Free Monads, Intrinsic Scoping, and Higher-Order Preunification

Nikolai Kudasov

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. 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 [14], Jensen-Pietrzykowski’s full unification [15] procedures, procedures for decidable fragments [25, 12] and a recent efficient implementation of full HOU [33].

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 [28] (HOAS): any binding construction can be encoded in λ-calculus. Unfortunately, HOAS and its variants [7, 34] are not always comfortable to work with as witnessed by both language implementors [17, 8] and formalization researchers [12].

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 \cite{11} have recently been an attractive alternative to HOAS. It has been successfully used to generate metatheory in Agda \cite{12} and a full $E$-unification procedure \cite{18} 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 parametrized metavariables. Each syntactic construction can be parametrized by a sequence of (potentially, scoped) subterms. For example, SOAS for simply typed lambda calculus \cite{11, Example 1} is generated from a family of constructors for all types $\sigma$ and $\tau$:

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

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

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):

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

Free monads \cite{31} generate (first-order) abstract syntax trees with a monadic binding operation serving as substitution. Following Swierstra \cite{31}, 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 \lambda-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 \cite{10} are commonly used in practice. Generalized de Bruijn indices \cite{4} have also been used (e.g. in Epigram \cite{24}) 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 generalized 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:

\footnote{The double arrow ($\Rightarrow$) here corresponds to the function types in the object language, while single arrow ($\rightarrow$) is a part of the type signature (in the metatheory) separating types of subterms from the type of the resulting term for each of the syntactic constructors.}

\footnote{such as implemented in the bound package, available at \url{http://hackage.haskell.org/package/bound}}
data TermF scope term
  = LambdaF scope -- \lambda x . T
  | AppF term term -- T_1 T_2
  | PiF term scope -- \Pi_{x : T_1} T_2(x)

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 [18].

1.1 Related Work

Unification and Free Monads. In his 2001 pearl [30], 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 [31]: terms with metavariables are constructed using free monads.

Axelsson and Vezzosi [4] use data types à la carte approach in their syntax for higher-order rewrite rules. However, their implementation of capture avoiding substitution explicitly demands specific syntax for variables, lambda abstractions, and applications, preventing correct treatment of other potential binding constructions. Our approach adds variables freely and does not impose any further restrictions on the syntax of the object language, allowing arbitrary binders.

Second-Order Abstract Syntax. Fiore and Szamosvancev [12] 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 [13] 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 Paszke have introduced the Foil [23], making it possible to have intrinsic scoping while maintaining the efficiency
benefits of the Barendregt convention, as implemented in the Rapier\cite{Rapier}, 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 assistants\cite{ProofAssistants}, in 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\cite{FreeScopedMonads} 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\cite{TermReduction} we propose an approach to the implementation of term reduction that mixes well with the data types à la carte approach.
3. In Section\cite{HigherOrderPreunification} 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 \cite{E-unification} and extract the preunification component of it.
4. In Section\cite{TypeChecking} 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\cite{Bellegarde}, and monadic structure (substitution) for untyped terms was developed by Bird and Patterson\cite{Bird}. Some later work has been done for typed terms\cite{TypedSOAS}, but those use intrinsic typing which we are not using in this paper\cite{IntrinsicTyping}.

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 parametrize 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.

\textsuperscript{3} or, at the very least, prototyping of programming languages and proof assistants

\textsuperscript{4} 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.
2.1 Intrinsically Well-Scoped Terms

Following Bird and Patterson [6] we start with the following definitions. First, we define a type constructor to extend the type of variables with one more name:

```haskell
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):

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

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

```haskell
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
```

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

One could use Scope directly to define, for example, the type of λΠ-terms, parametrized over the type of free variables:

```haskell
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.
```

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

```haskell
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
```

Compared with traditional de Bruijn indices, relying on nested data types using Scope has two great advantages. First, it is safer since ill-scoped terms are also ill-typed. Second, programming with scopes is now more type-driven and allows for more straightforward implementations (with substitute being a prime example).
Binding Multiple Variables. It will be useful to us in Section 2.3 to have a variation of `Scope` that supports binding of many variables at once:

```haskell
data IncMany var
    = BoundVar Int -- an Int-indexed bound variable
    | FreeVar 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.

```haskell
substituteMany :: Monad term var -> (Int -> term var) -> IntScope term var -> term var
substituteMany f s = s >>= \x ->
    case x of
        BoundVar n -> f n
        FreeVar 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 [31]. 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:

```haskell
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`\(^5\)), with the bind operation `(>>=)` corresponding to the substitution. Importantly, unlike regular free monads [31], the substitution in free scoped monads respects bound variables.

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

\(^5\) Note that `Bifunctor` and some other instances can be automatically derived for a user-defined types using GHC extensions, such as `DeriveFunctor`, or Template Haskell utility functions like `deriveBifunctor` from bifunctors package.
We now reformulate our type for untyped lambda terms, defining `TermF` to specify all syntactic constructions and using `FS` to give us the type of terms:

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

type Term a = FS TermF a

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

The `PatternSynonyms` extensions help us keep `whnf` implementation without any changes after the switch to free scoped monads representation:

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

It will be useful to us sometimes to apply a transformation to all nodes, changing from one signature to another. For that, we introduce this function:

```haskell
trans :: Bifunctor f => (forall x y. f x y -> g x y) -> FS f a -> FS g a

trans _phi (Pure x) = Pure x
trans phi (Free t) = Free (phi (bimap (trans phi) (trans phi) t))
```

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 trade-offs. 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 [32] 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 [11] and use parametrized metavariables. Instead of embedding metavariables directly into `FS` data type, we use data types à la carte approach [31] and extend any given bifunctor `term` with metavariables. The following datatype generates parametrized metavariables:

```haskell
data MetaAppF v scope term = MetaAppF v [term]
```
Parametrization provides independence from object language syntax (we do not require having function application in the object language), but it also keeps all “dependencies” of a metavariable bundled with it.

Following [18], we write $M_i[t_1, t_2, \ldots, t_n]$ to mean application of metavariable $M_i$ to terms $t_1, t_2, \ldots, t_n$. 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 $\text{Sum } f \ g$ that supports constructions from both original signatures.

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

Now, we can extend any signature $t$ with parametrized metavariables:

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

Here, $\text{SOAS}$ stands for “Second-Order Abstract Syntax” with $v$ being the type of metavariables, $t$ — the term signature, and $a$ — the type of free variables.

**Metavariable Substitution.** Following SOAS [11], we define substitution for parametrized metavariables by mapping each metavariable to a scoped term, with $n$ bound variables. In the implementation, we rely on $\text{IntScope}$, allowing arbitrarily many bound variables (not statically checked):

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

Here, the first component of type $\text{Int}$ 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.

We represent substitution $M_i[x_1, x_2, \ldots, x_n] \mapsto t$ as a pair of metavariable $M_i$ and the scoped term $t$, represented using $\text{MetaAbs}$ for the extended language. A simultaneous substitution [11, Section 2] is represented by a list of substitutions:

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

To apply $\text{Substs}$ to a term, we merely traverse the term replacing every occurrence of $\text{MetaAppF}$ that has a corresponding substitution:

```haskell
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
```

---

6 Here we are using a list for simplicity, but it is also possible to use other data structures, such as $\text{Data.HashMap}$ or $\text{Data.Map}$. 
Free (InR (MetaAppF v args)) -> -- metavariables are replaced according to subs
  -- substitute metavariables in arguments
let args' = map (applySubsts subs) args
in case lookup v (getSubsts subs) 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 subs)

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

1. SOAS v t a is the type of second-order terms generated from t;
2. Pure x corresponds to a (free) variable x;
3. Free (InR (MetaAppF v [t1, . . . , tN])) corresponds to M[t1, . . . , tN]
4. Free (InL t) corresponds to some syntactic construction F(x1,t1,. . . ,xn,tn);
5. function applySubsts performs simultaneous metavariable substitution.

3 Term Reduction à la Carte

In this section, we organize term reduction for extensible languages following
data types à la carte [31]. The motivation is twofold. On the one hand, we want
to be able to specify reduction in object languages without having to deal with
metavariables (indeed, it is natural for the reduction rules to be independent of
metavariables). On the other hand, we want to be able to easily extend languages
with new syntactic constructions (e.g. pairs and projections).

In general, assuming the constructions from the two signatures are not sup-
posed to "interfere" with each other, we can define term reduction for each compo-
nent independently. To get reduction working for terms generated from signature
'Sum f g', we need to specify reduction for 'f' and 'g', however, it is important
that we give that anticipating an extension:

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

Here reduceL reduces a term of a language, generated by t extended with
ext, assuming terms generated by ext are reducible. Using reduceL and com-
muting left and right languages, we get reduceR:

Fiore and Hur call these operators [11, Section 2]

Hence the use of Sum in the type signature. The importance of this definition becomes
clear when we consider the instance of Reducible for Sum f g

We can also make reduceR a part of the Reducible class.
reduceR = commute . reduceL . bimap commute commute

reduceL = trans $ \ case
  InL x -> InR x
  InR y -> InL y

Empty Signature  A particularly important language is an empty one:

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

Note that language generated by Empty is not actually empty: free variables are always added with FS. Without any constructors reduceL is trivial:

instance Reducible Empty where
  reduceL e = case e of {}

We can express term reduction in an (unextended) object language as a special case of reduceSum, extending it with an Empty signature:

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

Sum of Signatures  Combining two reducible languages with Sum yields a reducible language. Here, we rely on commutativity and associativity of Sum:

instance (Reducible f, Reducible g) => Reducible (Sum f g) where
  reduceL (InL t) = assoc' (reduceL (bimap assoc assoc t))
  reduceL (InR t) = from (reduceL (bimap to to t))
  where
    to = assoc . commute . assoc
    from = assoc' . commute . assoc'

Reducing \(\lambda\Pi\)-Terms  To adapt whnf to reduce terms in a language generated by TermF with arbitrary extension, we can introduce patterns for extended language:

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

Implementation of whnf is transferred almost letter for letter, with an important addition being the case when the root node belongs to the extension — here, we delegate reduction to the extension by using reduceR:

instance Reducible TermF where
  reduceL = \ case
    AppF fun arg -> case reduceSum fun of
      LamE body -> reduceSum (substitute arg body)
      fun' -> AppE fun' arg
    t -> Free (InL t)
Reducing with Metavariables  Parametrized metavariables reduce to themselves, however we can choose to reduce or keep their parameters, yielding two possible definitions. First one leaves parameters unevaluated:

```haskell
-- "lazy" reduction (arguments remain unevaluated)
instance Reducible (MetaAppF v) where
  reduceL t = Free (InL t)
```

The second possible implementation reduces the parameters:

```haskell
-- "strict" reduction (arguments are evaluated)
instance Reducible (MetaAppF v) where
  reduceL (MetaAppF m args) = Free (InL (MetaAppF m (map reduceSum args)))
```

Note that even though the second implementation is “strict” in the object language, using Haskell as a host language makes evaluation somewhat lazy in the sense that actual evaluation of parameters might still be delayed. This kind of lazy evaluation is used in some HOU algorithms [33] and from now we assume the second instance implementation for `MetaAppF`.

4 Higher-Order Unification

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

The algorithm is loosely based on E-unification for second-order abstract syntax [18], with the following important differences:

1. we forego the (mutate) rule [18, Definition 28], and instead assume that terms can be normalized (via `reduce`);
2. we combine the (imitate) and (project) rules [18, Definitions 24–25] into a single rule with generalized Huet-style bindings [19];
3. we only implement preunification, leaving unsolved constraints between two metavariables, so we are not using (eliminate), (identify), and (iterate) rules;
4. we implement unification itself in an untyped setting, i.e. our implementation of higher-order unification does not (directly) take types of terms into account; technically, type information can be embedded into the terms themselves and can be used by `reduce`, but in this paper we do not make extra assumptions when generating Huet-style bindings and leave development of an algorithm for type-directed generalized Huet-style bindings for future work.

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

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

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

For higher-order unification, we will require generalized Huet-style bindings [19]. Essentially, for a given language we need to know the following information:

1. For each argument (subterm) of a syntactic construction, is there a certain shape of a term that allows further reduction? For example, given a term $\pi_1 \text{M}_1[]$ (where $\text{M}_1[]$ is a metavariable), we should understand that $\text{M}_1$ can be substituted by a tuple $(\text{M}_2, \text{M}_3)$ (where $\text{M}_2$ and $\text{M}_3$ are fresh metavariables) to allow further reduction.

2. For parametrized metavariables, what are possible ways to construct a term that will use the parameters via reduction? For example, when unifying $\text{M}_1[(t_1, t_2)]$ with $t_1$, we should be able to try the substitution $\text{M}_1[x] := \pi_1 \text{M}_2[x]$ to find the solution $\text{M}_1[x] := \pi_1 x$.

First-Order Unification For first-order unification, we need to be able to match individual nodes of the syntax tree. Similarly to Wren Romano’s unification-fd package[10], we define a type class with a single method:

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

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

```haskell
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
```

[10] 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.
Implementing `Unifiable` is usually mechanical and can in fact be fully automated using GHC’s Generics (as is done in `unification-fd`) or Template Haskell.

**Higher-Order Unification** Note that, following SOAS [11], we do not require the signature to have lambda abstractions or applications as there is more than one way the user might want to introduce those. For example, application can be defined as binary or taking a list of arguments. Moreover, some theories have not one but several syntactic abstractions or applications (such as \( \Pi \)-types and extensions types in Riehl and Shulman’s type theory with shapes [29], or \( \mu \)-abstraction in Parigot’s \( \lambda \mu \)-calculus [26]).

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

```haskell
-- | Placeholder for a subterm that may or may not contain the head.
data IsHead = HasHead | NoHead

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

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

Implementing this for `TermF` we get the following:

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

  shapes = [AppF HasHead NoHead]
```

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

For many type theories, one only needs to identify introduction-elimination pairs to implement `HigherOrderUnifiable`. Given an instance of `Reducible`, one can go over all possible combinations of syntactic constructions to figure out 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. Importantly, the same metavariable can be used with different parameters and in different scopes. This means that a metavariable substitution cannot depend on the bound variables (otherwise they may “leak”).

Consider constraint involving λ-abstraction: \((\lambda f.\lambda x. M_1[f x]) \equiv (\lambda f.\lambda x.f x)\). Going under λ-abstraction in both terms might reduce the original constraint to

\[ M_1[f x] \equiv f x. \]

However, treating \(f\) and \(x\) now as free variables is incorrect as this constraint can be satisfied with two different substitutions: \(M_1[z] \mapsto f x\) (leaks \(f\) and \(x\)) and \(M_1[z] \mapsto z\) (correct).

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

```haskell
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. 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:

```haskell
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.
Simplifying Constraints  Simplification of a single constraint consists of three steps:

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

Given a collection of constraints, we perform simplification on each of them recursively, accumulating and applying generated substitutions from \texttt{guessMeta}, until we end up with a collection of constraints that cannot be simplified any further.

Simplified constraints are expected to be

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

The third potential type of constraints, where both sides are not metavariable applications, are called rigid-rigid constraints. These constraints are guaranteed to be simplified in step 3 with \texttt{zipMatch}. Indeed, \texttt{zipMatch} either returns \texttt{Nothing} (which means that two nodes do not match), or it pairs syntactic sub-trees to match recursively, ensuring structural recursion.

Extracting Head of a Term  If, according to \texttt{guessMetas}, there is a structural guess for some subterm position of a syntactic construction, we call a subterm in that position a head subterm. We say that a term $h$ is a head of a term $t$ if it is a head subterm of $t$ or if it is a head of any head subterm of $t$. For example, the term $\lambda z. f z$ is the head subterm of the term $\pi_1(\lambda z.f z (\pi_2 y))$.

Solving Flex-Rigid Constraints  Preunification algorithm starts off with a list of constraints, reduces rigid-rigid constraints and solves flex-rigid constraints, leaving only flex-flex constraints to be dealt with by the user.

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

- the head of $t$, where each variable bound by $\forall$ is replaced with a fresh metavariable application $M_k[x_1, \ldots, x_n]$;
- a bound variable of $M_i$: $x_j$;
- a candidate shape (one of \texttt{shapes}), where each \texttt{HasHead} position is filled with $T'$ and \texttt{NoHead} position is filled with a fresh metavariable application $M_k[x_1, \ldots, x_n]$. 
The entire algorithm is packed into a single function with the following type signature:

\[
\text{unify} :: ( \text{HigherOrderUnifiable} \ t, \text{Reducible} \ t \\
, \text{MonadPlus} \ m, \text{MonadLogic} \ m, \text{MonadFresh} \ v \ m \\
, \text{Eq} \ a, \text{Eq} \ v ) \\
\Rightarrow \text{Substs} \ v \ t \ a \\
\Rightarrow \text{[Constraint} \ v \ t \ a], \text{Substs} \ v \ t \ a
\]

5 Applications

In this section, we see the application of our approach to implementation of type inference for a couple of type theories. The implementation is available as part of version 0.1.0 of the proof assistant Rzk\color{red}{\textsuperscript{11}} and contains the following relevant modules:

1. module Rzk.Free.Syntax.FreeScoped introduces the free scoped monads, Sum, and utility functions as described in Section 2;
2. module Rzk.Free.Syntax.FreeScoped.Unification2 implements higher-order preunification as described in Section 4;
3. module Rzk.Free.Syntax.FreeScoped.TypeCheck implements type checking and type inference algorithms based on higher-order preunification;
4. module Rzk.Free.Syntax.Example.ULC implements untyped \(\lambda\)-calculus with higher-order unification;
5. module Rzk.Free.Syntax.Example.STLC implements a version of simply typed \(\lambda\)-calculus (STLC) with type inference via higher-order unification; this version differs from the standard STLC by allowing computation at the type level;
6. module Rzk.Free.Syntax.Example.MLTT contains the implementation of intensional Martin-Löf dependent type theory with type inference.

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

We now outline the key moments in the implementation of Martin-Löf type theory, more details on this and the implementation of simply typed \(\lambda\)-calculus can be found in Appendix B and in the corresponding implementation files.

\color{red}{\textsuperscript{11}} see relevant modules in https://github.com/rzk-lang/rzk/tree/v0.1.0/rzk/src/Rzk/Free
5.1 Typed terms

Many implementors define a single type in the host language for both terms and types in the object language \[22,9\]. 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:

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

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

```hs
-- | 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 `BigUniverse`, which is an explicit universe type \(U_\infty\). 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 \(U\), and \(f, A, B\) are free variables):

\[
\lambda x. f \ x \\
(\lambda x. (f \ x : B)) : (A \to B : (U : U_\infty))
\]

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.

5.2 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:

```hs
class Inferable t where
  inferF :: MonadTypecheck v t a m
        => t (m (Scope (TFS v t) a)) (m ((TFS v t) a))
        -> m (t (Scope (TFS v t) a) ((TFS v t) a))
```
Once we know how to perform a single step of type inference, all we need to do is traverse the entire term:

\[
infer :: (Inferable t, MonadTypecheck v t a m) \Rightarrow FS t a \rightarrow m (TFS v t a)
\]

\[
infer\ term = \text{case }\ term\ \text{of}
\]

\[
\Var x \rightarrow\ \text{do}
\]

\[
\text{addKnownFreeVar } x
\]

\[
\text{return } (\Var x)
\]

\[
\Free t \rightarrow\ \text{do}
\]

\[
\text{ty }<- \text{Free }<\$>\ \text{inferTypeFor }\text{(bimap inferScope infer } t)\n\]

\[
\text{clarifyTypedTerm ty}
\]

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.

### 5.3 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:

\[
data\ \text{TermF}\ \text{scope}\ \text{term} \\
=\ \text{UniverseF} \quad \text{-- Universe type: } U \\
|\ \PiF\ \text{term}\ \text{scope} \quad \text{-- Dependent product } \Pi_{x:T_1}T_2 \\
|\ \LamF\ \text{scope} \quad \text{-- Abstraction: } \lambda x.T_2 \\
|\ \AppF\ \text{term}\ \text{term} \quad \text{-- Application: } (T_1 T_2) \\
|\ \SigmaF\ \text{term}\ \text{scope} \quad \text{-- Dependent sum } \Sigma_{x:T_1}T_2 \\
|\ \PairF\ \text{term}\ \text{term} \quad \text{-- Pair: } (T_1 T_2) \\
|\ \FirstF\ \text{term} \quad \text{-- First projection: } \pi_1 T \\
|\ \SecondF\ \text{term} \quad \text{-- Second projection: } \pi_2 T \\
|\ \IdTypeF\ \text{term}\ \text{term} \quad \text{-- Identity type: } x = y \\
|\ \Ref1F\ \text{term} \quad \text{-- Reflexivity: } \text{refl}_T \\
|\ \JF\ \text{term}\ \text{term}\ \text{term}\ \text{term}\ \text{term}\ \text{term} \\
\quad \text{-- Identity type eliminator: } J(A, a, C, d, x, p)
\]

\[
\text{-- / An MLTT term with free variables in a.}
\]

\[
type\ \text{Term}\ a =\ FS\ \text{TermF}\ a
\]

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

1. We omit type annotations for the bound variable of \(\lambda\)-abstraction.
2. Both types and terms are generated with TermF.

In this particular implementation we use a single universe type and assume type-in-type: \(U : U\). It is possible to introduce a hierarchy of universes \(U_0 : U_1 : U_2 : \ldots\) instead by using UniverseF Natural constructor.
Next step is to introduce helpful pattern synonyms. We will immediately work with typed terms, so we only create patterns for those. We remind that these can be automatically generated using Template Haskell:

```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 t4 t5 t6 = Typed ty (JF t1 t2 t3 t4 t5 t6)
```

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

```haskell
instance Reducible TermF where
  reduceL = \case
    JF tA a tC d x p ->
    case reduce p of
      Refl{} -> reduce d
      p' -> J tA a tC d x p'
    ...
```

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

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

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

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

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

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

```haskell
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))
```

The rest of syntactic constructors is fairly straightforward to handle similarly. Completing Inferable brings dependent type inference to MLTT.

6 Conclusion and Future Work

We have presented an approach to abstract syntax representation with free scoped monads and demonstrated its effectiveness for the implementation of Martin-Löf Type Theory. Our example demonstrates that the approach does not require the user to have a deep understanding of higher-order unification to enable type inference for their language.

We have also devised a few directions for future work. First, we would like to extend to full higher-order unification or, better yet, full $E$-unification for second-order abstract syntax [18]. Implementing generic $E$-unification for second-order abstract syntax would be instrumental to implementing proof assistants for type theories with non-trivial or extensible definitional equalities. In particular, we think this might be useful for the implementation of extension types in Riehl and Shulman’s type theory for synthetic $\infty$-categories [29].

Second, we should make higher-order unification more efficient by optimizing the representation of free scoped monads, taking into account the types of unified terms, and recognizing efficient/decidable fragments of unification problems with oracles as in the work of Vukmirovic, Bentkamp, and Nummelin [33].

Acknowledgements I am grateful to Benedikt Ahrens and Daniel de Carvalho for their invaluable feedback throughout my early work towards the implementation of typecheckers for dependently typed languages. I thank Oksana Zhirosh, Ruslan Saduov, and Benedikt Ahrens for proofreading earlier versions of this paper. I also thank the anonymous reviewer of TFP 2024 for an improvement suggestion for Reducible.
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

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. Axelsen, E., Vezzosi, A.: Lightweight higher-order rewriting in haskell. In: Trends in Functional Programming: 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers 16. pp. 1–21. Springer (2016)

5. 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)90022-0

6. 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

7. 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

8. Cockx, J.: 1001 representations of syntax with binding. https://jesper.sikanda.be/posts/1001-syntax-representations.html (Nov 2021), accessed: 2023-01-21

9. 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/CBO9780511770524.007

10. 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

11. 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

12. 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

13. 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

14. Huet, G.: A unification algorithm for typed λ-calculus. Theoretical Computer Science 1(1), 27–57 (1975). https://doi.org/https://doi.org/10.1016/0304-3975(75)90011-0
15. Jensen, D.C., Pietrzykowski, T.: Mechanizing \(\omega\)-order type theory through unification. Theor. Comput. Sci. 3(2), 123–171 (1976). https://doi.org/10.1016/0304-3975(76)90021-9
16. Kiselyov, O., Shan, C.-c., Friedman, D.P., Sabry, A.: Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). SIGPLAN Not. 40(9), 192–203 (sep 2005). https://doi.org/10.1145/1090189.1086390
17. Kmett, E.: Bound. https://www.schoolofhaskell.com/user/edwardk/bound (Dec 2015), accessed: 2023-01-21
18. Kudasov, N.: E-Unification for Second-Order Abstract Syntax. In: Gaboardi, M., van Raamsdonk, F. (eds.) 8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 260, pp. 10:1–10:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.FSCD.2023.10
19. Kudasov, N.: Generalising huet-style projections in e-unification for second-order abstract syntax. In: UNIF 2023-37th International Workshop on Unification (2023)
20. Liang, S., Hudak, P., Jones, M.: Monad Transformers and Modular Interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 333–343. POPL ’95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199528
21. Libal, T., Miller, D.: Functions-as-Constructors Higher-Order Unification. In: Kesner, D., Pientka, B. (eds.) 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 52, pp. 26:1–26:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.FSCD.2016.26
22. Löh, A., McBride, C., Swierstra, W.: A Tutorial Implementation of a Dependent Typed Lambda Calculus. Fundamenta Informaticae 102, 177–207 (01 2010). https://doi.org/10.3233/FI-2010-304
23. Maclaurin, D., Radul, A., Paszke, A.: The Foil: Capture-Avoiding Substitution With No Sharp Edges. In: Proceedings of the 34th Symposium on Implementation and Application of Functional Languages. IFL ’22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3587216.3587224
24. McBride, C.: Epigram: Practical Programming with Dependent Types. In: Proceedings of the 5th International Conference on Advanced Functional Programming. p. 130–170. AFP’04, Springer-Verlag, Berlin, Heidelberg (2004). https://doi.org/10.1007/11546382_3
25. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. In: Schroeder-Heister, P. (ed.) Extensions of Logic Programming, pp. 253–281. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
26. Parigot, M.: \(\lambda\mu\)-calculus: An algorithmic interpretation of classical natural deduction. In: Voronkov, A. (ed.) Logic Programming and Automated Reasoning, pp. 190–201. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
27. Peyton Jones, S., Marlow, S.: Secrets of the Glasgow Haskell Compiler inliner. Journal of Functional Programming 12, 393–434 (July 2002). https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner
28. 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
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 [22,9]. 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:

```haskell
-- |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 $\mathcal{U}_\infty$. 

### References

29. Richel, E., Shulman, M.: A type theory for synthetic $\infty$-categories. Higher Structures 1 (2017)
30. 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]
31. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423–436 (2008). [https://doi.org/10.1017/S0956796808006758]
32. 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]
33. 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]
34. 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]
Consider term $\lambda x. f\ x$. Adding type annotations (written $t : T$) according to $\text{TyF}$ 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):

\[
\lambda x. f\ x \quad \text{(untyped term)}
\]

\[
(\lambda x. (f\ x : B) : (A \rightarrow B : (\mathcal{U} : \mathcal{U}_\infty))) \quad \text{(typed term)}
\]

Since we have modified the type of nodes in the syntax tree, with $\text{TFS} t a$, 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:

\[
\text{type } \text{TSOAS} v t a = \text{SOAS} v (\text{TyF} t) a
\]

Note that the universe $\mathcal{U}_\infty$ 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 $\mathcal{U}_\infty$.

### 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 $\text{Inc} a$ into $\text{IncMany} a$, 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 $\text{TSOAS} v t a$.
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 $\text{TSOAS} v t a$.
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: $[\text{TSOAS} v t (\text{IncMany} a)]$.
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: $\text{Substs} v (\text{TyF} t) a$.
5. Leftover unification constraints. Again, similar to the unification algorithm, each constraint has type $\text{Constraint} v (\text{TyF} t) a$.
6. A stream of fresh meta variable identifiers.

All of this is collected into a single data type:
Free Monads, Intrinsic Scoping, and Higher-Order Preunification

```
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]
}
```

To go through candidate substitutions for meta variables we rely on \texttt{MonadPlus} type class. Moreover, we require the monad to obey the left distributive law, as it is essential for backtracking:

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

A well-established implementation for backtracking is Kiselyov, et al.’s \texttt{LogicT} monad transformer \cite{Kiselyov}. To deal with state and possible type errors we use \texttt{StateT} and \texttt{ExceptT} transformers \cite{Hutton} correspondingly.

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

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

Using \texttt{Logic s} as the type of state allows for \texttt{MonadPlus} instance that supports 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
```

We note that it is also possible to use \texttt{interleave} instead of \texttt{mplus} to force interleaving of branches in the search space. But it is also possible to leave more control on the user side, deriving \texttt{MonadLogic} instance. It is also fairly straightforward to implement \texttt{MonadState s} instance for \texttt{SEL s e}. With all those instances in place, the monad for type checking and type inference becomes merely a special case of \texttt{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 $TFS \nu \tau \alpha$ we will have $m \langle TFS \nu \tau \alpha \rangle$ where $m$ is some typechecking monad. This is done to give the implementor of a particular language more control over typechecking and constraint resolution:

```haskell
class Inferable ty t where
  inferF :: MonadTypecheck v t a m
          \rightarrow t (m (Scope (TFS \nu t) a))
          (m ((TFS \nu \tau a) \alpha))
          \rightarrow m (t (Scope (TFS \nu t) a)
          ((TFS \nu \tau a) \alpha))
```

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

```haskell
infer :: (Inferable t, MonadTypecheck v t a m)
       \rightarrow FS t a \rightarrow m (TFS \nu \tau a)
infer term = case term of
  Var x \rightarrow do
    addKnownFreeVar x
    return (Var x)
  Free t \rightarrow 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.

Unifying types

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

```haskell
typecheck term ty = infer term >>= shouldHaveType ty
shouldHaveType term expected = do
  actual <- typeof term
  unifyWithExpected actual expected
```
Here, `typeOf` is a helper that either extracts the type annotation directly from `TyF`, or, when the term is a variable, extracts it from the `typesOfFreeVars` in current type information state.

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

```haskell
unifyWithExpected actual expected = do
  substs <- gets metaVarSubsts
  cs ← gets constraints
  (cs', substs') ← unify substs ((actual `~` expected) : cs)
  modify (λ info → info
    { metaVarSubsts = substs'
        , constraints = cs'
    })
  clarifyTypedTerm actual
```

**Fresh type meta variables**

Whenever a fresh type meta variable is created, we take into account all the bound variables present in scope. In other words, we generate a meta variable application with all bound variables as arguments: $M[x_1, \ldots, x_n]$. Note that we could also add all free variables, but in practice that is rarely wanted.

```haskell
freshTypeMetaVar :: MonadTypecheck v t a m =⇒ m (TFS v t a)
```

**Entering and exiting scopes**

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

```haskell
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)
```

With these helpers we are finally ready to consider implementations of specific type theories.
B Examples

B.1 Simply typed lambda calculus

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

```haskell
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

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

1. We do not have an explicit universe type, as it is introduced automatically with TyF.
2. We have an optional type annotation for the bound variable of $\lambda$-abstraction. The annotation is optional to illustrate how our type inference mixes with type annotations provided by the user.
3. Both types and terms are generated with TermF.

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

```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)
...
```

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

```haskell
instance Reducible TermF where
  reduce = \case
    FirstF t -> case reduce t of
      Pair f _ -> reduce f
      t' -> First t'
    SecondT t -> case reduce t of
      Pair _ s -> reduce s
      t' -> Second t'
```

App fun arg -> case reduce fun of
  Lam body -> reduce (substitute arg body)
  fun' -> App fun' arg
term -> Free (InL term)

First-order unification requires Unifiable instance, which has a straightforward 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

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

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

instance HigherOrderUnifiable TermF where
  guessMetas term = case term of
  AppF f arg -> AppF (f, [LamF ()]) (arg, [])
  -- " $M[x] t$ implies $M[x] := \lambda x. M'[x, \tau]"
  FirstF t -> FirstF (t, [PairF () ()])
  -- " $\pi_1 M[\tau]$ implies $M[\tau] := \langle M_1[\tau], M_2[\tau] \rangle"
  SecondF t -> SecondF (t, [PairF () ()])
  -- " $\pi_2 M[\tau]$ implies $M[\tau] := \langle M_1[\tau], M_2[\tau] \rangle"
  _ -> bimap (,[]) (,[]) term

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

As we mention in Section[11] the HigherOrderUnifiable instance can be automated entirely using either Template Haskell or GHC Generics given Reducible instance for the underlying bifunctor.

Finally, for type inference we specify relationships between terms and types:
instance Inferable t
    inferF term = case term of

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

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

    Infering the type of a lambda abstraction requires checking the type annotation if it is exists, inferring the type of the body, and producing the final 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))

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

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

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

Completing \texttt{inferF} for the rest of syntactic constructors in \texttt{TermF} is straightforward, and we omit the implementation to save space. After all the preparation we get type inference for simply typed lambda calculus:

\[
\begin{align*}
> & t = \text{LamE (LamE (Var (S Z)))} -- \lambda x.\lambda y.y \\
> & \text{infer t} \\
& \text{LamT (SomeType} \\
& (\text{FunT Universe (MetaAppT Universe 1 []}) \\
& (\text{FunT Universe (MetaAppT Universe 2 []}) \\
& (\text{MetaAppT Universe 2 []})))) \\
& (\text{LamT (SomeType} \\
& (\text{FunT Universe (MetaAppT Universe 2 []}) \\
& (\text{MetaAppT Universe 2 []})))) \\
& (\text{Var (S Z))})
\end{align*}
\]

The result above corresponds to the following typed term:

\[
\begin{align*}
\lambda x. (\lambda y.y : (M_2 \rightarrow \mathcal{U}_\infty)) & : (M_1 \rightarrow ((M_2 \rightarrow \mathcal{U}_\infty) : \mathcal{U}_\infty)) : \mathcal{U}_\infty \\
\text{(1)}
\end{align*}
\]

Or, omitting the \(\mathcal{U}_\infty\) annotations, we get:

\[
\begin{align*}
\lambda x. (\lambda y.y : M_2 \rightarrow M_2) & : M_1 \rightarrow (M_2 \rightarrow M_2) \\
\text{(2)}
\end{align*}
\]

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

```haskell
data TermF scope term =
  UniverseF -- Universe type: \(U\)
| 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: \((T_1, T_2)\)
| 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
  -- 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 \(\text{refl}_T\). We also note that it might be possible to avoid the term \(t\) in the annotation for \(\text{refl}_T\) as well, since the term \(t\) is present in the type \(t = t\) of \(\text{refl}_T\) and can be inferred in principle.

In this implementation we use a single universe type and assume type-in-type: \(U : U\). It is possible to introduce a hierarchy of universes \(U_0 : U_1 : U_2 : \ldots\) instead by using UniverseF Natural constructor.

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

```haskell
instance Reducible TermF where
  reduceL = \case
    JF tA a tC d x p ->
      case reduce p of
        Refl{} -> reduce d
```


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

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

To avoid infinite type annotations, we set the type of universe to be $U_{\infty}$:

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

Inferring types for $\Pi$-types and $\Sigma$-types involves dependent type checking. Given term $\Pi_{x:AB}$, where $B$ is a subterm that may refer to $x$, we have to check that both $A : \mathcal{U}$ and $B : \mathcal{U}$. Note that since $B$ is in the scope, its inferred type, by default, might also be dependent on $x$. For example, in the term $\Pi_{x:A} \text{refl}_x$ the algorithm would infer that $\text{refl}_x$ has type $x = x$, which captures the variable $x$. To make sure the body of a $\Pi$-type is always a type, we need to unify it with $\mathcal{U}$. 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)
```

Inferring the type for a dependent $\lambda$-abstraction is relatively straightforward. We generate a fresh type meta variable for the argument and infer the type of the body. In general, we should check that the inferred type is indeed a type, as many type theories, such as cubical type theory, have multiple universes. That 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))
```

The rest of syntactic constructors is fairly straightforward to handle similarly. Completing **Inferable** brings dependent type inference to MLTT.