

Free Monads, Intrinsic Scoping, and Higher-Order Preunification

Nikolai Kudasov^[0000–0001–6572–7292]

Innopolis University, Innopolis, Tatarstan Republic, Russia
`n.kudasov@innopolis.ru`

Abstract. Type checking algorithms and theorem provers rely on unification algorithms. In presence of type families or higher-order logic, higher-order (pre)unification (HOU) is required. Many HOU algorithms are expressed in terms of λ -calculus and require encodings, such as higher-order abstract syntax, which are sometimes not comfortable to work with for language implementors. To facilitate implementations of languages, proof assistants, and theorem provers, we propose a novel approach based on the second-order abstract syntax of Fiore, data types à la carte of Swierstra, and intrinsic scoping of Bird and Patterson. With our approach, an object language is generated freely from a given bifunctor, extending the free monads approach. Then, given an evaluation function and making a few reasonable assumptions on it, we derive a higher-order preunification procedure on terms in the object language. More precisely, we apply a variant of E -unification for second-order syntax. Finally, we briefly demonstrate an application of this technique to implement type checking (with type inference) for Martin-Löf Type Theory, a dependent type theory.

1 Introduction

When implementing a programming language, a proof assistant, or a theorem prover, one often relies on unification algorithms. Dealing with dependent types and/or higher-order logics requires higher-order unification (HOU) algorithms. Many such algorithms are available in the literature, most influential of which are, perhaps, Huet’s preunification [13], Jensen-Pietrzykowski’s full unification [14] procedures, procedures for decidable fragments [24,1,20] and a recent efficient implementation of full HOU [32].

HOU algorithms, such as mentioned above, are specified for a rather minimalistic version of λ -calculus. This is often justified by appealing to higher-order abstract syntax [27] (HOAS): any binding construction can be encoded in λ -calculus. Unfortunately, HOAS and its variants [6,33] are not always comfortable to work with as witnessed by both language implementors [16,7] and formalisation researchers [11].

Thus, supporting higher-order (pre)unification either forces one to use HOAS or to implement a version of a HOU algorithm from scratch for the chosen language. This appears to be one of the main reasons for prototype implementations

to omit or reduce support for type inference and demand more explicit type annotations for the user.

Second-order abstract syntax (SOAS) and second-order equational logic [10] have recently been an attractive alternative to HOAS. It has been successfully used to generate metatheory in Agda [11] and a full E -unification procedure [17] has been developed. Importantly, E -unification for SOAS is powerful enough to encode higher-order unification problems in languages with arbitrary binders.

SOAS is freely generated from a signature, which specifies the syntactic constructions available in the object language, by adding variables and parametrised metavariables. Each syntactic construction can be parametrised by a sequence of (potentially, scoped) subterms. For example, SOAS for simply typed lambda calculus is generated from a family of constructors for all types σ and τ :

$$\text{app}^{\sigma,\tau} : (\sigma \Rightarrow \tau, \sigma) \rightarrow \tau \quad \text{abs}^{\sigma,\tau} : (\sigma.\tau) \rightarrow \sigma \Rightarrow \tau$$

Here, $\text{app}^{\sigma,\tau}$ has two subterms (arguments) of types $\sigma \Rightarrow \tau$ and σ , while $\text{abs}^{\sigma,\tau}$ has a single *scoped* subterm of type τ with access to a local variable of type σ .

Although it should be possible to work with intrinsically typed SOAS as in the example above, in this paper, we consider only untyped SOAS since ultimately we are interested in explicit implementations of type checking and type inference for arbitrary languages, whose type system might not be properly embeddable in the host language. For example, we consider the following SOAS for preterms in $\lambda\Pi$ -calculus (i.e. well-scoped but not necessarily well-typed terms):

$$\text{app} : (T, T) \rightarrow T \quad \text{abs} : (T.T) \rightarrow T \quad \text{Pi} : (T, T.T) \rightarrow T$$

Free monads [30] generate (first-order) abstract syntax trees with a monadic binding operation serving as substitution. Following Swierstra [30], we want to generate SOAS with proper variable substitution and metavariable substitutions from a signature provided by a user-defined algebraic data type (ADT). To do that, we need to be able to specify and properly handle scoped terms.

For expressions with scopes (such as *let*-expressions or λ -abstractions), substitution (implemented manually or via free monads) is not safe by default since a name capture might happen. To avoid this, de Bruijn indices [9] are commonly used in practice. Generalized de Bruijn indices¹ have also been used (e.g. in Epigram [23]) to keep track of scoping in types and also to allow for the lifting entire subexpressions to optimize substitutions further.

Combining free monads with intrinsic scopes via generalised de Bruijn indices we are able to generate abstract syntax with proper substitution operations. For $\lambda\Pi$ -calculus the following ADT describes the signature of preterms:

```
data TermF scope term
= LambdaF scope      --  $\lambda x.T$ 
| AppF term term      --  $T_1 T_2$ 
| PiF term scope      --  $\prod_{x:T_1} T_2(x)$ 
```

¹ such as implemented in the `bound` package, available at <http://hackage.haskell.org/package/bound>

Here the `scope` parameter corresponds to scoped subterms, introducing local variable(s), and `term` corresponds to subterms without extra scope variables. It is possible to consider variations of our approach, supporting arbitrary indexing for bound variables and support for intrinsic typing. However, we find the suggested setting comfortable enough and defer variations for further work.

Since we can see the ADT above as a signature for SOAS, we can generate syntax for the object language (with and without metavariables), and provide higher-order preunification, adapting a version of *E*-unification for SOAS [17].

1.1 Related work

Unification and free monads. In his 2001 pearl [29], Sheard described an efficient and modularized implementation of single-sorted first-order unification. Wren Romano has implemented this approach in Haskell as the `unification-fd` library. Romano’s implementation also mixes well with Swierstra’s data types à la carte [30]: terms with metavariables are constructed using free monads.

Second-order abstract syntax. Fiore and Szamoszvincev [11] have developed a language-formalization framework in Agda. Their approach is based on SOAS and generates Agda code for a grammar of types, operations of weakening and substitution, correctness properties, and other utilities for the formalization of an equational/rewriting theory for a given language.

Makoto Hamana [12] has developed the framework of second-order computation systems and their algebraic semantics, laying out the foundation for the SOL system, a tool to check confluence and termination of polymorphic second-order computation systems. The SOL system is implemented in Haskell and relies on the quasiquotation feature of Template Haskell to specify a second-order signature and computation rules for a second-order computation system.

Whereas the aforementioned works are focused on the metatheory of languages, we are interested more in the implementation of languages, and in particular, type checkers for dependently typed languages.

Intrinsic scoping. Maclaurin, Radul, and Pászke have introduced the Foil [22], making it possible to have intrinsic scoping while maintaining the efficiency benefits of the Barendregt convention, as implemented in the Rapier [26], an approach to handling binders, which is implemented, in particular, in the Glasgow Haskell Compiler. It seems plausible that the Foil can be used instead of nested datatypes to ensure scope safety in the approach presented in this paper, but we leave this research for future work.

1.2 Contributions

We propose an approach to abstract syntax that relies on a combination of free monads and generalized de Bruijn indices. We argue that our approach facilitates the implementation of programming languages and proof assistants², in

² or, at the very least, prototyping of programming languages and proof assistants

particular, of dependently typed ones, by deriving a higher-order preunification algorithm for the object language. Our specific contributions are the following:

1. In Section 2, we introduce *free scoped monads*, a generic data type that serves as a basis for a family of languages with well-scoped terms.
2. In Section 3, we propose an approach to the implementation of term reduction that mixes well with the data types à la carte approach.
3. In Section 4, we formulate the necessary requirements for the signature to enable higher-order preunification of terms in the object language. We adapt an *E*-unification procedure for second-order abstract syntax [17] and extract the preunification component of it.
4. In Section 5, we demonstrate how our approach can be applied to implement type checking and type inference for Martin-Löf Type Theory.

2 Free monads with intrinsic scoping

In this section, we merge the ideas of free monads and intrinsically scoped terms to produce free scoped monads allowing us to generate the type of well-scoped terms with correctly defined substitution. We then add metavariables, generating SOAS from a signature given by an algebraic data type (ADT) in Haskell.

Intrinsically well-scoped de Bruijn terms were introduced by Bellegarde and Hook [4], and monadic structure (substitution) for untyped terms was developed by Bird and Patterson [5]. Some later work has been done for typed terms [3,2], but those use intrinsic typing which we are not using in this paper³.

Skipping intrinsic typing, we are not relying on dependent types in the host language, however our representation of abstract syntax still requires two important type system features. First, we require *nested* (also called *non-uniform* or *non-regular*) data types, whose definition involves a recursive component that is different from the type being defined. Second, we require higher-kinded types in order to parametrise the signature ADT by type constructors. We are using Haskell as our language of implementation, but the reader should be aware of these requirements, if they wish to port the code to another language.

2.1 Intrinsically well-scoped terms

Following Bird and Patterson [5] we start with the following definitions. First, we define a type constructor to extend the type of variables with one more name:

```
data Inc var = Z | S var
```

A scoped term is now a term defined in an extended context (i.e. over the type of variables extended with (bound) variable *Z*):

³ we are not relying on intrinsic typing since we would like to be able to implement languages with richer type systems in weaker or differently typed host languages; for example, we would like to implement Martin-Löf Type Theory in Haskell

```
type Scope term var = term (Inc var)
```

148 Note that `Scope` is a higher-kinded type since its argument `term` is a type
 149 constructor. As long as `term` is a `Monad`, we can perform substitution for the bound
 150 variable:

```
substitute :: Monad term => term a -> Scope term a -> term a
substitute u s = s >>= \x -> case x of
    Z   -> u           -- substitute bound variable
    S y -> return y    -- keep free variable
```

151 Note that intrinsic scoping here makes sure that we only substitute bound
 152 variables, and free variables (as well as the rest of the structure) are left intact.

153 One could use `Scope` directly to define, for example, the type of $\lambda\Pi$ -terms,
 154 parametrized over the type of free variables:

```
data Term a
  = Var a                -- ^ Free variable.
  | App (Term a) (Term a) -- ^ Application.
  | Lam (Scope Term a)    -- ^ Abstraction.
  | Pi (Term a) (Scope Term a) -- ^ Function type.
```

155 Assuming we have `Monad Term` instance, and equipped with `substitute`, it
 156 is straightforward to define evaluation of such terms. For example, this is how
 157 evaluation to weak head normal form (WHNF) can be implemented:

```
whnf :: Term a -> Term a
whnf term = case term of
    App fun arg -> case whnf fun of
        Lam body -> whnf (substitute arg body)
        fun' -> App fun' arg
    _ -> term
```

158 Compared with traditional de Bruijn indices, relying on nested data types
 159 using `Scope` has two great advantages. First, it is safer as ill-scoped terms are
 160 also ill-typed. Second, programming with scopes is now more type-driven and al-
 161 lows for more straightforward implementations (with `substitute` being a prime
 162 example).

163 **Binding multiple variables.** It will be useful to us in Section 2.3 to have a
 164 variation of `Scope` that supports binding of many variables at once:

```
data IncMany var
  = Bound Int -- an Int-indexed bound variable
  | Free var  -- a free variable

-- / A scope with arbitrarily many bound variables.
type IntScope term var = term (IncMany var)
```

Substitution for `IntScope` requires a mapping from a bound variable index to a term, but is otherwise straightforward.

```
substituteMany :: Monad term => (Int -> term var) -> IntScope term var -> Term var
substituteMany f s = s >>= \x ->
  case x of
    Bound n -> f n
    Free z -> return z
```

In this paper, we will use the regular `Scope` for the scopes in the object language and `IntScope` for metavariable substitution.

2.2 Free scoped monads

The use of `substitute` in the definition of `whnf` requires a `Monad` instance for `Term`. Although we could provide it explicitly, we would rather have it for free. One common technique to get it is to reformulate `Term` using free monads [30]. Unfortunately, our `Term` is used non-uniformly in its recursive definition, which is not compatible with standard free monad definitions. So, we introduce the *free scoped monad*:

```
data FS t a
  = Pure a
  | Free (t (Scope (FS t) a) (FS t a))
```

The main idea is that `t` in `FS t a` represents possible syntactic constructions of the language (similarly to generating functor in regular free monads), and it can explicitly mention both subterms and scopes. The free scoped monad is a `Monad` (whenever `t` is a `Bifunctor`), with the bind operation (`>>=`) corresponding to the substitution:

```
instance Bifunctor t => Monad (FS t) where
  return = Pure
  Pure x >>= f = f x
  Free t >>= f = Free (bimap ((>>= traverse f)) (>>= f) t)
```

We can now reformulate our type for untyped lambda terms. First, we define an auxiliary data type for syntactic constructions (sans variables). Then, we use `FS` to give us the type of terms:

```
data TermF scope term
  = AppF term term
  | LamF scope

type Term a = FS TermF a
type ScopedTerm a = Scope (FS TermF) a
```

Note that `Bifunctor` and some other instances can be automatically derived for `TermF` using GHC extensions, such as `DeriveFunctor`, or Template Haskell

utility functions like `deriveBifunctor` from `bifunctors` package. We can also make use of `PatternSynonyms` extensions to define pattern synonyms for all the necessary syntax, so that `whnf` implementation would not require any changes after we switch to free scoped monads:

```
pattern Var x = Pure x
pattern Lam s = Free (LamF s)
pattern App t1 t2 = Free (AppF t1 t2)
```

With free scoped monads, we now have the tools to generate types of well-scoped terms. Although `FS` provides an effective mechanism to automatically get substitution for our terms, the design as presented here has some tradeoffs. First, since we are using `Scope`, we are limiting ourselves to scopes that only introduce one bound variable. This can be improved by using *generalized de Bruijn indices* as implemented in the `bound` package. In this paper, we will use a simplified version for the sake of clarity and brevity. Second, the definition of `Scope` itself can be changed to reduce the number of required traversals of the syntax tree. Again, a more elaborate version, as seen in the `bound` package, can be used instead. Third, we could parametrize `FS` over the scope type constructor, but that would again unnecessarily complicate the code. Finally, we could use a different formulation of `FS`, such as a Church encoding, similar to Church-encoded free monads [31] for improved asymptotic complexity of substitution.

2.3 Metavariables, SOAS, and metavariable substitution

For unification, we need to add metavariables to our syntax. To avoid unnecessary assumptions about the object language while keeping the expressive power of higher-order unification, we follow SOAS [10] and use parametrised metavariables. Instead of embedding metavariables directly into `FS` data type, we use data types à la carte approach [30] and extend any given bifunctor `term` with metavariables. The following datatype generates parametrised metavariables:

```
data MetaAppF v scope term = MetaAppF v [term]
```

Parametrisation provides independence from object language syntax (we do not require to know of function application in the object language), but it also keeps all “dependencies” of a metavariable bundled with it.

Following [17], we write $M_i[t_1, t_2, \dots, t_n]$ to mean application of metavariable M_i to terms t_1, t_2, \dots, t_n . Note that $M_i[t_1][t_2]$ is invalid syntax and it is not possible in general to partially apply a metavariable.

To add metavariables to a language, we use a variant of Swierstra’s operator `(:+:)`. Given signatures `f` and `g`, we can get a new signature that supports constructions from both original signatures:

```
data Sum f g scope term
  = InL (f scope term) -- inject constructions of f
  | InR (g scope term) -- inject constructions of g
```

219 Note that `Sum f g` is a `Bifunctor` whenever `f` and `g` are.

220 Now, we can extend any signature `t` with parametrised metavariables:

```
type SOAS v t a = FS (Sum t (MetaAppF v)) a
```

221 Here, `SOAS` stands for “Unifiable Free Scoped” with `v` being the type of metavariables, `t` — the term signature, and `a` — the type of free variables.

223 **Metavariable substitution.** Following `SOAS` [10], we define substitution for parametrised metavariables by mapping each metavariable to a scoped term, with n bound variables. In the implementation, we rely on `IntScope`, allowing arbitrarily many bound variables (not statically checked):

```
data MetaAbs t a = MetaAbs Int (IntScope (FS t) a)
```

227 Here, the first component of type `Int` represents the arity of the metavariable, which is mostly useful for pretty-printing and debugging, and is not strictly necessary for the unification algorithm. The second component is the scoped term, with (up to) n distinct bound variables used.

231 We represent substitution $M_i[x_1, x_2, \dots, x_n] \mapsto t$ as a pair of metavariable M_i and the scoped term t , represented using `MetaAbs` for the extended language. A simultaneous substitution [10, Section 2] is represented by a list of substitutions⁴:

```
type Subst v t a = (v, MetaAbs (Sum t (MetaAppF v)) a)
newtype Substs v t a = Substs { getSubsts :: [Subst v t a] }
```

234 To apply `Substs` to a term, we merely traverse the term replacing every occurrence of `MetaAppF` that has a corresponding substitution:

```
applySubsts :: (Eq v, Bifunctor t) => Substs v t a -> SOAS v t a -> SOAS v t a
applySubsts substs = go where
  go term = case term of
    Pure{} -> term -- free variables remain
    Free (InR (MetaAppF v args)) -> -- metavariables are replaced according to substs
      -- substitute metavariables in arguments
      let args' = map (applySubsts substs) args
      in case lookup v (getSubsts substs) of
        Just (MetaAbs _arity body) -> substituteMany (args' !!) body
        Nothing -> Free (InR (MetaAppF v args'))
      -- recursively traverse other syntactic constructions
    Free (InL t) -> Free (InL (bimap goScope go t))
  goScope = applySubsts (fmap S substs)
```

236 This concludes the definition of `SOAS` generated from a signature provided in a form of a `Bifunctor` in Haskell:

⁴ here we are using a list for simplicity, but it is also possible to use other data structures, such as `Data.HashMap` or `Data.Map`

- 238 1. **SOAS** $v \ t \ a$ is the type of second-order terms generated from t ;
- 239 2. **Pure** x corresponds to a (free) variable x ;
- 240 3. **Free** (**InR** (**MetaAppF** $v \ [t_1, \dots, t_N]$)) corresponds to $M[t_1, \dots, t_N]$
- 241 4. **Free** (**InL** t) corresponds to some syntactic construction⁵ $F(\overline{x_1}.t_1, \dots, \overline{x_n}.t_n)$;
- 242 5. function **applySubsts** performs simultaneous metavariable substitution.

243 3 Term reduction à la carte

244 In this section, we organise term reduction for extensible languages following
 245 data types à la carte [30]. The motivation is twofold. On the one hand, we want
 246 to be able to specify reduction in object languages without having to deal with
 247 metavariables (indeed, it is natural for the reduction rules to be independent of
 248 metavariables). On the other hand, we want to be able to easily extend languages
 249 with new syntactic constructions. For an example of the latter, imagine extending
 250 a language with pairs and projections. For many languages, it is sufficient to
 251 specify syntax and reduction for pairs and projections in isolation and extend
 252 an arbitrary language with it.

253 At this stage, we do not impose specific restrictions of the semantics of “re-
 254 duction”, except that it should work for arbitrary terms, possibly with free vari-
 255 ables. Later, when dealing with (pre)unification we will require reduction to be
 256 confluent.

257 In general, assuming the constructions from the two signatures are not sup-
 258 posed to “interfere” with each other, we can define term reduction for each com-
 259 ponent independently. Unfortunately, it is not enough to define reduction for
 260 **FS** $f \ a$ and **FS** $g \ a$, and we need to define reduction in a more general setting:

```
class Reducible t where
  reduceL :: Reducible ext => FS (Sum t ext) a -> FS (Sum t ext) a
```

261 Here **reduceL** reduces a term of a language, generated by t extended with
 262 **ext**, assuming terms generated by **ext** are reducible.

263 **Empty signature** In the context of combining languages, a particularly im-
 264 portant language is an empty one:

```
-- this data type has no constructors
data Empty scope term
```

265 Note that language generated by **Empty** is not actually empty: free variables
 266 are always added with **FS**. Still, as variables reduce to themselves, **Empty** gener-
 267 ates a language with reducible terms:

```
instance Reducible Empty where
  reduceL (Pure x)      = Pure x
  reduceL (Free (InL e)) = case e of {}
  reduceL t              = reduceR t
```

⁵ Fiore and Hur call these *operators* [10, Section 2]

268 Extending a language with **Empty** yields essentially the same language, the
 269 only difference is in its representation in Haskell. Thus term reduction in an
 270 object language can be expressed as a special case of reduction in an extended
 271 language, where the extension is empty:

```
reduce :: Reducible t => FS t a -> FS t a
reduce = trans removeEmpty . reduceL . trans InL
  where removeEmpty (InL x) = x
        removeEmpty (InR e) = case e of {}
```

272 **Sum of signatures** Since **Sum** is symmetric, we can automatically get reduction
 273 for **InR** part, once we have one for **InL**:

```
reduceR = trans commute . reduceL . trans commute
  where commute (InL x) = InR x
        commute (InR y) = InL y
```

274 Combining two reducible languages with **Sum** yields a reducible language:

```
instance (Reducible f, Reducible g) => Reducible (Sum f g) where
  reduceL (Pure x) = Pure x
  reduceL t@(Free InL{}) = reduceL t
  reduceL t@(Free InR{}) = reduceR t
```

275 **Reducing λ -terms** To adapt **whnf** to reduce terms in a language gener-
 276 ated by **TermF** with arbitrary extension, we can introduce patterns for extended
 277 language:

```
pattern LamE body = Free (InL (LamF body))
pattern AppE t1 t2 = Free (InL (AppF t1 t2))
pattern ExtE t = Free (InR t)
```

278 Implementation of **whnf** is transferred almost letter for letter, with an im-
 279 portant addition being the case when the root node belongs to the extension —
 280 here, we delegate reduction to the extension by using **reduceR**:

```
instance Reducible TermF where
  reduceL term = case term of
    ExtE{} -> reduceR term -- handle extension
    AppE fun arg -> case reduceL fun of
      LamE body -> reduceL (substitute arg body)
      fun' -> AppE fun' arg
    _ -> term
```

281 **Reducing with metavariables** Parametrised metavariables reduce to them-
 282 selves, however we can choose to reduce or keep their parameters, yielding two
 283 possible definitions. First one leaves parameters unevaluated:

```

-- "lazy" reduction (arguments remain unevaluated)
instance Reducible (MetaAppF v) where
  reduceL (ExtE t) = reduceR (ExtE t)
  reduceL t = t

```

284 The second possible implementation reduces the parameters:

```

-- "strict" reduction (arguments are evaluated)
instance Reducible (MetaAppF v) where
  reduceL (ExtE t) = reduceR (ExtE t)
  reduceL (InL (MetaAppF m args)) = InL (MetaAppF m (map reduceL args))

```

285 Note that even though the second implementation is “strict” in the object
 286 language, using Haskell as a host language makes evaluation somewhat lazy in
 287 the sense that actual evaluation of parameters might still be delayed. This kind
 288 of lazy evaluation is used in some HOU algorithms [32] and from now we assume
 289 the second instance implementation for `MetaAppF`.

290 4 Higher-order unification

291 In this section, we describe a generic semi-decidable algorithm for single-sorted
 292 higher-order preunification. The algorithm stops when either the terms cannot
 293 be unified, or when the only constraints left are those between metavariables.

294 The algorithm is loosely based on *E*-unification for second-order abstract
 295 syntax [17], with the following important differences:

- 296 1. we forego the **(mutate)** rule [17, Definition 28], and instead assume that
 297 terms can be normalised (via `reduce`);
- 298 2. we combine the **(imitate)** and **(project)** rules [17, Definitions 24–25] into
 299 a single rule with generalised Huet-style bindings [18];
- 300 3. we only implement preunification, leaving unsolved constraints between two
 301 metavariables, so we are not using **(eliminate)**, **(identify)**, and **(iterate)**
 302 rules;
- 303 4. we implement *unification* itself in an untyped setting, i.e. our implemen-
 304 tation of higher-order unification does not (directly) take types of terms
 305 into account; technically, type information can be embedded into the terms
 306 themselves and can be used by `reduce`, but in this paper we do not make
 307 extra assumptions when generating Huet-style bindings and leave develop-
 308 ment of an algorithm for type-directed generalised Huet-style bindings for
 309 future work.

310 To achieve such an algorithm, we impose some constraints on the signature
 311 ADT. These constraints, formulated as type classes in Haskell, make sure that
 312 we can traverse the freely generated syntax tree, match individual nodes of that
 313 tree (enabling first-order unification), and make appropriate substitutions for
 314 metavariables (enabling higher-order unification).

315 4.1 Prerequisites

316 The unification process involves keeping track of metavariables and their values,
 317 which is a kind of effectful computation. Traversing an abstract syntax tree
 318 and making changes to the currently known values of metavariables requires
 319 not just **Bifunctor**, but **Biraversable** instance for the generating bifunctor.
 320 Fortunately, for most practical cases, we can derive those instances automatically
 321 with common GHC extensions or Template Haskell.

322 Apart from **Bitraversable**, we will require object language terms to be
 323 **Reducible**, and its syntactic constructions **Unifiable**. For first-order unifica-
 324 tion, it would be enough to match individual syntactic constructions and perform
 325 unification by recursive matching.

326 For higher-order unification, we will require generalised Huet-style bind-
 327 ings [18]. Essentially, for a given language we need to know the following in-
 328 formation:

- 329 1. For each argument (subterm) of a syntactic construction, is there a certain
 330 shape of a term that allows further reduction? For example, given a term
 331 $\pi_1 M_1[]$ (where $M_1[]$ is a metavariable), we should understand that M_1 can be
 332 substituted by a tuple (M_2, M_3) (where M_2 and M_3 are fresh metavariables)
 333 to allow further reduction.
- 334 2. For parametrised metavariables, what are possible ways to construct a term
 335 that will use the parameters via reduction? For example, when unifying
 336 $M_1[(t_1, t_2)]$ with t_1 , we should be able to try the substitution $M_1[x] :=$
 337 $\pi_1 M_2[x]$ to find the solution $M_1[x] := \pi_1 x$.

338 **First-order unification** For first-order unification, we need to be able to match
 339 individual nodes of the syntax tree. Similarly to Wren Romano’s **unification-fd**
 340 package⁶, we define a typeclass with a single method:

```
class Unifiable t where
  zipMatch :: t scope term -> t scope term -> Maybe (t (scope, scope) (term, term))
```

341 The method **zipMatch** takes two nodes as inputs and returns **Nothing** when
 342 they do not match. Otherwise, it returns a single node with subterms and sub-
 343 scopes paired. For example, matching **Lam** t with **Lam** u yields **Just** (**Lam** (t , u)),
 344 suggesting that the unification process should now proceed by going inside the
 345 lambda and attempting to unify t and u .

```
instance Unifiable TermF where
  zipMatch (AppF f1 x1) (AppF f2 x2) = Just (AppF (f1, f2) (x1, x2))
  zipMatch (LamF body1) (LamF body2) = Just (LamF (body1, body2))
  zipMatch _ _ = Nothing
```

⁶ In **unification-fd** and our implementation the type of **zipMatch** is a little more complicated to allow for an optimization, when one of the nodes omits a subterm and we can immediately take the necessary value from the second node. However, we decided to simplify the type here to increase readability of this paper.

346 Implementing **Unifiable** is usually mechanical and can in fact be fully au-
 347 tomated using GHC’s Generics (as is done in **unification-fd**) or Template
 348 Haskell.

349 **Higher-order unification** Note that, following SOAS [10], we do not require
 350 the signature to have lambda abstractions or applications as there is more than
 351 one way the user might want to introduce those. For example, application can
 352 be defined as binary or taking a list of arguments. Moreover, some theories
 353 have not one but several syntactic abstractions or applications (such as Π -types
 354 and extensions types in Riehl and Shulman’s type theory with shapes [28], or
 355 μ -abstraction in Parigot’s $\lambda\mu$ -calculus [25]).

356 So instead of forcing syntax onto the user, we instead ask them to provide a
 357 basic mechanism for generating valid structural guesses (generalised Huet-style
 358 bindings) for metavariables:

```
data IsHead = HasHead | NoHead

class Unifiable t => HigherOrderUnifiable t where
  guessMetas :: t scope term -> t (scope, [t () ()]) (term, [t () ()])
  shapes :: [t IsHead IsHead]
```

359 The role of **guessMetas** is to provide a list of valid partial guesses for each
 360 subterm and subscope in a given node of the syntax tree. For example, given a
 361 term $M_1 M_2$ where M_1 and M_2 are metavariables we can guess that M_1 is a function
 362 and so should be unified with a term λM_3 , where M_3 is a fresh metavariable. On
 363 the other hand, we do not have any information that would allow us to guess the
 364 structure of M_2 . Each returned guess for a particular subterm or subscope has
 365 type **t () ()**, which simply provides the general shape of the guess (e.g., that
 366 it should be a lambda abstraction). The unification algorithm will then replace
 367 each **()** in a guess with a fresh metavariable and continue the unification process.

368 Implementing this for **TermF** we get the following:

```
instance HigherOrderUnifiable TermF where
  guessMetas term = case term of
    AppF f arg -> AppF (f, [LamF ()]) (arg, [])
    _ -> bimap (,[]) (,[]) term

  shapes = [AppF HasHead NoHead]
```

369 Note that the type of **guessMetas** implies that a guess is based on a single
 370 syntactic construction (i.e. it cannot match a complex pattern). It also yields just
 371 one syntactic construction per guess (with placeholders for fresh metavariables).

372 For many type theories, one only needs to identify introduction-elimination
 373 pairs to implement **HigherOrderUnifiable**. Given an instance of **Reducible**,
 374 one can go over all possible combinations of syntactic constructions to figure out
 375 this information automatically, either using Template Haskell or GHC Generics.

4.2 Constraints

A constraint is essentially a pair of terms with metavariables that we would like to unify. However, upon entering the scope, the type of terms changes as it is extended with bound variables. Consider constraint involving λ -abstraction: $(\lambda x.M_1[x]) \equiv (\lambda z.z)$. Going under λ -abstraction in both terms might reduce the original constraint to $M_1[x] \equiv x$. Unfortunately, this transition is not faithful as the new constraint can be satisfied with two different substitutions: $M_1[z] \mapsto x$ and $M_1[z] \mapsto z$. The former contains x , which is not free in the original constraint, so is not a valid solution. To avoid leaking bound variables, some implementations introduce fresh bound variable when going into a scope, and have explicit checks to avoid using them in substitutions.

Since bound variables are not allowed to leak into solutions for unification problem, an appropriate simplification of the original constraint $(\lambda x.M_1[x]) \equiv (\lambda x.x)$ should look like $\forall x.(M_1[x] \equiv x)$. Here, x remains bound and is easy to avoid when generating substitutions for M_1 . Fortunately, we can leverage `Scope` to manage \forall quantifier and represent constraints properly:

```
data Constraint v t a
  = SOAS v t a :=: SOAS v t a
  | ForAll (Scope (Constraint v t) a)
```

The infix constructor `(:=:)` is used to construct a simple constraint with two terms. The constructor `ForAll` uses `Scope` to construct a scoped constraint. This representation does not solve the problem of leaking bound variables completely on its own, but it makes the compiler reject implementations that do not account for bound variables, as those substitutions will be impossible to lift outside of scopes.

4.3 Preunification algorithm

In this section, we describe an algorithm for single-sorted preunification algorithm. The algorithm relies on a backtracking-capable environment and the ability to generate fresh metavariables. In Haskell, we manage those capabilities via type classes `MonadPlus` and `MonadFresh`:

```
class Monad m => MonadFresh v m | m -> v where
  freshMeta :: m v
```

The main idea of the algorithm is straightforward:

1. starting with a collection of constraints,
2. attempt to simplify them into smaller constraints by using term reduction and structural guesses for metavariables, producing some flex-flex and flex-rigid constraints;
3. then take any flex-rigid constraint that could not be simplified further and try to solve it;
4. if cannot be solved — backtrack; otherwise — apply solution (substitution) to the rest of the constraints and
5. repeat until all flex-rigid constraints are resolved.

413 **Simplifying constraints** Simplification of a single constraint consists of three
 414 steps:

- 415 1. Terms are reduced using `reduce`.
- 416 2. Each term is traversed to see if any metavariables can be substituted using
 417 one of the structural guesses using `guessMetas`. If there are any potential
 418 substitutions, we apply them to both terms and repeat from step 1.
- 419 3. Finally, we `zipMatch` the two terms to break down constraint into a collection
 420 of smaller constraints.

421 Given a collection of constraints, we perform simplification on each of them
 422 recursively, accumulating and applying generated substitutions from `guessMeta`,
 423 until we end up with a collection of constraints that cannot be simplified any
 424 further.

425 Simplified constraints are expected to be

- 426 – of the form $\forall y_1 \dots y_m. (M_i[t_1, \dots, t_n] \equiv t)$, where t is not a metavariable
 427 application; these are called *flex-rigid* constraints;
- 428 – or of the form $\forall y_1 \dots y_m. (M_i[t_1^i, \dots, t_n^i] \equiv M_j[t_1^j, \dots, t_k^j])$; these are called
 429 *flex-flex* constraints.

430 The third potential type of constraints, where both sides are not metavariable
 431 applications, are called *rigid-rigid* constraints. These constraints are guar-
 432 anteed to be simplified in step 3 with `zipMatch`. Indeed, `zipMatch` either returns
 433 `Nothing` (which means that two nodes do not match), or it pairs syntactic sub-
 434 trees to match recursively, ensuring structural recursion.

435 **Extracting head of a term** If, according to `guessMetas`, there is a structural
 436 guess for some subterm position of a syntactic construction, we call a subterm
 437 in that position a *head subterm*. We say that a term h is a *head* of a term t if it
 438 is a head subterm of t or if it is a head of any head subterm of t . For example,
 439 the term $\lambda z. fz$ is the head of the term $\pi_1 ((\lambda z. fz) x (\pi_2 y))$.

440 **Solving flex-rigid constraints** Preunification algorithm starts off with a list
 441 of constraints, reduces *rigid-rigid* constraints and solves *flex-rigid* constraints,
 442 leaving only *flex-flex* constraints to be dealt with by the user.

443 To solve a flex-rigid constraint $\forall y_1 \dots y_m. (M_i[t_1, \dots, t_n] \equiv t)$, the algorithm
 444 goes through a sequence of candidate solutions. Each candidate solution is of
 445 the form $M_i[x_1, \dots, x_n] \mapsto T$, where T is one of the following:

- 446 – the head of t , where each variable bound by \forall is replaced with a fresh
 447 metavariable application $M_k[x_1, \dots, x_n]$;
- 448 – a bound variable of M_i : x_j ;
- 449 – a candidate shape (one of `shapes`), where each `HasHead` position is filled
 450 with T' and `NoHead` position is filled with a fresh metavariable application
 451 $M_k[x_1, \dots, x_n]$.

452 The entire algorithm is packed into a single function with the following type
 453 signature:

```
unify
  :: ( HigherOrderUnifiable t, Reducible t
      , MonadPlus m, MonadLogic m, MonadFresh v m
      , Eq a, Eq v )
  => Substs v t a
  -> [Constraint v t a]
  -> m ([Constraint v t a], Substs v t a)
```

454 5 Applications

455 In this section, we see the application of our approach to implementation of type
 456 inference for a couple of type theories. The implementation is available as part
 457 of version 0.1.0 of the proof assistant RZK⁷ and contains the following relevant
 458 modules:

- 459 1. `module Rzk.Free.Syntax.FreeScoped` introduces the free scoped monads,
 460 `Sum`, and utility functions as described in Section 2;
- 461 2. `module Rzk.Free.Syntax.FreeScoped.Unification2` implements higher-
 462 order preunification as described in Section 4;
- 463 3. `module Rzk.Free.Syntax.FreeScoped.TypeCheck` implements type check-
 464 ing and type inference algorithms based on higher-order preunification;
- 465 4. `module Rzk.Free.Syntax.Example.ULC` implements untyped λ -calculus with
 466 higher-order unification;
- 467 5. `module Rzk.Free.Syntax.Example.STLC` implements a version of simply
 468 typed λ -calculus (STLC) with type inference via higher-order unification;
 469 this version differs from the standard STLC by allowing computation at the
 470 type level;
- 471 6. `module Rzk.Free.Syntax.Example.MLTT` contains the implementation of
 472 intensional Martin-Löf dependent type theory with type inference.

473 The type inference algorithm follows the general structure of constraint-based
 474 typechecking, where higher-order preunification is used to resolve constraints.
 475 For the typechecking preunification usually suffices, since flex-flex constraints
 476 correspond to ambiguous typing which normally is considered a type error. The
 477 details of type inference algorithm can be found in Appendix A.

478 We now outline the key moments in the implementation of Martin-Löf type
 479 theory, more details on this and the implementation of simply typed λ -calculus
 480 can be found in Appendix B and in the corresponding implementation files.

⁷ see relevant modules in <https://github.com/rzk-lang/rzk/tree/v0.1.0/rzk/src/Rzk/Free>

481 5.1 Typed terms

482 Many implementors define a single type in the host language for both terms and
 483 types in the object language [21,8]. This means that typing is treated as a rela-
 484 tion between a term and another term. We take a similar approach, annotating
 485 every node in the syntax tree with another term, which represents the type and
 486 has annotations of its own. To achieve that, we extend the object language by
 487 modifying the generating bifunctor:

```
-- | Extending a type of types with universe.
data WithUniverse ty = BigUniverse | SomeType ty

data TyF t scope term = TyF
  { termF :: t scope term
  , typeF :: WithUniverse term
  }

-- | A typed term generated from t.
type TFS t a = FS (TyF t) a
```

488 We use the type `WithUniverse (TFS t a)` for type annotations, meaning
 489 that type terms themselves have type annotations. The recursive annotation
 490 stops either at variables, or at `BigUniverse`, which is an explicit universe type
 491 \mathcal{U}_∞ . Consider term $\lambda x.f x$. Adding type annotations (written $t : T$) according to
 492 `TyF` would produce the following typed term (here we assume the object language
 493 also has its own universe type \mathcal{U} , and f, A, B are free variables):

$$\begin{array}{ll} \lambda x.f x & \text{(untyped term)} \\ (\lambda x.(f x : B)) : (A \rightarrow B : (\mathcal{U} : \mathcal{U}_\infty)) & \text{(typed term)} \end{array}$$

494 Since we have modified the type of nodes in the syntax tree, with `TFS t a`,
 495 we have type annotation *for every subterm except variables*. This makes it easy
 496 to extract types of subterms when necessary without the need to repeatedly infer
 497 types.

498 5.2 Typing syntactic constructions

499 To perform type inference for any given language, it is enough to know how
 500 to perform a single step: given types of parts for single syntactic construction,
 501 compute the type of the whole. An important implementation detail is to provide
 502 not just the types of the parts, but an actual computation context for that type.
 503 In other words, instead of `TFS v t a` we will have `m (TFS v t a)` where `m` is
 504 some typechecking monad. This is done to give the implementor of a particular
 505 language more control over typechecking and constraint resolution:

```
class Inferable t where
  inferF :: MonadTypecheck v t a m
    => t (m (Scope (TFS v t) a)) (m ((TFS v t a) a))
    -> m (t (Scope (TFS v t) a) ((TFS v t a) a))
```

506 Once we know how to perform a single step of type inference, all we need to
 507 do is traverse the entire term:

```
infer :: (Inferable t, MonadTypecheck v t a m) => FS t a -> m (TFS v t a)
infer term = case term of
  Var x -> do
    addKnownFreeVar x
    return (Var x)
  Free t -> do
    ty <- Free <$> inferTypeFor (bimap inferScope infer t)
    clarifyTypedTerm ty
```

508 Here, `addKnownFreeVar` adds the free variable to the `TypeInfo` state with a
 509 fresh type meta variable, if it is the first time this variable is encountered. As per-
 510 forming inference for a single syntactic construction may result in new meta vari-
 511 able substitutions, we need to apply them across known type information and,
 512 perhaps, simplify the inferred typed term. For that we use `clarifyTypedTerm`,
 513 which has a straightforward implementation that we omit here.

514 5.3 Martin-Löf Type Theory

515 Let us now apply the approach to an actual dependent type theory — intensional
 516 Martin-Löf Type Theory (MLTT). We start with a generating bifunctor:

```
data TermF scope term
  = UniverseF          -- Universe type:  $\mathcal{U}$ 
  | PiF term scope      -- Dependent product  $\Pi_{x:T_1} T_2$ 
  | LamF scope          -- Abstraction:  $\lambda x. T_2$ 
  | AppF term term      -- Application:  $(T_1 T_2)$ 
  | SigmaF term scope   -- Dependent sum  $\Sigma_{x:T_1} T_2$ 
  | PairF term term     -- Pair:  $\langle T_1, T_2 \rangle$ 
  | FirstF term         -- First projection:  $\pi_1 T$ 
  | SecondF term        -- Second projection:  $\pi_2 T$ 
  | IdTypeF term term   -- Identity type:  $x = y$ 
  | ReflF term          -- Reflexivity:  $\text{refl}_T$ 
  | JF term term term term term term
    --  $\sim$  Identity type eliminator:  $J(A, a, C, d, x, p)$ 

-- / An MLTT term with free variables in a.
type Term a = FS TermF a
```

517 We note a couple of details about this particular presentation of MLTT:

- 518 1. We omit type annotations for the bound variable of λ -abstraction.
- 519 2. Both types and terms are generated with `TermF`.

520 In this particular implementation we use a single universe type and assume
 521 type-in-type: $\mathcal{U} : \mathcal{U}$. It is possible to introduce a hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 :$
 522 $\mathcal{U}_2 : \dots$ instead by using `UniverseF Natural` constructor.

523 Next step is to introduce helpful pattern synonyms. We will immediately
 524 work with typed terms, so we only create patterns for those. We remind that
 525 these can be automatically generated using Template Haskell:

```
pattern Typed ty t = Free (InL (TyF t ty))
pattern UniverseT ty = Typed ty UniverseF
pattern PiT ty t1 t2 = Typed ty (PiF t1 t2)
pattern LamT ty body = Typed ty (LamF body)
pattern AppT ty t1 t2 = Typed ty (AppF t1 t2)
...
pattern JT ty t1 t2 t3 t3 t4 t5 t6 = Typed ty (JF t1 t2 t3 t4 t5 t6)
```

526 Implementing WHNF reduction for MLTT is straightforward, we will focus
 527 here only on the case of J-eliminator:

```
instance Reducible TermF where
  reduceL term = case term of
    JT ty tA a tC d x p ->
      case reduceL p of
        ReflT{} -> reduceL d
        p' -> JT ty tA a tC d x p'
  ...
```

528 For **Unifiable** and **HigherOrderUnifiable** we also rely on a mechanical
 529 or automatic derivation and so omit it here to save space. Finally, we define
 530 inference for individual syntactic constructions:

```
instance Inferable TermF
  inferF term = case term of
```

531 To avoid infinite type annotations, we set the type of universe to be \mathcal{U}_∞ :

```
UniverseF -> pure (TyF UniverseF BigUniverse)
```

532 Inferring types for Π -types and Σ -types involves dependent type checking.
 533 Given term $\Pi_{x:A} B$, where B is a subterm that may refer to x , we have to check
 534 that both $A : \mathcal{U}$ and $B : \mathcal{U}$. Note that since B is in the scope, its inferred type,
 535 by default, might also be dependent on x . For example, in the term $\Pi_{x:A} \text{refl}_x$
 536 the algorithm would infer that refl_x has type $x = x$, which captures the variable
 537 x . To make sure the body of a Π -type is always a type, we need to unify it with
 538 \mathcal{U} . But for that we also need to make sure it is not dependent, so we use **nonDep**:

```
PiF inferA inferB -> do
  a <- inferA >=> shouldHaveType (UniverseT BigUniverse)
  typeOfA <- typeOf a
  b <- inScope typeOfA inferB
  typeOfB <- typeOfScope typeOfA b >=> nonDep
  typeOfB `shouldHaveType` UniverseT BigUniverse
  pure (TyF (PiF a b) (UniverseT BigUniverse))
```

539 Inferring the type for a dependent λ -abstraction is relatively straightforward.
 540 We generate a fresh type meta variable for the argument and infer the type of
 541 the body. In general, we should check that the inferred type is indeed a type, as
 542 many type theories, such as cubical type theory, have multiple universes. That
 543 said, in pure MLTT we can omit this check.

```
LamF inferBody -> do
  a <- freshTypeMetaVar
  typedBody <- inScope a inferBody
  b <- typeOfScope a typedBody
  typeOfScope a b >=> nonDep
  >=> shouldHaveType (UniverseT BigUniverse)
  pure $ TyF
    (LamF typedBody)
    (SomeType (PiT (UniverseT BigUniverse) a b))
```

544 The rest of syntactic constructors are fairly straightforward to handle in
 545 a similar fashion. Completing `Inferable` brings dependent type inference to
 546 MLTT.

547 6 Conclusion

548 We have presented an approach to abstract syntax representation with free
 549 scoped monads and demonstrated its effectiveness simply typed lambda calcu-
 550 lus and Martin-Löf Type Theory. These examples show that the approach does
 551 not require the user to have a deep understanding of higher-order unification to
 552 enable type inference for their language.

553 We have also devised a few directions for future work. First, we would like to
 554 extend to full higher-order unification or, better yet, full E -unification for second-
 555 order abstract syntax [17]. Implementing generic E -unification for second-order
 556 abstract syntax would be instrumental to implementing proof assistants for type
 557 theories with non-trivial or extensible definitional equalities. In particular, this
 558 might be useful for the implementation of extension types in Riehl and Shulman's
 559 type theory for synthetic ∞ -categories [28].

560 Second, we should make higher-order unification more efficient by optimizing
 561 the representation of free scoped monads, taking into account the types of unified
 562 terms, and recognising efficient/decidable fragments of unification problems with
 563 oracles as in the work of Vukmirovic, Bentkamp, and Nummelin [32].

564 **Acknowledgements** I am grateful to Benedikt Ahrens and Daniel de Carvalho
 565 for their invaluable feedback throughout my early work towards the implementa-
 566 tion of typecheckers for dependently typed languages. I thank Oksana Zhirosh,
 567 Ruslan Saduov, and Benedikt Ahrens for proofreading earlier versions of this
 568 paper.

References

1. Abel, A., Pientka, B.: Higher-order dynamic pattern unification for dependent types and records. In: Proceedings of the 10th International Conference on Typed Lambda Calculi and Applications. p. 10–26. TLCA’11, Springer-Verlag, Berlin, Heidelberg (2011)
2. Allais, G., Atkey, R., Chapman, J., McBride, C., McKinna, J.: A type and scope safe universe of syntaxes with binding: Their semantics and proofs. *Proc. ACM Program. Lang.* **2**(ICFP) (Jul 2018). <https://doi.org/10.1145/3236785>, <https://doi.org/10.1145/3236785>
3. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Flum, J., Rodriguez-Artalejo, M. (eds.) *Computer Science Logic*. pp. 453–468. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
4. Bellegarde, F., Hook, J.: Substitution: A formal methods case study using monads and transformations. *Sci. Comput. Program.* **23**(2–3), 287–311 (Dec 1994). [https://doi.org/10.1016/0167-6423\(94\)00022-0](https://doi.org/10.1016/0167-6423(94)00022-0), [https://doi.org/10.1016/0167-6423\(94\)00022-0](https://doi.org/10.1016/0167-6423(94)00022-0)
5. Bird, R.S., Paterson, R.: De bruijn notation as a nested datatype. *J. Funct. Program.* **9**(1), 77–91 (Jan 1999). <https://doi.org/10.1017/S0956796899003366>, <https://doi.org/10.1017/S0956796899003366>
6. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.* **43**(9), 143–156 (sep 2008). <https://doi.org/10.1145/1411203.1411226>, <https://doi.org/10.1145/1411203.1411226>
7. Cockx, J.: 1001 representations of syntax with binding. <https://jesper.sikanda.be/posts/1001-syntax-representations.html> (Nov 2021), accessed: 2023-01-21
8. Coquand, T., Kinoshita, Y., Nordstrom, B., Takeyama, M.: A simple type-theoretic language: Mini-tt. From Semantics to Computer Science: Essays in Honour of Gilles Kahn (01 2009). <https://doi.org/10.1017/CB09780511770524.007>
9. de Bruijn, N.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)* **75**(5), 381–392 (1972). [https://doi.org/10.1016/1385-7258\(72\)90034-0](https://doi.org/10.1016/1385-7258(72)90034-0)
10. Fiore, M., Hur, C.: Second-Order Equational Logic (Extended Abstract). In: Dawar, A., Veith, H. (eds.) *Computer Science Logic, 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23–27, 2010. Proceedings. Lecture Notes in Computer Science*, vol. 6247, pp. 320–335. Springer (2010). https://doi.org/10.1007/978-3-642-15205-4_26, https://doi.org/10.1007/978-3-642-15205-4_26
11. Fiore, M., Szamozvancev, D.: Formal Metatheory of Second-Order Abstract Syntax. *Proc. ACM Program. Lang.* **6**(POPL) (jan 2022). <https://doi.org/10.1145/3498715>, <https://doi.org/10.1145/3498715>
12. Hamana, M.: Theory and practice of second-order rewriting: Foundation, evolution, and SOL. In: Nakano, K., Sagonas, K. (eds.) *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14–16, 2020, Proceedings. Lecture Notes in Computer Science*, vol. 12073, pp. 3–9. Springer (2020). https://doi.org/10.1007/978-3-030-59025-3_1, https://doi.org/10.1007/978-3-030-59025-3_1
13. Huet, G.: A unification algorithm for typed λ -calculus. *Theoretical Computer Science* **1**(1), 27–57 (1975). [https://doi.org/10.1016/0304-3975\(75\)90011-0](https://doi.org/10.1016/0304-3975(75)90011-0)

- 619 14. Jensen, D.C., Pietrzykowski, T.: Mechanizing ω -order type theory through uni-
620 fication. *Theor. Comput. Sci.* **3**(2), 123–171 (1976). [https://doi.org/10.1016/](https://doi.org/10.1016/0304-3975(76)90021-9)
621 [0304-3975\(76\)90021-9](https://doi.org/10.1016/0304-3975(76)90021-9), [https://doi.org/10.1016/0304-3975\(76\)90021-9](https://doi.org/10.1016/0304-3975(76)90021-9)
- 622 15. Kiselyov, O., Shan, C.c., Friedman, D.P., Sabry, A.: Backtracking, interleaving,
623 and terminating monad transformers: (functional pearl). *SIGPLAN Not.* **40**(9),
624 192–203 (sep 2005). <https://doi.org/10.1145/1090189.1086390>, [https://doi.](https://doi.org/10.1145/1090189.1086390)
625 [org/10.1145/1090189.1086390](https://doi.org/10.1145/1090189.1086390)
- 626 16. Kmett, E.: Bound. <https://www.schoolofhaskell.com/user/edwardk/bound> (Dec
627 2015), accessed: 2023-01-21
- 628 17. Kudasov, N.: E-Unification for Second-Order Abstract Syntax. In: Gaboardi, M.,
629 van Raamsdonk, F. (eds.) 8th International Conference on Formal Structures for
630 Computation and Deduction (FSCD 2023). *Leibniz International Proceedings in*
631 *Informatics (LIPIcs)*, vol. 260, pp. 10:1–10:22. Schloss Dagstuhl – Leibniz-Zentrum
632 für Informatik, Dagstuhl, Germany (2023). [https://doi.org/10.4230/LIPIcs.](https://doi.org/10.4230/LIPIcs.FSCD.2023.10)
633 [FSCD.2023.10](https://doi.org/10.4230/LIPIcs.FSCD.2023.10), <https://drops.dagstuhl.de/opus/volltexte/2023/17994>
- 634 18. Kudasov, N.: Generalising huet-style projections in e-unification for second-order
635 abstract syntax. In: UNIF 2023-37th International Workshop on Unification (2023)
- 636 19. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In:
637 *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of*
638 *Programming Languages*. p. 333–343. POPL '95, Association for Computing Mach-
639 inery, New York, NY, USA (1995). <https://doi.org/10.1145/199448.199528>,
640 <https://doi.org/10.1145/199448.199528>
- 641 20. Libal, T., Miller, D.: Functions-as-Constructors Higher-Order Unification. In:
642 Kesner, D., Pientka, B. (eds.) 1st International Conference on Formal Structures
643 for Computation and Deduction (FSCD 2016). *Leibniz International Proceedings*
644 *in Informatics (LIPIcs)*, vol. 52, pp. 26:1–26:17. Schloss Dagstuhl–Leibniz-Zentrum
645 fuer Informatik, Dagstuhl, Germany (2016). [https://doi.org/10.4230/LIPIcs.](https://doi.org/10.4230/LIPIcs.FSCD.2016.26)
646 [FSCD.2016.26](https://doi.org/10.4230/LIPIcs.FSCD.2016.26), <http://drops.dagstuhl.de/opus/volltexte/2016/5981>
- 647 21. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently
648 typed lambda calculus. *Fundamenta Informaticae* **102**, 177–207 (01 2010). [https:](https://doi.org/10.3233/FI-2010-304)
649 [//doi.org/10.3233/FI-2010-304](https://doi.org/10.3233/FI-2010-304)
- 650 22. Maclaurin, D., Radul, A., Paszke, A.: The foil: Capture-avoiding substitution with
651 no sharp edges. In: *Proceedings of the 34th Symposium on Implementation and*
652 *Application of Functional Languages*. IFL '22, Association for Computing Machin-
653 ery, New York, NY, USA (2023). <https://doi.org/10.1145/3587216.3587224>,
654 <https://doi.org/10.1145/3587216.3587224>
- 655 23. McBride, C.: Epigram: Practical programming with dependent types. In: *Pro-*
656 *ceedings of the 5th International Conference on Advanced Functional Program-*
657 *ming*. p. 130–170. AFP'04, Springer-Verlag, Berlin, Heidelberg (2004). [https:](https://doi.org/10.1007/11546382_3)
658 [//doi.org/10.1007/11546382_3](https://doi.org/10.1007/11546382_3), https://doi.org/10.1007/11546382_3
- 659 24. Miller, D.: A logic programming language with lambda-abstraction, function vari-
660 ables, and simple unification. In: Schroeder-Heister, P. (ed.) *Extensions of Logic*
661 *Programming*. pp. 253–281. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
- 662 25. Parigot, M.: $\lambda\mu$ -calculus: An algorithmic interpretation of classical natural deduc-
663 tion. In: Voronkov, A. (ed.) *Logic Programming and Automated Reasoning*. pp.
664 190–201. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
- 665 26. Peyton Jones, S., Marlow, S.: Secrets of the glasgow haskell com-
666 piler inliner. *Journal of Functional Programming* **12**, 393–434 (July
667 2002), [https://www.microsoft.com/en-us/research/publication/](https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/)
668 [secrets-of-the-glasgow-haskell-compiler-inliner/](https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/)

27. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Wexelblat, R.L. (ed.) Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988. pp. 199–208. ACM (1988). <https://doi.org/10.1145/53990.54010>, <https://doi.org/10.1145/53990.54010>
28. Riehl, E., Shulman, M.: A type theory for synthetic ∞ -categories. Higher Structures **1** (2017)
29. Sheard, T.: Generic unification via two-level types and parameterized modules - functional pearl. Sigplan Notices - SIGPLAN **36**, 86–97 (10 2001). <https://doi.org/10.1145/507546.507648>
30. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (2008). <https://doi.org/10.1017/S0956796808006758>
31. Voigtländer, J.: Asymptotic improvement of computations over free monads. pp. 388–403 (07 2008). https://doi.org/10.1007/978-3-540-70594-9_20
32. Vukmirovic, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. Log. Methods Comput. Sci. **17**(4) (2021). [https://doi.org/10.46298/lmcs-17\(4:18\)2021](https://doi.org/10.46298/lmcs-17(4:18)2021), [https://doi.org/10.46298/lmcs-17\(4:18\)2021](https://doi.org/10.46298/lmcs-17(4:18)2021)
33. Washburn, G., Weirich, S.: Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. J. Funct. Program. **18**(1), 87–140 (2008). <https://doi.org/10.1017/S0956796807006557>, <https://doi.org/10.1017/S0956796807006557>

A Type inference

In this section, we describe a generic type inference algorithm for languages generated using free scoped monads. As we follow a common bottom-up constraint based type inference approach, similar to existing implementations, we do not go into all the details, and instead point out the most significant definitions and aspects.

Typed terms

Many implementors define a single type in the host language for both terms and types in the object language [21,8]. This means that typing is treated as a relation between a term and another term. We take a similar approach, annotating every node in the syntax tree with another term, which represents the type and has annotations of its own. To achieve that, we extend the object language by modifying the generating bifunctor:

```
-- | Extending a type of types with universe.
data WithUniverse ty = Universe | SomeType ty

data TyF t scope term = TyF
  { termF :: t scope term
  , typeF :: WithUniverse term
  }
```

```
-- / A typed term generated from t.
type TFS t a = FS (TyF t) a
```

We use the type `WithUniverse (TFS t a)` for type annotations, meaning that type terms themselves have type annotations. The recursive annotation stops either at variables, or at `Universe`, which is an explicit universe type \mathcal{U}_∞ . Consider term $\lambda x.f x$. Adding type annotations (written $t : T$) according to `TyF` would produce the following typed term (here we assume the object language also has its own universe type \mathcal{U} , and f, A, B are free variables):

$$\begin{array}{ll} \lambda x.f x & \text{(untyped term)} \\ (\lambda x.(f x : B)) : (A \rightarrow B : (\mathcal{U} : \mathcal{U}_\infty)) & \text{(typed term)} \end{array}$$

Since we have modified the type of nodes in the syntax tree, with `TFS t a`, we have type annotation *for every subterm except variables*. This makes it easy to extract types of subterms when necessary without the need to repeatedly infer types.

With type inference, we also need to take into account meta variables. Extending typed terms with meta variables yields the following type:

```
type TSOAS v t a = SOAS v (TyF t) a
```

Note that the universe \mathcal{U}_∞ is not available as a term, it can only be used in the type position. This, in particular, means that no variable or meta variable can be instantiated with \mathcal{U}_∞ .

Type checking context

We implement bottom-up type inference and keep track of currently available type information. As we traverse a given term and solve arising constraints, this information is updated. In this subsection, we explain what kind of type information we need to store and how we mix stateful computations with backtracking.

At any given moment in the algorithm, we are considering a subterm, possibly located inside several scopes. For the type inference algorithm, we translate nested data types with `Inc a` into `IncMany a`, effectively merging individual scopes into one.

1. Known types of free variables. Types of free variables cannot depend on any bound variables, so for each free variable we store its type as `TSOAS v t a`.
2. Known types of meta variables. Meta variables are global and, similarly to free variables, cannot depend on bound variables. So for each meta variable we store its type as `TSOAS v t a`.
3. Known types of bound variables. Types of bound variables may depend on previously introduced bound variables. We store these as a list of types: `[TSOAS v t (IncMany a)]`.
4. Known substitutions for meta variables. This is the same as in the unification algorithm with only difference being that substitutions are happening for typed terms: `Substs v (TyF t) a`.

- 738 5. Leftover unification constraints. Again, similar to the unification algorithm,
 739 each constraint has type
 740 `Constraint v (TyF t) a`.
 741 6. A stream of fresh meta variable identifiers.
- 742 All of this is collected into a single data type:

```
data TypeInfo v t a = TypeInfo
  { typesOfFreeVars  :: [(a, TSOAS v t a)]
  , typesOfBoundVars :: [TSOAS v t (IncMany a)]
  , typesOfMetaVars  :: [(v, TSOAS v t a)]
  , metaVarSubsts     :: Substs v (TyF t) a
  , constraints       :: [Constraint v (TyF t) a]
  , freshMetaVars     :: [v]
  }
```

743 To go through candidate substitutions for meta variables we rely on `MonadPlus`
 744 type class. Moreover, we require the monad to obey the left distributive law, as
 745 it is essential for backtracking:

```
mplus a b >=> f = mplus (a >=> f) (b >=> f)
```

746 A well-established implementation for backtracking is Kiselyov, et al.'s `LogicT`
 747 monad transformer [15]. To deal with state and possible type errors we use
 748 `StateT` and `ExceptT` transformers [19] correspondingly.

749 Unfortunately, `StateT` does not mix well with non-deterministic nature of
 750 `LogicT`. In particular, neither `StateT s (LogicT m)` nor `LogicT (StateT s m)`
 751 support the left distributive law of `MonadPlus`. A common workaround is to make
 752 the state itself nondeterministic. More specifically, we use the following data type
 753 to represent stateful computation with backtracking:

```
newtype SEL s e x = SEL
  { runSEL :: StateT (Logic s) (ExceptT e Logic) x }
```

754 Using `Logic s` as the type of state allows for `MonadPlus` instance that sup-
 755 ports left distributive law:

```
instance MonadPlus (SEL s e) where
  mzero = SEL (lift (lift mzero))
  mplus (SEL l) (SEL r) = SEL $ do
    states <- get
    (x, s') <- lift $ ExceptT $ mplus
      (runExceptT (runStateT l states))
      (runExceptT (runStateT r states))
    put s'
    return x
```

756 We note that it is also possible to use `interleave` instead of `mplus` to force
 757 interleaving of branches in the search space. But it is also possible to leave more

control on the user side, deriving `MonadLogic` instance. It is also fairly straightforward to implement `MonadState s` instance for `SEL s e`. With all those instances in place, the monad for type checking and type inference becomes merely a special case of `SEL`:

```
type TypeCheck v t a =
  SEL (TypeInfo v t a)
      (TypeError (TSOAS v t a))
```

Typing syntactic constructions

To perform type inference for any given language, it is enough to know how to perform a single step: given types of parts for single syntactic construction, compute the type of the whole. An important implementation detail is to provide not just the types of the parts, but an actual computation context for that type. In other words, instead of `TFS v t a` we will have `m (TFS v t a)` where `m` is some typechecking monad. This is done to give the implementor of a particular language more control over typechecking and constraint resolution:

```
class Inferable ty t where
  inferF :: MonadTypecheck v t a m
    => t (m (Scope (TFS v t) a))
        (m ((TFS v t) a) a))
    -> m (t (Scope (TFS v t) a)
          ((TFS v t) a) a))
```

Once we know how to perform a single step of type inference, all we need to do is traverse the entire term:

```
infer :: (Inferable t, MonadTypecheck v t a m)
    => FS t a -> m (TFS v t a)
infer term = case term of
  Var x -> do
    addKnownFreeVar x
    return (Var x)
  Free t -> do
    ty <- Free <$>
      inferTypeFor (bimap inferScope infer t)
    clarifyTypedTerm ty
```

Here, `addKnownFreeVar` adds the free variable to the `TypeInfo` state with a fresh type meta variable, if it is the first time this variable is encountered. As performing inference for a single syntactic construction may result in new meta variable substitutions, we need to apply them across known type information and, perhaps, simplify the inferred typed term. For that we use `clarifyTypedTerm`, which has a straightforward implementation that we omit here.

778 **Unifying types**

779 Type checking in our implementation is merely a combination of type inference
 780 and unification:

```

typecheck term ty = infer term >= shouldHaveType ty

shouldHaveType term expected = do
  actual <- typeOf term
  unifyWithExpected actual expected

```

781 Here, `typeOf` is a helper that either extracts the type annotation directly
 782 from `TyF`, or, when the term is a variable, extracts it from the `typesOfFreeVars`
 783 in current type information state.

784 For the unification, we take all known substitutions and constraints and run
 785 the pre-unification algorithm with `unify`, updating the type information and
 786 refining the types:

```

unifyWithExpected actual expected = do
  substs <- gets metaVarSubsts
  cs <- gets constraints
  (cs', substs') <-
    unify substs ((actual :~: expected) : cs)
  modify (\info -> info
    { metaVarSubsts = substs'
      , constraints = cs'
    })
  clarifyTypedTerm actual

```

787 **Fresh type meta variables**

788 Whenever a fresh type meta variable is created, we take into account all the
 789 bound variables present in scope. In other words, we generate a meta variable
 790 application will all bound variables as arguments: `M[x1, ..., xn]`. Note that we
 791 could also add all free variables, but in practice that is rarely wanted.

```

freshTypeMetaVar
  :: MonadTypecheck v t a m => m (TFS v t a)

```

792 **Entering and exiting scopes**

793 To infer types inside scopes we introduce a couple of helpers. First, `inScope`
 794 one adds information about the type of a bound variable to the current state
 795 before running given computation in scope, then it exits the scope, removing
 796 information about the bound variable. Second, we introduce a helper, similar to
 797 `typeOf`, that figures out types for scopes.

```

inScope :: MonadTypecheck v t a m
=> TFS v t a -> m r -> m r

typeOfScope :: MonadTypecheck v t a m
=> TFS v t a
-> Scope (TFS v t) a -> m (Scope (TFS v t) a)

```

798 With these helpers we are finally ready to consider implementations of specific
799 type theories.

800 B Examples

801 B.1 Simply typed lambda calculus

802 Here we apply our approach to an implementation of simply typed lambda cal-
803 culus (STLC) with pairs. We start with a generating bifunctor:

```

data TermF scope term
= FunF term term           -- Function type:  $T_1 \rightarrow T_2$ 
| LamF (Maybe term) scope -- Abstraction:  $\lambda(x : T_1).T_2$ 
| AppF term term           -- Application:  $(T_1 T_2)$ 
| PairTyF term term        -- Pair type:  $\langle T_1, T_2 \rangle$ 
| PairF term term          -- Pair:  $\langle T_1, T_2 \rangle$ 
| FirstF term              -- First projection:  $\pi_1 T$ 
| SecondF term             -- Second projection:  $\pi_2 T$ 

-- / An STLC term with free variables in a.
type Term a = FS TermF a

```

804 We note a couple of details about this particular presentation of STLC:

- 805 1. We do not have an explicit universe type, as it is introduced automatically
806 with **TyF**.
- 807 2. We have an optional type annotation for the bound variable of λ -abstraction.
808 We make the annotation optional to illustrate how our type inference mixes
809 with type annotations provided by the user.
- 810 3. Both types and terms are generated with **TermF**.

811 Next step is to introduce helpful pattern synonyms. We will immediately
812 work with typed terms, so we only create patterns for those. We remind that
813 these can be automatically generated using Template Haskell:

```

pattern Typed ty t = Free (InL (TyF t ty))
pattern FunT ty t1 t2 = Typed ty (FunF t1 t2)
pattern LamT ty body = Typed ty (LamF body)
pattern AppT ty t1 t2 = Typed ty (AppF t1 t2)
...

```

814 Using these patterns we implement WHNF reduction for typed STLC terms:

```
instance Reducible TermF where
  reduceL term = case term of
    ExtE{} -> reduceR term -- handle extension
    FirstT ty t -> case reduceL t of
      PairT _ty f _ -> reduceL f
      t' -> FirstT ty t'
    SecondT ty t -> case reduceL t of
      PairT _ty _ s -> reduceL s
      t' -> SecondT ty t'
    AppT ty fun arg -> case reduceL fun of
      LamT _ty body -> reduceL (substitute arg body)
      fun' -> AppT ty fun' arg
    _ -> term
```

815 First-order unification requires `Unifiable` instance, which has a straightfor-
 816 ward implementation. Here we show the less trivial case for `LamF`:

```
instance Unifiable TermF where
  zipMatch (LamF ty1 body1) (LamF ty2 body2)
    = Just (LamF ty (body1, body2))
  where
    ty = case (ty1, ty2) of
      (Nothing, Nothing) -> Nothing
      (Just t1, Just t2) -> Just (t1, t2)
      (Just t, Nothing) -> Just (t, t)
      (Nothing, Just t) -> Just (t, t)
    ...
  zipMatch _ _ = Nothing
```

817 *Remark 1.* Since the type annotation for the bound variable is optional, it is
 818 possible that during unification we have the annotation on the left but not on
 819 the right, or vice versa. In this case we intend to keep the type annotation, so
 820 we pair it with itself. A more refined version of `Unifiable` type class, such as
 821 in Wren Romano’s `unification-fd`, could handle this more gracefully, avoiding
 822 generating the unnecessary constraint of the form $t \equiv t$.

823 Next, for higher-order unification we need to establish structural guesses.
 824 This boils down to identifying introduction-elimination pairs of syntactic con-
 825 structions:

```
instance HigherOrderUnifiable TermF where
  guessMetas term = case term of
    AppF f arg -> AppF (f, [LamF ()]) (arg, [])
    -- ^ M[z] t implies M[z] := λx.M'[x, z]
    FirstF t -> FirstF (t, [PairF () ()])
    -- ^ π1 M[z] implies M[z] := ⟨M1[z], M2[z]⟩
```

```

SecondF t -> SecondF (t, [PairF () ()])
-- ^  $\pi_2 M[\bar{z}]$  implies  $M[\bar{z}] := \langle M_1[\bar{z}], M_2[\bar{z}] \rangle$ 
_ -> bimap (,[]) (,[]) term

```

```

shapes = [ AppF HasHead NoHead
          , FirstF HasHead, SecondF HasHead ]

```

826 As we mention in Section 4.1, the `HigherOrderUnifiable` instance can be au-
 827 tomated entirely using either Template Haskell or GHC Generics given `Reducible`
 828 instance for the underlying bifunctor.

829 Finally, for type inference we specify relationships between terms and types:

```

instance Inferable t
  inferF term = case term of

```

830 To infer types of types, we simply need to check the types of components.
 831 For example, for the function type we only have to check that both argument
 832 and result types are indeed types:

```

FunF inferA inferB -> do
  a <- inferA >=> shouldHaveType Universe
  b <- inferB >=> shouldHaveType Universe
  pure (TyF (FunF a b) Universe)

```

833 Inferring the type of a lambda abstraction requires checking the type an-
 834 notation if it is exists, inferring the type of the body, and producing the final
 835 function type:

```

LamF minferA inferBody -> do
  typeOfArg <- case minferA of
    Just inferA ->
      inferA >=> shouldHaveType Universe
    Nothing -> freshTypeMetaVar
  typedBody <- inScope typeOfArg inferBody
  typeOfBody <-
    typeOfScope typeOfArg typedBody >=> nonDep
  pure $ TyF
    (LamF (typeOfArg <$ minferA) typedBody)
    (SomeType
      (FunT Universe typeOfArg typeOfBody))

```

836 Note the use of `nonDep` — we have to explicitly limit the inference to make
 837 sure that the type of the body does not depend on the variable bound by the
 838 lambda abstraction.

839 For an application term $f x$, we have to infer the type F of the function F
 840 and the type X of its argument x . Then, if the function type $F \equiv A \rightarrow B$, then
 841 we simply need to unify argument type X with the expected type A . Otherwise,
 842 we need to unify the type of function F with type $X \rightarrow M$, where M is a fresh
 843 type meta variable:

```

AppF inferFun inferArg -> do
  f <- inferFun -- f : F
  x <- inferArg -- x : X
  typeOfApp <- do
    typeOfFun <- typeOf f
    case typeOfFun of
      -- if F ≡ A → B
      FunT _ expected result -> do
        -- then X ≡ A
        shouldHaveType (SomeType expected) x
        return result
    - -> do -- otherwise
      result <- freshTypeMetaVar -- M : U∞
      argType <- typeOf x
      -- F ≡ X → M
      unifyWithExpected typeOfFun
        (mkFun argType result)
      result
  return (TyF (AppF f x) (SomeType typeOfApp))

```

844 Completing `inferF` for the rest of syntactic constructors in `TermF` is straight-
 845 forward, and we omit the implementation to save space. After all the preparation
 846 we get type inference for simply typed lambda calculus:

```

> t = LamE (LamE (Var (S Z))) -- λx.λy.y
> infer t
LamT (SomeType
      (FunT Universe (MetaAppT Universe 1 [])
        (FunT Universe (MetaAppT Universe 2 [])
          (MetaAppT Universe 2 []))))
(LamT (SomeType
      (FunT Universe (MetaAppT Universe 2 [])
        (MetaAppT Universe 2 []))))
(Var (S Z))

```

847 The result above corresponds to the following typed term:

$$\begin{aligned}
 &\lambda x. (\lambda y. y : (M_2[] : \mathcal{U}_\infty) \rightarrow (M_2[] : \mathcal{U}_\infty) : \mathcal{U}_\infty) \\
 &\quad : M_1[] \rightarrow ((M_2[] : \mathcal{U}_\infty) \rightarrow (M_2[] : \mathcal{U}_\infty) : \mathcal{U}_\infty) : \mathcal{U}_\infty
 \end{aligned} \tag{1}$$

848 Or, omitting the \mathcal{U}_∞ annotations, we get:

$$\lambda x. (\lambda y. y : M_2[] \rightarrow M_2[]) : M_1[] \rightarrow (M_2[] \rightarrow M_2[]) \tag{2}$$

849 Since we mix terms and types of STLC and use dependent type inference
 850 engine, our version of STLC has a couple of unique features, differentiating it
 851 from a classical STLC:

1. We explicitly prevent the type of body in a lambda abstraction to depend on the argument. For users of STLC this means that they can input terms like $\lambda A. \lambda(x : A).x$ and get a type error saying that the inferred type of $\lambda(x : A).x$, which is $A \rightarrow A$ is dependent on the bound variable A , which is not allowed in STLC.
2. We do not forbid computation in types. Indeed, a term $\lambda(f : ((\lambda x.x)A) \rightarrow B).fx$ is valid, and we can infer its type to be $(A \rightarrow B) \rightarrow B$, computing $(\lambda x.x)A \equiv A$ in the process. It is possible to add validation pass to ensure that types only consist of certain syntactic constructions, disallowing non-type terms. However, we see the ability to perform computation in types as a bonus feature for our implementation of STLC.

Overall, we had to write down definitions of `TermF`, implement WHNF reduction for STLC terms in `Reducible` and specify how to infer types in `Inferable`. Everything else could be generated automatically with Template Haskell or GHC Generics. For this fairly little effort we have gotten an implementation of a variation of STLC with type inference and computation available in types.

B.2 Martin-Löf Type Theory

Let us now apply the approach to an actual dependent type theory — intensional Martin-Löf Type Theory (MLTT). We start with a generating bifunctor:

```
data TermF scope term
= UniverseF      -- Universe type:  $\mathcal{U}$ 
| PiF term scope -- Dependent product  $\Pi_{x:T_1} T_2$ 
| LamF scope     -- Abstraction:  $\lambda x. T_2$ 
| AppF term term -- Application:  $(T_1 T_2)$ 
| SigmaF term scope -- Dependent sum  $\Sigma_{x:T_1} T_2$ 
| PairF term term -- Pair:  $\langle T_1, T_2 \rangle$ 
| FirstF term     -- First projection:  $\pi_1 T$ 
| SecondF term    -- Second projection:  $\pi_2 T$ 
| IdTypeF term term -- Identity type:  $x = y$ 
| ReflF term      -- Reflexivity:  $\text{refl}_T$ 
| JF term term term term term term
  --  $\sim$  Identity type eliminator:  $J(A, a, C, d, x, p)$ 

-- | An MLTT term with free variables in a.
type Term a = FS TermF a
```

Remark 2. Note that in this representation we chose to not have any type annotations for bound variables in abstraction and for the type of terms in the identity type or refl_t . We also note that it might be possible to avoid the term t in the annotation for refl_t as well, since the term t is present in the type $t = t$ of refl and can be inferred in principle.

876 In this implementation we use a single universe type and assume type-in-
 877 type: $\mathcal{U} : \mathcal{U}$. It is possible to introduce a hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$
 878 instead by using `UniverseF Natural` constructor.

879 Similarly to STLC implementation, we expect the relevant pattern synonyms
 880 to be written out in a mechanical way or derived automatically. Implementing
 881 WHNF reduction for MLTT is straightforward as it only differs from STLC in
 882 the use of J-eliminator:

```
instance Reducible TermF where
  reduceL term = case term of
    JT ty tA a tC d x p ->
      case reduceL p of
        ReflT{} -> reduceL d
        p'      -> JT ty tA a tC d x p'
    ...
```

883 For `Unifiable` and `HigherOrderUnifiable` we also rely on a mechanical
 884 or automatic derivation and so omit it here to save space. Finally, we define
 885 inference for individual syntactic constructions:

```
instance Inferable t
  inferF term = case term of
```

886 To avoid infinite type annotations, we set the type of universe to be \mathcal{U}_∞ :

```
UniverseF -> pure (TyF UniverseF Universe)
```

887 Inferring types for Π -types and Σ -types involves dependent type checking.
 888 Given term $\Pi_{x:A} B$, where B is a subterm that may refer to x , we have to check
 889 that both $A : \mathcal{U}$ and $B : \mathcal{U}$. Note that since B is in the scope, its inferred type,
 890 by default, might also be dependent on x . For example, in the term $\Pi_{x:A} \text{refl}_x$
 891 the algorithm would infer that refl_x has type $x = x$, which captures the variable
 892 x . To make sure the body of a Π -type is always a type, we need to unify it with
 893 \mathcal{U} . But for that we also need to make sure it is not dependent, so we use `nonDep`:

```
PiF inferA inferB -> do
  a <- inferA >>= shouldHaveType Universe
  typeOfA <- typeOf a
  b <- inScope typeOfA inferB
  typeOfB <- typeOfScope typeOfA b >>= nonDep
  typeOfB `shouldHaveType` Universe
  pure (TyF (PiF a b) Universe)
```

894 Inferring the type for a dependent λ -abstraction is relatively straightforward.
 895 We generate a fresh type meta variable for the argument and infer the type of
 896 the body. In general, we should check that the inferred type is indeed a type, as
 897 many type theories, such as cubical type theory, have multiple universes. That
 898 said, in pure MLTT we can omit this check.

```

LamF inferBody -> do
  a <- freshTypeMetaVar
  typedBody <- inScope a inferBody
  b <- typeOfScope a typedBody
  typeOfScope a b >=> nonDep
    >=> shouldHaveType Universe
  pure $ TyF
    (LamF typedBody)
    (SomeType (PiT Universe a b))

```

899 The rest of syntactic constructors are fairly straightforward to handle in
900 a similar fashion. Completing `Inferable` brings dependent type inference to
901 MLTT.