Heapless Functional Programming
(Extended Abstract, Student Research Paper)

Ellis Kesterton (Student)[0009−0006−9369−8672] and
Edwin Brady (Supervisor)[0000−0002−9734−367X]

University of St Andrews, UK
{erk4,ecb10}@st-andrews.ac.uk

Abstract. Typed functional programming languages like Haskell and OCaml make heavy use of the heap at run-time. This makes them largely unsuitable for *systems programming*, where resources are limited and programs are often expected to run on bare-metal. This paper demonstrates how a (slightly restricted) high-level, pure, functional language can be compiled to machine code which does not use the heap at all. Despite usually requiring a heap at run-time, features such as higher-order functions, polymorphism and typeclasses are all supported by the surface language. This is made possible through *partial evaluation* [4]: by carefully reducing the program at compile-time, we can eliminate these high-level features entirely, resulting in a residual program which is trivial to compile. This paper describes the operation of this partial evaluator in detail, and shows how it can be implemented efficiently using an algorithm based on NbE. To demonstrate the practicality of this approach, we give several high-level example programs which can be compiled using this method.

1 Introduction

At the core of most functional languages is a set of high-level features including higher-order functions, polymorphism and algebraic data types. Most of the time these features are a boon: they allow the programmer to write concise, maintainable, readable code. However, all of these features typically require a heap at run-time. This is problematic if we want to use functional languages for *systems programming*, where it is common to target environments with extremely limited resources. Consider the following Haskell program:

```
main :: IO ()
main = do
    n :: Int <- readLn
    m :: Int <- readLn
    print (n + m)
```

While this program may appear simple, it implicitly makes use of a wide variety of high-level (heap-using) features. Alone, using `do`-notation requires typeclasses (to resolve the `Monad` instance), higher-order functions (for the implicit

calls to >>=) and polymorphism (also for >>=)! It is clear that every non-trivial Haskell program will use the heap in some capacity — features such as higher-order functions are too ubiquitous to avoid. Does this mean that high-level functional programming is entirely reliant on the heap? This paper demonstrates that this is not the case.

Reconsider our example from a computational perspective: it is unclear why a program which adds two integers should need a heap at all! In this case, features such as typeclasses and higher-order functions are only abstractions for the programmer; they are not adding any real computational value. We show that most of the time we can actually use these abstractions for free. Through careful manipulation at compile-time, it is possible to transform a large class of high-level functional programs into equivalent low-level programs which do not require the heap at all. We make use of the type system to restrict input programs, ensuring that this transformation is always possible. Compiling the earlier example with our prototype implementation produces identical machine code to the following C program compiled with `clang`.

```
int main() {
    int n m;
    scanf("%d", &n);
    scanf("%d", &m);
    printf("%d", n + m);
}
```

2 Contributions

The goal of this paper is to describe a method for designing and implementing high-level functional languages which do not use the heap at run-time. We believe this goal is worthwhile as not every low-level environment can feasibly support the high amount of heap allocations typically required by a functional language — by only using the stack, it becomes possible to use functional programming in these restricted environments. We hope that this work will help facilitate the design of functional languages which are suitable for systems programming on low-resource devices such as microcontrollers.

The full paper will describe the complete implementation of a practical surface language. However, for conciseness, this draft only focuses on how to compile a more minimal core language, $\mathcal{C}$. Specifically, we make the following contributions:

- We define a polymorphic, higher-order core language $\mathcal{C}$ which serves as an elaboration target for surface languages. $\mathcal{C}$ has a novel type system which guarantees that it can be compiled to stack-based code.
- We describe a pipeline which compiles $\mathcal{C}$ programs down to machine code. We primarily focus on how partial evaluation can be used eliminate $\mathcal{C}$'s high-level features (higher-order functions, polymorphism) at compile-time.

– In the full paper, we describe how to build a practical surface language which elaborates to $\mathcal{C}$. We cover common features such as typeclasses (incl. monads), implicit polymorphism and data types.

We have implemented the approach described in this paper in a prototype language, Strata, whose source code is available publicly[1].

3 Core Language

3.1 Outline

This section will define the core language $\mathcal{C}$ and describe how it can be compiled to stack-based machine code. The compilation process from $\mathcal{C}$ to machine code is broken down into several distinct steps:

1. Partially Evaluate
2. Uncurry
3. Lambda Lift
4. Compile

At the core of our approach lies a *partial evaluator*. By carefully performing reductions on the input program at compile-time, we can completely eliminate high-level features such as higher-order functions (HOFs) and polymorphism. By eliminating these features at compile-time, we no longer need to deal with the problem of representing them at run-time. The result of partial evaluation is therefore a program written in a very restricted subset of $\mathcal{C}$, greatly simplifying the rest of the compilation process. So that we can guarantee that partial evaluation will eliminate all occurences of these high-level features, we slightly restrict $\mathcal{C}$ programs through the type system. These extra/augmented rules will be justified in detail throughout the remainder of this section.

After partial evaluation, we follow up with two auxiliary source-to-source transformations: uncurrying and lambda-lifting. These steps remove curried and nested functions respectively, and are implemented using relatively standard algorithms. Unlike standard approaches to uncurrying [2], the type system and partial evaluator guarantee that the input is in a restricted form where function definitions are maximally η-expanded and there are no partial applications. This greatly simplifies the process, resulting in a trivial algorithm. Our approach to lambda-lifting is entirely standard, and ultimately the choice of algorithm is irrelevant. Quadratic-time algorithms have been specified in detail in existing literature [7].

Finally, the result of lambda-lifting is compiled to machine code. At this point in the pipeline the language is extremely restricted, with only first-order top-level function definitions remaining. Compilation is therefore trivial using standard techniques for first-order languages — our sample implementation uses a straightforward conversion to LLVM [6] to complete the compilation process.

[1] https://gitlab.com/strata-lang/compiler

3.2 Syntax

The core language $\mathcal{C}$ is an extension of System F_ω with let-bindings, products, fixed points, base types, and primitive operations. We assume the obvious strict semantics. The full syntax is omitted for brevity, but our notation is standard. In general, we use a single colon for annotating expressions with their type (e.g. $e : \tau$), and we use a double colon for annotating types with their kind (e.g. $\tau :: \kappa$).

Throughout the paper, we will often give examples in Haskell-like syntax rather than $\mathcal{C}$. This is to aid readability, as programming directly in a core language is very terse. We assume the natural elaboration from the Haskell-like syntax to $\mathcal{C}$.

3.3 Type System

The kinding and typing rules for $\mathcal{C}$ are given in Figure 1 and 2 respectively. The typing rules for constants and arithmetic/logic operations are omitted for brevity, assuming the usual types.

The most unusual part of the type system is the kind $\star_o$, indexed by an *order*. The full justification for this system is given in Section 4.5, but we give an intuitive overview here. The basic premise is that we want to differentiate first-order and higher-order terms. Types are categorised as follows

$\star_1$ Basic types, such as `Int`, `Bool`, etc.
$\star_2$ First-order function types, modulo currying. `Int -> Int -> Bool` is included in this kind, but `(Int -> Int) -> Int` is not.
$\star_3$ Everything else, including higher-order function types and types using universal quantification.

Intuitively, the kinds form a hierarchy where kinds with a higher number are more lenient in the types they allow. This hierachy is realised through sub-typing rule for kinds, *kind-sub*, meaning each kind is included in the one above.

3.4 Handling IO

IO functions are an important part of any practical language. In general, impurity does not mix well with program manipulation and partial evaluation — even something as simple as inlining a let-binding can become problematic if the bound expression has a side-effect. The most popular solution in modern functional programming is the IO monad, but it is difficult to represent at run-time without a heap. Instead, we thread a linear/unique[2] `World` token throughout the program, similar to the approach used by Clean [5]. At run-time the `World` token can be treated as the unit type.

[2] In this context, the distinction between linear and unique types is irrelevant.

$$\text{KIND-BASE}\ \frac{}{\Gamma \vdash \iota :: \star_1} \qquad \text{KIND-FORALL}\ \frac{\Gamma, \alpha :: \kappa \vdash \tau :: \star_i}{\Gamma \vdash (\forall \alpha :: \kappa . \tau) :: \star_3} \qquad \text{KIND-VAR}\ \frac{(\alpha :: \kappa) \in \Gamma}{\Gamma \vdash \alpha :: \kappa}$$

$$\text{KIND-ARROW}\ \frac{\Gamma \vdash \tau_1 :: \star_i \qquad \Gamma \vdash \tau_2 :: \star_j \qquad k = max(i+1, j, 3)}{\Gamma \vdash \tau_1 \to \tau_2 :: \star_k}$$

$$\text{KIND-ABS}\ \frac{\Gamma, \alpha :: \kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash (\lambda \alpha :: \kappa_1 . \tau) :: \kappa_1 \to \kappa_2} \qquad \text{KIND-APP}\ \frac{\Gamma \vdash \tau_1 :: \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \tau_2 :: \kappa_1}{\Gamma \vdash \tau_2 :: \kappa_1}$$

$$\text{KIND-PROD}\ \frac{\Gamma \vdash \tau_1 :: \star_i \qquad \Gamma \vdash \tau_2 :: \star_j \qquad k = max(i, j)}{\Gamma \vdash \tau_1 \times \tau_2 :: \star_k} \qquad \text{KIND-SUB}\ \frac{\Gamma \vdash \tau :: \star_i \qquad i < j}{\Gamma \vdash \tau :: \star_j}$$

Fig. 1. Kinding rules for $\mathcal{C}$.

4 Partial Evaluator

4.1 Rules

We characterise the partial evaluator by a set of reduction rules. The notation $t \rightsquigarrow u$ means that subterms matching the pattern t should be replaced with u during partial evaluation. Unless specified otherwise, there are no restrictions on where these reductions can take place, meaning that we reduce under binders by default. There is no stipulation on the order that reductions should take place, as the rules are always designed to preserve confluence. Occasionally, rules may have an attached side-condition which restricts when the rule should be applied — this means the rules may not form a true *rewriting system* [3], but we borrow the notation and terminology for conciseness.

Figures 3 and 4 detail the partial evaluator's reduction rules for types and expressions respectively.

4.2 Elimination

Ultimately, the partial evaluator removes high-level features by applying the appropriate elimination rule. For example, higher-order functions are eliminated through β-reduction; polymorphic functions are eliminated through monomorphisation. Almost all of the other rules are there to facilitate these eliminators by rewriting expressions into a form which is more amenable to further reduction. The most straightforward example of a "facilitating rule" is let-inlining. Consider the following expression (assume g is a free variable of the appropriate type):

```
let twice = λf : Int -> Int. λx : Int. f (f x)
in twice g n
```

$$\textsc{TYPE-VAR}\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{TYPE-ABS}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{TYPE-APP}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{TYPE-FORALL}\ \frac{\Gamma, \alpha :: \kappa \vdash e : \tau}{\Gamma \vdash (\Lambda\alpha :: \kappa.e) : (\forall\alpha :: \kappa.\tau)} \qquad \textsc{TYPE-INST}\ \frac{\Gamma \vdash e : \forall\alpha :: \kappa.\tau_1 \qquad \Gamma \vdash \tau_2 :: \kappa}{\Gamma \vdash e\ \tau_2 : \tau_1[\alpha \mapsto \tau_2]}$$

$$\textsc{TYPE-FIX}\ \frac{\Gamma \vdash e : \tau \to \tau \qquad \Gamma \vdash \tau :: \star_2}{\Gamma \vdash \text{fix}\ e : \tau} \qquad \textsc{TYPE-PROD}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\textsc{TYPE-LET-PROD}\ \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau_3}{\Gamma \vdash (\texttt{let}\ \langle x, y \rangle = e_1\ \texttt{in}\ e_2\ ) : \tau_3} \qquad \textsc{TYPE-LET}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ ) : \tau_2}$$

$$\textsc{TYPE-IF}\ \frac{\Gamma \vdash e_b : \texttt{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}$$

Fig. 2. Typing rules for $\mathcal{C}$.

$$(\lambda\alpha :: \kappa.\tau_1)\ \tau_2 \rightsquigarrow \tau_1[\alpha \mapsto \tau_2] \tag{1}$$

Fig. 3. Reduction rules for partially evaluating types.

At the moment there are no possible β-reductions, yet a higher-order function remains. We must therefore inline the higher-order function bound by `let` so that β-reduction can take place. After running the above program through the partial evaluator we are left with a first-order residual program:

$$g\ (g\ n)$$

The same idea applies to polymorphic functions which quantify over type variables — they must be inlined and monomorphised. This process is closely related to approaches which specialize higher-order functions to their functional arguments [1]. A deeper comparison with specialization is given in the full paper.

4.3 Distribution

We must be careful that other constructs in the core language do not interfere with the elimination of high-level features such as higher-order functions and polymorphism. Specifically, language constructs satisfying all of the following criteria must be paid extra attention:

– The construct is supported at run-time and may appear in the residual program.

$$
\begin{aligned}
e &\rightsquigarrow \lambda x.e\ x \quad \text{if } e : \tau_1 \to \tau_2 && \textit{(}\eta\textit{)} \\
(\lambda x.e_1)e_2 &\rightsquigarrow e_1[x \mapsto e_2] && \textit{(}\beta\textit{)} \\
\texttt{(if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\texttt{)}\ e_4 &\rightsquigarrow \texttt{if}\ e_1\ \texttt{then}\ e_2\ e_4\ \texttt{else}\ e_3\ e_4 && \textit{(if-distr)} \\
\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 &\rightsquigarrow e_2[x \mapsto e_1] \quad \text{if } e_1 \text{ is higher-order} && \textit{(let-inl)} \\
\texttt{(let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\texttt{)}\ e_3 &\rightsquigarrow \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ e_3 && \textit{(let-distr)} \\
\texttt{(let}\ \langle x, y\rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\texttt{)}\ e_3 &\rightsquigarrow \texttt{let}\ \langle x, y\rangle\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ e_3 && \textit{(let-prod-distr)} \\
(\Lambda\alpha.e)\tau &\rightsquigarrow e[\alpha \mapsto \tau] && \textit{(mono)}
\end{aligned}
$$

Fig. 4. Reduction rules for partially evaluating expressions.

– The construct can have a higher-order type if a higher-order functions is used as direct sub-expression of the construct.

Intuitively, the reasoning behind this criteria is that a higher-order function may get "stuck" as a sub-expression of the construct — if the construct may appear in the residual program, it is possible that the higher-order function will never get reduced and eliminated.

There are three constructs satisfying this criteria: fixed points, if-expressions and let-bindings (both regular and pair-destructuring). Fixed points are a large source of complications, and are separately discussed in Section 4.4. The remaining constructs will be addressed here.

If-Expressions If-expressions allow any type of data in their branches as long as both branches have the same type. We therefore should consider what happens if both branches contain higher-order functions, like in the following expression:

$$\texttt{(if}\ e_0\ \texttt{then}\ \lambda x.e_1\ \texttt{else}\ \lambda y.e_2\texttt{)}\ \lambda z.e_3$$

With just β-reduction, there is no way to make progress, yet higher-order functions still remain in the residual program! One might try adding extra reduction rules for `if`:

$$
\begin{aligned}
\texttt{if true then}\ e_t\ \texttt{else}\ e_f &\rightsquigarrow e_t \\
\texttt{if false then}\ e_t\ \texttt{else}\ e_f &\rightsquigarrow e_f
\end{aligned}
$$

However, if e_0 is a free variable (whose value may not be known until runtime), then we are still stuck. Instead, we add the rule *if-distr* which distributes applications into the branches of the `if`. The higher-order functions can then be β-reduced and eliminated, leaving the following residual expression:

$$\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

Let Bindings Our rule for inlining let-bindings covers the case when the bound expression is a higher-order function, but what about when the body of the `let` is a higher-order function? The following expression shows that there is a similar problem as with if-expressions:

$$\texttt{(let } n \texttt{ = 0 in } \lambda f.e_1\texttt{)} \ \lambda x.e_2$$

The partial evaluator will not inline the `let` as it does not bind a higher-order term to avoid code duplication. We are once again stuck, with no way to β-reduce the higher-order function bound in the body of the `let`. The solution is analogous to the one for if-expressions: introduce a rule which distributes applications inside the `let`'s body. The rules *let-distr* and *let-prod-distr* implement this behaviour for regular and product-destructuring let-bindings respectively.

4.4 Recursion

Practical functional programming languages almost always support general recursion, usually in the form of recursive definitions. General recursion is a particularly interesting feature as it makes the language turing-complete, and means that the language is no longer strongly-normalizing. This can cause problems for a partial evaluator, as we can no longer blindly reduce expressions without risking non-termination (this is true even if the input program is total).

In general, any rule which indiscriminantly unfolds `fix` will be problematic. However, if we do not unfold `fix`, then we run into problems when recursion is used in combination with higher-order functions. If we define a function which is both higher-order and recursive, then it will remain in the residual program (which should be first-order). Unlike non-recursive higher-order functions, we cannot inline and reduce the definition due to the `fix` blocking further evaluation.

To resolve this, we introduce the following simple restriction: *recursive functions must not be higher-order*. This is realised through the kind system, where the typing rule for `fix` requires that the type of the recursive definition must have kind $\star_2$. The reason why we use the kind system to enforce this rather than a syntactic check is explained in Section 4.5.

With this set up, no problematic reduction rules are needed for `fix`, as it can only bind first-order functions which are easy to compile. However, this restrction does seem impractical: plenty of useful functions are both higher-order and recursive! What about favourites like `map` and `fold`? Fortunately, this restriction is not a barrier to writing these functions. By abstracting out the recursive part of these functions into an inner definition, we can carefully avoid introducing a function which would violate the rule. Consider the following definition of `map` in Haskell:

```
map :: forall a b. (a -> b) -> [a] -> [b]
map f = go
  where
```

```
go :: [a] -> [b]
go []       = []
go (x : xs) = f x : go xs
```

The definition is functionally equivalent to the regular definition of `map`, but avoids introducing a function which is both recursive *and* higher-order. The outer function (`map`) is higher-order but not recursive, while the inner function (`go`) is recursive but not higher-order.

It is possible to do this refactoring for the majority of higher-order functions. Precisely, as long as the functional argument (in this case, `f :: a -> b`) remains constant throughout all recursive calls, the refactoring is possible and straightforward[3].

4.5 Polymorphism

Another trademark feature of modern functional programming languages is *polymorphism*. So that one function can operate on many different types of data, most functional languages use a uniform representation for all data — a pointer to a heap allocated cell. Polymorphic functions can then indiscriminantly treat all inputs in the same way, regardless of type. Since this is not an option without a heap, we use a variant of the other predominant method: *monomoprhisation*. This is realised through the *mono* rule in the partial evaluator.

Recursive Functions Like higher-order functions, we are relying on polymorphism being eliminated at compile-time by the partial evaluator. However, once again, recursive function definitions can cause issues when used in conjunction with polymorphism. As already discussed, recursive functions cannot be unfolded due to issues with termination. Unfortunately, this means that any function that is both recursive and polymorphic will remain in the residual program, as the partial evaluator cannot monomorphise without inlining first. Similar to higher-order functions, the typing rule for `fix` prevents this from happening as the kind of any type which uses universal quantification will be $\star_3$.

To implement standard functions which are both polymorphic and recursive we can employ a similar trick to the one for higher-order functions: abstract out the recursive part into an inner auxiliary definition. This is always possible provided recursive calls are made with the same type variables to what the function was given originally. For example, the following function composes an endofunction with itself n times.

```
nTimes :: forall a. Int -> (a -> a) -> (a -> a)
nTimes n f = go n
  where
    go :: Int -> a -> a
    go 0 x = x
    go m x = f (go (m - 1) x)
```

[3] A practical compiler might choose to perform this refactoring automatically.

Kind System In addition to the direct interaction between polymorphism and recursion, there is a more subtle nuance. When the typing rule for `fix` was introduced, you may have wondered why a kind system is used to ensure the function is first-order (modulo currying), as opposed to a syntactic condition. There are a few reasons for this.

Primarily, the problem with a syntactic check relates to polymorphic type variables. It raises the question — should a type variable be considered a first-order or higher-order type? Consider a recursive function which takes an argument whose type is a type variable:

```
choose :: forall a. Int -> a -> a -> a
choose = go
  where
    go :: Int -> a -> a -> a
    go 0 x y = x
    go n x y = go (n - 1) y x
```

Depending on whether the variable `a` is later instantiated with a basic type or function type, the type of the monomorphised function may be either first-order (modulo currying) or higher-order. Because of this, we cannot treat all type variables as being basic types: doing so would allow the programmer to circumvent the restrictions on higher-order functions by hiding functions types behind a type variable.

The problem stems from the fact that a type variable can represent a first-order type *or* a higher-order type. By separating type variables based on whether they represent functions or not, we can reason about whether a polymorphic function definition is truly first-order for every possible type instantiation.

References

1. Chin, W.N., Darlington, J.: A higher-order removal method. Lisp and Symbolic Computation **9**, 287–322 (1996)
2. Hannan, J., Hicks, P.: Higher-order uncurrying. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 1–11 (1998)
3. Jay, C.B.: Long pn normal forms and confluence. Tech. rep., Tech. Rep. LFCS-91-183 (and its revised version) (1991)
4. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Peter Sestoft (1993)
5. Koopman, P., Plasmeijer, R., van Eekelen, M., Smetsers, S.: Functional programming in clean (2002)
6. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
7. Morazán, M.T., Schultz, U.P.: Optimal lambda lifting in quadratic time. In: Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers 19. pp. 37–56. Springer (2008)