closed function type.

The [VarOpen] rule derives an open function type for a variable of a closed function type. The variable must reside in Δ, because we can only open the type of named functions. As an example, we can use [VarOpen] to make $id\,(\mathsf{perform}\ \mathit{ask\text{-}int}\,())$ well-typed, because we can derive $\Xi \mid \Gamma \mid \Delta \vdash id : \mathsf{int} \to \langle \mathsf{read}_2 \rangle\, \mathsf{int}$ by [VarOpen] and $\Xi \mid \Gamma \mid \Delta \vdash \mathsf{perform}\ \mathit{ask\text{-}int}\,() : \mathsf{int} \mid \langle \mathsf{read}_2 \rangle$ by [Perform] and [App].

The [Lam] rule derives a function type for a lambda abstraction. Observe that the effect ϵ of the body e is integrated into the type $\tau_1 \to \epsilon\, \tau_2$ in the conclusion. Observe also that the lambda-bound variable x is added to Γ, not Δ. Therefore, in the derivation of the body e, the type of x cannot be opened using [VarOpen].

The [App] rule requires that the function e_1, the argument e_2, and the body of the function have the same effect ϵ.

The [Gen] rule derives a polymorphic type. Similar to the value restriction in ML we only allow values v [20], but type soundness is actually guaranteed by restricting the effect type to be total $\langle\,\rangle$ [8, 12]. The [Inst] rule is completely standard.

The [Let] rule is used to bind values with polymorphic types. As in the [Gen] rule, the expression being bound is a value, and must have a total effect $\langle\,\rangle$. Notice that bound variable x is added to the context Δ, not Γ. Therefore, in the derivation of the body e_2, the type of x can be opened via [VarOpen].

The [Perform] rule is used to type an operation call. The type in the conclusion is a monotype, which is instantiated by using a sequence of types $\overline{\tau}$. The notation $op\colon \forall \overline{\alpha}^{\overline{k}}.\tau_1 \to \tau_2 \in \Sigma(l)$ of the premise means that the operation $op\colon \forall \overline{\alpha}^{\overline{k}}.\tau_1 \to \tau_2$ belongs to the effect signature corresponding to the effect label l.

The [Hanlder] rule is used to type a handler. It takes a thunked computation (*action*) of type $() \to \langle l \mid \epsilon \rangle\, \tau$ and handles the effect l. The [Ops] rule takes care of operation clauses of a handler. The type of each operation clause is a nested

$$\boxed{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \qquad \Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon}$$

$$\frac{x : \sigma \in \Gamma, \Delta \quad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \sigma \text{ not a closed function type}}{\Xi \mid \Gamma \mid \Delta \vdash x : \sigma \mid \epsilon} \ [\textsc{Var}]$$

$$\frac{f : \forall \overline{\alpha}^{\overline{k}}. \tau_1 \to \langle l_1, \ldots, l_n \rangle \tau_2 \in \Delta \qquad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash f : \forall \overline{\alpha}^{\overline{k}}. \tau_1 \to \langle l_1, \ldots, l_n \mid \epsilon \rangle \tau_2 \mid \epsilon'} \ [\textsc{VarOpen}]$$

$$\frac{\Xi \mid \Gamma, x : \tau_1 \mid \Delta \vdash e : \tau_2 \mid \epsilon \quad \Xi \vdash_{\mathsf{wf}} \tau_1 : * \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x. e : \tau_1 \to \epsilon\, \tau_2 \mid \epsilon'} \ [\textsc{Lam}] \qquad \frac{\Xi \mid \Gamma \mid \Delta \vdash e_1 : \tau_2 \to \epsilon\, \tau \mid \epsilon \quad \Xi \mid \Gamma \mid \Delta \vdash e_2 : \tau_2 \mid \epsilon}{\Xi \mid \Gamma \mid \Delta \vdash e_1\, e_2 : \tau \mid \epsilon} \ [\textsc{App}]$$

$$\frac{\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v : \sigma \mid \langle\rangle \quad k \not\equiv \mathsf{lab} \quad \alpha^k \notin \mathsf{ftv}(\Gamma, \Delta)}{\Xi \mid \Gamma \mid \Delta \vdash v : \forall \alpha^k. \sigma \mid \langle\rangle} \ [\textsc{Gen}] \qquad \frac{\Xi \mid \Gamma \mid \Delta \vdash e : \forall \alpha^k. \sigma \mid \epsilon \quad \Xi \vdash_{\mathsf{wf}} \tau : k}{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma[\alpha^k := \tau] \mid \epsilon} \ [\textsc{Inst}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash v_1 : \sigma_1 \mid \langle\rangle \quad \Xi \mid \Gamma \mid \Delta, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 : \tau_2 \mid \epsilon} \ [\textsc{Let}]$$

$$\frac{op : \forall \overline{\alpha}^{\overline{k}}. \tau_1 \to \tau_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \Xi \quad \Xi \vdash_{\mathsf{wf}} \overline{\tau} : \overline{k} \quad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{ef}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{perform}\, op : id[\overline{\alpha}^{\overline{k}} := \overline{\tau}](\tau_1 \to \langle l \mid \epsilon \rangle \tau_2) \mid \epsilon'} \ [\textsc{Perform}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{handler}\, h : (() \to \langle l \mid \epsilon \rangle \tau) \to \epsilon\, \tau) \mid \epsilon'} \ [\textsc{Handler}]$$

$$\frac{\overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon, \tau) \quad op_i : \forall \overline{\alpha}_i^{\overline{k}}. \tau_1^i \to \tau_2^i \in \Sigma(l) \quad \Xi \mid \Gamma \mid \Delta \vdash v_i : \forall \overline{\alpha}_i^{\overline{k}}. \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau) \mid \langle\rangle}{\Xi \mid \Gamma \mid \Delta \vDash \{ op_1 \to v_1, \ldots, op_n \to v_n \} : \tau \mid l \mid \epsilon} \ [\textsc{Ops}]$$

Figure 4. Declarative Type Inference Rules

$\vdash \theta : \Xi \Rightarrow \Xi' \qquad \Xi \vdash \sigma_1 \sqsubseteq \sigma_2$

$$\frac{}{\vdash \emptyset : \cdot \Rightarrow \Xi}\ [\textsc{Empty}] \qquad \frac{\vdash \theta : \Xi_1 \Rightarrow \Xi_2 \quad \Xi_2 \vdash \tau : k}{\vdash \theta[\alpha^k := \tau] : (\Xi_1, \alpha^k) \Rightarrow \Xi_2}\ [\textsc{Extend}]$$

$$\frac{\Xi \vdash_{\mathsf{wf}} \overline{\tau} : \overline{k} \quad \tau_2 = id[\overline{\alpha}^{\overline{k}} := \overline{\tau}]\tau_1 \quad \overline{\beta}^{\overline{k'}} \notin \mathsf{ftv}(\forall \overline{\alpha}^{\overline{k}}.\, \tau_1)}{\Xi \vdash \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \sqsubseteq \forall \overline{\beta}^{\overline{k'}}.\, \tau_2}\ [\textsc{TypeOrdering}]$$

Figure 5. Type Substitution and Type Ordering

function type of the form $\forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau)$, where τ_1^i is the input type of the operation, and $\tau_2^i \to \epsilon\, \tau$ is the type of the continuation. The condition $\overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon, \tau)$ is necessary for type preservation of the translation from ImpKoka to System F$^{\epsilon}$+restrict.

4. Syntax-Directed Type Inference Rules

In this section, we formalize the syntax-directed type inference rules with open and close, following [6, 11]. These rules allow us to determine which typing rule to apply to an expression from the syntax of that expression. In what follows, we first define type substitution and type ordering, and then elaborate the key cases of the inference rules.

4.1. Type Substitution

Figure 5 shows the definition of type substitution, which is inspired by [4]. The judgment $\vdash \theta : \Xi_1 \Rightarrow \Xi_2$ means substitution θ replaces type variables in context Ξ_1 with types well-formed under context Ξ_2. There are two formation rules for substitutions: [Empty] forms an empty substitution, and [Extend] extends a substitution θ with $[\alpha^k := \tau]$. As a convention, we write id to mean the identity substitution.

4.2. Type Ordering

Figure 5 shows the definition of type ordering, which is similar to that of System F. The judgment $\Xi \vdash \sigma_1 \sqsubseteq \sigma_2$ means type σ_2 is more specific than type σ_1 under context Ξ. The context Ξ is used to inspect the kinds of the types $\overline{\tau}$ that replace the type variables $\overline{\alpha}^{\overline{k}}$. For example, the following relationship holds.

$\Xi \vdash \forall \alpha\, .\, \alpha \to \langle\,\rangle\, \alpha \sqsubseteq \mathsf{int} \to \langle\,\rangle\, \mathsf{int}$
$\Xi \vdash \forall \alpha\, \mu\, .\, \alpha \to \mu\, \alpha \sqsubseteq \forall \beta.\, (\beta \to \langle \mathsf{exn} \rangle\, \beta) \to \langle \mathsf{exn} \rangle\, (\beta \to \langle \mathsf{exn} \rangle\, \beta)$

Using type ordering, we define type equivalence as follows.

$\sigma_1 = \sigma_2 \Leftrightarrow \sigma_1 \sqsubseteq \sigma_2 \wedge \sigma_2 \sqsubseteq \sigma_1$

$$\boxed{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e : \tau \mid \epsilon \qquad \Xi \mid \Gamma \mid \Delta \vDash_{\mathsf{s}} h : \tau \mid l \mid \epsilon}$$

$$\frac{x : \sigma \in \Gamma, \Delta \quad \Xi \vdash \sigma \sqsubseteq \tau \quad \Xi \Vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} x : \tau \mid \epsilon} \ [\textsc{Var}]$$

$$\frac{\begin{array}{l} f : \forall \overline{\alpha^k}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \in \Delta \\ \quad \Xi \vdash \forall \overline{\alpha^k}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \sqsubseteq \tau_1' \to \langle l_1, \ldots, l_n \rangle\, \tau_2' \\ \quad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff} \end{array}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} f : \tau_1' \to \langle l_1, \ldots, l_n \mid \epsilon \rangle\, \tau_2' \mid \epsilon'} \ [\textsc{VarOpen}]$$

$$\frac{\begin{array}{l} \Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} v_1 : \tau_1 \mid \langle\,\rangle \quad \Xi \setminus \overline{\alpha^k} \mid \Gamma \mid \Delta, x : \mathsf{Close}(\sigma_1) \Vdash_{\mathsf{s}} e_2 : \tau_2 \mid \epsilon \\ \quad \sigma_1 = \mathsf{Gen}(\Gamma, \Delta, \tau_1) = \forall \overline{\alpha^k}.\, \tau_1 \end{array}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 : \tau_2 \mid \epsilon} \ [\textsc{Let}]$$

$$\frac{\begin{array}{l} op_i : \forall \overline{\alpha_i^k}.\, \tau_1^i \to \tau_2^i \in \Sigma(l) \quad \overline{\alpha_i^k} \notin \mathsf{ftv}(\epsilon, \tau) \\ \Xi, \overline{\alpha_i^k} \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} v_i : \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau) \mid \langle\,\rangle \quad \overline{\alpha_i^k} \notin \mathsf{ftv}(\Gamma, \Delta) \end{array}}{\Xi \mid \Gamma \mid \Delta \vDash_{\mathsf{s}} \{ op_1 \to v_1, \ldots, op_n \to v_n \} : \tau \mid l \mid \epsilon} \ [\textsc{Ops}]$$

Figure 6. Syntax-directed Type Inference Rules (excerpt)

4.3. Inference Rules with open and close

Figure 6 is an excerpt of the syntax-directed type inference rules, consisting of those rules that are changed from the declarative rules. Compared to the declarative rules we saw in Section 3.2, there are no rules corresponding to [Gen] and [Inst], because these rules are not syntax directed. All other rules are identical to the declarative rules. Another difference is that we use two auxiliary functions $\mathsf{Gen}(\cdot, \cdot, \cdot)$ and $\mathsf{Close}(\cdot)$. The Gen function is the standard generalization function, with a standard definition of free type variables. The Close function closes the effect row of a function type. For example, $\mathsf{Close}(\forall \mu\, \alpha.\, \alpha \to \mu\, \alpha) = \forall \alpha.\, \alpha \to \langle\,\rangle\, \alpha$.

$$\mathsf{Gen}(\Gamma, \Delta, \tau) = \forall (\mathsf{ftv}(\tau) - \mathsf{ftv}(\Gamma, \Delta)).\, \tau$$

$$\mathsf{Close}(\forall \mu\, \overline{\alpha^k}.\, \tau_1 \to \langle l_1, \ldots, l_n \mid \mu \rangle\, \tau_2) = \forall \overline{\alpha^k}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \quad \text{iff } \mu \notin \mathsf{ftv}(\tau_1, \tau_2)$$
$$\mathsf{Close}(\sigma) = \sigma$$

Among the key rules, [Var] is standard and derives the type instantiated by $\overline{\tau}$. The [VarOpen] rule derives an open function type from a closed one by inserting an arbitrary effect row. The [Let] rule generalizes the type of the bound expression using Gen, and derives the type of the body under an extended type context $\Delta, x : \mathsf{Close}(\sigma)$. In the [Ops] rule, we derive a monomorphic type without the type variables bound by the quantifier.

It is important that the Close function is applied only in the [Let] rule. The reason is that, if Close is used to a function type that is not universally quantified, the type system cannot track the effects to be handled. Let us illustrate the problem using a variation of Close and [Lam] rule. We first define Close_1 as a

function that closes the effect variable of a monomorphic function type. For example, $\mathsf{Close}_1(\alpha \to \mu\,\alpha) = \alpha \to \langle\,\rangle\,\alpha$, where $\Gamma = \emptyset$ and $\Delta = \emptyset$.

$$\mathsf{Close}_1(\tau_1 \to \langle l_1, \ldots, l_n \mid \mu\rangle\,\tau_2) = \tau_1 \to \langle l_1, \ldots, l_n\rangle\,\tau_2 \quad \mathsf{iff}\,\mu \notin \mathsf{ftv}(\tau_1, \tau_2)$$
$$\mathsf{Close}_1(\tau) = \tau$$

We next define [Lam1] as a typing rule that closes the monomorphic function type of a λ-bound variable using Close_1 and derives the type of the body under an extended type context $\Delta, x : \tau_1'$. This allows more functions to have a closed function type as their domain.

$$\frac{\Xi \mid \Gamma \mid \Delta, x : \tau_1' \Vdash_{\mathsf{s}} e : \tau_2 \mid \epsilon \quad \tau_1' = \mathsf{Close}_1(\tau_1) \quad \Xi \vdash_{\mathsf{wf}} \tau_1 : * \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} \lambda x.\, e : \tau_1 \to \epsilon\,\tau_2 \mid \epsilon'}\ [\textsc{Lam1}]$$

With [Lam1], it is possible to derive wrong effects. Consider the following expression.

$$\mathsf{let}\, f = \lambda g.\ g\,()\,\mathsf{in}\ f\,(\lambda\_.\,\mathsf{perform}\ \textit{ask-int}\ ())$$

First, we derive $\emptyset \mid \emptyset \mid \emptyset \Vdash_{\mathsf{s}} \lambda g.\ g\,() : (() \to \mu\,\alpha) \to \langle\,\rangle\,\alpha$ by the [Lam1] and [VarOpen] rules. Next, we extend the type context Δ with $f : \forall \mu\,\alpha.\,(() \to \mu\,\alpha) \to \langle\,\rangle\,\alpha$ by the [Let] rule. Then, we obtain the following derivation by the [App] rule.

$$\emptyset \mid \emptyset \mid f : \forall \mu\,\alpha.\,(() \to \mu\,\alpha) \to \langle\,\rangle\,\alpha \Vdash_{\mathsf{s}} f\,(\lambda\_.\,\mathsf{perform}\ \textit{ask-int}\ ()) : \mathsf{int} \mid \langle\,\rangle$$

This type judgment is clearly wrong, because the read_2 effect is not tracked. By using Close only in the [Let] rule, we can avoid this problem.

The syntax-directed type inference rules are sound and complete with respect to the declarative type inference rules.

Theorem 1. (*syntax-directed inference rules is sound*)
If $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e : \tau \mid \epsilon$ then $\Xi \mid \Gamma \mid \Delta \vdash e : \tau \mid \epsilon$.

Theorem 2. (*syntax-directed inference rules is complete*)
If $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon$ then $\Xi \mid \Gamma \mid \Delta \Vdash_{\mathsf{s}} e : \tau \mid \epsilon$,
where $\Xi \subseteq \Xi'$ and $\Xi' \vdash \mathsf{Gen}(\Gamma, \Delta, \tau) \sqsubseteq \sigma$.

4.4. Fragility of the [Let] Rule

An unfortunate aspect of our current rules is that the [Let] rule is fragile in the sense that insertion of a `let` binding may change the typability of programs. Let us consider the following functions:

$$\textit{remote} = \lambda f.\ \mathsf{perform}\ \textit{ask-bool}\ ()$$
$$\textit{foo} = \lambda f.\ \textit{remote}\ f; f\ ()$$

In our type system, the type of the function *remote* is inferred as $\alpha \to \langle \mathsf{read}_2 \mid \mu\rangle\,\mathsf{bool}$. The function *foo* is also well-typed. First, the lambda-bound variable f is added to the context Γ, second, the type of *remote* f is inferred as $\Xi \mid \Gamma \mid \Delta \vDash_{\mathsf{s}} \textit{remote}\ f : \mathsf{bool}$

$\mid \langle \mathsf{read}_2 \mid \mu \rangle$ by the [App] rule, and finally, the type of f is inferred as $\Xi \mid \Gamma \mid \Delta \models_{\mathsf{s}} f : () \to \langle \mathsf{read}_2 \mid \mu \rangle\, \beta \mid \langle \mathsf{read}_2 \mid \mu \rangle$ by the [App] rule, where $e_1; e_2$ is a syntax suger of $(\lambda\_.\ e_2)\ e_1$. Therefore, the type of function *foo* is inferred as $((() \to \langle \mathsf{read}_2 \mid \mu \rangle\, \beta) \to \langle \mathsf{read}_2 \mid \mu \rangle\, \beta$ and *foo* is judged well-typed.

Suppose though that we have a function *remote* that is explicitly annotated with type $((() \to \langle\,\rangle\,()) \to \langle \mathsf{read}_2 \rangle\, \mathsf{bool}$.

$$
\begin{aligned}
&\mathit{remote} \ :\ ((() \to \langle\,\rangle\,()) \to \langle \mathsf{read}_2 \rangle\, \mathsf{bool} \\
&\mathit{remote} = \lambda f.\ \mathsf{perform}\ \mathit{ask\text{-}bool}\ ()
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foo} = \lambda f.\ \mathit{remote}\ f; f\ () \\
&\mathit{bar} = \lambda f.\ \mathit{remote}\ f; \mathsf{let}\ g = f\ \mathsf{in}\ g\ ()
\end{aligned}
$$

Here *foo* and *bar* only differ in the explicit let-binding for f, but our inference rules reject the *foo* definition. The type of $\mathit{remote}\ f$ is inferred as $\Xi \mid \Gamma \mid \Delta \models_{\mathsf{s}} \mathit{remote}\ f : \mathsf{bool} \mid \langle \mathsf{read}_2 \rangle$, and the type of f is unified to a closed type $() \to \langle\,\rangle\,()$ by *remote*. Its type cannot be opened to $() \to \langle \mathsf{read}_2 \rangle\, ()$, because the lambda-variable f is included in Γ and cannot be applied the [VarOpen] rule. Hence, we cannot apply the [App] rule to $f\ ()$. On the other hand, the *bar* definition is well-typed. The function f is now a `let`-bound variable, thus its type can be opened to $() \to \langle \mathsf{read}_2 \rangle\, ()$ by the [VarOpen] rule. Hence, we can apply the [App] rule to $f\ ()$.

This is clearly not desirable and we would like to address this in future work. On the other hand, it is not uncommon to find this form of fragility in practical type systems (like the monomorphism restriction in Haskell, inference for GADT's [15], etc) and it may work out fine in practice. Experiments on all the standard libraries of Koka (~15000 lines) showed only 2 instances where a let binding was required.

5. Type Inference Algorithm

In this section, we formalize the type inference algorithm with open and close as an extension of Algorithm W [3]. We first present the unification algorithm (Section 5.1), and then discuss the type inference algorithm (Section 5.2).

5.1. Unification Algorithm

In Figure 7, we define the unification algorithm. The algorithm is a natural extension of the standard Robinson unification algorithm [19]. It consists of three functions $\mathit{unify}(\cdot,\cdot,\cdot)$, $\mathit{unifyEffect}(\cdot,\cdot,\cdot)$, and $\mathit{unifyOp}(\cdot,\cdot,\cdot)$, which take care of value types, effect types, and the type of operation clauses, respectively. Among these functions, *unify* and *unifyEffect* are standard. The *unifyOp* function originates from the special treatment of Koka's operation clauses.

Let us look at the *unify* and *unifyEffect* functions. Given a triple $(\Xi,\ \epsilon,\ l)$ of a type context, an effect row type, and an effect label, *unifyEffect* returns a triple $(\Xi_1,\ \theta_1,\ \epsilon_1)$ of a new type context, a substitution, and an effect row. We

$$
\begin{array}{l}
\mathit{unify} \;:\; (\mathsf{KCtx} \times \mathsf{MType} \times \mathsf{MType}) \to (\mathsf{KCtx} \times \mathsf{Subst}) \\
\mathit{unify}(\Xi, \alpha^k, \alpha^k) \;=\; \mathsf{return}\,(\Xi,\; \mathit{id}) \\
\mathit{unify}((\Xi, \alpha^k), \alpha^k,\; \tau) \;=\; \mathsf{assert}\,\Xi \vdash_{\mathsf{wf}} \tau \;:\; k;\; \mathsf{return}\,(\Xi,\; \mathit{id}[\alpha^k := \tau]) \\
\mathit{unify}((\Xi, \alpha^k), \tau, \alpha^k) \;=\; \mathsf{assert}\,\Xi \vdash_{\mathsf{wf}} \tau \;:\; k;\; \mathsf{return}\,(\Xi,\; \mathit{id}[\alpha^k := \tau]) \\
\mathit{unify}(\Xi, \langle\,\rangle, \langle\,\rangle) \;=\; \mathsf{return}\,(\Xi,\; \mathit{id}) \\
\mathit{unify}(\Xi,\; \langle l \mid \epsilon_1\rangle,\; \langle l' \mid \epsilon_2\rangle) \;= \\
\qquad \mathrm{let}\,(\Xi_1,\; \theta_1,\; \epsilon_3) \;=\; \mathit{unifyEffect}(\Xi,\; \langle l' \mid \epsilon_2\rangle,\; l) \\
\qquad \mathsf{assert}\; \mathit{tail}(\epsilon_1) \;\notin\; \mathit{dom}(\theta_1) \\
\qquad \mathrm{let}\,(\Xi_2,\; \theta_2) \;=\; \mathit{unify}(\Xi_1,\; \theta_1(\epsilon_1),\; \epsilon_3) \\
\qquad \mathsf{return}\,(\Xi_2,\; \theta_2 \circ \theta_1) \\
\mathit{unify}(\Xi,\; \tau_1 \to \epsilon\, \tau_2,\; \tau_1' \to \epsilon'\, \tau_2') \;= \\
\qquad \mathrm{let}\,(\Xi_1,\; \theta_1) \;=\; \mathit{unify}(\Xi,\; \tau_1,\; \tau_1') \\
\qquad \mathrm{let}\,(\Xi_2,\; \theta_2) \;=\; \mathit{unify}(\Xi_1,\; \theta_1(\epsilon),\; \theta_1(\epsilon')) \\
\qquad \mathrm{let}\,(\Xi_3,\; \theta_3) \;=\; \mathit{unify}(\Xi_2,\; (\theta_2 \circ \theta_1)(\tau_2'),\; (\theta_2 \circ \theta_1)(\tau_2')) \\
\qquad \mathsf{return}\,(\Xi_3,\; \theta_3 \circ \theta_2 \circ \theta_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyEffect} \;:\; (\mathsf{KCtx} \times \mathsf{Eff} \times \mathsf{Lab}) \to (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{Eff}) \\
\mathit{unifyEffect}(\Xi,\; \langle l' \mid \epsilon\rangle,\; l) \;= \\
\quad \mid\; l' \equiv l \Rightarrow \mathsf{return}\,(\Xi,\; \mathit{id},\; \epsilon) \\
\quad \mid\; l' \not\equiv l \Rightarrow \mathrm{let}\,(\Xi_1,\; \theta_1,\; \epsilon_1) \;=\; \mathit{unifyEffect}(\Xi,\; \epsilon,\; l) \\
\qquad\qquad\qquad \mathsf{return}\,(\Xi_1,\; \theta_1,\; \langle l' \mid \epsilon_1\rangle) \\
\mathit{unifyEffect}((\Xi,\; \mu),\; \mu,\; l) \;= \\
\quad \mathsf{assume}\; \mu_1 \text{ is fresh} \\
\quad \mathsf{return}\,((\Xi,\; \mu_1),\; \mathit{id}[\mu := \langle l \mid \mu_1\rangle],\; \mu_1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyOp} \;:\; (\mathsf{KCtx} \times \mathsf{Type} \times \mathsf{MType}) \to (\mathsf{KCtx} \times \mathsf{Subst}) \\
\mathit{unifyOp}(\Xi,\; \forall \overline{\alpha^k}.\, \tau_1,\; \tau_2) \;= \\
\quad \mathsf{assume}\; \overline{\mathsf{s}^k} \text{ are fresh (skolem constants)} \\
\quad \mathrm{let}\,(\Xi_1,\; \theta_1) \;=\; \mathit{unify}(\Xi,\; \mathit{id}[\overline{\alpha^k} := \overline{\mathsf{s}^k}]\tau_1,\; \tau_2) \\
\quad \mathsf{assert}\; \overline{\mathsf{s}^k} \;\notin\; \mathsf{const}(\mathsf{codom}(\theta_1 \;-\; \mathsf{ftv}(\tau_2))) \\
\quad \mathsf{return}\,((\Xi_1,\; \overline{\alpha^k}),\; [\overline{\mathsf{s}^k \mapsto \alpha^k}] \circ \theta_1)
\end{array}
$$

Figure 7. Unification Algorithm

can transfom ϵ into an effect row whose head effect label is l. As an example, let us consider the following unification problem.

$$\mathit{unifyEffect}(\Xi,\ \langle \mathsf{read}_2 \mid \mu \rangle,\ \mathsf{exn})$$

This succeeds and returns the following effect row and substitution:

$$\epsilon_1 \ =\ \langle \mathsf{read}_2 \mid \mu_1 \rangle \qquad \theta_1 \ =\ \mathit{id}[\mu := \langle \mathsf{exn} \mid \mu_1 \rangle]$$

The *unifyEffect* function is sound. That is, if $\mathit{unifyEffect}(\Xi,\ \epsilon,\ l)$ succeeds, it returns a substitution θ_1 and an effect row ϵ_1 that satisfy $\theta_1(\epsilon) \equiv \langle l \mid \theta_1(\epsilon_1) \rangle$.

Theorem 3. (*unifyEffect is sound*)
If $\Xi \vdash_{\mathsf{wf}} \epsilon :: \mathsf{eff}$, $\Xi \vdash_{\mathsf{wf}} l :: \mathsf{lab}$ and $\mathit{unifyEffect}(\Xi,\ \epsilon,\ l) = (\Xi_1,\ \theta_1,\ \epsilon_1)$,
then $\vdash \theta_1 :: \Xi \Rightarrow \Xi_1$ and $\theta_1(\epsilon) \equiv \langle l \mid \theta_1(\epsilon_1) \rangle$.

The *unifyEffect* function is also complete. That is, if ϵ can be rewritten to an effect row of the form $\langle l \mid \theta(\epsilon') \rangle$ by the substitution θ, $\mathit{unifyEffect}(\Xi,\ \epsilon,\ l)$ succeeds and returns the most general substitution θ_1.

Theorem 4. (*unifyEffect is complete*)
If $\Xi \vdash_{\mathsf{wf}} \epsilon :: \mathsf{eff}$, $\Xi \vdash_{\mathsf{wf}} l :: \mathsf{lab}$, $\vdash \theta :: \Xi \Rightarrow \Xi_2$ and $\theta(\epsilon) \equiv \langle l \mid \theta(\epsilon') \rangle$,
then $\mathit{unifyEffect}(\Xi,\ \epsilon,\ l) = (\Xi_1,\ \theta_1,\ \epsilon_1)$ and there exists $\vdash \theta_2 :: \Xi_1 \Rightarrow \Xi_2$ such that $\theta = \theta_2 \circ \theta_1$.

The *unify* function is similar to *unifyEffect*. Given a triple $(\Xi,\ \tau_1,\ \tau_2)$ of a type context and two monomorphic types, *unify* returns a pair $(\Xi_1,\ \theta_1)$ of a new type context and a substitution.

The unification algorithm is sound: if $\mathit{unify}(\Xi,\ \tau_1,\ \tau_2)$ succeeds, it returns the substitution θ_1 that unifies τ_1 and τ_2.

Theorem 5. (*unify is sound*)
If $\Xi \vdash_{\mathsf{wf}} \tau_1 :: k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 :: k$, and $\mathit{unify}(\Xi,\ \tau_1,\ \tau_2) = (\Xi_1,\ \theta_1)$, then $\vdash \theta_1 :: \Xi \Rightarrow \Xi_1$ and $\theta_1(\tau_1) = \theta_1(\tau_2)$.

The unification algorithm is also complete: if two types τ_1 and τ_2 are unifiable, $\mathit{unify}(\Xi,\ \tau_1,\ \tau_2)$ succeeds and it returns the most general substitution θ_1.

Theorem 6. (*unify is complete*)
If $\Xi \vdash_{\mathsf{wf}} \tau_1 :: k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 :: k$, $\vdash \theta :: \Xi \Rightarrow \Xi_2$ and $\theta(\tau_1) = \theta(\tau_2)$,
then $\mathit{unify}(\Xi,\ \tau_1,\ \tau_2) = (\Xi_1,\ \theta_1)$ and there exists $\vdash \theta_2 :: \Xi_1 \Rightarrow \Xi_2$ such that $\theta = \theta_2 \circ \theta_1$.

We now look at the *unifyOp* function. The function takes a triple $(\Xi,\ \forall \overline{\alpha^k}.\, \tau_1,\ \tau_2)$ of a type variable context, a type scheme, and a monomorphic type τ, and returns a pair $(\Xi_1,\ \theta_1)$ of a new type context and a substitution. Following [11], we use *skolem constants* to ensure that bound type variables $\overline{\alpha^k}$ do not escape or occur free in the substitution $[\overline{\mathsf{s}^k \mapsto \alpha^k}] \circ \theta_1$. The algorithm reads as follows. We first replace the free type variables $\overline{\alpha^k}$ of τ_1 with fresh skolem constants $\overline{\mathsf{s}^k}$, and unify the resulting type with τ_2. When we obtain a substitution θ_1 as the result, we check the codomain of $\theta_1 - \mathsf{ftv}(\tau_2)$ does not contain skolem constants $\overline{\mathsf{s}^k}$, using the const function. If this checking succeeds, we construct a new substitution by

replacing the skolem constants $\overline{\mathsf{s}}^{\overline{k}}$ in θ_1 back to type variables $\overline{\alpha}^{\overline{k}}$, and return the resulting substitution. As an example, consider the following unification problem:

$$\mathit{unifyOp}(\Xi,\ \forall\alpha.\,\alpha \to \mathsf{exn}\,((\alpha \to \langle\mathsf{exn}\rangle\,\mathsf{int}) \to \langle\mathsf{exn}\rangle\,\mathsf{int}),\ \beta_1 \to \mu\,\beta_2)$$

This succeeds and returns the following substitution:

$$[\mathsf{s} \mapsto \alpha] \circ \theta_1 \;=\; \mathit{id}[\beta_1 := \alpha,\ \beta_2 := ((\alpha \to \langle\mathsf{exn}\rangle\,\mathsf{int}) \to \langle\mathsf{exn}\rangle\,\mathsf{int}),\ \mu := \langle\mathsf{exn}\rangle]$$

where $\theta_1 \;=\; \mathit{id}[\beta_1 := \mathsf{s},\ \beta_2 := ((\mathsf{s} \to \langle\mathsf{exn}\rangle\ \mathit{int}) \to \langle\mathsf{exn}\rangle\ \mathit{int}),\ \mu := \langle\mathsf{exn}\rangle]$.

The soundness and completeness of *unifyOp* can be proven in a similar way to that of *unify* and *unifyEffect*.

Theorem 7. (*unifyOp is sound*)
If $\Xi \vdash_{\mathsf{wf}} \forall\overline{\alpha}^{\overline{k}}.\,\tau_1 \;:\; k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 \;:\; k$ and $\mathit{unifyOp}(\Xi,\ \forall\overline{\alpha}^{\overline{k}}.\,\tau_1,\ \tau_2) \;=\; (\Xi_1,\ \theta_1)$, then $\vdash\ \theta_1 \;:\; \Xi \Rightarrow \Xi_1$ and $\theta_1(\forall\overline{\alpha}^{\overline{k}}.\,\tau_1) \;=\; \forall\overline{\alpha}^{\overline{k}}.\,\theta_1(\tau_2)$.

Theorem 8. (*unifyOp is complete*)
If $\Xi \vdash_{\mathsf{wf}} \forall\overline{\alpha}^{\overline{k}}.\,\tau_1 \;:\; k$, $\Xi \vdash_{\mathsf{wf}} \tau_2 \;:\; k$, $\vdash\ \theta \;:\; \Xi \Rightarrow \Xi_2$ and $\theta(\forall\overline{\alpha}^{\overline{k}}.\,\tau_1) \;=\; \forall\overline{\alpha}^{\overline{k}}.\,\theta(\tau_2)$, then $\mathit{unifyOp}(\Xi,\ \forall\overline{\alpha}^{\overline{k}}.\,\tau_1,\ \tau_2) \;=\; (\Xi_1,\ \theta_1)$ and there exists $\vdash\ \theta_2 \;:\; \Xi_1 \Rightarrow \Xi_2$ such that $\theta \;=\; \theta_2 \circ \theta_1$.

5.2. Type Inference Algorithm

In Figures 8 and 9, we define the type inference algorithm. The algorithm is an extension of Algorithm W [3] with kinding and a row-based effect system. The functions $\mathit{infer}(\cdot,\ \cdot,\ \cdot,\ \cdot)$ and $\mathit{inferHandler}(\cdot,\ \cdot,\ \cdot,\ \cdot)$ are defined by mutual induction. Given a quadruple $(\Xi,\ \Gamma,\ \Delta,\ e)$ of a type variable context, two typing contexts, and an expression, *infer* returns a quadruple $(\Xi,\ \theta,\ \tau,\ \epsilon)$ of a new type variable context, a substitution, a monomorphic type, and an effect row. The effect row ϵ represents the effect performed by the expression e.

Let us go through individual cases. In the variable case, if x has a closed function type and resides in Δ, *infer* yields the most general type by opening the closed effect row to μ_1. The *infer* function also yields an arbitrary effect μ_2 as x is a value. If x does not have a closed function type, *infer* generates a new type variable $\overline{\beta}^{\overline{k}}$ as in a standard type inference algorithm.

The abstraction and application cases are standard, except that they involve inference of effect rows.

In the case of a `let` expression, the bound expression is generalized by Gen and the effect row of the function type is closed by Close. Then, the type context Δ is extended with the closed effect row and used for the inference of the body of the `let` expression. Consider the inference of the following `let` expression.

$$\mathsf{let}\, f \;=\; \lambda x.\, x\, \mathsf{in}\, f\ 1$$

The type of f is inferred to be $\forall\alpha.\,\alpha \to \mu\,\alpha$, where μ is an effect variable. Since the effect row of the function is closed immediately after generalization, the inference of the body $f\ 1$ is done by $\mathit{infer}(\Xi,\ \Gamma,\ (\Delta,\ f :\ \forall\alpha.\,\alpha \to \langle\,\rangle\,\alpha)\,,\ f\ 1)$.

$$
\begin{array}{l}
\mathit{infer} \ : \ (\mathsf{KCtx} \times \mathsf{TCtx} \times \mathsf{TCtx} \times \mathsf{Exp}) \to (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{MType} \times \mathsf{Eff}) \\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ x) \ = \\
\quad \mid x \ : \ \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, .., l_n \rangle\, \tau_2 \ \in \Delta \\
\qquad \Rightarrow \ \mathsf{assume}\, \overline{\beta}^{\overline{k}},\ \mu_1 \,\text{and}\, \mu_2 \,\text{are fresh.} \\
\qquad\quad \mathsf{return}\ ((\Xi,\ \overline{\beta}^{\overline{k}},\ \mu_1,\ \mu_2),\ \mathit{id},\ \mathit{id}[\overline{\alpha}^{\overline{k}} := \overline{\beta}^{\overline{k}}](\tau_1 \to \langle l_1, .., l_n \mid \mu_1 \rangle\, \tau_2),\ \mu_2) \\
\quad \mid x \ : \ \forall \overline{\alpha}^{\overline{k}}.\, \tau \ \in \Gamma,\ \Delta \\
\qquad \Rightarrow \ \mathsf{assume}\, \overline{\beta}^{\overline{k}} \,\text{and}\, \mu \,\text{are fresh.} \\
\qquad\quad \mathsf{return}\ ((\Xi,\ \overline{\beta}^{\overline{k}},\ \mu),\ \mathit{id},\ \mathit{id}[\overline{\alpha}^{\overline{k}} := \overline{\beta}^{\overline{k}}]\tau,\ \mu) \\
\\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \lambda x.\, e) \ = \\
\quad \mathsf{assume}\, \alpha^* \,\text{and}\, \mu \,\text{are fresh.} \\
\quad \text{let}\ (\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1) \ = \ \mathit{infer}((\Xi,\ \alpha^*),\ (\Gamma,\ x \ : \ \alpha^*),\ \Delta,\ e) \\
\quad \mathsf{return}\ (\Xi_1,\ \theta_1,\ \theta_1(\alpha^*) \to \epsilon_1\, \tau_1,\ \mu) \\
\\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e_1\ e_2) \ = \\
\quad \mathsf{assume}\, \alpha^* \,\text{is fresh.} \\
\quad \text{let}\ (\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1) \ = \ \mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ e_1) \\
\quad \text{let}\ (\Xi_2,\ \theta_2,\ \tau_2,\ \epsilon_2) \ = \ \mathit{infer}(\Xi_1,\ \theta_1(\Gamma),\ \theta_1(\Delta),\ e_2) \\
\quad \text{let}\ (\Xi_3,\ \theta_3) \ = \ \mathit{unify}(\Xi_2,\ \theta_2(\tau_1),\ \tau_2 \to \epsilon_2\, \alpha^*) \\
\quad \text{let}\ (\Xi_4,\ \theta_4) \ = \ \mathit{unify}(\Xi_3,\ (\theta_3 \circ \theta_2)(\epsilon_1),\ \theta_3(\epsilon_2)) \\
\quad \mathsf{return}\ (\Xi_4,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\ (\theta_4 \circ \theta_3)(\alpha^*),\ (\theta_4 \circ \theta_3)(\epsilon_2)) \\
\\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{let}\ x \ = \ v_1\ \mathsf{in}\ e_2) \ = \\
\quad \text{let}\ (\Xi_1,\ \theta_1,\ \tau_1,\ \epsilon_1) \ = \ \mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ v_1) \\
\quad \text{let}\ (\Xi_2,\ \theta_2) \ = \ \mathit{unify}(\Xi_1,\ \epsilon_1,\ \langle\,\rangle) \\
\quad \text{let}\ \forall \overline{\alpha}^{\overline{k}}.\, \theta_2(\tau_1) \ = \ \mathsf{Gen}((\theta_2 \circ \theta_1)(\Gamma),\ (\theta_2 \circ \theta_1)(\Delta),\ \theta_2(\tau_1)) \\
\quad \text{let}\ \sigma \ = \ \mathsf{Close}(\forall \overline{\alpha}^{\overline{k}}.\, \theta_2(\tau_1)) \\
\quad \text{let}\ (\Xi_3,\ \theta_3,\ \tau_2,\ \epsilon_2) \ = \ \mathit{infer}(\Xi_2 \setminus \overline{\alpha}^{\overline{k}},\ (\theta_2 \circ \theta_1)(\Gamma),\ ((\theta_2 \circ \theta_1)(\Delta),\ x \ : \ \sigma),\ e_2) \\
\quad \mathsf{return}\ (\Xi_3,\ (\theta_3 \circ \theta_2 \circ \theta_1),\ \tau_2,\ \epsilon_2) \\
\\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{perform}\ \mathit{op}) \ = \\
\quad \mathsf{assume}\, \overline{\beta}^{\overline{k}},\ \mu_1 \,\text{and}\, \mu_2 \,\text{are fresh.} \\
\quad \text{let}\ \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \tau_2 \ = \ \mathsf{Op}(\Sigma,\ \mathit{op}) \\
\quad \mathsf{return}\ ((\Xi,\ \overline{\beta}^{\overline{k}},\ \mu_1,\ \mu_2),\ \mathit{id},\ \mathit{id}[\overline{\alpha}^{\overline{k}} := \overline{\beta}^{\overline{k}}](\tau_1 \to \mu_1\, \tau_2),\ \mu_2) \\
\\
\mathit{infer}(\Xi,\ \Gamma,\ \Delta,\ \mathsf{handler}\ h) \ = \\
\quad \mathsf{assume}\, \mu \,\text{is fresh.} \\
\quad \text{let}\ (\Xi_1,\ \theta_1,\ \tau_1,\ l_1,\ \epsilon_1) \ = \ \mathit{inferHandler}(\Xi,\ \Gamma,\ \Delta,\ h) \\
\quad \mathsf{return}\ ((\Xi,\ \mu),\ \theta_1,\ (() \to \langle l_1 \mid \epsilon_1 \rangle\, \tau_1) \to \epsilon_1\, \tau_1,\ \mu)
\end{array}
$$

Figure 8. Type Inference Algorithm

$$
\begin{array}{l}
\mathit{inferHandler} \,:\, (\mathsf{KCtx} \times \mathsf{TCtx} \times \mathsf{TCtx} \times \mathsf{Hnd}) \to (\mathsf{KCtx} \times \mathsf{Subst} \times \mathsf{MType} \times \mathsf{Lab} \times \mathsf{Eff}) \\
\mathit{inferHandler}(\Xi,\, \Gamma,\, \Delta,\, \{ op_1 \to v_1,\, \ldots,\, op_n \to v_n \}) \,=\, \\
\quad \mathsf{assume}\, \beta^* \,\text{and}\, \mu \,\text{are fresh.} \\
\quad \text{let}\, l \,=\, \mathsf{Label}(\Sigma,\, op_1) \\
\quad \mathsf{assert}\, \Sigma(l) \,=\, \{ op_1, \ldots, op_n \} \\
\quad \text{let}\, (\Xi_0,\, \theta_0) \,=\, (\Xi,\, \mathit{id}) \\
\quad \text{for}\, i \in \{1, \ldots, n\} \\
\quad\quad \mathsf{assume}\, \overline{\alpha}_i^k \,\text{are fresh.} \\
\quad\quad \text{let}\, \forall \overline{\alpha}_i^k.\, \tau_1^i \to \tau_2^i \,=\, \mathsf{Op}(\Sigma,\, op_i) \\
\quad\quad \text{let}\, (\Xi_i^1,\, \theta_i^1,\, \tau_i,\, \epsilon_i) \,=\, \mathit{infer}(\Xi_{i-1}^3,\, \theta_{i-1}(\Gamma),\, \theta_{i-1}(\Delta),\, v_i) \\
\quad\quad \text{let}\, (\Xi_i^2,\, \theta_i^2) \,=\, \mathit{unify}(\Xi_i^1,\, \epsilon_i,\, \langle\,\rangle) \\
\quad\quad \text{let}\, (\Xi_i^3,\, \theta_i^3) \\
\quad\quad\quad =\, \mathit{unifyOp}(\Xi_i^2,\, (\theta_i^2 \circ \theta_i^1 \circ \theta_{i-1})(\forall \overline{\alpha}^k.\, \tau_i^1 \to \mu\, ((\tau_i^2 \to \mu\, \beta^*) \to \mu\, \beta^*)),\, \theta_i^2(\tau_i)) \\
\quad\quad \text{let}\, \theta_i \,=\, \theta_i^3 \circ \theta_i^2 \circ \theta_i^1 \circ \theta_{i-1} \\
\quad\quad \mathsf{assert}\, \overline{\alpha}_i^k \notin \mathsf{ftv}(\theta_i(\Gamma),\, \theta_i(\Delta)) \\
\quad \mathsf{return}\, (\Xi_n^3,\, \theta_n,\, \theta_n(\beta^*),\, l,\, \theta_n(\mu))
\end{array}
$$

Figure 9. Type Inference Algorithm for Handlers

In the case of an operation call, *infer* simply returns the type instantiated with a new effect variable. Here, $\mathsf{Op}(\cdot, \cdot)$ is an auxiliary function that selects from Σ the signature of the operation *op*.

In the handler case, we use two auxiliary functions *inferHandler* and Label. The *inferHandler* function infers the type of a handler. It receives a quadruple (Ξ, Γ, Δ, h) of a type variable context, two typing contexts, and a handler, and returns a quintuple $(\Xi, \theta, \tau, l, \epsilon)$ of a new type variable context, a substitution, a monomorphic type, an effect label, and an effect row. Here, τ is the return type of the continuation captured by the handler h, l is the effect label handled by h, and ϵ is the rest of the effect row. The Label function returns the effect label corresponding to the given operation.

It is important to use *unifyOp* instead of *unify* in the type inference of handlers. For example, consider the following effect signature and handler:

$$
\begin{array}{l}
\Sigma \,=\, \{\, l_1 \,:\, \{\, op \,:\, \forall \alpha.\, \alpha \to \alpha \,\}\} \\
\mathsf{handler}\, \{\, op \to \lambda x\, k.\, k\, (x \,+\, 1) \,\}
\end{array}
$$

This handler should be rejected for the following reason. First, the operation *op* is defined as having type $\forall \alpha.\, \alpha \to \alpha$. Therefore, the operation clause of *op* must have a type of the form $\forall \alpha.\, \alpha \to \mu\, ((\alpha \to \mu\, \beta) \to \mu\, \beta)$, where the input and output types of the operation are universally quantified. Second, the operation clause $\lambda x\, k.\, k\, (x + 1)$ is inferred to have type $\mathsf{int} \to \mu\, ((\mathsf{int} \to \mu\, \beta) \to \mu\, \beta)$, where the input and output types are a concrete type int. If we use *unifyOp*, unification of $\forall \alpha.\, \alpha \to \mu\, ((\alpha \to \mu\, \beta) \to \mu\, \beta$ and $\mathsf{int} \to \mu\, ((\mathsf{int} \to \mu\, \beta) \to \mu\, \beta)$ fails, because the bound variable α and int cannot be unified. On the other hand, if we use *unify*, unification of the two types succeeds, because we would pass a monomorphic type $\alpha \to \mu\, ((\alpha \to \mu\, \beta) \to \mu\, \beta)$ to *unify*, which can be unified with $\mathsf{int} \to \mu\, ((\mathsf{int} \to \mu\, \beta) \to \mu\, \beta)$.

The soundness and completeness with respect to the syntax-directed inference rules of *infer* can be proven by induction on the structure of e.

Theorem 9. (*infer is sound with respect to syntax-directed inference rules*)
If $\mathit{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta, \tau, \epsilon)$, then $\vdash \theta : \Xi \Rightarrow \Xi_1$ and $\Xi_1 \mid \theta(\Gamma) \mid \theta(\Delta) \Vdash_{\mathsf{s}} e : \tau \mid \epsilon$.

Theorem 10. (*infer is complete with respect to syntax-directed inference rules*)
If $\vdash \theta : \Xi \Rightarrow \Xi_2$ and $\Xi_2 \mid \theta(\Gamma) \mid \theta(\Delta) \Vdash_{\mathsf{s}} e : \tau \mid \epsilon$,
then $\mathit{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta_1, \tau, \epsilon)$, and there exists $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$ such that $\theta = \theta_2 \circ \theta_1$.

Using the results so far, we can prove the main theorems: the type inference algorithm for the declarative inference rules is sound and complete.

Theorem 11. (*infer is sound*)
If $\mathit{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta, \tau, \epsilon)$, then $\vdash \theta : \Xi \Rightarrow \Xi_1$ and $\Xi_1 \mid \theta(\Gamma) \mid \theta(\Delta) \vdash e : \tau \mid \epsilon$.

Proof. By Theorem 1 and Theorem 9.

Theorem 12. (*infer is complete*)
If $\vdash \theta : \Xi \Rightarrow \Xi_2$ and $\Xi_2 \mid \theta(\Gamma) \mid \theta(\Delta) \vdash e : \tau \mid \epsilon$,
then $\mathit{infer}(\Xi, \Gamma, \Delta, e) = (\Xi_1, \theta_1, \tau, \epsilon)$, and there exists $\vdash \theta_2 : \Xi_1 \Rightarrow \Xi_2$ such that $\theta = \theta_2 \circ \theta_1$.

Proof. By Theorem 2 and Theorem 10.

6. Related Work

There are a variety of languages supporting effect handlers in the literature. Eff [1] is an ML-like language that employs the Hindley-Milner type inference [9, 17]. Differently from Koka, Eff has an effect system based on subtyping. As a result, the type inference algorithm [9] of Eff is more complex than the one presented in this paper.

Frank [2, 14] is a language that has effect rows and effect polymorphism similar to Koka. The difference is that Frank treats all effect rows as open ones by implicitly inserting effect variables. This means the user does not need to write type variables to express effect polymorphism, but it also means error messages may contain effect variables that the user did not write.

Links [5] is another language with a row-based effect system and effect polymorphism. What is different from Koka is that effect rows in Links are based on Remy's record types [18], where each effect label is annotated with a presence type. Presence types increase the expressiveness of the language, but they also complicate the inference algorithm.

There is a type inference algorithm for an older version of Koka [12], which solely supports built-in effects such as exceptions and references. Similar to the current Koka, it has effect rows [10] and effect polymorphism. The type inference algorithm is an extension of the Hindley-Milner algorithm, but it infers an open effect row for all functions due to the lack of open and close.

7. Conclusion and Future Work

In this paper, we formalized a type inference algorithm with open and close and proved its soundness and completeness. The inference algorithm helps the Koka compiler statically determine handlers, and thus improve performance. Moreover, it allows the compiler to display precise signatures.

In future work, we plan to improve the current typing rules in order to make the typability of programs robust against small syntactic rewrites.

Appendix

In this appendix, we present an extension of System F^ϵ, which we call System F^ϵ+restrict, and a type-directed translation from ImpKoka to System F^ϵ+restrict. The translation allows us to prove the type soundness of ImpKoka without directly defining an operational semantics for ImpKoka. This is a well-known technique, and is used in [11], for instance. The target calculus of the translation has a new construct restrict, which is necessary for establishing soundness of the translation.

A. System F^ϵ+restrict

In Figures 11 to 13, we define the syntax, operational semantics, and typing rules of System F^ϵ+restrict. The typing rule [Restrict] extends the closed effect row $\langle l_1, .., l_n \rangle$ of the expression e to $\langle l_1, \ldots l_n \mid \epsilon \rangle$, where ϵ is an arbitrary effects.

We can prove the type soundness of Sysmtem F^ϵ+restrict by showing the fllowing theorems.

Theorem 13.
If $\emptyset \vdash e_1 : \sigma \mid \langle\,\rangle$ then either e_1 is a value or $e_1 \longmapsto e_2$.

Theorem 14.
If $\emptyset \vdash e_1 : \sigma \mid \langle\,\rangle$ and $e_1 \longmapsto e_2$ then $\emptyset \vdash e_2 : \sigma \mid \langle\,\rangle$.

B. Type-directed Translation to System F^ϵ

In Figure 10, we next define the type-directed translation from ImpKoka to System F^ϵ+restrict. The judgment $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$ states that an expression e has type σ and effect ϵ under the type variable context Ξ and typing contexts Γ and Δ, and translates to an expression e' of System F^ϵ+restrict. We can easily prove the soundness of the type-directed translation.

Theorem 15.
If $\Xi \mid \Gamma \mid \Delta \vdash e : \sigma \mid \epsilon \rightsquigarrow e'$ then $\Gamma, \Delta \vdash e' : \sigma \mid \epsilon$.

Proof. By straightforward induction on the typing derivation.

$$\boxed{\underset{\uparrow}{\Xi} \mid \underset{\uparrow}{\Gamma} \mid \underset{\uparrow}{\Delta} \vdash \underset{\uparrow}{e} : \underset{\uparrow}{\sigma} \mid \underset{\uparrow}{\epsilon} \rightsquigarrow \underset{\downarrow}{e'} \qquad \underset{\uparrow}{\Xi} \mid \underset{\uparrow}{\Gamma} \mid \underset{\uparrow}{\Delta} \vDash \underset{\uparrow}{h} : \underset{\uparrow}{\tau} \mid \underset{\uparrow}{l} \mid \underset{\uparrow}{\epsilon} \rightsquigarrow \underset{\downarrow}{h'}}$$

$$\frac{x : \sigma \in \Gamma,\, \Delta \quad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \qquad \sigma \text{ not a closed function type}}{\Xi \mid \Gamma \mid \Delta \vdash x : \sigma \mid \epsilon \rightsquigarrow x} \;[\textsc{Var}]$$

$$\frac{f : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \rangle\, \tau_2 \in \Delta \qquad \Xi \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\begin{array}{l}\Xi \mid \Gamma \mid \Delta \vdash f : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \langle l_1, \ldots, l_n \mid \epsilon \rangle\, \tau_2 \mid \epsilon' \\ \quad \rightsquigarrow \Lambda \overline{\alpha}^{\overline{k}}.\, \lambda^{\langle l_1, \ldots, l_n \mid \epsilon \rangle}\, x : \tau_1.\, \mathsf{restrict}^{\langle l_1, \ldots, l_n \rangle}\, f[\overline{\alpha}^{\overline{k}}]\, x\end{array}} \;[\textsc{VarOpen}]$$

$$\frac{\Xi \mid \Gamma, x : \tau_1 \mid \Delta \vdash e : \tau_2 \mid \epsilon \rightsquigarrow e' \quad \Xi \vdash_{\mathsf{wf}} \tau_1 : * \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \lambda x.\, e : \tau_1 \to \epsilon\, \tau_2 \mid \epsilon' \rightsquigarrow \lambda^{\epsilon} x : \tau_1.\, e'} \;[\textsc{Lam}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e_1 : \tau_2 \to \epsilon\, \tau \mid \epsilon \rightsquigarrow e_1' \quad \Xi \mid \Gamma \mid \Delta \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow e_2'}{\Xi \mid \Gamma \mid \Delta \vdash e_1\, e_2 : \tau \mid \epsilon \rightsquigarrow e_1'\, e_2'} \;[\textsc{App}]$$

$$\frac{\Xi, \alpha^k \mid \Gamma \mid \Delta \vdash v : \sigma \mid \langle\rangle \rightsquigarrow v' \quad k \not\equiv \mathsf{lab}}{\Xi \mid \Gamma \mid \Delta \vdash v : \forall \alpha^k.\, \sigma \mid \langle\rangle \rightsquigarrow \Lambda \overline{\alpha}^{\overline{k}}.\, v'} \;[\textsc{Gen}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash e : \forall \alpha^k.\, \sigma \mid \epsilon \rightsquigarrow e' \quad \Xi \vdash_{\mathsf{wf}} \tau : k}{\Xi \mid \Gamma \mid \Delta \vdash e : \sigma[\alpha^k := \tau] \mid \epsilon \rightsquigarrow e'[\tau]} \;[\textsc{Inst}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vdash v_1 : \sigma_1 \mid \langle\rangle \rightsquigarrow v_1' \quad \Xi \mid \Gamma \mid \Delta, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow e_2'}{\Gamma \vdash \mathsf{let}\, x = v_1\, \mathsf{in}\, e_2 : \sigma_2 \mid \epsilon \rightsquigarrow (\lambda^{\epsilon} x : \sigma_1.\, e_2')\, v_1'} \;[\textsc{Let}]$$

$$\frac{op : \forall \overline{\alpha}^{\overline{k}}.\, \tau_1 \to \tau_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \Xi \quad \Xi \vdash_{\mathsf{wf}} \overline{\tau} : \overline{k} \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{perform}\, op : (\tau_1 \to \langle l \mid \epsilon \rangle\, \tau_2)[\overline{\alpha}^{\overline{k}} := \overline{\tau}] \mid \epsilon' \rightsquigarrow \mathsf{perform}^{\epsilon}\, op\, \overline{\tau}} \;[\textsc{Perform}]$$

$$\frac{\Xi \mid \Gamma \mid \Delta \vDash h : \tau \mid l \mid \epsilon \rightsquigarrow h' \quad \Xi \vdash_{\mathsf{wf}} \epsilon' : \mathsf{eff}}{\Xi \mid \Gamma \mid \Delta \vdash \mathsf{handler}\, h : (() \to \langle l \mid \epsilon \rangle\, \tau) \to \epsilon\, \tau) \mid \epsilon' \rightsquigarrow \mathsf{handler}^{\epsilon}\, h'} \;[\textsc{Handler}]$$

$$\frac{\begin{array}{c} op_i : \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \tau_2^i \in \Sigma(l) \quad \overline{\alpha}_i^{\overline{k}} \notin \mathsf{ftv}(\epsilon, \tau) \\ \Xi \mid \Gamma \mid \Delta \vdash v_i : \forall \overline{\alpha}_i^{\overline{k}}.\, \tau_1^i \to \epsilon\, ((\tau_2^i \to \epsilon\, \tau) \to \epsilon\, \tau) \mid \langle\rangle \rightsquigarrow v_i' \end{array}}{\Xi \mid \Gamma \mid \Delta \vDash \{op_1 \to v_1, \ldots, op_n \to v_n\} : \tau \mid l \mid \epsilon \rightsquigarrow \{op_1 \to v_1', \ldots, op_n \to v_n'\}} \;[\textsc{Ops}]$$

Figure 10. Translation to System F^{ϵ}+restrict

Types				Kinds		
σ	$::=$ α^k	(type variables of kind k)	k	$::=$ $*$	(value type)
	$\mid$ $c^k\,\tau\ldots\tau$	(type constructor of kind k)		$\mid$ $k\to k$	(type constructors)
	$\mid$ $\sigma\to\epsilon\,\sigma$	(function type)		$\mid$ eff	(effect type (μ,ϵ))
	$\mid$ $\forall\alpha^k.\,\sigma$	(quantified type)		$\mid$ lab	(basic effect (l))

Effect signature	sig	$::=$	$\{\, op_1 \;:\; \forall\overline{\alpha}_1.\,\sigma_1\to\sigma'_1,\ \ldots,\ op_n \;:\; \forall\overline{\alpha}_n.\,\sigma_n\to\sigma'_n\}$	
Effect signatures	Σ	$::=$	$\{l_1 \;:\; sig_1,\ \ldots,\ l_n \;:\; sig_n\}$	
Type Constructors	$\langle\,\rangle$	$:$	eff	empty effect row (total)
	$\langle \_ \mid \_ \rangle$	$:$	$\mathsf{lab}\to\mathsf{eff}\to\mathsf{eff}$	effect row extension
Syntax	$\langle l_1,\ldots,l_n\rangle$	$\doteq$	$\langle l_1 \mid \ldots \mid \langle l_n \mid \langle\,\rangle\rangle \ldots \rangle$	
	$\langle l_1,\ldots,l_n \mid \mu\rangle$	$\doteq$	$\langle l_1 \mid \ldots \mid \langle l_n \mid \mu\rangle \ldots \rangle$	
	$\epsilon ::= \sigma^{\mathsf{eff}}, \quad \mu ::= \alpha^{\mathsf{eff}}, \quad l ::= c^{\mathsf{lab}}$			

Figure 11. Types of System F$^\epsilon$+restrict

Expressions				Values		
e	$::=$ v	(value)	v	$::=$ x	(variables)
	$\mid$ $e\,e$	(application)		$\mid$ $\lambda^\epsilon x:\sigma.\,e$	(abstraction)
	$\mid$ $e[\sigma]$	(type application)		$\mid$ $\Lambda\alpha^k.\,e$	(type abstraction)
	$\mid$ $\mathsf{handle}\,h\,e$	(handler instance)		$\mid$ $\mathsf{handler}^\epsilon\,h$	(effect handler)
	$\mid$ $\mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\,v$	(restrict)		$\mid$ $\mathsf{perform}^\epsilon\,op\,\overline{\sigma}$	(operation)

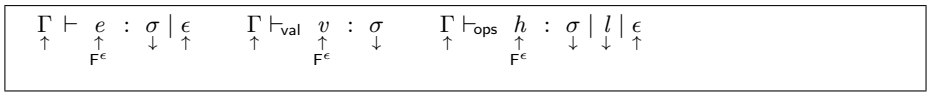
Handlers	h	$::=$	$\{op_1\to f_1,\ \ldots,\ op_n\to f_n\}$
Evaluation Context	F	$::=$	$\square \mid \mathsf{F}\,e \mid v\,\mathsf{F} \mid \mathsf{F}\,\sigma$
		$\mid$	$\mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\,\mathsf{F}$
	E	$::=$	$\square \mid \mathsf{E}\,e \mid v\,\mathsf{E} \mid \mathsf{E}\,\sigma$
		$\mid$	$\mathsf{handle}\,h\,\mathsf{E} \mid \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\,\mathsf{E}$

(*app*)	$(\lambda^\epsilon x:\sigma.\,e)\,v$	$\longrightarrow$	$e[x{:=}v]$	
(*handler*)	$(\mathsf{handler}^\epsilon\,h)\,v$	$\longrightarrow$	$\mathsf{handle}\,h\,(v\,())$	
(*return*)	$\mathsf{handle}\,h\,v$	$\longrightarrow$	v	
(*perform*)	$\mathsf{handle}\,h\,\mathsf{E}[\mathsf{perform}\;op\,\overline{\sigma}\;v]$	$\longrightarrow$	$f[\overline{\sigma}]\,v\,k$	iff $op\notin\mathsf{bop}(\mathsf{E})\ \wedge\ (op\to f)\in h$ where $op\ :\ \forall\overline{\alpha}.\,\sigma_1\to\sigma_2\ \in\Sigma(l)$
(*restrict*)	$\mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\,v$	$\longrightarrow$	v	

$$\frac{e\longrightarrow e'}{\mathsf{E}[e]\longmapsto\mathsf{E}[e']}\ [\textsc{step}]$$

$$\begin{aligned}
&\mathsf{bop}(\square) &&= \emptyset\\
&\mathsf{bop}(\mathsf{E}\,e) &&= \mathsf{bop}(\mathsf{E})\\
&\mathsf{bop}(v\,\mathsf{E}) &&= \mathsf{bop}(\mathsf{E})\\
&\mathsf{bop}(\mathsf{handle}\,h\,\mathsf{E}) &&= \mathsf{bop}(\mathsf{E})\ \cup\ \{op\mid(op\to f)\in h\}\\
&\mathsf{bop}(\mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\,\mathsf{E}) &&= \mathsf{bop}(\mathsf{E})
\end{aligned}$$

Figure 12. Expressions and Operational Sematics of F$^\epsilon$+restrict



$$\frac{\Gamma \vdash e : \sigma \mid \langle l_1, \ldots, l_n \rangle \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash \mathsf{restrict}^{\langle l_1,\ldots,l_n\rangle}\, e : \sigma \mid \langle l_1, \ldots, l_n \mid \epsilon \rangle}\ [\textsc{Restrict}] \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x : \sigma}\ [\textsc{Var}]$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \sigma_2 \mid \epsilon \quad \Gamma \vdash_{\mathsf{wf}} \sigma_1 : *}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon} x : \sigma_1.\, e : \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{Lam}] \qquad \frac{\Gamma \vdash_{\mathsf{val}} v : \sigma \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash v : \sigma \mid \epsilon}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash e_1 : \sigma_2 \to \epsilon\, \sigma \mid \epsilon \quad \Gamma \vdash e_2 : \sigma_2 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \sigma \mid \epsilon}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash e : \sigma \mid \langle\,\rangle \quad \alpha^k \notin \mathsf{ftv}(\Gamma) \quad k \not\equiv \mathsf{lab}}{\Gamma \vdash_{\mathsf{val}} \Lambda\alpha^k.e : \forall\alpha.\, \sigma}\ [\textsc{TAbs}]$$

$$\frac{\Gamma \vdash e : \forall\alpha^k.\sigma_1 \mid \epsilon \quad \Gamma \vdash_{\mathsf{wf}} \sigma' : k}{\Gamma \vdash e\,[\sigma'] : \sigma[\alpha{:=}\sigma'] \mid \epsilon}\ [\textsc{TApp}]$$

$$\frac{op : \forall\overline{\alpha}^{\overline{k}}.\, \sigma_1 \to \sigma_2 \in \Sigma(l) \quad \overline{\alpha} \notin \mathsf{ftv}(\Gamma) \quad \Gamma \vdash_{\mathsf{wf}} \epsilon : \mathsf{eff}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}^{\epsilon}\, \mathsf{op}\, \overline{\sigma} : \sigma_1[\overline{\alpha}^{\overline{k}} := \overline{\sigma}] \to \langle l \mid \epsilon \rangle\, \sigma_2[\overline{\alpha}^{\overline{k}} := \overline{\sigma}]}\ [\textsc{Perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon \quad \Gamma \vdash e : \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \mathsf{handle}^{\epsilon}\, h\, e : \sigma \mid \epsilon}\ [\textsc{Handle}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\, h : (() \to \langle l \mid \epsilon \rangle\, \sigma) \to \epsilon\, \sigma}\ [\textsc{Handler}]$$

$$\frac{op_i : \forall\overline{\alpha}^{\overline{k}}.\, \sigma_1 \to \sigma_2 \in \Sigma(l) \quad \overline{\alpha}^{\overline{k}} \notin \mathsf{ftv}(\Gamma) \quad \Gamma \vdash_{\mathsf{val}} v_i : \forall\overline{\alpha}^{\overline{k}}.\, \sigma_1 \to \epsilon\, ((\sigma_2 \to \epsilon\, \sigma) \to \sigma)}{\Gamma \vdash_{\mathsf{ops}} \{op_1 \to v_1, \ldots, op_n \to v_n\} : \sigma \mid l \mid \epsilon}\ [\textsc{Ops}]$$

Figure 13. Typing Rules of F^{ϵ}+restrict

References

[1] Andrej Bauer, and Matija Pretnar. "Programming with Algebraic Effects and Handlers." *Journal of Logical and Algebraic Methods in Programming* 84 (1). Elsevier: 108–123. 2015.

[2] Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. "Doo Bee Doo Bee Doo." *In the Journal of Functional Programming*, January. Jan. 2020. To appear in the special issue on algebraic effects and handlers.

[3] Luís Damas, and Robin Milner. "Principal Type-Schemes for Functional Programs." In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, edited by Richard A. DeMillo, 207–212. ACM Press. 1982. doi:10.1145/582153.582176.

[4] Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. "FreezeML: Complete and Easy Type Inference for First-Class Polymorphism." In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, edited by Alastair F. Donaldson and Emina Torlak, 423–437. ACM. 2020. doi:10.1145/3385412.3386003.

[5] Daniel Hillerström, and Sam Lindley. "Liberating Effects with Rows and Handlers." In *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, edited by James Chapman and Wouter Swierstra, 15–27. ACM. 2016. doi:10.1145/2976022.2976033.

[6] Mark P. Jones. "A Theory of Qualified Types." In *4th. European Symposium on Programming (ESOP'92)*, 582:287–306. Lecture Notes in Computer Science. Springer-Verlag, Rennes, France. Feb. 1992. doi:10.1007/3-540-55253-7_17.

[7] Ohad Kammar, and Gordon D. Plotkin. "Algebraic Foundations for Effect-Dependent Optimisations." In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 349–360. POPL '12. ACM, New York, NY, USA. 2012. doi:10.1145/2103656.2103698.

[8] Ohad Kammar, and Matija Pretnar. "No Value Restriction Is Needed for Algebraic Effects and Handlers." *Journal of Functional Programming* 27 (1). Cambridge University Press. Jan. 2017. doi:10.1017/S0956796816000320.

[9] Georgios Karachalias, Matija Pretnar, Amr Hany Saleh, Stien Vanderhallen, and Tom Schrijvers. "Explicit Effect Subtyping." *J. Funct. Program.* 30: e15. 2020. doi:10.1017/S0956796820000131.

[10] Daan Leijen. "Extensible Records with Scoped Labels." In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312. 2005.

[11] Daan Leijen. "HMF: Simple Type Inference for First-Class Polymorphism." In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. Sep. 2008. doi:10.1145/1411204.1411245.

[12] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types." In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

[13] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

[14] Sam Lindley, Connor McBride, and Craig McLaughlin. "Do Be Do Be Do." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles*

of Programming Languages (POPL'17), 500–514. Paris, France. Jan. 2017. doi:10.1145/3009837.3009897.

[15] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. *Wobbly Types: Type Inference For Generalised Algebraic Data Types*. MS-CIS-05-26. Jul. 2004. Microsoft Research.

[16] Gordon D. Plotkin, and John Power. "Algebraic Operations and Generic Effects." *Applied Categorical Structures* 11 (1): 69–94. 2003. doi:10.1023/A:1023064908962.

[17] Matija Pretnar. "Inferring Algebraic Effects." *Log. Methods Comput. Sci.* 10 (3). 2014. doi:10.2168/LMCS-10(3:21)2014.

[18] Didier Rémy. "Type Inference for Records in Natural Extension of ML." In *Theoretical Aspects of Object-Oriented Programming*, 67–95. 1994. doi:10.1.1.48.5873.

[19] John Alan Robinson. "A Machine-Oriented Logic Based on the Resolution Principle." *J. ACM* 12 (1): 23–41. 1965. doi:10.1145/321250.321253.

[20] Andrew K. Wright. "Simple Imperative Polymorphism." *LISP Symb. Comput.* 8 (4): 343–355. 1995.

[21] Ningning Xie, and Daan Leijen. "Effect Handlers in Haskell, Evidently." In *Proceedings of the 2020 ACM SIGPLAN Symposium on Haskell*. Haskell'20. Jersey City, NJ. Aug. 2020. doi:10.1145/3406088.3409022.

[22] Ningning Xie, and Daan Leijen. "Generized Evidence Passing for Effect Handlers – Efficient Compilation of Effect Handlers to C." In *Proceedings of the 26th ACM SIGPLAN International Conference on Functional Programming (ICFP'2021)*. ICFP '21. Virtual. Aug. 2021.

Named Arguments as Records
(Research Paper)

Yaozhu Sun* and Bruno C. d. S. Oliveira

The University of Hong Kong, China
{yzsun,bruno}@cs.hku.hk

Abstract. Named arguments are commonly supported in mainstream programming languages. However, there has been little work on formalizing the design of named arguments.
This paper shows a minimal calculus that encodes named arguments as a form of records. Our design is based on a variant of record types, which allows *optional fields* and introduces two alternative notions of *open-bindings* and *failable projections*. With this design, we are able to model an expressive form of named arguments, which supports optional arguments as well. In our design, named arguments are commutative, and they are distinct from positional arguments. We present an extension to $\lambda_{<:}$ and discuss its semantics. Our main goal is to obtain a calculus that is as simple as possible but still captures most of the desirable features for named arguments.

Keywords: Named arguments · Optional arguments · Record types

1 Introduction

The λ-calculus, introduced by Alonzo Church, shows us how to model computation solely with function abstraction and application. In the λ-calculus, a function only has one parameter and can only be applied to one argument. Many programming languages in the ML family inherit this feature. If we want a function with multiple arguments in those languages, we need to turn it into a sequence of functions, each with a single argument, which is called *currying*. Currying brings brevity to functional programming and naturally allows partial application, but it also limits the flexibility of function application. For example, we cannot pass arguments in a different order nor omit some of them by providing default values. Both demands are not rare at all in practical programming and can be met in a language that supports *named arguments*.

Named arguments are commonly supported in mainstream programming languages, such as C#, Python, Ruby, and Scala, just to name a few. The earliest instance, to the best of our knowledge, is Smalltalk, where every argument must be associated with a keyword. The syntax of modern languages is usually less rigid, so programmers can choose whether to attach keywords to arguments or

* The main author is a student.

```python
def exp(x, base=math.e):
  return base ** x

exp(10) #= exp(x=10) = 22026
exp(base=2, x=10)   #=  1024
```

(a) The Python way

```ruby
def exp(x:, base: Math::E)
  base ** x
end
exp(10)    # ArgumentError!
exp(base: 2, x: 10) #= 1024
```

(b) The Ruby way

Fig. 1: Named arguments in two different ways

not. More specifically, there are two ways to handle named arguments. The more common way, shown in Fig. 1a, is to make variable names in the function definition as optional keywords. Thus every argument can be passed with or without keywords. As shown in the Python code, `exp(10)` is equivalent to `exp(x=10)`. To reconcile the two forms, a restriction is imposed that all positional arguments have to appear to the left of named ones. The other way, shown in Fig. 1b, is to strictly distinguish named arguments from positional ones. In Ruby, a named parameter should end with a colon even if it does not have a default value. Therefore, they are distinct from positional parameters, and their keywords cannot be omitted in a function call. In fact, these named arguments are desugared to a hash map.

Although named arguments are ubiquitous in practical programming, they do not attract enough attention in the research of programming languages. Among the few related papers, the work by Garrigue et al. [1,3,6] formalizes a label-selective λ-calculus and eventually applies it to OCaml [5]. We will discuss it in detail in Section 2.1. Another work by Rytz and Odersky [10] discusses the design of named and optional arguments in Scala but does not formalize it. In Haskell, the paradigm of *named arguments as records* is folklore, which will be elaborated in Section 2.2. From previous work in the literature, we identify four design choices related to named arguments:

1. *Commutativity*: whether the order of arguments can be different from that in the original definition.
2. *Optionality*: whether some arguments can be optional if their default values are given.
3. *Distinctness*: whether named arguments are distinct from positional arguments.
4. *Currying*: whether a function that takes more than one argument is always converted into a chain of functions that each take a single argument.

The first two properties hold for most mainstream programming languages that support named arguments. Commutativity and optionality are so useful that they should not be compromised. The third design is endorsed by Ruby and Racket, and we find it more intuitive than mixing two forms of arguments, so we advocate distinguishing them. In other words, we have to add argument keywords in the function application as long as they defined to be named. The last one,

currying, is important for functional languages. OCaml manages to integrate currying with commutativity, at the cost of introducing a very complicated core calculus. We agree that currying is very useful when we use normal positional arguments, but we argue that currying can be temporarily dropped when we use named arguments because the most common use case for named arguments is to represent a whole chunk of parameters like settings.

In this paper, we propose a new approach by desugaring named arguments into records in a minimal core calculus. The approach is simpler than the OCaml way and avoids some drawbacks of the design pattern used in Haskell. To some degree, we absorb the design pattern into the language support. Our approach supports commutativity and optionality but does not support currying when using named arguments. This is a trade-off between simplicity and expressiveness. Nevertheless, users have the right to choose either to use named arguments without currying or to use positional arguments with currying.

To make it clear, our goal is not to find a solution that is as powerful as OCaml has, but to propose a core calculus that is *as simple as possible* but still supports named and optional arguments via desugaring. We believe such a simple calculus leads to less feature interaction when integrated with other languages.

Our source language benefits from named arguments in two aspects. On the one hand, argument keywords serve as extra documentation at the language level. We do not need to open the code in an IDE and look up the definition of a function to figure out for what its arguments are used. This design is more friendly to those users who read code literally. Moreover, keywords are part of a function type, so we can know more from the type information without referring to ad-hoc documentation. On the other hand, named arguments lay the foundation for supporting commutativity and optionality. Without argument keywords, it is unclear how to naturally support these useful features.

In summary, the contributions of this paper are:

- We document the folklore paradigm of *named arguments as records* in Haskell.
- We demonstrate how a functional language with named and optional arguments can be desugared to a minimal core calculus.
- We propose two core calculi supporting optional fields in record types, which extend $\lambda_{<:}$ with open-bindings and failable projections respectively.
- In order to give a correct small-step operational semantics to open-bindings, we explore environment-based evaluation with closures instead of using textual substitution.

2 Named Arguments in Existing Functional Languages

In this section, we review the techniques used for named arguments in two existing functional languages, namely OCaml and Haskell. We will explain why their approaches still have some drawbacks.

2.1 OCaml

Originally, just like other languages in the ML family, OCaml did not support named arguments. Later, Garrigue et al. [1,3,6] conducted research on the label-selective λ-calculus and implemented it in OLabl [4]. OLabl extended OCaml with labeled and optional arguments, among others. All features of OLabl were merged into OCaml 3, though with subtle differences [5].

Here is an example of the exponential function defined in a labeled style:

```
let exp ?(base = 2.71828) x = base ** x
(* val exp : ?base:float → float → float = <fun> *)
exp 10.0              (*= 22026. *)
exp 10.0 ~base:2.0    (*=  1024. *)
(exp 10.0) ~base:2.0  (* TypeError! *)
```

In the definition of `exp`, `base` is an optional labeled parameter while `x` is a normal positional parameter. We cannot change `x` into a second labeled parameter here because OCaml imposes a restriction that there must be a positional parameter after all optional parameters. This restriction is at the heart of how OCaml resolves the ambiguity introduced by currying. For example, consider the function application `exp 10.0`. Is it a partially applied function or a fully applied one using the default value of `base`? The presence of the positional argument (`x` in this example) is used to decide whether such a function has been fully applied. So `exp 10.0` is considered to be a full application because `x` is already given. However, this feature may confuse users since `(exp 10.0) ~base:2.0` will raise a type error but `exp 10.0 ~base:2.0` will not. Currying does not seem to hold in such a situation.

In OCaml, optional arguments are internally implemented as `option` types. Here is an equivalent definition for `exp`, together with two examples of the transformation of optional arguments:

```
let exp ?base x =
  let base = match base with None → 2.71828 | Some b → b in
  base ** x
(* val exp : ?base:float → float → float = <fun> *)
exp 10.0            (*> exp 10.0 ~base:None *)
exp 10.0 ~base:2.0  (*> exp 10.0 ~base:(Some 2.0) *)
```

This encoding is quite natural, but it assumes `option` types to be built in. Unfortunately, there are plenty of languages that do not regard `option` as a built-in type, especially in those languages that do not support algebraic data types.

In short, the OCaml way cannot scale smoothly. OCaml has a very powerful label-selective core calculus that reconciles commutativity and currying, but it is quite complicated, hindering its integration with other languages. The assumption of `option` types makes the situation even worse. In contrast to OCaml, we want a minimal core calculus that supports named and optional arguments via desugaring.

```
data Settings = Settings
  { settingsPort :: Port
  , settingsHost :: HostPreference
  , settingsTimeOut :: Int
  , ...
  }
```

(a) The record type containing settings

```
defaultSettings = Settings
  { settingsPort = 3000
  , settingsHost = "*4"
  , settingsTimeout = 30
  , ...
  }
```

(b) Default values

```
runSettings :: Settings → Application → IO ()
runSettings = ...

main :: IO ()
main = runSettings settings app
  where settings = defaultSettings { settingsPort = 4000
                                   , settingsHost = "*6" }
```

(c) Update some settings before running a server application

Fig. 2: Named arguments as records in Haskell

2.2 Haskell

Unlike OCaml, Haskell does not support named arguments natively. However, the paradigm of *named arguments as records* has long existed in the Haskell community. Although we have to uncurry a function to have all parameters labeled in a record, it is clearer and more human-readable, especially when different parameters have the same type. For example, in the web server library *warp* [11], various server settings are bundled in the data type `Settings`, as shown in Fig. 2. It is obvious how named arguments correspond to record fields, but it needs some thought on how to encode default values for optional arguments. The simplest approach, also used by *warp*, is to define a record `defaultSettings`. Users can update whatever fields they want to change while keeping others.

Such an approach works fine here but still has two drawbacks. The first issue is the dependency on `defaultSettings`. It is awkward for users to look for a record containing particular default values, especially when there are quite a few similar records in a library. A better solution is to change the parameter from concrete `Settings` to a function that updates `Settings`:

```
runSettings' :: (Settings → Settings) → Application → IO ()
runSettings' update = runSettings (update defaultSettings)

main :: IO ()
main = runSettings' update app
  where update settings = settings { settingsPort = 4000
                                   , settingsHost = "*6" }
```

With the new interface, users do not need to look for default values anymore, and the use of `runSettings'` is fully decoupled from `defaultSettings`. However, this design still has the second drawback: all arguments are optional. Sometimes we do not want to provide any default value for some argument, `settingsPort` for example, and users are required to fill it in. A workaround employed by `SqlBackend` in the library *persistent* [7] is to have another function that takes required arguments and supplements default values for optional arguments:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data ReqSettings = ReqSettings { settingsPort :: Port }

mkSettings :: ReqSettings → Settings
mkSettings ReqSettings {..} =
  Settings { settingsHost = "*4", settingsTimeout = 30, .. }
```

Although the new `mkSettings` function solves the second issue, there is a regression concerning the first issue: users have to look for `mk*` functions now. Fortunately, we can harmonize the essence of both design patterns to develop a third approach:

```
{-# language DuplicateRecordFields, RecordWildCards #-}

data OptSettings = OptSettings { settingsHost :: HostPreference
                               , settingsTimeOut :: Int }

runSettings'' :: (OptSettings → Settings) → Application → IO ()
runSettings'' update = runSettings (update defaultSettings)
  where defaultSettings = OptSettings { settingsHost = "*4"
                                      , settingsTimeout = 30 }

main :: IO ()
main = runSettings'' update app
  where update OptSettings {..} =
          Settings { settingsPort = 4000, .. }
                   { settingsHost = "*6" }
```

This last approach is probably the best practice at the moment in Haskell, though it is already complicated for novices and requires two GHC language extensions. Of course, there could be other approaches we did not mention to encoding named and optional arguments in Haskell. Users may get users confused about the various available design patterns. This is partly due to the lack of the language support for named arguments. We believe it is better for a functional language to provide some standard syntax instead.

3 Encoding Named Arguments as Records

In this section, we demonstrate our approach to encoding named and optional arguments. Two possible ways of desugaring are presented, both of which are based on a minimal extension of $\lambda_{<:}$. The first one adds *open-bindings* while the second one adds *failable projections*. A new kind of record type with *optional fields* is also needed for type safety.

3.1 Desugaring with Open-Bindings

Let us revisit the example of the exponential function. This time, `exp` is defined and applied using our source language in ML-like syntax:

```
exp { x: Double; base: Double = 2.71828 } = base ** x

exp { x = 10.0 }              --> 22026.
exp { x = 10.0; base = 2.0 }  -->  1024.
exp { base = 2.0; x = 10.0 }  -->  1024.
```

In the definition of `exp`, we provide the default value for `base`. When applying `exp`, we can choose whether to pass `base` or not as long as the required argument `x` is present. We can freely swap the order of `x` and `base` while keeping the meaning of each argument clear.

Desugared code. How does it work? Actually, we desugar the previous definition of `exp` to a core expression of this form:

```
exp = λargs: { x: Double }. let base = 2.71828 in
  open args in  -- let x = 10.0 in let base = 2.0 in
  base ** x                      ----------------- shadowing!
```

The first thing to note is that the type of `args` does not contain any optional argument. Here, we leverage *width subtyping* between record types to accept additional arguments. That is why the record type only contains required arguments like `x`.

In the desugared definition, the most interesting part is the *open-binding* on the second line. It will dynamically convert each field within the record into a corresponding let-binding. While optional arguments are already bound to their default values, the new values in the argument record, if provided by users, will *shadow* the previous let-bindings. Note that we cannot statically convert open-bindings to let-bindings because we cannot know what optional fields are available in the record `args` until run time. In other words, `open` is a *dynamic* operation that inspects the evaluated value of the argument record.

A stricter open. Although our desugaring works fine so far, one may have a concern about accidental shadowing caused by open-bindings. For example, if a user applies `exp` to `{ x = 10.0; foo = "bar" }`, the field `foo` may cause accidental shadowing if this variable has already been defined. To avoid such an

embarrassing situation, we propose a stricter version of `open` with a permitted label set. The given labels limit the range of fields that can be opened. With this version of open, the desugared code can be rewritten like this:

```
exp = λargs: { x: Double }. let base = 2.71828 in
  open base, x of args in
  base ** x
```

However, this stricter version is still unsatisfying in terms of type safety. We will revisit open-bindings in Section 3.3.

3.2 Desugaring with Failable Projections

In the first way of desugaring, we said that open-bindings cannot be statically converted to let-bindings because the presence of optional fields is unknown until run time. Thus we cannot guarantee that record projections are always safe. But what if we allow failable projections? In OCaml, `option` types are used to represent such a failable result. As we have argued, we want to keep the core calculus as simple as possible, so we choose to provide default values for failable projections instead. It can be regarded as eliminating `option` values with the `Option.value` function immediately. (`Option.value` is equivalent to `fromMaybe` in Haskell.)

Desugared code. With failable projections, the previous definition of `exp` is desugared into a core expression of this form:

```
exp = λargs: { x: Double }.
  let base = args.base ? 2.71828 in  -- failable projection
  let x = args.x in           -- definitely safe projection
  base ** x
```

The syntax of failable projection is $e_1.\ell$? e_2. If a field of the form $\{\ell = v\}$ is present in e_1 then v is returned, otherise use e_2 as a default. Note that the second projection is definitely safe because the type of `args` ensures that `x` is present.

Lazy evaluation. In most situations, failable projections and the stricter version of open-bindings are equally valid. But one thing to note is that failable projections have better compatibility with lazy evaluation. This issue is about the strict evaluation of e_1 in the expression `open` e_1 `in` e_2. For example, consider the following code:

```
const = λargs: Top. let foo = 0 in open args in 48
const { foo = undefined }
```

Here, we assume `undefined` to be a stuck term. The application of `const` is certainly stuck since the argument is strictly evaluated. This is not a bug but a feature in the *call-by-value* λ-calculus. To avoid this, we can employ *call-by-name* evaluation, but our open-bindings do not have a lazy version. The evaluation will still be stuck when evaluating `args` in the expression `open args in ...`.

Therefore, we have to choose an alternative way using failable projections and lazy let-bindings:

```
const = λargs: Top. let foo = args.foo ? 0 in 48
const { foo = undefined }
```

Since `foo` is unused, the expression `args.foo ? 0` is never evaluated. Thus the code terminates in a call-by-name semantics.

3.3 Toward Type Safety

There is still a serious problem in our approach: *neither* open-bindings *nor* failable projections are type-safe! To address this problem, we need to provide the type system with more information about optional fields. The way of desugaring should be changed a bit:

```
exp : { x: Double; base: Double = 2.71828 } = ...
-- will be desugared to:
exp = λargs: { x: Double | base?: Double }. ...
```

In the type of `args` in the desugared code, the optional argument `base` is appended after a vertical bar. The question mark after the label is used to visually distinguish optional fields from required ones. With the extra type information, we can do more checks to ensure type safety.

Open-bindings. Even with the stricter version of `open`, we could not guarantee that every opened argument has the same type as its default value. After we include optional fields in the parameter type, the desugared code is now:

```
exp = λargs: { x: Double | base?: Double }.
  let base = 2.71828 in open args in base ** x
```

Since we statically know the names and types of the optional arguments from the type of `args`, we can check if they are bound with default values of appropriate types before `args` are opened. At call sites, optional arguments are also checked against the parameter type to ensure that they have the correct types. Furthermore, passing undeclared arguments is forbidden to avoid accidental shadowing.

Failable projections. A similar issue can be found in the approach with failable projections: we cannot guarantee that the value we obtain from a projection has the same type as the default value. But with the new way, the desugared code is now:

```
exp = λargs: { x: Double | base?: Double }.
  let base = args.base ? 2.71828 in let x = args.x in base ** x
```

When type checking `args.base ? 2.71828`, we can statically know the potential type of the `base` field. It is easy to make sure the failable projection is type-safe by comparing that type and the type of the default value.

4 Formalization of Core Calculi

After the informal introduction of language constructs in the core calculi, we go deep into the formalization in this section. We formalize failable projections and open-bindings in λ_{proj} and λ_{open}, respectively. As demonstrated before, *either* of the two calculi is enough to encode named arguments. Furthermore, a special record type with extra information about optional fields is added to keep both calculi type-safe.

4.1 Syntax and Semantics of λ_{proj}

The syntax of λ_{proj} is presented in Fig. 3. The $\lambda_{<:}$ components are standard as we follow the formalization in *Software Foundations* [8], except that we use bidirectional typing [2] to make typing rules clear and evaluation contexts [9] to simplify evaluation rules. Therefore, we focus on the novel parts about failable projections and record types with optional fields. It is worth noting that all of our new rules are *modularly* added, which means no existing rules need modification.

Subtyping. The subtyping rules are inherited from $\lambda_{<:}$ intact, including ordinary record subtyping (width, depth, and permutation subtyping). Note that there is no subtyping relation with respect to the record types with optional fields. Consequently, these types will never go through the rule of subsumption.

Typing. Following the convention of bidirectional typing, we use $\Gamma \vdash e \Rightarrow A$ to denote type inference and $\Gamma \vdash e \Leftarrow A$ to denote type checking. There is no rule that infers an expression to be a record type with optional fields, so such a type can only occur in parameters that are annotated by users. When a function is applied to optional arguments, the argument is checked against the parameter type, that is, a record type with optional fields. Such a check is handled by T-OptRcd in Fig. 4. In essence, optional fields add a *lower bound* to a record

Types	$A, B ::= \top \mid A \to B \mid \{\overline{\ell : A}\} \mid \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\}$
Expressions	$e ::= x \mid \lambda x : A.\ e \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid$
	$\{\overline{\ell = e}\} \mid e.\ell \mid e_1.\ell\ ?\ e_2$
Values	$v ::= \lambda x : A.\ e \mid \{\overline{\ell = v}\}$
Evaluation Contexts	$E ::= \square \mid E\ e \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid$
	$\{\overline{\ell_i = v_i};\ \ell = E;\ \overline{\ell_j = e_j}\} \mid E.\ell \mid E.\ell\ ?\ e$
Typing Environments	$\Gamma ::= \cdot \mid \Gamma, e : A$

Fig. 3: Syntax of λ_{proj}

$$\textsc{T-OptRcd}\quad \frac{\Gamma \vdash e \Rightarrow A \qquad \{\overline{\ell_i : A_i}; \overline{\ell_j : A_j}\} <: A <: \{\overline{\ell_i : A_i}\}}{\Gamma \vdash e \Leftarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\}}$$

$$\textsc{T-OptProj}\quad \frac{\Gamma \vdash e_1 \Rightarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}; \ell ? : A; \overline{\ell_{j'} ? : A_{j'}}\} \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1.\ell \ ?\ e_2 \Rightarrow A}$$

Fig. 4: Typing rules of optional fields and failable projections

$$\textsc{E-ProjSome}\quad \{\overline{\ell_i = v_i}; \ell = v; \overline{\ell_j = v_j}\}.\ell \ ?\ e_2 \longrightarrow v \qquad \textsc{E-ProjNone}\quad \frac{\ell \notin \{\overline{\ell_i}\}}{\{\overline{\ell_i = v_i}\}.\ell \ ?\ e_2 \longrightarrow e_2}$$

Fig. 5: Evaluation rules of failable projections

type. The record type $\{\overline{\ell_i : A_i}\}$ without optional fields still acts as the upper bound; meanwhile, we construct another ordinary record type $\{\overline{\ell_i : A_i}; \overline{\ell_j : A_j}\}$ consisting of both optional and required fields as the lower bound. The inferred type of a record should lie between the two bounds. The typing rule of failable projections is also shown in Fig. 4. T-OptProj checks if the field with label ℓ has the same type as e_2. If not, type checking fails.

Operational semantics. As shown in Fig. 5, there are two evaluation rules about failable projections: E-ProjSome succeeds in finding the field $\{\ell = v\}$ and steps to v, while E-ProjNone steps to e_2 since ℓ is absent in the record.

4.2 Syntax and Semantics of λ_{open}

The syntax of λ_{open} is presented in Fig. 6. The components about optional fields are omitted since they are the same as those introduced in λ_{proj}. The extension of open-bindings is more difficult than failable projections because we have to employ a different operational semantics. The root cause is that open-bindings are incompatible with *textual substitution*. Since a substitution eagerly replaces all occurrences of a variable by traversing the syntax tree, open-bindings cannot foresee whether a substitution is shadowed by the labels to be opened before the record is evaluated. For example, consider such an expression:

```
let x = 1 in let args = { x = 2 } in open args in x
```

It evaluates to 1 if we evaluate let-bindings with substitution. The substitution of `let x = 1` is not shadowed by `open args` because the record `args` has not been evaluated and the labels it contains are unknown at this moment. Therefore, we abandon substitution and propose a environment-based operational semantics with closures.

$$
\begin{array}{lll}
\text{Expressions} & e ::= & x \mid \lambda x : A.\ e \mid e_1\ e_2 \mid \{\overline{\ell = e}\} \mid e.\ell \mid \\
& & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{open}\ e_1\ \texttt{in}\ e_2 \mid \langle \Delta \mid e \rangle \\
\text{Values} & v ::= & \langle \Delta \mid \lambda x : A.\ e \rangle \mid \{\overline{\ell = v}\} \\
\text{Evaluation Contexts} & E ::= & \square \mid E\ e \mid v\ E \mid \{\overline{\ell_i = v_i};\ \ell = E;\ \overline{\ell_j = e_j}\} \mid E.\ell \mid \\
& & \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{open}\ e_1\ \texttt{in}\ e_2 \\
\text{Valuation Environments} & \Delta ::= & \cdot \mid \Delta, x \mapsto v
\end{array}
$$

Fig. 6: Syntax of λ_{open}

$$\textsc{E-Var}\ \frac{x \mapsto v \in \Delta}{\Delta \vdash x \longrightarrow v} \qquad \textsc{E-Abs}\ \Delta \vdash \lambda x : A.\ e \longrightarrow \langle \Delta \mid \lambda x : A.\ e \rangle$$

$$\textsc{E-App}\ \Delta \vdash \langle \Delta' \mid \lambda x : A.\ e \rangle\ v \longrightarrow \langle \Delta', x \mapsto v \mid e \rangle \qquad \textsc{E-Proj}\ \Delta \vdash \{\overline{\ell_i = v_i};\ \ell = v;\ \overline{\ell_j = v_j}\}.\ell \longrightarrow v$$

$$\textsc{E-Let}\ \Delta \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \longrightarrow \langle \Delta, x \mapsto v_1 \mid e_2 \rangle$$

$$\textsc{E-Open}\ \Delta \vdash \texttt{open}\ \{\overline{\ell = v}\}\ \texttt{in}\ e \longrightarrow \texttt{let}\ \overline{\ell = v}\ \texttt{in}\ e \qquad \textsc{E-Closure}\ \frac{\Delta' \vdash e \longrightarrow e'}{\Delta \vdash \langle \Delta' \mid e \rangle \longrightarrow \langle \Delta' \mid e' \rangle}$$

$$\textsc{E-ClosureV}\ \Delta \vdash \langle \Delta' \mid v \rangle \longrightarrow v \qquad \textsc{E-Context}\ \frac{\Delta \vdash e \longrightarrow e'}{\Delta \vdash E[e] \longrightarrow E[e']}$$

Fig. 7: Environment-based evaluation rules of λ_{open}

$$\textsc{T-Open}\ \frac{\Gamma, \overline{\ell_j : A_j} \vdash e_1 \Rightarrow \{\overline{\ell_i : A_i} \mid \overline{\ell_j ? : A_j}\} \qquad \Gamma, \overline{\ell_j : A_j}, \overline{\ell_i : A_i} \vdash e_2 \Rightarrow B}{\Gamma, \overline{\ell_j : A_j} \vdash \texttt{open}\ e_1\ \texttt{in}\ e_2 \Rightarrow B}$$

Fig. 8: The typing rule of open-bindings

Operational semantics. As shown in Fig. 7, a valuation environment Δ, which binds variable names to their corresponding values, is added to each evaluation rule. The expression $\langle \Delta \mid e \rangle$ saves an environment inside so that evaluation can later resume with a saved environment, among which $\langle \Delta | \lambda x : A.\ e \rangle$ is well knwon as a *function closure*. Briefly speaking, closures are used to ensure lexical scoping. The extension of open-bindings is rather simple: E-Open converts open-bindings to let-bindings depending on the evaluated result of the record.

Typing. The typing rule of open-bindings in Fig. 8 may need some explaining. T-Open first figures out the optional fields from the type of e_1 and checks if these names are already in the typing environment. This is because we assume that all optional arguments have their default values defined in advance. If the requirement is met, we go on to calculate the type of e_2 with the type information of all fields appended to the environment. By the way, the check of an open-binding degenerate into something like a check of multiple let-bindings if there is no optional fields.

Remarks. Overall, we prefer λ_{proj} to λ_{open} because we want to keep the extension to $\lambda_{<:}$ as simple as possible. Although the operational semantics of λ_{open} is unusual, the implementation of open-bindings should not be harder than failable projections since we seldom use textual substitution owing to inefficiency. Instead, a practical implementation is probably more close to our closure-based operational semantics. Moreover, an open-binding itself is a useful language construct, similar to the `open` directive in the ML module system or record wildcards in Haskell. It is also interesting to us that a seemingly concise design can finally lead to a relatively sophisticated formalization.

5 Conclusion

Named and optional arguments are widely supported in object-oriented programming languages but are hardly formalized. Garrigue et al. formalized a label-selective λ-calculus for OCaml that combines commutativity and currying, but it is non-trivial to be integrated with other sophisticated λ-calculi. OCaml goes to the extreme of pursuing fancy features, while Haskell goes to the other extreme of lacking native support for named arguments. It is well known in Haskell that named arguments can be encoded as records, but it requires a lot of boilerplate code to support both required and optional arguments. In this paper, we presented a minimal extension to $\lambda_{<:}$ that serves as a type-safe core calculus. Based on two alternative ways of desugaring, named and optional arguments can be encoded as records.

Although we keep the calculus as simple and modular as possible, it is impossible to avoid *every* potential conflict caused by feature interaction. If a language is simply incompatible with subtyping, for an extreme example, our approach does not work. Nevertheless, we believe that our approach works in most situations. We hope that functional language designers who are concerned about named arguments can benefit from this paper.

Future work. We plan to prove the type soundness of λ_{proj} and λ_{open} using Coq in the near future. Moreover, it is worth investigating how to adapt our approach for a record calculus that uses row polymorphism rather than subtyping.

Acknowledgments. This work has been sponsored by the Hong Kong Research Grant Council project numbers 17209519, 17209520, and 17209821.

References

1. Aït-Kaci, H., Garrigue, J.: Label-selective lambda-calculus: syntax and confluence. Theor. Comput. Sci. **151**(2) (1995)
2. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5) (2021)
3. Furuse, J.P., Garrigue, J.: A label-selective lambda-calculus with optional arguments and its compilation method. Tech. rep., Kyoto University (1995)
4. Garrigue, J.: Objective Label trilogy, `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/`
5. Garrigue, J.: Labeled and optional arguments for Objective Caml. In: JSSST SIGPPL (2001)
6. Garrigue, J., Aït-Kaci, H.: The typed polymorphic label-selective lambda-calculus. In: POPL (1994)
7. Parsons, M.: Persistent: type-safe, multi-backend data serialization, `https://hackage.haskell.org/package/persistent`
8. Pierce, B.C., et al.: Programming Language Foundations, Software Foundations, vol. 2. `https://softwarefoundations.cis.upenn.edu/plf-current/`
9. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2) (1975)
10. Rytz, L., Odersky, M.: Named and default arguments for polymorphic object-oriented languages. In: SAC (2010)
11. Snoyman, M.: Warp: a fast, light-weight web server for WAI applications, `https://hackage.haskell.org/package/warp`

Towards Efficient Adjustment of Effect Rows

Naoya Furudono, Youyou Cong, Hidehiko Masuhara, and Daan Leijen

[1] Tokyo Institute of Technology, Japan
naoyafurudono@prg.is.titech.ac.jp
[2] Tokyo Institute of Technology, Japan
cong@c.titech.ac.jp
[3] Tokyo Institute of Technology, Japan
masuhara@acm.org
[4] Microsoft Research, USA
daan@microsoft.com

Abstract. Koka is a functional programming language with native support for algebraic effects and handlers. To implement effect handler operations efficiently, Koka employs a semantics where the handlers in scope are passed down to each function as an evidence vector. At runtime, these evidence vectors are adjusted using the `open` constructs to match the evidence for a particular function. All these adjustments can cause significant runtime overhead. In this paper, we present a novel transformation on the Koka core calculus that we call *open floating*. This transformation aims to float up `open` constructs and combine them in order to minimize the adjustments needed at runtime. Open floating improves performance by 2.5× in an experiment. Furthermore, we formalize an aspect of row-based effect typing, including the *closed prefix* relation on effect rows that is required for our transformation to be sound.

1. Introduction

Several implementation strategies of algebraic effect handlers [13] have been proposed for performance [2, 14, 18, 19]. Algebraic effect handlers are language feature for user-defined effects. For instance, using effect handlers, we can support exception, asynchronous programming, nondeterminism, and so on not as builtin features but as libraries. While effect handlers are convenient, they incur more runtime overhead compared to native effects. In order to fill in the gap of the performance, suitable semantics have been explored. For example, the Koka language [7, 8, 19] employs the evidence passing semantics [19] for the core calculus.

The key idea of the semantics is to pass around a vector of handler implementations, called an *evidence vector*, and propagate it to algebraic effect operation calls, exposing optimization opportunities. The row-based type-and-effect system ensures the correctness of the dynamic semantics, where the static effect row type corresponds directly to the shape of the dynamic evidence vector at runtime.

The Koka compiler automatically inserts `open` constructs during type inference so that adjust evidence vectors at runtime. Let us consider a function call

as an example. Suppose that a function of type $int \to \langle exn \rangle\, int$, which expects an evidence vector of shape $\langle\!\langle exn : .. \rangle\!\rangle$ to be passed, is called in the context with effect $\langle exn, read \rangle$, which provides an evidence vector of $\langle\!\langle exn : .., read : .. \rangle\!\rangle$. The compiler detects the mismatch and wraps the function by open to adjust the runtime vector from $\langle\!\langle exn : .., read : .. \rangle\!\rangle$ to $\langle\!\langle exn : .. \rangle\!\rangle$.

Unfortunately, each adjustment incurs runtime cost and, being type-directed, the automatic insertion tends to generate many redundant open calls around function applications. In this paper, we present the *open floating* algorithm to remove such opens as a compiler optimization. The algorithm first removes existing opens in a top-down way and then re-assign effect adjustment constructs back in a bottom-up traversal. The re-assignment is driven by the (now) explicit effect types, ensuring preservation of the meaning of programs. To give the reader a rough idea of the algorithm, we present programs before and after open floating.

$$
\begin{array}{l}
\mathsf{handler}\,\{\, ask \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\, \lambda\_. \\
\quad \mathsf{let}\, x = \mathsf{open}\, \langle read \rangle\, safediv(3, 2)\, \mathsf{in} \\
\quad \mathsf{let}\, y = \mathsf{open}\, \langle read \rangle\, safediv(3, x)\, \mathsf{in} \\
\quad \mathsf{open}\, \langle read \rangle\, safediv(3,\ y)
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{l}
\mathsf{handler}\,\{\, ask \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\, \lambda\_. \\
\quad \mathsf{restrict}\, \langle\,\rangle\, (\\
\quad\quad \mathsf{let}\, x = safediv(3, 2)\, \mathsf{in} \\
\quad\quad \mathsf{let}\, y = safediv(3, x)\, \mathsf{in} \\
\quad\quad safediv(3,\ y))
\end{array}
$$

Observe that the program on the left has three opens, whereas the one on the right only has a single restrict construct. Both open and restrict perform the same adjustment thus the transformation decreases the number of adjustments. The restrict construct can be applied to general expressions, not just functions. This is essential to share a single adjustment over multiple function applications.

In Section 2, we give the overview of our study, and following sections include listed contributions.

- We define System F_{pwo}, a system of effect handlers with the open construct. The system is the core calculus of the Koka language, and is an extension of System F_{pw} by Xie and Leijen [19]. We elaborate on effect typing in the extended system, showing
  - an advantage of using rows for effect types rather than sets, which are popular in effect handler calculi [1, 15]
  - how the type system checks the use of effect handlers
  - an effect type restriction on open, which we call the *closed prefix* relation (Section 3.4).

  In particular, the closed prefix relation clarifies what program transformations are allowed. This helps us define open floating.
- We give a formal definition of the open floating algorithm, which floats up redundant opens (Section 4).
- We implemented open floating in the Koka compiler [10]. Here we did not extend the internal compiler core syntax with restrict but were able to express the necessary scoping by simply using open and lambda expressions and matching on this particular form in the backend.

– We evaluated open floating via a preliminary benchmark (Section 5). where it improves performance by 2.5×. Based on the results, we make it clear that what kind of programs benefit from open floating.

We discuss future work in Section 6 and related work in Section 7.

2. Overview

In this section, we give an overview of open floating. We first give an overview of the calculus of study, and explain the need for open. We then show an example with redundant opens and describe the idea of our solution.

2.1. Effect Handlers

Algebraic effects are declared with an effect label l and a list of operation signatures. For instance, suppose we have a *read* effect with a single operation *ask*, which takes a unit argument and returns an integer value.

$$\mathit{read} \ : \ \{\, \mathit{ask} \ : \ () \to \mathit{int} \,\}$$

An effect handler for *read* specifies what the *ask* operation should do when it is called in the handled expression.

$$\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{ask}\,()$$

In this example, $\mathsf{perform}\ \mathit{ask}\,()$ calls operation *ask* with argument (), which is evaluated to 3 according to the handler. The behavior of an operation call is formalized as follows: (1) find the innermost handler of the effect, (2) capture the *resumption* – continuation delimited by the handler –, and (3) apply the handler clause to the argument and the resumption.

1. $\underline{\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, \mathsf{perform}\ \mathit{ask}\,()} \ + \ \mathsf{perform}\ \mathit{ask}\,()$
2. $\mathit{resume} \ = \ \lambda z.\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, z \ + \ \mathsf{perform}\ \mathit{ask}\,()$
3. $(\lambda\_.\, \lambda k.\, k\,3)\,()\ \mathit{resume}$

The handler resumes the resumption with argument 3, so that the example is evaluated to the following.

$$\mathsf{handler}\,\{\, \mathit{ask} \to \lambda\_.\, \lambda k.\, k\,3 \,\}\, \lambda\_.\, 3 \ + \ \mathsf{perform}\ \mathit{ask}\,()$$

The two occurrences of $\mathsf{perform}\ \mathit{ask}\,()$ are both evaluated to 3, therefore the whole expression is evaluated to 6.

Using effect handlers, it is easy to combine different effects in a single program. Let us combine *read* with the exception effect *exn*, which has an operation *throw* of type $\forall\alpha.\mathit{int} \to \alpha$

$$\mathit{exn} \ : \ \{\, \mathit{throw} \ : \ \forall\alpha.\mathit{int} \to \alpha \,\}$$

Using a handler for *exn* as we did for *read*, we can perform the *throw* operation.

$$\begin{array}{l}\mathsf{handler}\,\{\, \mathit{ask} \mapsto \lambda x.\, \lambda k.\, k\,3 \,\}\, (\lambda\_. \\ \quad \mathsf{handler}\,\{\, \mathit{throw} \mapsto \lambda x.\, \lambda k.\, x \,\}\, (\lambda\_. \\ \quad\quad \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{ask}\,() \ + \ \mathsf{perform}\ \mathit{throw}\ 1))\end{array}$$

In this program, perform *ask*() is evaluated to 3 as before, but the entire expression is evaluated to 1 due to perform *throw* 1. The handler for the *throw* operation discards the resumption k and returns the argument x, which exits the computation out of the handler expression.

2.2. Evidence Passing Semantics and Row-based Effect System

Among different formalizations of the dynamic semantics of effect handlers, we adopt the *evidence passing semantics* (EPS) [19], which utilizes the invariants given by the row-based effect system and allows the compiler to translate programs to efficient code.

Under the EPS, we pass all handlers in scope to the handled expression so that operation calls can access their handler locally. The operation clauses passed to expressions are represented as an *evidence vector* [18]. For instance, in the *read* and *exn* example discussed above, the expression perform *ask*() is performed with the evidence vector of the form $\langle\!\langle \mathit{exn} : \{\mathit{throw} \mapsto \ldots\}, \mathit{read} : \{\mathit{ask} \mapsto \ldots\} \rangle\!\rangle$. Xie and Leijen [19] show that EPS often allow us to avoid lookups and resumption captures, which are one of main sources of inefficiency in effect handler execution.

The static semantics of our calculus is defined by a *row-based effect system*. In the effect system, every expression is related to an effect row type in addition to a usual type. Effect rows indicate what kind of evidence vector is provided from the context to evaluate the expression. For instance, in the example with *read* and *exn*, the expression perform *ask*() + perform *ask*() + perform *throw* 1 has type *int* and effect row $\langle \mathit{exn}, \mathit{read} \rangle$.

The typing rules maintain the correspondence between evidence vectors and effect rows. For instance, a function application uses the same evidence vector for the function, the argument, and the β-reduced expression. Correspondingly, the typing rule for function application requires the effect rows of the three parts (occurrences of ϵ in the premises) to agree with that of the entire expression (occurrence of ϵ in the conclusion).

$$\frac{\Gamma \vdash e_1 : \sigma_1 \to \epsilon\, \sigma_2 \mid \epsilon \qquad \Gamma \vdash e_2 : \sigma_1 \mid \epsilon}{\Gamma \vdash e_1\, e_2 : \sigma_2 \mid \epsilon}\ [\textsc{app}]$$

We ignore the order of labels in effect rows (except for *parameterized effect labels* that are discussed in Section 3.1.2 and Section 3.4). For instance, we regard two rows $\langle \mathit{exn}, \mathit{read} \rangle$ and $\langle \mathit{read}, \mathit{exn} \rangle$ as equivalent. This flexible row equivalence allows the type system to ignore the order of handlers in evaluation contexts. As a consequence, both programs below are judged well-typed as one would expect.

handler { *ask* ↦ λ*x*. λ*k*. *k* 3 } (λ_.
  handler { *throw* ↦ λ*x*. λ*k*. *x* } *f*)

handler { *throw* ↦ λ*x*. λ*k*. *x* } (λ_.
  handler { *ask* ↦ λ*x*. λ*k*. *k* 3 } *f*)

where $f =$ λ_.perform *ask*() + perform *ask*() + perform *throw* 1))

We formally define the effect row equivalence in Section 3.1.2 and discuss it in Section 3.4.

2.3. Effect Type Adjustment for Function Types

The typing rule [APP] is too restrictive on some occasions. Consider a function *safediv* of type $(int,\ int) \to \langle\,\rangle\ maybe\ int$, which returns *Nothing* if the divider is 0, instead of throwing an exception. This function causes no effect, hence we should be able to call the function in any context. However, the type system prevents us from calling *safediv* in certain contexts. For instance, the following expression is judged ill-typed.

$$\mathsf{handler}\,\{\ ask \mapsto \lambda x.\,\lambda k.\,k\,3\ \}\ \lambda\_.safediv(3,\ 2)$$

The expression $safediv(3, 2)$ expects an evidence vector of type $\langle\,\rangle$, whereas the context provides an evidence vector of type $\langle read \rangle$. Due to this inconsistency, the expression is rejected by the type system.

To call functions with a "smaller" effect, we introduce the expression $\mathsf{open}\ \epsilon'\ v$ into the calculus. At compile time, open allows a function to have a "bigger" effect type according to the following typing rule.

$$\frac{\Gamma \vdash_{\mathsf{val}}\ v\ :\ \sigma_1 \to \epsilon\,\sigma_2 \qquad \epsilon \leqslant \epsilon'}{\Gamma \vdash_{\mathsf{val}}\ \mathsf{open}\,\epsilon'\ v\ :\ \sigma_1 \to \epsilon'\,\sigma_2}\ [\textsc{open}]$$

At runtime, open adjusts evidence vectors so that the callee receives an evidence vector of the expected shape and thus runs correctly. Using open, we can make the above example well-typed.

$$\mathsf{handler}\,\{ask \mapsto \lambda x.\,\lambda k.\,k\,3\}\ \lambda\_.(\mathsf{open}\ \langle read \rangle\ safediv)(3,\ 2)$$

We call the smaller effect row a *closed prefix* of the larger one. The closed prefix relation is defined as:

$$\langle l_1, \ldots, l_n \rangle \leqslant \langle l_1, \ldots, l_n \mid \epsilon \rangle \quad (n \geq 0)$$

It turns out that different formulations are possible but some seemingly benign generalizations can make the type system unsound – we discuss this in detail the closed prefix relation in Section 3.3 and Section 3.4.

2.4. Motivating Open Floating

Our calculus is designed as an intermediate language of a compiler. This means the user does not need to explicitly write opens; they are automatically inserted by the type inferencer. The user expression $\mathsf{handler}\,\{\ ask \mapsto \ldots\ \}\ \lambda\_.safediv(3, 2)$ is translated to $\mathsf{handler}^{\langle\,\rangle}\ \{\ ask \mapsto \ldots\ \}\ \lambda\_.(\mathsf{open}\ \langle read \rangle\ safediv)\ (3, 2)$, for example.

Unfortunately, naive insertion of open makes programs inefficient. Consider a program that calls *safediv* three times. The compiler inserts opens into each function call as follows.

$$\begin{array}{l}\mathsf{handler}^{\langle\,\rangle}\ \{\ ask \mapsto \lambda x\ :\ int.\,\lambda k\ :\ int \to \langle\,\rangle\ int.\,k\,3\ \}\ \lambda\_. \\ \quad \mathsf{let}\ x\ =\ \mathsf{open}\ \langle read \rangle\ safediv(3, 2)\ \mathsf{in} \\ \quad \mathsf{let}\ y\ =\ \mathsf{open}\ \langle read \rangle\ safediv(3, x)\ \mathsf{in} \\ \quad \mathsf{open}\ \langle read \rangle\ safediv(3,\ y)\end{array}$$

As open causes evidence vector adjustment at runtime, having many open calls makes execution slow. In order to avoid this inefficiency, we design the *open floating* optimization that eliminates redundant open calls. By open floating, the above program is transformed to the following one.

$$
\begin{array}{l}
\mathsf{handler}^{\langle\,\rangle}\,\{\,\mathit{ask} \mapsto \lambda x : \mathit{int}.\,\lambda k : \mathit{int} \to \langle\,\rangle\,\mathit{int}.\,k\,3\,\}\,\lambda^{\langle \mathit{read} \rangle}\_. \\
\quad \mathsf{restrict}\,\langle\,\rangle\,(\\
\quad\quad \mathsf{let}\,x = \mathit{safediv}(3, 2)\,\mathsf{in} \\
\quad\quad \mathsf{let}\,y = \mathit{safediv}(3, x)\,\mathsf{in} \\
\quad\quad \mathit{safediv}(3,\ y))
\end{array}
$$

Here, $\mathsf{restrict}\,\epsilon\,e$ allows e to be typed with effect ϵ, which is smaller than the effect of the context. In this particular example, e is typed with $\langle\,\rangle$, not $\langle \mathit{read} \rangle$. At runtime, as open does, $\mathsf{restrict}\,\epsilon\,e$ changes the shape of the evidence vector to fit ϵ and pass it to e. In general, open floating erases open in β-redex and reassigns appropriate open or restrict to make the whole expression type check in a bottom-up way. The closed prefix relation plays an essential role in determining the new effect type of each subexpression. We formally define open, restrict and closed prefix in Section 3, the algorithm in Section 4, and show the preliminary benchmark in Section 5. Although the benchmark code is artificial, the results are promising and open floating may potentially encourage use of effect handlers in various fields.

3. System F_{pwo}

In this section, we present System F_{pwo}, a calculus with algebraic effect handlers and the open. The calculus is an extension of System F_{pw} [19], an explicitly typed polymorphic lambda calculus with effect handlers. The semantics is based on *evidence passing semantics* [19], which leads to an efficient implementation of effect handlers. Furthermore, both calculi have row-based effect types, which denote the static meaning of evidence vectors. We extend F_{pw} with open, restrict and parameterized effect labels. These features have previously discussed by Leijen [9]. In that work, the idea of open was formalized as a typing rule, and parameterized effect labels were formalized in a way that does not fit well to our calculus.

We have confirmed the soundness of the type system through the compiler implementation. We are currently developing formal proofs. We first introduce the syntax, dynamic semantics, and static semantics of F_{pwo}. After that, we describe the typing with effect rows, including the closed prefix.

3.1. Syntax

The syntax is defined in Figure 1.

3.1.1. Expressions Expressions e include values v, applications $e\,e$, type applications $e\,\sigma$, let-bindings $\mathsf{let}\,x = e\,\mathsf{in}\,e$, prompt $\mathsf{prompt}\,m\,h\,e$, yield $\mathsf{yield}\,m\,v$, and restrict $\mathsf{restrict}\,\epsilon\,e$. Prompt and yields are *internal constructs* that only appear as an intermediate result of evaluation. We will formally define internal constructs in Definition 1.

Expression	e	::=	$v \mid e\,e \mid e\,\sigma$
		$\mid$	$\mathsf{let}\, x = e\, \mathsf{in}\, e$
		$\mid$	$\mathsf{prompt}\, m\, h\, e$
		$\mid$	$\mathsf{yield}\, m\, v$
		$\mid$	$\mathsf{restrict}\, \epsilon\, e$
Value	v	::=	$x \mid \lambda^{\epsilon} x : \sigma.\, e$
		$\mid$	$\Lambda\alpha^{\mathsf{k}}.\, v$
		$\mid$	$\mathsf{handler}^{\epsilon}\, h$
		$\mid$	$\mathsf{perform}\, op\, \epsilon\, \overline{\sigma}$
		$\mid$	$\mathsf{open}\, \epsilon\, v$
Hnd. clauses	h	::=	$\{\, \overline{op \mapsto f} \,\}$

Type	σ	::=	$\alpha^{\mathsf{k}} \mid c^{\mathsf{k}}\, \overline{\sigma} \mid \sigma \to \epsilon\, \sigma$
		$\mid$	$\forall \alpha^{\mathsf{k}}.\, \sigma$
Kind	k	::=	$* \mid \mathsf{k} \to \mathsf{k} \mid \mathsf{lab} \mid \mathsf{eff}$
Effect label	l	::=	σ^{lab}
Effect constant	c_l	::=	c^{k}
			$\mathsf{k} = * \to \ldots \to * \to \mathsf{lab}$
Effect row	ϵ	::=	$\langle\,\rangle \mid \langle\, l \mid \epsilon\,\rangle \mid \alpha^{\mathsf{eff}}$
Type env.	Γ	::=	$\bullet \mid \Gamma,\, x : \sigma$
Effect ctx.	Σ	::=	$\{\, \overline{c_l : sig} \,\}$
Effect sig.	sig	::=	$\{\, \overline{op : \forall \overline{\alpha^{\mathsf{k}}}.\, \tau \to \tau} \,\}$
Type in effect sig.	τ	::=	$\sigma \mid \lambda \alpha^{*}.\tau$
Evidence	ev	::=	$(m,\, h,\, w)$
Evidence vec.	w	::=	$\langle\!\langle\,\rangle\!\rangle \mid \langle\!\langle\, l : ev \mid w \,\rangle\!\rangle$

zero or more x	$\overline{x}$
internal constructs	internal
invariants	invariants

Figure 1. Syntax of System F_{pwo}

Values v include variables x, lambda abstractions $\lambda^{\epsilon} x : \sigma.e$, type abstractions $\Lambda\alpha^{\mathsf{k}}.v$, effect handlers $\mathsf{handler}^{\epsilon}\, h$, operation calls $\mathsf{perform}\, op\, \epsilon\, \overline{\sigma}$, and open $\mathsf{open}\, \epsilon\, v$.

Handlers clauses h consist of a sequence of pairs of an operation name op and a function value f. The meta-variable f is syntactically a value, but we use it specifically as a function, which takes (type) arguments. The type system maintains the intention.

3.1.2. Types Types σ include type variables α^{k} of kind k, type application for type constructors $c^{\mathsf{k}}\, \overline{\sigma}$ (where c^{k} is applied to arguments $\overline{\sigma}$), function types $\sigma_1 \to \epsilon\, \sigma_2$ (indicating the body of the function can cause effect ϵ), and polymorphic types $\forall \alpha^{\mathsf{k}}.\sigma$.

Kinds k include the regular kind $*$, functions $\mathsf{k} \to \mathsf{k}$, effect labels lab, and effect rows eff. Types of the function kind $\mathsf{k} \to \mathsf{k}$ are either an effect constant c_l, a row type constructor $\langle \_ \mid \_ \rangle$, or a type τ in an effect signature. Effect labels l are types of kind lab. Effect constants c_l are of kind lab parameterized with zero or more types of regular kind. Effect labels are used to structure effect rows and evidence vectors, while effect constants are used for effect contexts and effect labels.

Effect rows ϵ include the empty effect $\langle\,\rangle$, extension with effect label $\langle\, l \mid \epsilon\,\rangle$, and type variables α^{eff} of kind eff. We use the following abbreviation for rows: $\langle l_1, \ldots, l_n \mid \epsilon \rangle \doteq \langle l_1 \mid \ldots \langle\, l_n \mid \epsilon\,\rangle \ldots \rangle$ and $\langle l_1, \ldots, l_n \rangle \doteq \langle l_1, \ldots, l_n \mid \langle\,\rangle \rangle$. The equivalence of effect rows is defined in Figure 2. We can ignore the order of two effect labels in a row if two labels consist of different effect constants. Here are examples of row equivalence. See Section 3.4 for detail about effect rows.

$\langle ask,\ exn \rangle \cong \langle exn, ask \rangle$
$\langle ask,\ polyexn\, string,\ polyexn\, int \rangle \cong \langle polyexn\, string,\ ask,\ polyexn\, int \rangle$
$\langle ask,\ polyexn\, string,\ polyexn\, int \rangle \ncong \langle ask,\ polyexn\, int,\ polyexn\, string \rangle$

$$\frac{}{\epsilon \cong \epsilon}\ [\text{EQ-REFL}] \qquad \frac{\epsilon_1 \cong \epsilon_2 \quad \epsilon_2 \cong \epsilon_3}{\epsilon_1 \cong \epsilon_3}\ [\text{EQ-TRANS}] \qquad \frac{\epsilon_1 \cong \epsilon_2}{\langle l \mid \epsilon_1 \rangle \cong \langle l \mid \epsilon_2 \rangle}\ [\text{EQ-HEAD}]$$

$$\frac{l_1 \neq l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon \rangle \rangle \cong \langle l_2 \mid \langle l_1 \mid \epsilon \rangle \rangle}\ [\text{EQ-SWAP}] \qquad \frac{c_{l1} \neq c_{l2}}{c_{l1}\,\overline{\sigma}_1 \not\equiv c_{l2}\,\overline{\sigma}_2}\ [\text{UNEQ-LAB}]$$

Figure 2. Effect Row Type Equivalence

An effect context Σ is a sequence of pairs of an effect constant and an effect signature. It maintains the relation between the name of an effect and the type of its operations. We assume Σ is given externally in this calculus, while in practical language one may define Σ by top-level definitions.

Effect signatures *sig* are a sequence of a pair of an operation name and its type. The type τ in an effect signature takes zero or more type arguments of regular kind. The type arguments will be passed if the effect is parameterized. We will show an example with type rule [PERFORM] in Section 3.3.

3.1.3. Evidence Vectors Evidence vectors w include the empty vector $\langle\!\langle\,\rangle\!\rangle$ and extension $\langle\!\langle l : ev \mid w \rangle\!\rangle$ with a pair of an effect label l and an evidence ev. Evidences ev are a triple (m, h, w) of a marker m, a handler h, and an evidence vector w where h is defined. The evidence vector of the triple is key to general use of effect handlers, but the discussion is out of the scope of this paper. See [19] for details.

3.2. Dynamic Semantics

The dynamic semantics is defined in Figure 3. The semantics consists of three rules: stepping $\longmapsto$, multi-stepping $\longmapsto^*$, and reduction $\longrightarrow$.

3.2.1. Evaluation Steps The rules (*∗stepwR*) and (*∗stepwT*) defines multi-stepping as the reflexive transitive closure of stepping. The rules (*step*) and (*stepw*) reduce a redex without and with an evidence vector w, respectively. In these rules, the evaluation context of the redex must be F, not E. F excludes prompt frames and restrict frames.

The (*promptw*) rule extends the evidence vector. Conversely, the (*restrictw*) rule shortens the evidence vector using the *select* meta-function so that the shape of the new evidence vector fits the effect row ϵ' of the restrict frame.

3.2.2. Reduction Rules The (*app*), (*let*) and (*tapp*) rules are standard. The (*handler*) rule reduces a handler application by calling the passed function f under prompt with fresh marker m. The marker acts as a control delimiter [3]. The (*promptv*) rule removes the prompt frame if the handled expression is a value.

Operation call is divided into two rules: (*perform*) and (*prompt*). The (*perform*) rule prepare the marker m and handler clause f using the evidence vector. In the right-hand side of the (*prompt*), the handler clause is applied to (type) arguments and wrapped by a lambda to take a resumption. The (*prompt*) rule captures the

Evaluation Contexts:

$$\begin{array}{lll} \mathsf{E} & ::= & \Box \mid \mathsf{E}\, e \mid v\mathsf{E} \mid \mathsf{E}\,\sigma \mid \mathsf{let}\, x = \mathsf{E}\, \mathsf{in}\, e \mid \mathsf{prompt}\, m\, h\, \mathsf{E} \mid \mathsf{restrict}\, \epsilon\, \mathsf{E} \\ \mathsf{F} & ::= & \Box \mid \mathsf{F}\, e \mid v\mathsf{F} \mid \mathsf{F}\,\sigma \mid \mathsf{let}\, x = \mathsf{F}\, \mathsf{in}\, e \end{array}$$

Evaluation Steps:

$$\frac{e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\ (step) \qquad \frac{w \vdash e \longrightarrow e'}{w \vdash \mathsf{F}[e] \longmapsto \mathsf{F}[e']}\ (stepw) \qquad \frac{}{w \vdash e \longmapsto^* e}\ (*stepwR)$$

$$\frac{\langle\!\langle\, l : (m, h, w) \mid w \,\rangle\!\rangle \vdash e \longmapsto e'}{w \vdash \mathsf{F}[\mathsf{prompt}\, m\, h\, e] \longmapsto \mathsf{F}[\mathsf{prompt}\, m\, h\, e']}\ (promptw)$$

$$\frac{\begin{array}{l}\mathsf{restrict}\,\epsilon'\, e : \sigma \mid \epsilon \quad \epsilon' \leqslant \epsilon \quad \vdash w : \epsilon \\ select\,\epsilon'\, w \vdash e \longmapsto e'\end{array}}{w \vdash \mathsf{F}[\mathsf{restrict}\,\epsilon'\, e] \longmapsto \mathsf{F}[\mathsf{restrict}\,\epsilon'\, e']}\ (restrictw) \qquad \frac{w \vdash e \longmapsto^* e' \quad w \vdash e' \longmapsto e''}{w \vdash e \longmapsto^* e''}\ (*stepwT)$$

Evidence Vector Operations:

$$\begin{array}{lll} select\,\langle\,\rangle\, w & = & \langle\!\langle\,\rangle\!\rangle \\ select\,\langle\, l \mid \epsilon\,\rangle\, w & = & \langle\!\langle\, l : w.l \mid select\,\epsilon\,(w - l)\,\rangle\!\rangle \end{array} \qquad \begin{array}{lll} \langle\!\langle\, l : ev \mid w\,\rangle\!\rangle.l & = & ev \\ \langle\!\langle\, l_1 : ev \mid w\,\rangle\!\rangle.l_2 & = & w.l_2 \quad \text{if } l_1 \neq l_2 \end{array}$$

$$\begin{array}{lll} \langle\!\langle\, l : ev \mid w\,\rangle\!\rangle - l & = & w \\ \langle\!\langle\, l_1 : ev \mid w\,\rangle\!\rangle - l_2 & = & w - l_2 \quad \text{if } l_1 \neq l_2 \end{array}$$

Reduction Rules:

$$\begin{array}{llll} (app) & (\lambda^{\epsilon} x : \sigma.\, e)\, v & \longrightarrow & e[x{:=}v] \\ (let) & \mathsf{let}\, x = v\, \mathsf{in}\, e & \longrightarrow & e[x{:=}v] \\ (tapp) & (\Lambda\alpha^{\mathsf{k}}.\, v)\,\sigma & \longrightarrow & v[\alpha{:=}\sigma] \\ (handler) & \mathsf{handler}^{\epsilon}\, h f & \longrightarrow & \mathsf{prompt}\, m\, h\, (f()) \quad \text{with unique } m \\ (promptv) & \mathsf{prompt}\, m\, h\, v & \longrightarrow & v \\ (perform) & w \vdash \mathsf{perform}\, op\, \epsilon_0\, \overline{\sigma}\, v & \longrightarrow & \mathsf{yield}\, m\, (\lambda^{\epsilon} k{:}\ (\sigma_2\, \overline{\sigma'})[\overline{\alpha}{:=}\overline{\sigma}] \to \epsilon\, \sigma.\, f\overline{\sigma}\, v\, k) \\ & & \text{with} & (m, h, \_) = w.l \wedge (op \mapsto f) \in h \quad l = c_l\, \overline{\sigma'} \\ & & & (op : \forall\overline{\alpha}.\, \sigma_{in} \to \sigma_{out}) \in \Sigma(c_l) \wedge \bullet \vdash h : \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon \\ (prompt) & \mathsf{prompt}\, m\, h\, \mathsf{E}[\mathsf{yield}\, m\, f] & \longrightarrow & f(\lambda^{\epsilon} x{:}\ \sigma_2.\, \mathsf{prompt}\, m\, h\, \mathsf{E}[x]) \\ & & \text{with} & \bullet \vdash_{\mathsf{val}} f : (\sigma_2 \to \epsilon\, \sigma) \to \epsilon\, \sigma \\ (open) & (\mathsf{open}\, \epsilon'\, f)\, v & \longrightarrow & \mathsf{restrict}\, \epsilon\, (f\, v) \\ & & \text{with} & \epsilon = effectof(f) \quad \bullet \vdash f : \sigma_1 \to \epsilon\, \sigma_2 \\ (restrictv) & \mathsf{restrict}\, \epsilon\, v & \longrightarrow & v \end{array}$$

Effect Annotation Extractor:

$$\begin{array}{lll} effectof(\lambda^{\epsilon} x : \sigma.e) & = & \epsilon \\ effectof(\mathsf{handler}^{\epsilon} h) & = & \epsilon \end{array} \qquad effectof(\mathsf{perform}\, \epsilon\, \overline{\sigma}\, op) = \epsilon$$

Figure 3. Dynamic Semantics of System F_{pwo}

resumption $\lambda^{\epsilon} x{:}\ \sigma_2.\, \mathsf{prompt}\, m\, h\, \mathsf{E}[x]$ by finding the marker m and applies the operation clause to it, which is instantiated by the (*perform*) rule.

The (*open*) rule generates a restrict frame using th effect annotation of the function value. Here, *effectof* meta-function is used to extract the effect type

from the function value, which is either a lambda abstraction, an operation call, or a handler.

The effect row of an open expression is used for type checking. The (*restrictv*) rule removes the restrict frame if the subexpression is a value.

3.2.3. Example In System F_{pwo}, we can express the second program example of Section 2.1 in the following way.

$$
\begin{array}{l}
\mathsf{handler}^{\langle\rangle}\ h^{read}\,\lambda^{\langle read\rangle}\_.\mathsf{handler}^{\langle read\rangle}\ h^{exn}\,f \quad \text{where} \\
\quad f = \lambda^{\langle exn,\ read\rangle}\_.\ \mathsf{perform}\,\langle exn,\ read\rangle\ ask\,()\ +\ \mathsf{perform}\,\langle exn,\ read\rangle\ ask\,() \\
\quad h^{read} = \{\ ask \mapsto \lambda^{\langle\rangle}\ \_.\lambda^{\langle\rangle}\ k : int \to \langle\rangle int.\ k\ 3\} \\
\quad h^{exn} = \{\ throw \mapsto \Lambda\alpha.\lambda^{\langle read\rangle} x : string.\lambda^{\langle read\rangle}\ k : \alpha \to \langle read\rangle\ int.\ 127\ \}.
\end{array}
$$

This expression is evaluated to 6 through the following steps (we omit some type annotations for space reasons).

$$
\begin{array}{ll}
 & \mathsf{handler}^{\langle\rangle}\ h^{read}\,\lambda^{\langle read\rangle}\_\ :\ unit.\mathsf{handler}^{\langle\rangle}\ h^{exn}\,f. \\
\longmapsto^* & \mathsf{prompt}\ m_1\ h^{read}\,(\mathsf{prompt}\ m_2\ h^{exn}\,(\\
 & \quad \underline{\mathsf{perform}\,\langle exn,\ read\rangle\ ask\,()}\ +\ \mathsf{perform}\,\langle exn,\ read\rangle\ ask\,())) \\
\longmapsto & \mathsf{prompt}\ m_1\ h^{read}\,(\mathsf{prompt}\ m_2\ h^{exn}\,(\\
 & \quad \mathsf{yield}\ m_1\ (\lambda^{\langle\rangle} k.\,(\lambda^{\langle\rangle}\ x.\lambda^{\langle\rangle}\ k_1.\,k_1\ 3)\ ()\ k)\ +\ \mathsf{perform}\,\langle exn\rangle\ ask\,())) \\
\longmapsto & (\lambda^{\langle\rangle} k.\,(\lambda^{\langle\rangle}\ x.\lambda^{\langle\rangle}\ k_1.\,k_1\ 3)\ ()\ k) \\
 & \quad (\lambda^{\langle\rangle} z\ :\ int.\ \mathsf{prompt}\ m_1\ h^{read}\,(\mathsf{prompt}\ m_2\ h^{exn}\,(z\ +\ \mathsf{perform}\,\langle exn\rangle\ ask\,()))) \\
\longmapsto^* & \mathsf{prompt}\ m_1\ h^{read}\,(\mathsf{prompt}\ m_2\ h^{exn}\,(\underline{3}\ +\ \mathsf{perform}\,\langle exn\rangle\ ask\,())) \\
\longmapsto^* & \mathsf{prompt}\ m_1\ h^{read}\,(\mathsf{prompt}\ m_2\ h^{exn}\,(3\ +\ 3)) \\
\longmapsto^* & 6
\end{array}
$$

The first step shows the reduction of handlers. Specifically, the two handlers are instantiated to prompts with fresh markers m_1 and m_2. The gray part indicates the body of f, which is evaluated with the evidence vector $\langle\!\langle\, exn : (m_2, h^{exn}, w) \mid w \,\rangle\!\rangle$ representing surrounding handlers, where $w = \langle\!\langle\, read : (m, h^{read}, \langle\!\langle\,\rangle\!\rangle)\,\rangle\!\rangle$. The subsequent three steps complete the *ask* operation call. In the next line, the (*perform*) rule prepares the marker m_1 and the wrapped handler clause. After that, the resumption is captured and the wrapped handler clause is applied to it via the (*prompt*) rule. After reducing some β-redexes, 3 is obtained as the result of the operation call.

3.3. Static Semantics

The static semantics is mutually defined with three relations $\vdash$, $\vdash_{\mathsf{val}}$, and $\vdash_{\mathsf{ops}}$ in Figure 4.

- $\Gamma \vdash e : \sigma \mid \epsilon$ means expression e is typed σ under type environment Γ and contextual effect ϵ, i.e., the type of the evidence vector provided for evaluation of e.
- $\Gamma \vdash_{\mathsf{val}} v : \sigma$ means value v is typed σ under type environment Γ. Note that if value v can be typed with $\vdash_{\mathsf{val}}$ relation, then it can be typed with *any* effect type ϵ with $\vdash$ relation, according to type rule [VAL].

– $\Gamma \vdash_{\mathsf{ops}} h \,:\, \sigma \mid l \mid \epsilon$ means that the sequence of operation clauses h has return type σ and handles effect operation of label l under effect type ϵ.

We also use well-formedness relation $\vdash_{\mathsf{wf}}$ and definitional equality of types $\vdash_{\mathsf{eq}}$ defined in Appendix 8

Let us now look at the typing rules (Figure 4). These rules are syntax directed in the sense that the syntax of the expressions determines the applicable type rule.

$$\boxed{\Gamma \vdash e \,:\, \sigma \mid \epsilon \qquad \Gamma \vdash_{\mathsf{val}} v \,:\, \sigma \qquad \Gamma \vdash_{\mathsf{ops}} h \,:\, \sigma \mid l \mid \epsilon}$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v \,:\, \sigma}{\Gamma \vdash v \,:\, \sigma \mid \epsilon}\ [\textsc{val}] \qquad \frac{\Gamma,\ x \,:\, \sigma_1 \vdash e \,:\, \sigma_2 \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon} x \,:\, \sigma_1.\, e \,:\, \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{abs}]$$

$$\frac{x \,:\, \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x \,:\, \sigma}\ [\textsc{var}] \qquad \frac{\Gamma \vdash e_1 \,:\, \sigma_1 \to \epsilon\, \sigma \mid \epsilon \quad \Gamma \vdash e_2 \,:\, \sigma_1 \mid \epsilon}{\Gamma \vdash e_1\, e_2 \,:\, \sigma \mid \epsilon}\ [\textsc{app}]$$

$$\frac{\Gamma \vdash v \,:\, \sigma \quad \mathsf{k} \neq \mathsf{lab} \quad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{val}} \Lambda\alpha^{\mathsf{k}}.\ v \,:\, \forall\alpha^{\mathsf{k}}.\sigma}\ [\textsc{tabs}] \qquad \frac{\Gamma \vdash e \,:\, \forall\alpha^{\mathsf{k}}.\sigma_1 \mid \epsilon \quad \vdash_{\mathsf{wf}} \sigma \,:\, \mathsf{k}}{\Gamma \vdash e\,\sigma \,:\, \sigma_1[\alpha{:=}\sigma] \mid \epsilon}\ [\textsc{tapp}]$$

$$\frac{\begin{array}{l} op \,:\, \forall\overline{\alpha}.\tau_1 \to \tau_2 \in \Sigma(c_l) \\ \vdash_{\mathsf{eq}} (\tau_1[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_1 \,:\, * \qquad \vdash_{\mathsf{eq}} (\tau_2[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_2 \,:\, * \end{array}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}\, op\, \langle c l\ \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} \,:\, \sigma_1 \to \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \sigma_2}\ [\textsc{perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h \,:\, \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\, h \,:\, (() \to \langle l \mid \epsilon \rangle\, \sigma) \to \epsilon\, \sigma}\ [\textsc{handler}]$$

$$\frac{\begin{array}{l} \{op_i \,:\, \forall\overline{\alpha}_i.\tau_i^{in} \to \tau_i^{out}\}_{i=1}^{n} = \Sigma(c_l) \quad \overline{\alpha}_i \not\cap \mathsf{ftv}(\epsilon, \overline{\sigma}, \overline{\sigma'}) \\ \vdash_{\mathsf{eq}} \tau_i^{in}\, \overline{\sigma'} \equiv \sigma_i^{in} \,:\, * \quad \vdash_{\mathsf{eq}} \tau_i^{out}\, \overline{\sigma'} \equiv \sigma_i^{out} \,:\, * \\ \Gamma \vdash_{\mathsf{val}} f_i \,:\, \forall\overline{\alpha_i}.\sigma_i^{in} \to \epsilon\, ((\sigma_i^{out} \to \epsilon\, \sigma) \to \epsilon\, \sigma) \end{array}}{\Gamma \vdash_{\mathsf{ops}} \{op_1 \to f_1, \ldots, op_n \to f_n\} \,:\, \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon}\ [\textsc{ops}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} f \,:\, (\sigma \to \epsilon'\, \sigma') \to \epsilon'\, \sigma'}{\Gamma \vdash \mathsf{yield}\, m f \,:\, \sigma \mid \epsilon}\ [\textsc{yield}] \qquad \frac{\Gamma \vdash_{\mathsf{ops}} h \,:\, \sigma \mid l \mid \epsilon \quad \Gamma \vdash e \,:\, \sigma \mid \langle l \mid \epsilon \rangle}{\Gamma \vdash \mathsf{prompt}\, m\, h\, e \,:\, \sigma \mid \epsilon}\ [\textsc{prompt}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v \,:\, \sigma_1 \to \epsilon'\, \sigma_2 \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{open}\, \epsilon\, v \,:\, \sigma_1 \to \epsilon\, \sigma_2}\ [\textsc{open}] \qquad \frac{\Gamma \vdash e \,:\, \sigma \mid \epsilon' \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash \mathsf{restrict}\, \epsilon'\, e \,:\, \sigma \mid \epsilon}\ [\textsc{restrict}]$$

Closed Prefix: $\langle l_1, \ldots, l_n \rangle \leqslant \langle l_1, \ldots, l_n \mid \epsilon \rangle \quad (n \geq 0)$

Figure 4. Typing Rules of System F_{pwo}

The [VAL] rule types values as expressions with any effect type ϵ. The [VAR] rule is usual. The [ABS] rule type check the body e with the effect annotation ϵ of the lambda abstraction. The [APP] is standard except for the effect type: the operator (e_1) and operand (e_2) need to be typed under the contextual effect of entire expression ($e_1\ e_2$). Furthermore, the effect type of the body of the operator also needs to agree with the one of the entire expression. The restriction guarantees that the evidence vector is passed correctly to sub-expressions. This may seem too restrictive, but open construct liberates the restriction. The [TABS] and [TAPP] rules are standard except for bound type variables, which cannot have kind lab.

The [PERFORM] rule determines the type of the operation call referring to the effect context Σ and the effect row $\langle cl\ \overline{\sigma'} \mid \epsilon \rangle$. The signature of the operation is found in the effect context and the type arguments $\overline{\sigma}$ are substituted for the type variables in the argument type τ_1 and the result type τ_2. Furthermore, the type arguments $\overline{\sigma'}$ from the effect row are applied to $\tau_1[\overline{\alpha} := \overline{\sigma}]$ and $\tau_2[\overline{\alpha} := \overline{\sigma}]$. As an example, assuming $\Sigma = \{\ exn\ :\ \{\ throw\ :\ \forall\alpha.string \to \alpha\ \}\}$, we can write $\lambda^{\langle exn \rangle}\ x\ :\ string.\ 1\ +\ \mathsf{perform}\ throw\,\langle exn \rangle\ int\,x$ as a well-typed function. The operation call in the body is typed with an instance of the [PERFORM] rule as follows.

$$\frac{\begin{array}{c} throw\ :\ \forall\beta.string \to \beta\ \in \Sigma(exn) \\ \vdash_{\mathsf{eq}}\ string[\beta := int] \equiv string \qquad \vdash_{\mathsf{eq}}\ \beta[\beta := int] \equiv int \end{array}}{x\ :\ string \vdash_{\mathsf{val}}\ \mathsf{perform}\ throw\,\langle exn \rangle\ int\ :\ string \to \langle exn \rangle\ int}$$

In this case, $c_l = exn$ and $\overline{\sigma'}$ is an empty sequence of types. If we replace *throw* with $polythrow\,string$, $c_l = polyexn$ and $\overline{\sigma'}$ is a singleton sequence *string*.

$$\frac{\begin{array}{c} polythrow\ :\ \forall\beta.(\lambda\alpha.\alpha) \to (\lambda\alpha.\beta)\ \in \Sigma(polyexn) \\ \vdash_{\mathsf{eq}}\ (\lambda\alpha.\alpha)[\beta := int]\ string \equiv string \qquad \vdash_{\mathsf{eq}}\ (lambda\,\alpha.\beta)[\beta := int]\ string \equiv int \end{array}}{x\ :\ string \vdash_{\mathsf{val}}\ \mathsf{perform}\ polythrow\,\langle\,\rangle\ int\ :\ string \to \langle polyexn\ string \rangle\ int}$$

The [HANDLER] rule is defined for handler expressions. A handler takes a computation of type $(() \to \langle l \mid \epsilon \rangle\ \sigma)$ and handles the effect l. Hence, the effect row of the entire function type is ϵ, not $\langle l \mid \epsilon \rangle$.

The [OPS] rule determines the effect labels $c_l\,\overline{\sigma'}$, which indicate the handled effect. Each operation clause f_i takes an operation argument of type σ_i^{in} and a resumption of type $\sigma_i^{out} \to \epsilon\,\sigma$. The result type ($\sigma$) of the handler is the result type of all resumptions and operation clauses, because handlers in System F_{pwo} are *deep* ones. The condition $\overline{\alpha}_i\ \not\cap \mathsf{ftv}(\epsilon, \sigma, \overline{\sigma'})$ avoids unexpected binding in the type of f_i. The arguments of effect constants $\overline{\sigma'}$ are derived from the typing of each operation clause. By combining them with the effect constant c_l, we derive the effect label $c_l\,\overline{\sigma'}$.

The [YIELD] rule requires careful reading. Recall that f is a wrapped handler clause that will be applied to a resumption. The result type of f, which is the result type of the handler clause, must agree with the result type of the resumption. Therefore the two σ' need to agree. The two σ indicate that the input type of f, which is the type of the "result" of the operation call, must agree with the

type of the yield expression. Note that the effect type ϵ of yield is not related to the effect ϵ' of the operation clause, as the evaluation contexts of yield and prompt (in which f will be evaluated) are different in general.

The [PROMPT] rule extends the contextual effect ϵ with the effect label l to type check subexpression e. The result type of e and that of handler clauses h need to agree, and it becomes the type of the entire prompt expression.

The [OPEN] rule opens (make bigger) the effect type to the given effect ϵ. The original effect type ϵ' must be a *closed prefix* of the resulting effect type ϵ. An effect row ϵ_1 is a closed prefix of ϵ_2 if and only if ϵ_1 consists of labels in a prefix of ϵ_2 and ends in $\langle\,\rangle$. We discuss the definition of the closed prefix relation in the next section. The [RESTRICT] rule is similar to [OPEN] and allows expression e to be typed under a closed prefix ϵ'.

3.4. Effect Rows and Closed Prefix Relation

In this section, we discuss how the type system exploits effect rows to perform type checking against effect handlers and discuss requirement for open and restrict to entail type safety.

Recall the typing rule [PERFORM] for operation calls. The conclusion of the rule has an effect row $\langle c_l\,\overline{\sigma'} \mid \epsilon\,\rangle$, which tells us that evaluation of the operation call needs to access a handler of effect label $c_l\,\overline{\sigma'}$. The accessibility of the required handler is guaranteed by the row equivalence rules defined in Figure 2.

For instance, among the two examples below, the first one is correctly rejected due to the inapplicability of [EQ-SWAP], and the second one is accepted as desired.

$$\mathsf{handler}\ h^{polyexn\ int}\,(\mathsf{handler}\ h^{polyexn\ string}\,(\lambda\_.\ throw\ 1))$$
$$\mathsf{handler}\ h^{polyexn\ int}\,(\mathsf{handler}\ h^{polyexn\ string}\,(\lambda\_.\ throw\text{"}hello\text{"}))$$

We design the closed prefix relation so that restrict does not increase handlers accessible from the sub-expressions. This is stated as the following property.

Proposition 1.
If $\epsilon.l$ is defined and $\epsilon \leqslant \epsilon'$, then $\epsilon'.l$ is also defined and $\epsilon.l = \epsilon'.l$.

It is obvious that the closed prefix relation satisfies this property, but what about other candidates? Initially, we considered an *open prefix* relation defined as follows.

$$\langle l_1, \ldots, l_k \mid \mu\,\rangle \leqslant_?\ \langle l_1, \ldots, l_k, l_{k+1}, \ldots, l_n \mid \mu\,\rangle$$

Here, μ is a type variable of kind eff. This relation leads to loss of type safety, as shown by the following example.

$$(\Lambda\mu.\lambda^{\langle polyexn\ int|\mu\,\rangle}\ f :\ ()\to\mu\,().\ \mathsf{restrict}\ \mu\,(f()))$$
$$\langle polyexn\ string\rangle\ \lambda^{\langle polyexn\ string\,\rangle}\_.polythrow\text{"}blame!\text{"}$$

Here, the example would be accepted with an open prefix relation because $\mu \leqslant_?\ \langle polyexn\ int \mid \mu\,\rangle$ holds. However, the effect annotation $\langle polyexn\ int \mid \mu\,\rangle$ indicates that the innermost *polyexn* handler expects *polyexn int*, while *polythrow* raises a string value, causing a type mismatch at runtime! Fortunately, using the closed prefix relation will reject the example and preserve soundness.

Effect Requirement: $\varphi ::= \varnothing \mid \epsilon$, with an order defined as:

$$\frac{}{\varnothing \sqsubseteq \varphi}\ [\textsc{bottom}] \qquad \frac{}{\varphi \sqsubseteq \varphi}\ [\textsc{refl}] \qquad \frac{\epsilon_1 \leqslant \epsilon_2}{\epsilon_1 \sqsubseteq \epsilon_2}\ [\textsc{prefix}]$$

Smart open and restrict:

$$\begin{array}{lll}
open'\ \varphi\ \varphi'\ e = e & (if \varphi \cong \varphi') \\
open'\ \varnothing\ \varphi'\ e = e & \\
open'\ \epsilon\ \epsilon'\ e = & \\
\quad \mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{open}\ \epsilon'\ x & (if \epsilon \leqslant \epsilon') \\
open'\ \varphi\ \varphi'\ e = undefined & (otherwise)
\end{array}
\qquad
\begin{array}{lll}
restrict'\ \varphi\ \varphi'\ e = e & (if \varphi \cong \varphi') \\
restrict'\ \varnothing\ \varphi'\ e = e & \\
restrict'\ \epsilon\ \epsilon'\ e = \mathsf{restrict}\ \epsilon\ e & (if \epsilon \leqslant \epsilon') \\
restrict'\ \varphi\ \varphi'\ e = undefined & (otherwise)
\end{array}$$

Figure 5. Effect Requirement and Auxiliary Definitions

4. Open Floating

In this section, we present open floating (in Figure 6), a transformation algorithm defined on System F_{pwo}. We first overview the definition here. In Section 4.1, we describe the auxiliary definitions and in Section 4.2, explain the main definition of open floating in detail.

The algorithm consists of three kinds of rules taking care of expressions, values, and operation clauses.

- $\Gamma \vdash e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi$
- $\Gamma \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma$
- $\Gamma \vdash_{\mathsf{ops}} h \rightsquigarrow h' : \sigma \mid l \mid \epsilon$

In this study, proper programs are well-typed and *internal-safe* expressions.

Definition 1. (*Internal-free expression, Internal-safe expression*)
An expression is *internal-safe* if it is (1) internal-free (contains no prompt or yield expressions) or (2) reduced from an internal-safe expression.

The input of open floating is a proper program fragment. The output is a proper program fragment with the *required* effect φ. Required effects are defined in Section 4.1.

Open floating adjusts effect types at higher nodes of the abstract syntax tree by inserting restrict, rather than at lower nodes and leaves. To achieve this, rule [Open-App] removes open call and the entire algorithm maintains the effect type of programs using auxiliary definitions given in Section 4.1.

We designed the algorithm so that it only changes open, restrict, and the effect type of expressions. This ensures that the algorithm preserves the type of expressions. The type environment given to the algorithm and the type returned from the algorithm are used to examine the effect type of the function body (in the [App] case). The effect row given to the expression case is used to calculate a new effect type (in the [App] and [Bind] cases).

4.1. Auxiliary Definitions

We extend effect rows to represent required effect as *effect requirement* φ in Figure 5. This extension allows us to communicate the fact that an expression imposes *no requirement* on the context, using the symbol $\varnothing$. For example, values

can be evaluated with any evidence vector, hence the effect requirement of values is always $\varnothing$. If we use $\langle\,\rangle$ instead, open floating would generate redundant restrict as in $div(\mathsf{restrict}\,\langle\,\rangle\,1, \mathsf{restrict}\,\langle\,\rangle\,0)$. Using $\varnothing$, the algorithm can generate $div(1, 0)$ as desired.

We also extend the closed prefix relation, open, and restrict so that they take an effect requirement instead of a plain effect row, as in Figure 5. Smart open ($open'$) and smart restrict ($restrict'$) are *meta-level* (partial) functions; they act as auxiliary functions of the algorithm. They generate open or restrict only if two effect rows are different. Smart open may seem strange; it generates a let-binding. Recall that, when we open an expression, the expression needs to be a value as specified by the syntax rules.

We use the *sup* meta function in the [App] and [Bind] cases of the algorithm, in order to calculate the result requirement satisfying the requirements generated from the subexpressions and the original effect type. The partial function *sup* is defined as follows.

$$sup \;:\; Requirement \longrightarrow 2^{Requirement} \longrightarrow Requirement$$
$$sup_{\epsilon}\,\{\varphi_i\}_i \;=\; \mathsf{min}\,\{\,\varphi \mid \varphi_i \sqsubseteq \varphi \sqsubseteq \epsilon\,\}$$

Here, ϵ is the original effect type of the expression (application or let-binding) and each φ_i is the requirement from the subexpression. The result requirement of the expression need to be smaller than the original effect and bigger than the requirements of subexpressions. The *sup* function choose the best (smallest) requirement from the candidates.

4.2. The Definition

We define the open floating algorithm in Figure 6. The light propositions are invariants, not side conditions; we write them to clarify the intention of the algorithm design.

Rule [Var] simply returns the variable with its type. Rule [Lam] recursively applies the algorithm to the body e. We do not simply return $\lambda^{\varphi} x : \sigma_1.e'$ but return $\lambda^{\epsilon}\, restrict'\,\varphi\epsilon\; e'$ in order to preserve the type of the lambda abstraction. Rule [Val] assigns a null requirement $\varnothing$ to value. Rule [App] treats three effect requirements: the requirements of the two sub-expressions (φ_1 and φ_2) and the effect type of the function body. The rule uses the supremum of these requirements for the result. Rule [Bind] also takes the supremum of the two effect rows. Rules [TAbs], [TApp], [Perform], [Handler], and [Ops] simply recursively call the algorithm and propagate the results.

Rule [Open-App] processes application of an opened function. It removes open and may assign a smaller effect to the expression. This rule conflicts with the [App] rule, hence we give priority to [Open-App]. Rule [Open-Preserve] recursively calls the algorithm while keeping open. This rule is required for function arguments, for instance. Let us consider the following example.

$$\begin{array}{l} safemap = \\ \quad \lambda^{\langle\,\rangle}\, lst : list\langle int\rangle.\lambda^{\langle\,\rangle}\, f : int \to \langle exn\rangle\, int. \\ \quad\quad \mathsf{handler}^{\langle\,\rangle}\,\{\, throw \longmapsto \lambda^{\langle\,\rangle}\, x : string.\, \lambda^{\langle\,\rangle}\, k : int \to int.\,[\,]\,\}\,(map\, int\, int\, \langle exn\rangle\, lst\, f) \\ safemap\,[1, 2, 3, 4]\,(\mathsf{open}\,\langle exn\rangle\, addone) \end{array}$$

$$\boxed{\Gamma \vdash e \mid \epsilon \rightsquigarrow e' : \sigma' \mid \varphi \qquad \Gamma \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma \qquad \Gamma \vdash_{\mathsf{ops}} h \rightsquigarrow h' : \sigma \mid l \mid \epsilon}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_{\mathsf{val}} x \rightsquigarrow x : \sigma} \ [\textsc{Var}] \qquad \frac{\Gamma \vdash e \mid \epsilon \rightsquigarrow e' : \forall \alpha^k.\sigma_1 \mid \varphi}{\Gamma \vdash e\,[\sigma'] \mid \epsilon \rightsquigarrow e'\,[\sigma'] : \sigma[\alpha{:=}\sigma'] \mid \varphi} \ [\textsc{TApp}]$$

$$\frac{\Gamma \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma}{\Gamma \vdash v \rightsquigarrow v' : \sigma \mid \varnothing} \ [\textsc{Val}] \qquad \frac{\Gamma,\, \alpha^k \vdash_{\mathsf{val}} v \rightsquigarrow v' : \sigma}{\Gamma \vdash_{\mathsf{val}} \Lambda\alpha^k.v \rightsquigarrow \Lambda\alpha^k.v' : \forall\alpha.\,\sigma} \ [\textsc{TAbs}]$$

$$\frac{\Gamma,\, x : \sigma_1 \vdash e \mid \epsilon \rightsquigarrow e' : \sigma_2 \mid \varphi \quad \varphi \sqsubseteq \epsilon}{\Gamma \vdash_{\mathsf{val}} \lambda^{\epsilon}\, x : \sigma_1.\, e \ \rightsquigarrow \lambda^{\epsilon}\, x : \sigma_1.\, \mathit{restrict}'\, \varphi\epsilon\, e' : \sigma_1 \to \epsilon\, \sigma_2} \ [\textsc{Lam}]$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \mid \epsilon \rightsquigarrow e_1' : \sigma_2 \to \epsilon_{\mathsf{body}}\, \sigma \mid \varphi_1 \quad \Gamma \vdash e_2 \mid \epsilon \rightsquigarrow e_2 : \sigma_2 \mid \varphi_2 \\ \epsilon_s = \mathit{sup}_{\epsilon}\, \{\epsilon_{\mathsf{body}},\, \varphi_1,\, \varphi_2\}\end{array}}{\Gamma \vdash e_1\, e_2 \mid \epsilon \rightsquigarrow (\mathit{open}'\, \epsilon_{\mathsf{body}}\, \epsilon_s\, (\mathit{restrict}'\, \varphi_1\, \epsilon_s\, (e_1'))\,)\, (\mathit{restrict}'\, \varphi_2\, \epsilon_s\, e_2') : \sigma \mid \epsilon} \ [\textsc{App}]$$

$$\frac{\Gamma \vdash e_1 \mid \epsilon \rightsquigarrow e_1' : \sigma_1 \mid \varphi_1 \quad \Gamma \vdash e_2 \mid \epsilon \rightsquigarrow e_2' : \sigma_2 \mid \varphi_2 \quad \varphi = \mathit{sup}_{\epsilon}\, \{\varphi_1,\, \varphi_2\}}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \epsilon \rightsquigarrow \mathsf{let}\, x = \mathit{restrict}'\, \varphi_1\, \varphi e_1'\, \mathsf{in}\, \mathit{restrict}'\, \varphi_2\, \varphi e_2' : \sigma_2 \mid \varphi} \ [\textsc{Bind}]$$

$$\frac{\begin{array}{l}\mathit{op} : \forall\overline{\alpha}.\tau_1 \to \tau_2 \in \Sigma(c_l) \qquad \overline{\alpha} \not\cap \mathsf{ftv}(\Gamma) \\ \vdash_{\mathsf{eq}} (\tau_1[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_1 : * \quad \vdash_{\mathsf{eq}} (\tau_2[\overline{\alpha}{:=}\overline{\sigma}]\ \overline{\sigma'}) \equiv \sigma_2 : *\end{array}}{\Gamma \vdash_{\mathsf{val}} \mathsf{perform}\ \mathit{op}\, \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} \rightsquigarrow \mathsf{perform}\ \mathit{op}\, \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \overline{\sigma} : \sigma_1 \to \langle c_l\, \overline{\sigma'} \mid \epsilon \rangle\, \sigma_2} \ [\textsc{Perform}]$$

$$\frac{\Gamma \vdash_{\mathsf{ops}} h \rightsquigarrow h' : \sigma \mid l \mid \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{handler}^{\epsilon}\, h \rightsquigarrow \mathsf{handler}^{\epsilon}\, h' : (() \to \langle l \mid \epsilon \rangle\, \sigma) \to \epsilon\, \sigma} \ [\textsc{Handler}]$$

$$\frac{\begin{array}{l}\{\mathit{op}_i : \forall\overline{\alpha}_i.\tau_i^{in} \to \tau_i^{out}\}_{i=1}^{n} = \Sigma(c_l) \quad \overline{\alpha}_i \not\cap \mathsf{ftv}(\epsilon, \overline{\sigma}, \overline{\sigma'}) \\ \vdash \mathit{ti}^{in}\, \overline{\sigma'} \equiv \mathit{si}^{in} : * \ \vdash \mathit{ti}^{out}\, \overline{\sigma'} \equiv \mathit{si}^{out} : * \\ \Gamma \vdash_{\mathsf{val}} f_i \rightsquigarrow f_i' : \forall\overline{\alpha_i}.\sigma_i^{in} \to \epsilon\, ((\sigma_i^{out} \to \epsilon\, \sigma) \to \epsilon\, \sigma)\end{array}}{\Gamma \vdash_{\mathsf{ops}} \{\mathit{op}_1 \to f_1, \ldots, \mathit{op}_n \to f_n\} \rightsquigarrow \{\mathit{op}_1 \to f_1', \ldots, \mathit{op}_n \to f_n'\} : \sigma \mid c_l\, \overline{\sigma'} \mid \epsilon} \ [\textsc{Ops}]$$

$$\frac{\Gamma \vdash v\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi \quad \varphi \sqsubseteq \epsilon}{\Gamma \vdash (\mathsf{open}\, \epsilon\, v)\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi} \ [\textsc{Open-App}]$$

$$\frac{\Gamma \vdash v \rightsquigarrow v' : \sigma_1' \to \epsilon'\, \sigma_2' \quad \epsilon' \leqslant \epsilon}{\Gamma \vdash_{\mathsf{val}} \mathsf{open}\, \epsilon\, v \rightsquigarrow \mathsf{open}\, \epsilon\, v' : \sigma_1 \to \epsilon\, \sigma_2} \ [\textsc{Open-Preserve}]$$

$$\frac{\Gamma \vdash e \mid \epsilon_1 \rightsquigarrow e' : \sigma \mid \varphi}{\Gamma \vdash \mathsf{restrict}\, \epsilon_1\, e \mid \epsilon \rightsquigarrow e' : \sigma \mid \varphi} \ [\textsc{Restrict}]$$

Figure 6. Open Floating Algorithm

The *safemap* function expects a function argument of $\mathit{int} \to \langle \mathit{exn} \rangle\, \mathit{int}$ and handles the exception effect. If we want to pass a function of effect $\langle\,\rangle$ as the argument (*addone* in this case), we need to open it to make the entire program type check. The last rule [Restrict] ignores the existing restrict.

```
fun test-one() : ⟨exn,read₁,read₂⟩ int
  val x = square(1) + ... + square(1)  // call `square(1)` 20 times
  val y = square-ask₁()
  val z = square-ask₂()
  x + y + z

noinline fun square( i : int ) : ⟨exn, read₁⟩ int
  if True then i * i else throw("impossible: " ++ ask₁().show)
noinline fun square-ask₁() : read₁ int
  ask₁() * ask₁()
noinline fun square-ask₂() : read₂ int
  ask₂() * ask₂()
```

Open floating	Exec. time (sec.)	Open call (static)	Open call (dynamic)
Enable	0.957	22	220,000
Disable	2.388	3	30,000

Figure 7. Benchmarking open floating (the ideal case)

5. Evaluation

We implemented our open floating algorithm in the Koka compiler [10] and evaluated open floating with several artificial programs. The results show that, in the best case, open floating makes programs about 2.5 times faster, while in some cases, it makes programs slower. In this section, we summarize the experiments and discuss what kind of programs are made faster by open floating.

5.1. Ideal Case

Figure 7 includes a fragment of a small benchmark with the execution times. The number of open calls is the sum of open and restrict expressions in the program. As we can see, open floating is very effective here and the enabling open floating improves performance by 2.5×. The execution times are averaged over 3 runs, on an Intel Core i5 at 3Ghz with 8GiB memory running macOS 11.6.3, with Koka v2.3.9 extended with open floating. In the program, we use three kinds of effects: `read`$_1$, `read`$_2$, and `exn`. The function `test-one` has the effect ⟨`read`$_1$`,read`$_2$`,exn`⟩, while the other functions use smaller effects. In the body of `test-one`, the Koka type inferencer inserts the following open calls around the `square`, `square-ask`$_1$, and `square-ask`$_2$ functions:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{open}(\mathit{square})(1) + \ldots + \mathsf{open}(\mathit{square})(1) \quad (\text{call}\, \mathsf{open}(\mathit{square})(1)\, 20\ \text{times}) \\ \mathsf{let}\, y = \mathsf{open}(\mathit{squareask}_1)() \\ \mathsf{let}\, z = \mathsf{open}(\mathit{squareask}_2)() \\ x + y + z\end{array}$$

This program contains 22 open calls initially and open floating reduces them to 3 restricts as follows:

$$\begin{array}{l}\mathsf{let}\, x = \mathsf{restrict}(\, \mathit{square}(1) + \ldots + \mathit{square}(1)\,) \quad (\text{call}\, \mathit{square}(1)\, 20\ \text{times}) \\ \mathsf{let}\, y = \mathsf{restrict}(\mathit{squareask}_1()) \\ \mathsf{let}\, z = \mathsf{restrict}(\mathit{squareask}_2()) \\ x + y + z\end{array}$$

Open floating	Exec. time (sec.)	Open call (static)	Open call (dynamic)
Enable	0.500	22	220,000
Disable	0.182	3	30,000

Figure 8. Benchmark of Failure Case

Here all individual open calls around each *square* invokation are floated up to a single restrict, leading to the improved performance.

5.2. Failure Case

However, making a small modification to the program can make open floating less effective. Consider a modified version of the `square` function in Figure 7.

```
noinline fun square'( i : int ) : ⟨exn⟩ int
  if True then i * i else throw("impossible: ")
```

We have removed `++ ask₁().show` and changed the effect from ⟨`ask₁, exn`⟩ to ⟨`exn`⟩. Just as before, the open floating reduces the number of open calls in the `test-one` function, but now the program runs slightly slower as shown in Figure 8.

The reason why this happens is that the opened function (`square'`) has an effect row type of length one. In such case, the Koka runtime already optimizes the use of open by avoiding allocating an explicit evidence vector and directly using the single evidence *as is*. Since no allocation happens, this can be faster, in particular since the current implementation of the restrict operation is not optimized in a similar fashion yet. Instead, it is implemented combining a lambda abstraction and an open operation, as $\mathsf{restrict}\, e \doteq \mathsf{open}(\lambda\_.\, e)()$. We plan to improve the implementation of restrict in which case it should always be beneficial to perform open floating.

6. Future Work

Even though our open floating algorithm is effective, there are still situations where it can be improved. In particular, for certain higher-order programs, such as calls of map and fold, open floating can be improved. Consider the following program.

```
fun map( xs : list⟨a⟩, f : a → e b ) : e list⟨b⟩
fun g( x : int) : ⟨ask, ndet⟩ int

fun f() : ⟨ask, exn, ndet⟩ int
  ...
  map(lst, g)
  ...
```

The Koka compiler wraps the function g with an open call to make the program type check. This gives us the following expression.

$$map\ int\ int\langle ask, exn, ndet\rangle\ lst\ (\mathsf{open}\langle ask, exn, ndet\rangle\ g)$$

The current open floating algorithm does not change the program due to the rule [Open-Preserve] as we discussed in Section 4.2. At runtime, the opened function is applied to each element in the list `lst` by the `map` function. It would be better

to float the open to surround the the entire *map* call which reduces the number of open calls to one.

$$\mathsf{restrict}\langle ask,\ ndet\rangle(\ map\ int\ int\langle ask,\ ndet\rangle\ lst\ g)$$

7. Related Work

Our work is in the context of passing dynamic evidence vectors at runtime that correspond to the static effect types, as described by Xie and Leijen [19]. The idea of passing dynamic runtime evidence for static properties is not new and is a standard way of implementing qualified types and type classes [5, 17]. Here, the evidence takes the form of a dictionary of overloaded operations and corresponds to the qualified type constraints. In our work, open adjusts the runtime evidence vectors, while with type classes *instances* are used to modify runtime dictionaries. For example, a function with a `Show a` constraint may call a function with a `Show [a]` constraint. To call this, the received dictionary for `Show a` is transformed at runtime to a `Show [a]` dictionary using the `instance Show a => Show [a]` declaration. Usually, these "evidence adjustments" are called context reduction and generalized by the entailment relation in the theory of qualified types [6]. Peyton Jones et al. [12] explore the design space of sound context reduction in Haskell.

Gaster and Jones [4] present a system for extensible records based on the theory of qualified types. Here, a *lacks* constraint l/r corresponds to a runtime evidence, providing the offset in the record r where the label l would be inserted. When modifying the record, the evidence is also adjusted at runtime to reflect a new offset. For example, if another label is inserted before l, its offset is incremented. A similar mechanism is used in the system of type-indexed rows developed by Shields and Meijer [16].

In all of the above examples, we can imagine transformations similar to open floating that try to minimize the evidence adjustments, although we are not aware of any previous work that addresses this issue specifically.

Our formalization of effect row can roughly be understood as an instance of scoped rows discussed by Morris and McKinna [11]. They define a general *row theory* and *row algebra* with qualified types. Scoped rows are shown as an instance of them. Closed prefix relation is almost represented as left containment relation. Note that our calculus uses polymorphic effect rows such as $\forall\mu.\langle l_1,\ l_2 \mid \mu\rangle$ while scoped rows in [11] do not seem to entail it.

8. Conclusion

In this paper, we formalized open and restrict with their restriction of the closed prefix relation. The formalization is useful for determining the validity of a program transformation, such as open floating. We also defined the open floating algorithm on the formalized calculus and developed an implementation in Koka. The benchmark shows the effectiveness of open floating and point out room to improve the implementation.

$$\frac{\vdash_{\mathsf{eq}} \ \lambda\alpha^{\mathsf{k}_2}.\tau_1 \equiv \tau_1' \ : \ \mathsf{k}_2 \to \mathsf{k} \quad \vdash_{\mathsf{eq}} \ \tau_2 \equiv \tau_2' \ : \ \mathsf{k}_2}{\vdash_{\mathsf{eq}} \ \tau_1 \, \tau_2 \equiv \tau_n[\alpha^{\mathsf{k}_2} \coloneqq \tau_2'] \ : \ \mathsf{k}} \ [\textsc{kapp}]$$

$$\frac{}{\vdash_{\mathsf{eq}} \ \alpha^{\mathsf{k}} \equiv \alpha^{\mathsf{k}} \ : \ \mathsf{k}} \ [\textsc{kvar}]$$

$$\frac{}{\vdash_{\mathsf{eq}} \ c^{\mathsf{k}} \equiv c^{\mathsf{k}} \ : \ \mathsf{k}} \ [\textsc{kconst}]$$

$$\frac{\vdash_{\mathsf{eq}} \ \tau_1 \equiv \tau_1' \ : \ \mathsf{k}_1}{\vdash_{\mathsf{eq}} \ \lambda\alpha^*.\tau_1 \equiv \lambda\alpha^*.\tau_1' \ : \ * \to \mathsf{k}_1} \ [\textsc{kabs}]$$

$$\frac{\vdash_{\mathsf{wf}} \ \tau \ : \ *}{\vdash_{\mathsf{eq}} \ \tau \equiv \tau \ : \ *}$$

$$\frac{\vdash_{\mathsf{eq}} \ \tau \equiv \tau \ : \ *}{\vdash_{\mathsf{wf}} \ \tau \ : \ *}$$

$$\frac{\vdash_{\mathsf{wf}} \ \sigma_1 \ : \ * \quad \vdash_{\mathsf{wf}} \ \sigma_2 \ : \ * \quad \vdash_{\mathsf{wf}} \ \epsilon \ : \ \mathsf{eff}}{\vdash_{\mathsf{wf}} \ \sigma_1 \to \epsilon \, \sigma_2 \ : \ *} \ [\textsc{kfun}]$$

$$\frac{\vdash_{\mathsf{wf}} \ \sigma \ : \ *}{\vdash_{\mathsf{wf}} \ \forall\alpha^{\mathsf{k}}.\sigma \ : \ *} \ [\textsc{kall}]$$

$$\frac{\vdash_{\mathsf{wf}} \ l \ : \ \mathsf{lab} \quad \vdash_{\mathsf{wf}} \ \epsilon \ : \ \mathsf{eff}}{\vdash_{\mathsf{wf}} \ \langle \, l \mid \epsilon \, \rangle \ : \ \mathsf{eff}} \ [\textsc{keffect}]$$

Figure 9. Well-formedness and Definitional Equality of Types of System F_{pwo}

Appendix

We present the well-formedness relation $\vdash_{\mathsf{wf}}$ and definitional equality of types $\vdash_{\mathsf{eq}}$ in Figure 9. The type rules (Figure 4) use these relations.

References

[1] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. *Effekt: Lightweight Effect Polymorphism for Handlers*. Technical Report. University of Tübingen, Germany. 2020.

[2] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. "Effective Concurrency through Algebraic Effects." In *OCaml Workshop*, 13. 2015.

[3] Mattias Felleisen. "The Theory and Practice of First-Class Prompts." In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 180–190. POPL '88. Association for Computing Machinery, New York, NY, USA. 1988. doi:10.1145/73560.73576.

[4] Ben R. Gaster, and Mark P. Jones. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham. 1996.

[5] Mark P. Jones. "A Theory of Qualified Types." In *4th. European Symposium on Programming (ESOP'92)*, 582:287–306. Lecture Notes in Computer Science. Springer-Verlag, Rennes, France. Feb. 1992. doi:10.1007/3-540-55253-7_17.

[6] Mark P. Jones. "Simplifying and Improving Qualified Types." In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, 160–169. FPCA '95. La Jolla, California, USA. 1995. doi:10.1145/224164.224198.

[7] Daan Leijen. "Koka: Programming with Row Polymorphic Effect Types." In *MSFP'14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

[8] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 486–499. 2017.

[9] Daan Leijen. "Type Directed Compilation of Row-Typed Algebraic Effects." In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

[10] Daan Leijen. "Koka Repository." 2019. https://github.com/koka-lang/koka.

[11] J Garrett Morris, and James McKinna. "Abstracting Extensible Data Types: Or, Rows by Any Other Name." *Proceedings of the ACM on Programming Languages* 3 (POPL). ACM New York, NY, USA: 1–28. 2019.

[12] Simon Peyton Jones, Mark Jones, and Erik Meijer. "Type Classes: An Exploration of the Design Space." In *In Haskell Workshop*. 1997.

[13] Gordon D. Plotkin, and Matija Pretnar. "Handling Algebraic Effects." In *Logical Methods in Computer Science*, volume 9. 4. 2013. doi:10.2168/LMCS-9(4:23)2013.

[14] Matija Pretnar, Amr Hany Shehata Saleh, Axel Faes, and Tom Schrijvers. *Efficient Compilation of Algebraic Effects and Handlers*. CW Reports. Department of Computer Science, KU Leuven; Leuven, Belgium. 2017. https://lirias.kuleuven.be/retrieve/472230.

[15] Amr Hany Shehata Saleh, Georgios Karachalias, Matija Pretnar, and Tom Schrijvers. "Explicit Effect Subtyping." In *27th European Symposium on Programming (ESOP)*. 2018.

[16] Mark Shields, and Erik Meijer. "Type-Indexed Rows." In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 261–275. POPL'01. London, United Kingdom. 2001. doi:10.1145/360204.360230.

[17] Philip Wadler, and Stephen Blott. "How to Make Ad-Hoc Polymorphism Less Ad Hoc." In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, 60–76. POPL '89. ACM, Austin, Texas, USA. 1989. doi:10.1145/75277.75283.

[18] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. "Effect Handlers, Evidently." *Proceedings of the ACM on Programming Languages* 4 (ICFP). ACM New York, NY, USA: 1–29. 2020.

[19] Ningning Xie, and Daan Leijen. "Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C." *Proc. ACM Program. Lang.* 5 (ICFP). Association for Computing Machinery, New York, NY, USA. Aug. 2021. doi:10.1145/3473576.