Elaboration with First-Class Implicit Function Types

ANONYMOUS AUTHOR(S)

Implicit functions are dependently typed functions, such that arguments are provided (by default) by inference machinery instead of programmers of the surface language. Implicit functions in Agda are an archetypal example. In the Haskell language as implemented by the Glasgow Haskell Compiler (GHC), polymorphic types are another example. Implicit function types are *first-class* if they are treated as any other type in the surface language. This holds in Agda and partially holds in GHC. Inference and elaboration in the presence of first-class implicit functions poses a challenge; in the context of Haskell and ML-like languages, this has been dubbed "impredicative instantiation" or "impredicative inference". We propose a new framework for elaborating first-class implicit functions, which is applicable for full dependent type theories and compares favorably to prior solutions in terms of power, generality and conceptual simplicity. We build atop Norell's bidirectional elaboration algorithm for Agda, and we note that the key issue is incomplete information about insertions of implicit abstractions and applications. We make it possible to track and refine information related to such insertions, by adding a function type to a core Martin-Löf type theory, which supports strict (definitional) currying. This allows us to represent undetermined domain arities of implicit function types, and we can decide at any point during elaboration whether implicit abstractions should be inserted.

Additional Key Words and Phrases: impredicative polymorphism, type theory, elaboration, type inference

1 INTRODUCTION

Programmers and users of proof assistants do not like to write out obvious things. Type inference and elaboration serve the purpose of filling in tedious details, translating terse surface-level languages to explicit core languages. Modern compilers such as Agda have gotten quite adept at this task. However, in practice, programmers still have to tell the compiler when and where to try filling in details on its own.

Implicit function types are a common mechanism for conveying to the compiler that particular function arguments should be inferred by default. In Agda and Coq, one can use bracketed function domains for this purpose:

$$\begin{aligned} &\mathit{id} : \{A : \mathsf{Set}\} \to A \to A \qquad\qquad \mathsf{Definition}\ \mathit{id}\,\{A : \mathsf{Type}\}(x : A) := x. \\ &\mathit{id}\,x = x \end{aligned}$$

In GHC, one can use forall to define implicit function types[1]

$$\begin{aligned} &\mathit{id} :: \mathsf{forall}\,(a :: *).\, a \to a \\ &\mathit{id}\,x = x \end{aligned}$$

In all of the above cases, if we apply *id* to an argument, the implicit type argument is provided by elaboration. For example, in Agda, *id true* is elaborated to *id* {*Bool*} *true*, and analogously in GHC and Coq. In all three systems, there is also a way to explicitly specify implicit arguments: in Agda we may put arguments in brackets as we have seen, in Coq we can prefix a name with @ to make every implicit argument explicit, as in @*id bool true*, and in GHC we can enable the language extension TypeApplications and write *id* @*Bool True*.

Implicit functions are **first-class** if they can be manipulated like any other type. Coq is an example for a system where this is *not* the case. In Coq, the core language does not have an actual

[1]This notation requires language extensions KindSignatures and RankNTypes; one could also write the type $a \to a$ and GHC would silently insert the quantification.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

implicit function type, instead, implicitness is tied to particular *names*, and while we can write $\mathit{list}\,(\mathsf{forall}\,\{A : \mathsf{Type}\},\, A \to A)$ for a list type with polymorphic elements, the brackets here are simply ignored by Coq. For example, Coq accepts the following definition:

$$\begin{aligned}&\mathsf{Definition}\,\mathit{poly}\;:\mathsf{forall}\,(f : \mathsf{forall}\,\{A : \mathsf{Type}\},\, A \to A),\, \mathit{bool} * \mathit{nat} :=\\ &\quad \mathsf{fun}\, f \Rightarrow (f\,\mathit{bool}\,\mathit{true},\; f\,\mathit{nat}\,0)\end{aligned}$$

This is a higher-rank polymorphic function which returns a pair. Note that f is applied to two arguments, because the implicitness in $\mathsf{forall}\,\{A : \mathsf{Type}\},\, A \to A$ is silently dropped.

In GHC Haskell, forall types are more flexible. We can write the following, with RankNTypes enabled:

$$\begin{aligned}&\mathit{poly} :: (\mathsf{forall}\,a.\, a \to a) \to (\mathit{Bool},\, \mathit{Int})\\ &\mathit{poly}\, f = (f\, \mathit{True},\; f\, 0)\end{aligned}$$

However, polymorphic types are only supported in function domains and as fields of algebraic data constructors. We cannot instantiate an arbitrary type parameter to a forall, as in $[\mathsf{forall}\,a.\, a \to a]$ for a list type with polymorphic elements. While this type is technically allowed by the ImpredicativeTypes language extension, as of GHC 8.8 this extension is deprecated and is not particularly usable in practice.

In Agda, implicit functions are truly a first-class notion, and we may have $\mathit{List}\,(\{A : \mathsf{Set}\} \to A \to A)$ without issue. However, Agda's elaboration still has limitations when it comes to handling implicit functions. Assume that we have $[]$ for the empty list and $- :: -$ for list extension, and consider the following code:

$$\begin{aligned}&\mathit{polyList} : \mathit{List}\,(\{A : \mathsf{Set}\} \to A \to A)\\ &\mathit{polyList} = (\lambda\, x \to x)\; ::\; []\end{aligned}$$

Agda 2.6.0.1 does not accept this. However, it does accept $\mathit{polyList} = (\lambda\,\{A\}\, x \to x)\; ::\; []$. The issue is the following. Agda first infers a type for $(\lambda\, x \to x)\; ::\; []$, then tries to unify the inferred type with the given $\mathit{List}\,(\{A : \mathsf{Set}\} \to A \to A)$ annotation. However, when Agda elaborates $\lambda\, x \to x$, it does not yet know anything about the element type of the list; it is an undetermined unification variable. Hence, Agda does not know whether it should insert an extra $\lambda\,\{A\}$ or not. If the element type is later found to be an implicit function, then it should, otherwise it should not. To solve this conundrum, Agda simply assumes that any unknown type is *not* an implicit function type, and elects to not insert a lambda. This assumption is often correct, but sometimes — as in the current case — it is not.

There is significant literature on type inference in the presence of first-class polymorphic types, mainly in relation to GHC and ML-like languages; see e.g. [Leijen 2008, 2009; Serrano et al. 2018; Vytiniotis et al. 2006]. The above issue in Agda is a specific instance of the challenges described in the mentioned works. So far, none of the above solutions have landed in production compilers, for reasons of complexity, fragility and interaction with other language features. A recent GHC development [Serrano et al. 2020] offers a solution which is relatively simple, and which is likely to land in an official GHC release. However, none of these solutions support dependent types, and they also have other limitations which we would like to address.

The solution presented in this paper is to gradually accummulate information about implicit insertions, and to have a setup where insertions can be refined and performed at any time after a particular expression is elaborated. In the current example, our algorithm wraps $\lambda\, x \to x$ in an implicit lambda with unknown arity, whose domain is later refined to be $A : \mathsf{Set}$ when the inferred type is unified with the annotation.

1.1 Contributions

- We propose an elaboration algorithm which translates from a small Agda-like surface language to a small Martin-Löf type theory extended with implicit function types, telescopes and *strictly curried* function types with telescope domain. We use these extensions to accummulate information about implicit insertions.
- Our algorithm is a conservative extension of Norell's bidirectional elaborator for Agda [Norell 2007, Chapter 3]; it accepts strictly more programs, and does not require new constructs in the surface language.
- Our target language serves as a general platform for elaborating implicit functions. The concrete elaborator presented in this paper is a relatively simple one, and there is plenty of room to develop more advanced elaboration and unification. However, our simple algorithm is already comparable or superior to previous solutions for impredicative inference.
- We provide an executable implementation of the elaborator described in this paper.

1.1.1 Note on terminology. We prefer to avoid the term "impredicative inference" in order to avoid confusion with impredicativity in type theory. The two notions sometimes coincided historically, but currently they are largely orthogonal. In type theory, impredicativity is a property of a universe, i.e. closure of a universe under arbitrary products. In the type inference literature, impredicativity means the ability to instantiate type variables and metavariables to polymorphic types. In particular, we have that

- Agda has type-theory-predicative universes, but implements type-inference-impredicative elaboration with first-class implicit function types.
- Coq has type-theory-impredicative Prop universe (and optionally also Set), but implements type-inference-predicative elaboration, because of the lack of implicit function types.
- GHC is type-theory-impredicative with RankNTypes enabled and ImpredicativeTypes *disabled*, as we have $(\text{forall}\ (a :: *).\ a \to a) :: *$.

2 BIDIRECTIONAL ELABORATION

First, we present a variant of Norell's bidirectional elaborator [Norell 2007, Chapter 3]. Compared to ibid. we make some extensions and simplifications; what we end up with can be viewed as a toy version of the actual Agda elaborator. In this section, we use it to build the backbone of our algorithm and illustrate the key issues. We extend this elaborator in Section 5.

2.1 Surface syntax

Figure 1 shows the the possible constructs in the surface language. We only have terms, as we have Russell-style universe in the core, and we can conflate types and terms for convenience. The surface syntax does not have semantics or any well-formedness relations attached; its sole purpose is to serve as input to elaboration. Hence, the surface syntax can be also viewed as a small untyped tactic language which is interpreted by the elaborator.

The syntactic constructs are the almost the same in the surface language as in the core syntax. The difference is that _ holes only appear in surface syntax. The _ can be used to request a term to be inferred by elaboration, the same way as in Agda. This can be used to give let-definitions without type annotation, as in $\textbf{let}\,x : \_ = \mathsf{U}\,\textbf{in}\,x$.

2.2 Core syntax

Figure 2 lists selected rules of the core language. We avoid a fully formal presentation in this paper. Some notes on what is elided:

$$
\begin{array}{rcll}
t, u, v, A, B, C & ::= & x & \text{variable} \\
& | & (x : A) \to B & \text{function type} \\
& | & \{x : A\} \to B & \text{implicit function type} \\
& | & t\, u & \text{application} \\
& | & t\, \{u\} & \text{implicit application} \\
& | & \lambda\, x.\, t & \text{lambda abstraction} \\
& | & \lambda\, \{x\}.\, t & \text{implicit abstraction} \\
& | & \mathsf{U} & \text{universe} \\
& | & \mathbf{let}\, x : A = t\, \mathbf{in}\, u & \text{let-definition} \\
& | & \_ & \text{hole for inferred term}
\end{array}
$$

Fig. 1. Syntax of the surface language.

- We use nameful notation and implicit weakening, i.e. whenever a term is well-formed in some context, it is assumed to be well-formed (as it is) in extended contexts. We also assume that any specifically mentioned name is fresh, e.g. when we write $\Theta, \alpha : A$, we assume that α is fresh in Θ. Formally, we would use de Bruijn indices for variables, and define variable renaming and parallel substitution by recursion on presyntax, e.g. as in [Schäfer et al. 2015].
- Fixing any Θ metacontext, parallel substitutions of bound and defined variables form morphisms of a category, where the identity substitution id maps each variable to itself and composition $- \circ -$ is given by pointwise substitution. The action of parallel substitution on terms is functorial, i.e. $t[\sigma][\delta] \equiv t[\sigma \circ \delta]$ and $t[\mathrm{id}] \equiv t$, and typing is stable under substitution.
- Definitional equality is understood to be a congruence and an equivalence relation, which is respected by substitution and typing.
- We elide a number of well-formedness assumptions in rules. For instance, whenever a context appears in a rule, it is assumed to be well-formed. Likewise, whenever we have $\Theta|\Gamma \vdash t : A$, we assume that $\Theta|\Gamma \vdash A : \mathsf{U}$.

From now on, we will only consider well-formed core syntax, and unless otherwise mentioned, constructions on core syntax which respect definitional equality.

Alternatively, one could present the syntax as a generalized algebraic theory [Sterling 2019] or a quotient inductive-inductive type [Altenkirch and Kaposi 2016], in which case we would get congruences and quotienting for free, and we would also get a rich model theory for our syntax. However, it seems that there are a number of possible choices for giving an algebraic presentation of metacontexts, and existing works on algebraic presentations of dependent modal contexts (e.g. [Birkedal et al. 2018]) do not precisely cover the current use case. We leave this to future work, along with the investigation of elaboration from an algebraic perspective.

Metacontexts are used to record metavariables which are created during elaboration. In our case, metacontexts are simply a context prefix, and we have variables pointing into it. This corresponds to a particularly simple variant of *crisp type theory* [Licata et al. 2018], where we do not have modal type operators or functions with crisp ("meta") domain. The non-meta typing context additionally supports *defined variables*, which is used in the typing rule for **let**-definitions, and we have that any defined variable is equal to its definition. We mainly support this as a convenience feature in the surface language.

$\boxed{\Theta \vdash}$ *metacontext formation*

$\boxed{\Theta|\Gamma \vdash}$ *context formation*

$\boxed{\Theta|\Gamma \vdash t : A}$ *typing*

$\boxed{\Theta|\Gamma \vdash t \equiv u : A}$ *term equality*

METACON/EMPTY
$$\frac{}{\bullet \vdash}$$

METACON/BIND
$$\frac{\Theta \vdash \qquad \Theta \vdash A : \mathsf{U}}{\Theta, \alpha : A \vdash}$$

CON/EMPTY
$$\frac{\Theta \vdash}{\Theta|\bullet \vdash}$$

CON/BIND
$$\frac{\Theta|\Gamma \vdash \qquad \Theta|\Gamma \vdash A : \mathsf{U}}{\Theta|\Gamma,\ x : A \vdash}$$

CON/DEFINE
$$\frac{\Theta|\Gamma \vdash \qquad \Theta|\Gamma \vdash t : A}{\Theta|\Gamma,\ x : A = t \vdash}$$

METAVAR
$$\frac{}{\Theta_0,\ \alpha : A,\ \Theta_1|\Gamma \vdash \alpha : A}$$

BOUND-VAR
$$\frac{}{\Theta|\Gamma,\ x : A,\ \Delta \vdash x : A}$$

DEFINED-VAR
$$\frac{}{\Theta|\Gamma,\ x : A = t,\ \Delta \vdash x : A}$$

UNIVERSE
$$\frac{}{\Theta|\Gamma \vdash \mathsf{U} : \mathsf{U}}$$

LET
$$\frac{\Theta|\Gamma \vdash t : A \qquad \Theta|\Gamma,\ x : A = t \vdash u : B}{\Theta|\Gamma \vdash \mathbf{let}\, x : A = t\, \mathbf{in}\, u : B[x \mapsto t]}$$

FUN
$$\frac{\Theta|\Gamma \vdash A : \mathsf{U} \qquad \Theta|\Gamma,\ x : A \vdash B : \mathsf{U}}{\Theta|\Gamma \vdash (x : A) \to B : \mathsf{U}}$$

IMPLICIT-FUN
$$\frac{\Theta|\Gamma \vdash A : \mathsf{U} \qquad \Theta|\Gamma,\ x : A \vdash B : \mathsf{U}}{\Theta|\Gamma \vdash \{x : A\} \to B : \mathsf{U}}$$

APP
$$\frac{\Theta|\Gamma \vdash t : (x : A) \to B \qquad \Theta|\Gamma \vdash u : A}{\Theta|\Gamma \vdash t\, u : B[x \mapsto u]}$$

IMPLICIT-APP
$$\frac{\Theta|\Gamma \vdash t : \{x : A\} \to B \qquad \Theta|\Gamma \vdash u : A}{\Theta|\Gamma \vdash t\, \{u\} : B[x \mapsto u]}$$

LAM
$$\frac{\Theta|\Gamma,\ x : A \vdash t : B}{\Theta|\Gamma \vdash \lambda x.\, t : (x : A) \to B}$$

IMPLICIT-LAM
$$\frac{\Theta|\Gamma,\ x : A \vdash t : B}{\Theta|\Gamma \vdash \lambda\, \{x\}.\, t : \{x : A\} \to B}$$

FUN-β
$$\frac{\Theta|\Gamma,\ x : A \vdash t : B \qquad \Theta|\Gamma \vdash u : A}{\Theta|\Gamma \vdash (\lambda x.\, t)\, u \equiv t[x \mapsto u] : B[x \mapsto u]}$$

IMPLICIT-FUN-β
$$\frac{\Theta|\Gamma,\ x : A \vdash t : B \qquad \Theta|\Gamma \vdash u : A}{\Theta|\Gamma \vdash (\lambda\, \{x\}.\, t)\, \{u\} \equiv t[x \mapsto u] : B[x \mapsto u]}$$

FUN-η
$$\frac{\Theta|\Gamma \vdash t : (x : A) \to B}{\Theta|\Gamma \vdash (\lambda x.\, t\, x) \equiv t : (x : A) \to B}$$

IMPLICIT-FUN-η
$$\frac{\Theta|\Gamma \vdash t : \{x : A\} \to B}{\Theta|\Gamma \vdash (\lambda\, \{x\}.\, t\, \{x\}) \equiv t : \{x : A\} \to B}$$

DEFINITION
$$\frac{}{\Theta|\Gamma,\ x : A = t,\ \Delta \vdash x \equiv t : A}$$

Fig. 2. Selected rules of the core language.

The universe U is Russell-style, and we have the type-in-type rule. This causes our core syntax to be non-total, and our elaboration algorithm to be possibly non-terminating. We use type-in-type to simplify presentation, since consistent universe setups are orthogonal to the focus of this work.

Function types only differ from each other in notation: implicit functions have the same rules as "explicit" functions. The primary purpose of implicit function types is to *guide elaboration*: the elaborator will at times compute a type and branch on whether it is an implicit function.

Notation 1. Both in the surface and core syntax, we use Agda-like notation:

- We use $A \to B$ to refer to non-dependent functions.
- We group domain types together in functions, and omit function arrows, as in $\{A\,B : \mathsf{U}\}(x : A) \to B \to A$.
- We group multiple λ-s, as in $\lambda\,\{A\}\,\{B\}\,x\,y.\,x$.

Notation 2. We use a spine notation for neutral terms. A spine is a list of terms, noted as $\bar{t}$, where terms may be wrapped in brackets to signal implicit application. For example, if $\overline{u} \equiv (\{A\}, \{B\}, x)$, then $t\,\overline{u}$ denotes $t\,\{A\}\,\{B\}\,x$. In $t\,\overline{u}$, we call t the *head* of the neutral term. In particular, if t is a metavariable, the neutral term is *meta-headed.*

Example 2.1. The core syntax is quite expressive as a programming language, thanks to **let**-definitions[2] and the type-in-type rule which allows Church-encodings of a large class of inductive types. For example, the following term computes a list of types by mapping:

$$
\begin{aligned}
&\textbf{let}\,\mathit{List} : \mathsf{U} \to \mathsf{U}\\
&\quad = \lambda\,A.\,(L : \mathsf{U}) \to (A \to L \to L) \to L \to L\ \textbf{in}\\
&\textbf{let}\,\mathit{map} : \{A\,B : \mathsf{U}\} \to (A \to B) \to \mathit{List}\,A \to \mathit{List}\,B\\
&\quad = \lambda\,\{A\}\,\{B\}\,f\,\mathit{as}\,L\,\mathit{cons}\,\mathit{nil}.\,\mathit{as}\,L\,(\lambda\,a.\,\mathit{cons}\,(f\,a))\,\mathit{nil}\ \textbf{in}\\
&\mathit{map}\,\{\mathsf{U}\}\,\{\mathsf{U}\}\,(\lambda\,A.\,A \to A)\,(\lambda\,L\,\mathit{cons}\,\mathit{nil}.\,\mathit{cons}\,\mathsf{U}\,(\mathit{cons}\,\mathsf{U}\,\mathit{nil}))
\end{aligned}
$$

2.3 Metasubstitutions

Before we can move on to the description of the elaborator, we need to specify metasubstitutions. These are essentially just parallel substitutions of metacontexts, and their purpose is to keep track of meta-operations (e.g. fresh meta creation or solution of a meta).

- A metasubstitution $\boxed{\theta : \Theta_0 \Rightarrow \Theta_1}$ assigns to each variable in Θ_1 a term in Θ_0 , hence it is represented as a list of terms $(\alpha_1 \mapsto t_1,\ \ldots\ \alpha_i \mapsto t_i)$.
- We define the action of a metasubstitution on contexts and terms by recursion; we notate action on contexts as $\Gamma[\theta]$ and action on terms as $t[\theta]$. We remark that there is no abstraction for metavariables in the core syntax, so we do not have to handle variable capture (or index shifting).

[2]In dependent type theories, the **let** rule is not derivable from function application, unlike in simple type theories.

The following are admissible:

$$\frac{\Theta \vdash}{() : \Theta \Rightarrow \bullet}\ \textsc{metasub/empty} \qquad \frac{\theta : \Theta_0 \Rightarrow \Theta_1 \qquad \Theta_0|\bullet \vdash t : A[\theta]}{(\theta,\, \alpha \mapsto t) : \Theta_0 \Rightarrow (\Theta_1,\, \alpha : A)}\ \textsc{metasub/extended} \qquad \frac{\theta : \Theta_0 \Rightarrow \Theta_1 \qquad \Theta_1|\Gamma \vdash}{\Theta_0|\Gamma[\theta] \vdash}\ \textsc{metasub/con-action}$$

$$\frac{\theta : \Theta_0 \Rightarrow \Theta_1 \qquad \Theta_1|\Gamma \vdash t : A}{\Theta_0|\Gamma[\theta] \vdash t[\theta] : A[\theta]}\ \textsc{metasub/tm-action} \qquad \frac{}{\mathsf{id} : \Theta \Rightarrow \Theta}\ \textsc{metasub/identity} \qquad \frac{\theta_0 : \Theta_1 \Rightarrow \Theta_2 \qquad \theta_1 : \Theta_0 \Rightarrow \Theta_1}{\theta_0 \circ \theta_1 : \Theta_0 \Rightarrow \Theta_2}\ \textsc{metasub/composition}$$

$$\frac{}{\mathsf{p} : (\Theta,\, x : A) \Rightarrow \Theta}\ \textsc{metasub/weakening}$$

The identity substitution id maps each variable to itself. Composition is given by pointwise term substitution, id and $- \circ -$ yields a category, and the action of metasubstitution on contexts and terms is functorial. The weakening substitution p (the naming comes from categories-with-families terminology [Dybjer 1995]) can be defined as dropping the last entry from $\mathsf{id} : (\Theta,\, x : A) \Rightarrow (\Theta,\, x : A)$.

2.4 Fresh Metavariables

Using *contextual metavariables* is a standard practice in the implementation of dependently typed languages. This means that every "hole" in the surface language is represented as an unknown function which abstracts over all bound variables in the scope of a hole. Unlike [Nanevski et al. 2008] and similarly to [Gundry 2013], we do not have a first-class notion of contextual types, and instead reuse the standard dependent function type to abstract over enclosing contexts.

Definition 2.2 (Closing type). For each $\Theta|\Gamma \vdash A : \mathsf{U}$, we define $\Gamma \Rightarrow A$ by recursion on Γ, such that $\Theta|\bullet \vdash \Gamma \Rightarrow A : \mathsf{U}$.

$$\begin{aligned}
((\Gamma,\, x : A) \Rightarrow B) &:\equiv (\Gamma \Rightarrow ((x : A) \to B)) \\
((\Gamma,\, x : A = t) \Rightarrow B) &:\equiv (\Gamma \Rightarrow B[x \mapsto t]) \\
(\bullet \Rightarrow B) &:\equiv B
\end{aligned}$$

Definition 2.3 (Contextualization). For each $\Theta|\Gamma \vdash t : \Gamma \Rightarrow A$, we define the spine $\overline{\mathsf{vars}_\Gamma}$ such that that $\Theta|\Gamma \vdash t\,\overline{\mathsf{vars}_\Gamma} : A$. Informally, this is t applied to all bound variables in Γ.

$$\begin{aligned}
(t\,\overline{\mathsf{vars}_{\Gamma,\, x:A}}) &:\equiv (t\,\overline{\mathsf{vars}_\Gamma})\,x \\
(t\,\overline{\mathsf{vars}_{\Gamma,\, x:A=t}}) &:\equiv (t\,\overline{\mathsf{vars}_\Gamma}) \\
(t\,\overline{\mathsf{vars}_\bullet}) &:\equiv t
\end{aligned}$$

Example 2.4. If we have $\Gamma \equiv (\bullet,\, A : \mathsf{U},\, B : A \to \mathsf{U})$, then $(\Gamma \Rightarrow \mathsf{U}) \equiv ((A : \mathsf{U})(B : A \to \mathsf{U}) \to \mathsf{U})$ and $t\,\overline{\mathsf{vars}_\Gamma} \equiv t\,A\,B$.

Definition 2.5 (Fresh meta creation). We specify $\mathsf{freshMeta}_{\,\Theta|\Gamma}\,A$ as follows:

$$\frac{\Theta|\Gamma \vdash A : \mathsf{U}}{\mathsf{freshMeta}_{\,\Theta|\Gamma}\,A \in \{(\Theta',\, \theta,\, t) \mid (\theta : \Theta' \Rightarrow \Theta)\ \wedge\ (\Theta'|\Gamma[\theta] \vdash t : A[\theta])\}}$$

The definition $\mathsf{freshMeta}_{\,\Theta|\Gamma}\,A :\equiv ((\Theta,\, \alpha : \Gamma \Rightarrow A),\ \mathsf{p},\ \alpha\,\overline{\mathsf{vars}_\Gamma})$, where α is fresh in Θ, satisfies this specification. We extend Θ with a fresh meta, which has the closing type $\Gamma \Rightarrow A$. The p weakening relates the new metacontext to the old one, by "dropping" the new entry. Lastly, $\alpha\,\overline{\mathsf{vars}_\Gamma}$ is the new meta applied to all bound variables.

2.5 Implicit Argument Insertion

We define a helper function which inserts implicit applications around a core term. For example, if we have a defined name *id* with type $\{A : \mathsf{U}\} \to A \to A$ in a surface program, we usually want to expand *id* to *id* $\{\alpha\}$, where α is a fresh metavariable.

$$\frac{\mathit{ins} \in \{\mathsf{true},\ \mathsf{false}\} \qquad \Theta|\Gamma \vdash t : A}{\mathsf{insert}\ \mathit{ins}\ t\ A \in \{(\Theta',\ \theta\, t',\ A') \mid (\theta : \Theta' \Rightarrow \Theta)\ \wedge\ (\Theta'|\Gamma[\theta] \vdash t' : A')\}}$$

We have an additional $\mathit{ins} \in \{\mathsf{true},\ \mathsf{false}\}$ parameter, which simply toggles whether any insertion is to be performed. We will use this in the definition of elaboration. Insertion is defined by recursion on *ins* and the A type, as follows. Here, we notate Θ and Γ as subscripts but later we will leave them implicit. The clauses of the following definition are matched top-down.

$$\begin{array}{ll}
\mathsf{insert}_{\Theta|\Gamma}\ \mathsf{false}\ t\ A & :\equiv (\Theta,\ \mathsf{id},\ t,\ A) \\
\mathsf{insert}_{\Theta|\Gamma}\ \mathsf{true}\ t\ (\{x : A\} \to B) & :\equiv \mathbf{let}\ (\Theta',\ \theta,\ u) = \mathsf{freshMeta}_{\,\Theta|\Gamma}\ A \\
& \ \mathbf{in}\ \mathsf{insert}_{\Theta'|\Gamma[\theta']}\ \mathsf{true}\ ((t[\theta])\ \{u\})\ (B[\theta][x \mapsto u]) \\
\mathsf{insert}_{\Theta|\Gamma}\ \mathsf{true}\ t\ A & :\equiv (\Theta,\ \mathsf{id},\ t,\ A)
\end{array}$$

2.6 Unification

We assume that there is a unification procedure, which returns a unifying metasubstitution on success. We only have *homogeneous* unification, i.e. the two terms to be unified must have the same type. The specification is as follows:

$$\frac{\Theta|\Gamma \vdash t : A \qquad \Theta|\Gamma \vdash u : A}{\mathsf{unify}\ t\ u \in \{(\Theta',\ \theta) \mid (\theta : \Theta' \Rightarrow \Theta)\ \wedge\ (\Theta'|\Gamma[\theta] \vdash t[\theta] \equiv u[\theta] : A[\theta])\} \cup \{\mathsf{fail}\}}$$

For a simple example, assuming $\Theta :\equiv (\bullet,\ \alpha : \mathsf{U},\ \beta : \mathsf{U})$, $\mathsf{unify}\ \alpha\ (\beta \to \beta)$ yields $\Theta' :\equiv (\bullet,\ \beta : \mathsf{U})$ and the substitution $\theta :\equiv (\alpha \mapsto (\beta \to \beta),\ \beta \mapsto \beta)$, where $\theta : \Theta' \Rightarrow \Theta$.

Here, we do not require that unification returns most general unifiers, nor do we go into the details of how unification is implemented. Gundry describes unification in detail in [Gundry 2013, Chapter 4] for a similar syntax, with a similar (though more featureful) setup for metacontexts. See also [Abel and Pientka 2011] for a reference on unification. Note that our unification algorithm does not support *constraint postponing*, as we have not talked about constraints at all. In our concrete prototype implementation, unification supports basic pattern unification and metavariable pruning.

2.7 Elaboration

In this section we define the elaboration algorithm. First, we explain the used notations.

- We use a Haskell-like monadic pseudocode notation, where the side effect is failure via fail.
- We use pattern matching notation on core terms; e.g. we may match on whether a type is a function type. This assumes an evaluation/normalization procedure on core terms; but note that we already assume this in implicit argument insertion and unification.
- We abbreviate $\theta_1 \circ \theta_2$ as θ_{12}, $\theta_1 \circ \theta_2 \circ \theta_3$ as θ_{123} and analogously in other cases. We do this to reduce the visual noise caused by threading composed metasubstitutions everywhere in the elaboration algorithm.

Elaboration consists of two (partial) functions, checking and inferring, which are defined by mutual induction on surface syntax. They are specified as follows. We also have the additional $\mathit{ins} \in \{\mathsf{true},\ \mathsf{false}\}$ argument for inference, which toggles whether we perform implicit argument insertion on the output. We explain the cases of the definition in order.

CHECK

$$\frac{t \text{ is a surface expression} \qquad \Theta|\Gamma \vdash A : \mathsf{U}}{[\![t]\!]\Downarrow_{\Theta|\Gamma} A \in \{(\Theta', \theta, t') \mid (\theta : \Theta' \Rightarrow \Theta) \;\wedge\; (\Theta'|\Gamma[\theta] \vdash t' : A[\theta])\} \cup \{\mathsf{fail}\}}$$

INFER

$$\frac{ins \in \{\mathsf{true}, \mathsf{false}\} \qquad t \text{ is a surface expression} \qquad \Theta|\Gamma \vdash}{[\![t]\!]\Uparrow_{\Theta|\Gamma} ins \in \{(\Theta', \theta, t', A) \mid (\theta : \Theta' \Rightarrow \Theta) \;\wedge\; (\Theta'|\Gamma[\theta] \vdash t' : A)\} \cup \{\mathsf{fail}\}}$$

$$
\begin{array}{l}
[\![\lambda\, x.\, t]\!]\Downarrow_{\Theta|\Gamma} ((x : A) \to B) :\equiv \textbf{do} \\
\quad (\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\, x:A} B \\
\quad \textbf{return}\, (\Theta', \theta, \lambda\, x.\, t') \\
[\![\lambda\, \{x\}.\, t]\!]\Downarrow_{\Theta|\Gamma} (\{x : A\} \to B) :\equiv \textbf{do} \\
\quad (\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\, x:A} B \\
\quad \textbf{return}\, (\Theta', \theta, \lambda\, \{x\}.\, t') \\
[\![t]\!]\Downarrow_{\Theta|\Gamma} (\{x : A\} \to B) :\equiv \textbf{do} \\
\quad (\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\, x:A} B \\
\quad (\Theta', \theta, \lambda\, \{x\}.\, t') \\
[\![\textbf{let}\, x : A = t\, \textbf{in}\, u]\!]\Downarrow_{\Theta_0|\Gamma} B :\equiv \textbf{do} \\
\quad (\Theta_1, \theta_1, A') \leftarrow [\![A]\!]\Downarrow_{\Theta_0|\Gamma} \mathsf{U} \\
\quad (\Theta_2, \theta_2, t') \leftarrow [\![t]\!]\Downarrow_{\Theta_1|\Gamma[\theta_1]} A' \\
\quad (\Theta_3, \theta_3, u') \leftarrow [\![u]\!]\Downarrow_{\Theta_2|\Gamma[\theta_{12}]} (B[\theta_{12}]) \\
\quad \textbf{return}\, (\Theta_3, \theta_{123}, \textbf{let}\, x : A'[\theta_{23}] = t'[\theta_3]\, \textbf{in}\, u') \\
[\![\_]\!]\Downarrow_{\Theta|\Gamma} A :\equiv \textbf{do} \\
\quad \textbf{return}\, (\mathsf{freshMeta}_{\Theta|\Gamma}\, A) \\
[\![t]\!]\Downarrow_{\Theta_0|\Gamma} A :\equiv \textbf{do} \\
\quad (\Theta_1, \theta_1, t', B) \leftarrow [\![t]\!]\Uparrow_{\Theta_0|\Gamma} \mathsf{true} \\
\quad (\Theta_2, \theta_2) \leftarrow \mathsf{unify}\, (A[\theta_1])\, B \\
\quad \textbf{return}\, (\Theta_2, \theta_{12}, t'[\theta_2])
\end{array}
$$

$$
\begin{array}{l}
[\![x]\!]\Uparrow_{\Theta|\Gamma} ins :\equiv \textbf{do} \\
\quad \textbf{if}\, (\Gamma = (\Gamma_0, x : A, \Gamma_1)) \vee (\Gamma = (\Gamma_0, x : A = t, \Gamma_1)) \\
\quad\quad \textbf{then return}\, (\mathsf{insert}\, ins\, x\, A) \\
\quad\quad \textbf{else}\, \mathsf{fail} \\
[\![\mathsf{U}]\!]\Uparrow_{\Theta|\Gamma} ins :\equiv \textbf{do} \\
\quad \textbf{return}\, (\Theta, \mathsf{id}, \mathsf{U}, \mathsf{U}) \\
[\![(x : A) \to B]\!]\Uparrow_{\Theta_0|\Gamma} ins :\equiv \textbf{do} \\
\quad (\Theta_1, \theta_1, A') \leftarrow [\![A]\!]\Downarrow_{\Theta_1|\Gamma} \mathsf{U}
\end{array}
$$

$$
\begin{aligned}
&\quad (\Theta_2,\ \theta_2,\ B') \leftarrow \llbracket B \rrbracket \Downarrow\ _{\Theta_2\,|\,\Gamma[\theta_1],\ x:A'}\ \mathsf{U} \\
&\quad \mathbf{return}\ (\Theta_2,\ \theta_{12},\ ((x : A'[\theta_2]) \to B'),\ \mathsf{U}) \\
&\llbracket \{x : A\} \to B \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A') \leftarrow \llbracket A \rrbracket \Downarrow\ _{\Theta_1\,|\,\Gamma}\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ B') \leftarrow \llbracket B \rrbracket \Downarrow\ _{\Theta_2\,|\,\Gamma[\theta_1],\ x:A'}\ \mathsf{U} \\
&\quad \mathbf{return}\ (\Theta_2,\ \theta_{12},\ (\{x : A'[\theta_2]\} \to B'),\ \mathsf{U}) \\
&\llbracket \lambda\, x.\, t \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A) \leftarrow \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma}\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ t',\ B) \leftarrow \llbracket t \rrbracket \Uparrow\ _{\Theta_1\,|\,\Gamma[\theta_1],\ x:A}\ \mathsf{true} \\
&\quad \mathbf{return}\ (\Theta_2,\ \theta_{12},\ \lambda\, x.\, t',\ (x : A[\theta_2]) \to B) \\
&\llbracket \lambda\, \{x\}.\, t \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A) \leftarrow \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma}\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ t',\ B) \leftarrow \llbracket t \rrbracket \Uparrow\ _{\Theta_1\,|\,\Gamma[\theta_1],\ x:A}\ \mathsf{true} \\
&\quad \mathbf{return}\ (\mathsf{insert}\ \mathit{ins}\ (\lambda\, \{x\}.\, t')\ (\{x : A[\theta_2]\} \to B)) \\
&\llbracket t\ u \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ t',\ A) \leftarrow \llbracket t \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathsf{true} \\
&\quad (\Theta_2,\ \theta_2, A_0) \leftarrow \mathsf{freshMeta}_{\,\Theta_1\,|\,\Gamma[\theta_1]}\ \mathsf{U} \\
&\quad (\Theta_3,\ \theta_3, A_1) \leftarrow \mathsf{freshMeta}_{\,\Theta_2\,|\,\Gamma[\theta_{12}],\ x:A_0}\ \mathsf{U} \\
&\quad (\Theta_4,\ \theta_4) \leftarrow \mathsf{unify}\ (A[\theta_{23}])\ ((x : A_0[\theta_3]) \to A_1) \\
&\quad (\Theta_5,\ \theta_5,\ u') \leftarrow \llbracket u \rrbracket \Downarrow\ _{\Theta_4\,|\,\Gamma[\theta_{1234}]}\ (A_0[\theta_{34}]) \\
&\quad \mathbf{return}\ (\mathsf{insert}\ \mathit{ins}\ ((t'[\theta_{2345}])\ u')\ (A_1[\theta_{45}][x \mapsto u'])) \\
&\llbracket t\ \{u\} \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ t',\ A) \leftarrow \llbracket t \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathsf{false} \\
&\quad (\Theta_2,\ \theta_2, A_0) \leftarrow \mathsf{freshMeta}_{\,\Theta_1\,|\,\Gamma[\theta_1]}\ \mathsf{U} \\
&\quad (\Theta_3,\ \theta_3, A_1) \leftarrow \mathsf{freshMeta}_{\,\Theta_2\,|\,\Gamma[\theta_{12}],\ x:A_0}\ \mathsf{U} \\
&\quad (\Theta_4,\ \theta_4) \leftarrow \mathsf{unify}\ (A[\theta_{23}])\ (\{x : A_0[\theta_3]\} \to A_1) \\
&\quad (\Theta_5,\ \theta_5,\ u') \leftarrow \llbracket u \rrbracket \Downarrow\ _{\Theta_4\,|\,\Gamma[\theta_{1234}]}\ (A_0[\theta_{34}]) \\
&\quad \mathbf{return}\ (\mathsf{insert}\ \mathit{ins}\ ((t'[\theta_{2345}])\ \{u'\})\ (A_1[\theta_{45}][x \mapsto u'])) \\
&\llbracket \mathbf{let}\, x : A = t\ \mathbf{in}\ u \rrbracket \Uparrow\ _{\Theta_0\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A') \leftarrow \llbracket A \rrbracket \Downarrow\ _{\Theta_0\,|\,\Gamma}\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ t') \leftarrow \llbracket t \rrbracket \Downarrow\ _{\Theta_1\,|\,\Gamma[\theta_1]}\ A' \\
&\quad (\Theta_3,\ \theta_3,\ u',\ B) \leftarrow \llbracket u \rrbracket \Uparrow\ _{\Theta_2\,|\,\Gamma[\theta_{12}]}\ \mathit{ins} \\
&\quad \mathbf{return}\ (\Theta_3,\ \theta_{123},\ (\mathbf{let}\, x : A'[\theta_{23}] = t'[\theta_3]\ \mathbf{in}\, u'),\ B) \\
&\llbracket \_ \rrbracket \Uparrow\ _{\Theta\,|\,\Gamma}\ \mathit{ins} :\equiv \mathbf{do} \\
&\quad \mathbf{let}\ (\Theta',\ \theta,\ A) = \mathsf{freshMeta}_{\,\Theta\,|\,\Gamma}\ \mathsf{U} \\
&\quad \mathbf{return}\ (\mathsf{freshMeta}_{\,\Theta'\,|\,\Gamma[\theta]}\ A)
\end{aligned}
$$

2.7.1 Checking. The first two clauses are checking λ-s, where the expected type exactly matches the λ binders. Hence, we simply check under binders with $[\![t]\!] \Downarrow_{\Theta|(\Gamma,\, x:A)} B$, and wrap the resulting term in the appropriate (implicit or explicit) λ.

The third clause for $[\![t]\!] \Downarrow_{\Theta|\Gamma} (\{x : A\} \to B)$ is more interesting. Here, we are checking a surface term which is *not* a λ (this follows from our top-down pattern matching notation), with an implicit function expected type. Here, we check t in the extended $\Gamma,\ x : A$ context, and we insert a new implicit λ in the elaboration output. This is the only point where implicit λ-s are introduced by elaboration. Practically, this rule is commonly useful whenever we have a higher-order function where some arguments have implicit function type. For example, in the surface syntax, assume natural numbers, and an induction principle for them:

$$NatInd : \{P : Nat \to \mathsf{U}\} \to P\,\mathsf{zero} \to (\{n : Nat\} \to P\,n \to P\,(\mathsf{suc}\,n)) \to (n : Nat) \to P\,n$$

Then, define addition using induction:

$$\begin{aligned} &\mathbf{let}\, NatPlus : Nat \to Nat \to Nat \\ &\qquad = NatInd\,(\lambda\,n.\,\mathsf{Nat} \to Nat)\,(\lambda\,m.\,m)\,(\lambda\,f\,m.\,\mathsf{suc}\,(f\,m))\ \mathbf{in}\ \ldots \end{aligned}$$

When the above is elaborated, the $\lambda\,f\,m.\,\mathsf{suc}\,(f\,m)$ function is checked with the expected type $\{n : Nat\} \to (Nat \to Nat) \to (Nat \to Nat)$, and the elaboration output is $\lambda\,\{n\}\,f\,m.\,\mathsf{suc}\,(f\,m)$. Hence, in this case we do not have to write implicit λ in the surface syntax.

For $[\![\mathbf{let}\,x : A = t\,\mathbf{in}\,u]\!] \Downarrow_{\Theta_0|\Gamma} B$, we simply let checking fall through. The definition here is a bit noisy, because we need to thread metasubstitutions through, and we always have to "update" core terms and contexts with the current metasubstitution.

For $[\![\_]\!] \Downarrow_{\Theta|\Gamma} A$, we return a fresh metavariable with the expected type. In any other $[\![t]\!] \Downarrow_{\Theta_0|\Gamma} A$ case, we have a *change of direction*: we infer a type for t, then unify the expected and inferred types.

2.7.2 Inferring. For $[\![x]\!] \Uparrow_{\Theta|\Gamma} ins$, we look up the type of x in Γ, and insert implicit applications if needed. In the case of U, we always succeed and infer U as type. In the cases for function types, we check that the domains and codomains have type U.

For λ-s, we create a fresh meta for the domain type (since our surface λ-s are not annotated), and infer types for the bodies. In the case of $[\![\lambda\,\{x\}.\,t]\!] \Uparrow_{\Theta_0|\Gamma} ins$, we additionally perform insert o n the output.

The $[\![t\,u]\!] \Uparrow_{\Theta_0|\Gamma} ins$ and $[\![t\,\{u\}]\!] \Uparrow_{\Theta_0|\Gamma} ins$ cases are again interesting. Here, we first infer a type for the term which is being applied, then refine the type to a function type, and lastly check the argument with the domain type. Note the difference between the explicit and implicit case. In the former case, we use $[\![t]\!] \Uparrow_{\Theta_0|\Gamma} \mathsf{true}$, which inserts implicit applications. In the latter case we do no insertion. This, together with the inference definition for variables, ensures that implicit applications in the surface syntax behave similarly as in Agda. For example, given $id : \{A : \mathsf{U}\} \to A \to A$ in scope, we elaborate $id\,\mathsf{U}$ as follows:

(1) The expression is an explicit application, so we infer id and insert implicit arguments, returning $id\,\{\alpha\}$, where α is a fresh meta.
(2) We check that U has type α. Here we immediately change direction, inferring U as type for U and unifying α with U.
(3) Hence, the resulting output is $id\,\{\mathsf{U}\}\,\mathsf{U}$.

On the other hand, we elaborate $id\,\{\mathsf{U}\}$ as follows:

(1) This is an implicit application, so we infer a type for id without inserting implicit arguments. This yields the inferred type $\{A : \mathsf{U}\} \to A \to A$, and we check that U has type U.
(2) This changes direction again, and we infer U type for U, and successfully unify U with U.

Note that it would be more efficient (and also allow more user-friendly error messages) to not immediately refine the inferred type of t to a function type, but rather match on the inferred type, and only perform refining when the type is meta-headed. We present the unoptimized version here for the sake of brevity.

In the case of **let**, inference again just falls through, and we infer a type for the **let** body. For $[\![\_]\!] \Uparrow_{\Theta|\Gamma} ins$, we create a fresh meta for the type of the hole, and another fresh meta for the hole itself.

2.7.3 Starting and finishing elaboration. Given a surface term t, we initiate elaboration by computing $[\![t]\!] \Uparrow_{\bullet|\bullet}$ true. If this succeeds, we get a (Θ, θ, t') result. Elaboration is successful overall if $\Theta = \bullet$ and $\theta = \mathrm{id}$, i.e. no unsolved metas remain.

2.7.4 Properties of elaboration. First, elaboration is *sound* in the sense that it only produces well-formed output.

Theorem 2.6 (Soundness). *The definitions of $[\![-]\!] \Downarrow$ and $[\![-]\!] \Uparrow$ conform to the* check *and* infer *specifications. This follows by induction on surface syntax, while also relying on the properties of substitution, metasubstitution,* insert*,* unify *and* freshMeta*.* □

We remark that this notion of soundness is only a "sanity" or well-typing statement for elaboration. In fact, we could define elaboration as a constantly failing partial function, and it would also conform to the specification. The right way to view this, is that $[\![-]\!] \Downarrow$ and $[\![-]\!] \Uparrow$ together with their specification constitute the semantics of surface syntax. We do not give any other semantics to the surface syntax, nor does it support any other operation.

We do not present any *completeness* result for elaboration in this paper. For an example of what this would entail, in [Dunfield and Krishnaswami 2013], completeness means that whenever there is a way to fill in missing details in the surface syntax, algorithmic typechecking *always* finds it. In ibid. this means figuring out domain types for λ-s and inserting all implicit applications. However, our elaborator targets a far stronger theory, and it is beyond our reach to succinctly characterize which annotations are inferable in the surface language.

We can still say something about the behavior of our elaborator. For this, we consider a translation from core terms to surface terms, the obvious forgetful translation, which maps core terms to surface counterparts. Now, this is an "evil" construction on core terms, since it does not preserve definitional equality. We shall only use this evil notion in the following statement.

Theorem 2.7 (Conservativity). *Elaboration is conservative over the surface syntax, in the sense that for any surface term t, if checking or inference outputs t', then the forgetful translation of t' differs from t only by*

- *Having all _ holes filled in by expressions.*
- *Having extra implicit λ-s and implicit applications inserted.*

This follows by straightforward induction on surface syntax. □

2.7.5 Omitted features.

- *Let-generalization.* This is an open research topic in settings with dependent types, and we make no attempt at covering it. See [Eisenberg 2016] for a treatment in a proposed dependent version of Haskell.
- *Polymorphic subtyping.* In some prior works, e.g. in [Dunfield and Krishnaswami 2013; Vytiniotis et al. 2008], there is a subtyping relation arising along instantiations of polymorphic

types. In GHC 8 polymorphic subtyping is implemented for function types only. Polymorphic subtyping complicates type inference, and to our knowledge it has not been implemented in any dependently typed setting. We also believe that it is undesirable in dependent settings, because elaboration of subtyping must insert coercions which significantly change the intensional character of programs. For example, if we have covariant list types, then coercing $t : \mathit{List}\,(\{A : \mathsf{U}\} \to A \to A)$ to $t : \mathit{List}\,(\mathit{Bool} \to \mathit{Bool})$ requires mapping over t and inserting implicit applications to Bool for each list element. In System F, all such coercions are erasible, since types are computationally irrelevant, but in our core syntax we have implicit functions with arbitrary (relevant) domains. In GHC, subtyping coercions for functions change operational semantics; this is a reason for abandoning subtyping in recent developments of impredicative inference for GHC [Serrano et al. 2020].

3 ISSUES WITH FIRST-CLASS IMPLICIT FUNCTIONS

We revisit now the *polyList* example from Section 1. We assume the following:

$$
\begin{aligned}
\mathit{List} &: \mathsf{U} \to \mathsf{U} \\
\mathit{nil} &: \{A : \mathsf{U}\} \to \mathit{List}\,A \\
\mathit{cons} &: \{A : \mathsf{U}\} \to A \to \mathit{List}\,A
\end{aligned}
$$

In the following, we present a trace of checking $\mathit{cons}\,(\lambda\,x.\,x)\,\mathit{nil}$ at type $\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A)$. We omit context and metacontext parameters everywhere, and notate recursive calls by indentation. We also omit some checking, inference, implicit insertion and unification calls which are not essential for illustration.

$$
\begin{array}{l}
[\![\mathit{cons}\,(\lambda\,x.\,x)\,\mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A)) \\
\quad [\![\mathit{cons}\,(\lambda\,x.\,x)\,\mathit{nil}]\!] \Uparrow\ \mathsf{true} \\
\quad\quad [\![\mathit{cons}\,(\lambda\,x.\,x)]\!] \Uparrow\ \mathsf{true} \\
\quad\quad\quad [\![\mathit{cons}]\!] \Uparrow\ \mathsf{true} \\
\quad\quad\quad\ = \mathit{cons}\,\{\alpha_0\}\ :\ \alpha_0 \to \mathit{List}\,\alpha_0 \to \mathit{List}\,\alpha_0 \\
\quad\quad\quad [\![\lambda\,x.\,x]\!] \Downarrow\ \alpha_0 \\
\quad\quad\quad\ = \lambda\,x.\,x \\
\quad\quad\ = \mathit{cons}\,\{\alpha_1 \to \alpha_1\}\,(\lambda\,x.\,x) : \mathit{List}\,(\alpha_1 \to \alpha_1) \to \mathit{List}\,(\alpha_1 \to \alpha_1) \\
\quad\quad [\![\mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\alpha_1 \to \alpha_1)) \\
\quad\quad\ = \mathit{nil}\,\{\alpha_1 \to \alpha_1\} \\
\quad\ = \mathit{cons}\,\{\alpha_1 \to \alpha_1\}\,(\lambda\,x.\,x)\,(\mathit{nil}\,\{\alpha_1 \to \alpha_1\}) : \mathit{List}\,(\alpha_1 \to \alpha_1) \\
\quad \mathsf{unify}\,(\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A))\,(\mathit{List}\,(\alpha_1 \to \alpha_1)) \\
\quad\quad \mathsf{unify}\,(\{A : \mathsf{U}\} \to A \to A)\,(\alpha_1 \to \alpha_1) \\
\quad\quad\ = \mathsf{fail}
\end{array}
$$

Above, we first infer $\mathit{cons}\,(\lambda\,x.\,x)\,\mathit{nil}$, which inserts implicit applications to fresh metas in *cons* and *nil*, and returns $\mathit{cons}\,\{\alpha_1 \to \alpha_1\}\,(\lambda\,x.\,x)\,(\mathit{nil}\,\{\alpha_1 \to \alpha_1\}) : \mathit{List}\,(\alpha_1 \to \alpha_1)$. Here, the α_0 meta is refined to $\alpha_1 \to \alpha_1$ when we check $\lambda\,x.\,x$. In the end, we need to unify the expected and inferred types, which fails, since we have an implicit function type on one side and an explicit function on the other side.

Why does this fail? The culprit is line 5, where we call $[\![\lambda\,x.\,x]\!]\Downarrow\,\alpha_0$. At this point, the checking type is not an implicit function type (it is a meta), so we do not insert an implicit λ. At the heart of the issue is that elaboration makes insertion choices based on core types.

(1) $[\![t]\!]\Downarrow_{\Theta|\Gamma} A$ can insert a λ only if A is an implicit function type.
(2) $\text{insert true}\,A$ inserts an application only if A is an implicit function type.

In both of these cases, if A is of the form $\alpha\,\overline{u}$ (i.e. meta-headed), then it is possible that α is later refined to an implicit function, but at that point we have already missed our shot at implicit insertion.

At least for λ-insertion, there is a potential solution: just *postpone* checking a term until the shape of the checking type is known for sure. This was included as part of a proposed solution for smarter λ-insertions in [Johansson and Lloyd 2015]. This means that checking with a meta-headed type returns a "guarded constant" [Norell 2007, Chapter 3], an opaque stand-in which only computes to an actual core term when the checking type becomes known. In practice, this solution has a painful drawback: *we get no information at all from checked terms before the guard is unblocked.* For an example for unexpected behavior with this solution, let us assume $\mathit{Bool} : \mathsf{U}$ and $\mathit{true} : \mathit{Bool}$, and try to infer type for the following surface term:

$$\textbf{let}\,x : \_ = \mathit{true}\,\textbf{in}\,x$$

We first insert a fresh meta α for the hole, and then check true with α. We postpone this checking, returning a guarded constant, and then infer a type for x, which is α. Hence, this small example yields an unsolved meta and a guarded constant in the output.

Now, this particular example can be repaired by special-casing the elaboration of a **let**-definition without an explicit type annotation. However, the current author's experience from playing with an implementation of this solution, is that we are missing too much information by postponing, and this cascades in an unfortunate way: postponing yields more unsolved metas, which cause more postponing.

4 TELESCOPES AND STRICTLY CURRIED FUNCTIONS

As part of the proposed solution, we extend the core theory with telescopes and strictly curried functions. Figure 3 lists the typing rules and definitional equalities.

4.1 Telescopes

Telescopes can be viewed as a generic implementation of record types. We have Tel as the type of telescopes. Elements of Tel are right-nested telescopes of types, with ϵ denoting the empty telescope, and $- \triangleright -$ telescope extension. For example, we can define the signature of natural number algebras as follows:

$$\textbf{let}\,\mathit{NatAlgSig} : \mathsf{Tel} = (N : \mathsf{U}) \triangleright (\mathit{zero} : N) \triangleright (\mathit{suc} : N \to N) \triangleright \epsilon\,\textbf{in}\,\ldots$$

We interpret an $A : \mathsf{Tel}$ as a record type as $\mathsf{Rec}\,A$, which behaves as the evident iterated Σ-type corresponding to the telescope. Hence, $\mathsf{Rec}\,\epsilon$ is isomorphic to the unit type, with inhabitant $[]$, and $\mathsf{Rec}\,((x : A) \triangleright B)$ behaves as a Σ-type, with pairing constructor $- :: -$ and projections π_1 and π_2. We also have the β and η rules for record constructors and projections in Figure 3. We present definitional equalities in a compact form, but note that they still stand for $\boxed{\Theta|\Gamma \vdash t \equiv u : A}$ judgments. Hence, the sides of the equations must have the same types, and in particular the left side of the $[]$-η rule has type $\mathsf{Rec}\,\epsilon$.

Our telescopes and records are derivable from natural numbers, the unit type and Σ-types. We use Agda-like pattern matching notation in the following. First, we define length-indexed telescopes.

$$\begin{aligned}
&\mathsf{Tel}' : \mathsf{Nat} \to \mathsf{U}\\
&\mathsf{Tel}'\,\mathsf{zero} \quad :\equiv \top\\
&\mathsf{Tel}'\,(\mathsf{suc}\,n) :\equiv \Sigma(A : \mathsf{U}).\,(A \to \mathsf{Tel}'\,n)
\end{aligned}$$

Then, we have $\mathsf{Tel} :\equiv \Sigma(n : \mathsf{Nat}).\,(\mathsf{Tel}'\,n)$, and define records:

$$\begin{aligned}
&\mathsf{Rec} : \mathsf{Tel} \to \mathsf{U}\\
&\mathsf{Rec}\,(\mathsf{zero},\,_) \qquad :\equiv \top\\
&\mathsf{Rec}\,(\mathsf{suc}\,n,\,(A,\,B)) :\equiv \Sigma(a : A).\,(\mathsf{Rec}\,(n,\,B\,a))
\end{aligned}$$

From the above, ϵ, $- \triangleright -$, $- :: -$ and $[]$ are evident, and all expected equalities hold definitionally. Derivability is good news because we inherit nice properties of the type theory which only contains the base type formers. For instance, if we have a consistent universe setup[3], we inherit consistency, canonicity and decidability of conversion. We currently use native telescopes instead of Nat, $\top$ and Σ because in unification and elaboration it is convenient that we are able to restrict some types to records types.

4.2 Strictly Curried Functions

These are function types whose domains are telescopes, and they are immediately computed to iterated implicit function types when the domain telescope is canonical. See FUN-ϵ and FUN-$\triangleright$: a curried function with empty domain computes to simply the codomain, while a function with a non-empty domain computes to an implicit function type. We explicitly notate telescopes in both λ-abstractions and applications for strictly curried functions, since they are relevant in the computation rules.

Curried function types tend to be computed away, but they can persist if the domain telescope is neutral, and in particular when it is meta-headed. For example, assuming a meta α : Tel, the type $\{x : \overline{\alpha}\} \to B$ cannot be computed further. During elaboration, we will use strictly curried function types to represent unknown insertions, but these types are eventually computed away if a surface expression can be successfully elaborated. Since the surface language remains unchanged, telescopes and curried functions are merely an internal implementation detail from the perspective of programmers.

Curried functions are *mostly* derivable from Nat, $\top$ and Σ. The type former is defined as follows:

$$\begin{aligned}
&\Pi^C : (A : \mathsf{Tel}) \to (\mathsf{Rec}\,A \to \mathsf{U}) \to \mathsf{U}\\
&\Pi^C(\mathsf{zero},\,_)\,B \qquad :\equiv B\,\mathsf{tt}\\
&\Pi^C(\mathsf{suc}\,n,\,(A,\,B))\,C :\equiv \{a : A\} \to \Pi^C\,(n,\,B\,a)(\lambda\,b.\,C\,(a,\,b))
\end{aligned}$$

With this, we can also define $\mathsf{app} : \Pi^C\,A\,B \to (a : \mathsf{Rec}\,A) \to B\,a$ and $\mathsf{lam} : ((a : \mathsf{Rec}\,A) \to B\,a) \to \Pi^C\,A\,B$, and all equations in Figure 3 hold definitionally, except CURRIED-β and CURRIED-η. These do not hold strictly, because Π^C, app and lam are all defined by recursion on the A telescope, but the $\beta\eta$ rules are specified generically for an arbitrary (possibly neutral) telescope. CURRIED-β is still provable as a propositional equality, and assuming function extensionality CURRIED-η is provable as well. For details, see our Agda formalization of these definitions, which is included alongside the prototype implementation.

[3] Recall that we currently assume type-in-type, which causes consistency and normalization to fail.

$$\textsc{Tel}\ \frac{}{\Theta|\Gamma \vdash \mathsf{Tel} : \mathsf{U}} \qquad \textsc{Empty-Tel}\ \frac{}{\Theta|\Gamma \vdash \epsilon : \mathsf{Tel}} \qquad \textsc{Nonempty-Tel}\ \frac{\Theta|\Gamma \vdash A : \mathsf{U} \qquad \Theta|\Gamma,\, x : A \vdash B : \mathsf{Tel}}{\Theta|\Gamma \vdash (x : A) \triangleright B : \mathsf{Tel}}$$

$$\textsc{Record-Type}\ \frac{\Theta|\Gamma \vdash A : \mathsf{Tel}}{\Theta|\Gamma \vdash \mathsf{Rec}\, A : \mathsf{U}} \qquad \textsc{Empty-Record}\ \frac{}{\Theta|\Gamma \vdash [] : \mathsf{Rec}\, \epsilon} \qquad \textsc{Nonempty-Record}\ \frac{\Theta|\Gamma \vdash t : A \qquad \Theta|\Gamma \vdash u : \mathsf{Rec}\, (B[x \mapsto t])}{\Theta|\Gamma \vdash t :: u : \mathsf{Rec}\, ((x : A) \triangleright B)}$$

$$\textsc{Record-Projection}\ \frac{\Theta|\Gamma \vdash t : \mathsf{Rec}\, ((x : A) \triangleright B)}{\Theta|\Gamma \vdash \pi_1\, t : A \qquad \Theta|\Gamma \vdash \pi_2\, t : \mathsf{Rec}\, (B[x \mapsto \pi_1\, t])} \qquad \textsc{Curried-Fun}\ \frac{\Theta|\Gamma \vdash A : \mathsf{Tel} \qquad \Theta|\Gamma,\, x : \mathsf{Rec}\, A \vdash B : \mathsf{U}}{\Theta|\Gamma \vdash \{x : \overline{A}\} \to B : \mathsf{U}}$$

$$\textsc{Curried-Lam}\ \frac{\Theta|\Gamma,\, x : \mathsf{Rec}\, A \vdash t : B}{\Theta|\Gamma \vdash \lambda\, \{x : \overline{A}\}.\, t : \{x : \overline{A}\} \to B} \qquad \textsc{Curried-App}\ \frac{\Theta|\Gamma \vdash t : \{x : \overline{A}\} \to B \qquad \Theta|\Gamma \vdash u : \mathsf{Rec}\, A}{\Theta|\Gamma \vdash t\, \{u : \overline{A}\} : B[x \mapsto u]}$$

$$\begin{array}{lll}
\pi_1\text{-}\beta & \pi_1\, (t :: u) & \equiv t \\
\pi_2\text{-}\beta & \pi_2\, (t :: u) & \equiv u \\
::\text{-}\eta & (\pi_1\, t :: \pi_2\, t) & \equiv t \\
[]\text{-}\eta & t & \equiv [] \\
\textsc{fun-}\epsilon & \{x : \overline{\epsilon}\} \to B & \equiv B[x \mapsto []] \\
\textsc{fun-}\triangleright & \{x : \overline{(y : A) \triangleright B}\} \to C & \equiv \{y : A\} \to (\{b : \overline{B}\} \to C[x \mapsto (y :: b)]) \\
\textsc{lam-}\epsilon & \lambda\, \{x : \overline{\epsilon}\}.\, t & \equiv t[x \mapsto []] \\
\textsc{lam-}\triangleright & \lambda\, \{x : \overline{(y : A) \triangleright B}\}.\, t & \equiv \lambda\{y\}.\, \lambda\, \{b : \overline{B}\}.\, t[x \mapsto (y :: b)] \\
\textsc{app-}\epsilon & t\, \{u : \overline{\epsilon}\} & \equiv t \\
\textsc{app-}\triangleright & t\, \{u : \overline{(x : A) \triangleright B}\} & \equiv t\, \{\pi_1\, u\}\, \{\pi_2\, u : \overline{B[x \mapsto \pi_1\, u]}\} \\
\textsc{curried-}\beta & \lambda\, (\{x : \overline{A}\}.\, t)\, \{u : \overline{A}\} & \equiv t[x \mapsto u] \\
\textsc{curried-}\eta & \lambda\, \{x : \overline{A}\}.\, t\, \{x : \overline{A}\} & \equiv t
\end{array}$$

Fig. 3. Rules for telescopes and strictly curried functions.

Hence, we can derive a somewhat weaker version of curried functions, with propositional β and η. From this, we still get consistency almost for free. That is because whenever we have a model of the base theory with Nat, $\top$ and Σ, such that the model also validates equality reflection, then every propositionally provable equation is also validated as a definitional equation. And fortunately for us, standard models which prove consistency usually validate equality reflection, e.g. set-theoretical models or standard models in extensional type theory.

In contrast, showing canonicity, normalization and decidability of conversion would require some extra work. We leave this to future work, but we expect that it is straightforward to extend previous proofs to cover strict β and η for curried functions.

5 EXTENDING ELABORATION

We shall utilize the extended core theory to implement smarter elaboration. Recall from Section 3 that the old elaborator makes two kinds of unforced insertion choices:

(1) $[\![t]\!] \Downarrow_{\Theta|\Gamma} A$ does not insert an implicit λ when A is meta-headed.
(2) $\mathsf{insert\ true}\, A$ does not insert an implicit application if A is meta-headed.

In the following, we shall only enhance λ-insertions. This allows a simple implementation which only requires minimal changes to unification, and which is already remarkably powerful. It seems that handling implicit application insertions requires extending unification; we discuss this in Section **??**. First, we modify closing types and contextualization to take advantage of telescopes.

Definition 5.1 (Closing types). We use curried function types to close over record types in the scope. If a bound variable does not have a record type, then we do as before[4]. We prepend the following clause to Definition 2.2:

$$((\Gamma,\, x : \mathsf{Rec}\, A) \Rightarrow B) :\equiv (\Gamma \Rightarrow (\{x : \overline{A}\} \to B))$$

Definition 5.2 (Contextualization). We extend our spine notation to applications of curried functions. For example, we may have a spine $\bar{t} \equiv (\{x : \overline{A}\}\,\{y : \overline{B}\})$. We accordingly revise Definition 2.3 for $\overline{\mathsf{vars}_\Gamma}$ so that we use curried function application for each record type in Γ.

5.1 Handling Superfluous Implicit Functions

Before we can move on to unification, checking and inference, we have to address a curious issue. Assuming $Bool : \mathsf{U}$, $true : Bool$ and $false : Bool$, consider the following surface expression:

$$\mathbf{let}\, x : \_ = true\, \mathbf{in}\, x$$

What should this expression elaborate to? We would expect the result to be simply

$$\mathbf{let}\, x : Bool = true\, \mathbf{in}\, x$$

However, there are infinitely many core terms which are conservative over the surface expression in the sense of Theorem 2.7. That is, we can wrap definitions with any number of implicit λ-s, and add implicit applications accordingly to usage sites of the defined name. For example, we could have

$$\mathbf{let}\, x : \{y : Bool\} \to Bool = \lambda\,\{y\}.\, true\, \mathbf{in}\, x\,\{true\}$$

This is clearly undesirable. With the type $\{y : Bool\} \to Bool$, the implicit argument y is never inferable, because the codomain type does not depend on the domain, and the argument is never constrained. Hence, with the above definition, we always have to write $x\,\{true\}$ or $x\,\{false\}$ when we want to use x. In order to avoid such nonsense, we adopt the following principle: *elaboration should never invent non-dependent implicit function types.*

This was a non-issue in the old elaborator, because it was not able to invent implicit function types; it was only utilizing the type annotations present in the surface input. In the case of $\mathbf{let}\, x : \_ = true$, the old elaborator checks $true$ with a fresh meta, and just assumes that the meta does not stand for an implicit function type.

[4] This implies that we close over meta-headed types using plain functions. In theory, this causes a higher-order version of the basic implicit insertion problem: we are uncertain about whether we should be uncertain about implicit insertions. So far, this higher-order insertion problem seems to seems to be irrelevant in practice, in the prototype implementation.

$$\textsc{metacon/constancy}\quad \frac{\Theta|\Gamma,\, x : \mathsf{Rec}\, A \vdash B : \mathsf{U}}{\Theta,\, \mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B \vdash}$$

$$\textsc{constancy-}\equiv\quad \frac{x \notin \mathsf{FreeVars}(B)}{\Theta_0,\, \mathsf{constancy}_{\Gamma_0,\, x:\mathsf{Rec}\, A}\, B,\, \Theta_1|\Gamma_1 \vdash A \equiv \epsilon : \mathsf{Tel}}$$

$$\textsc{metasub/constancy}\quad \frac{\theta : \Theta_0 \Rightarrow \Theta_1 \qquad x \notin \mathsf{FreeVars}(B[\theta]) \text{ implies } \Theta_0|\Gamma[\theta] \vdash A[\theta] \equiv \epsilon : \mathsf{Tel}}{(\theta,\, \mathsf{solve}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B) : \Theta_0 \Rightarrow (\Theta_1,\, \mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B)}$$

$$\textsc{metasub/weaken-constancy}\quad \frac{}{\mathsf{p} : (\Theta,\, \mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B) \Rightarrow \Theta}$$

Fig. 4. Rules for constancy constraints

5.1.1 Constancy constraints. We use these constraints to get rid of curried function types as soon as we learn that they are non-dependent. They are constraints in the usual sense in unification algorithms (e.g. as in [Abel and Pientka 2011] or [Vytiniotis et al. 2011]). We formalize them in a compact way, by adding a new kind of context extension for metacontexts. The rules are given in Figure 4.

In the rule metacon/constancy we specify extension of a metacontext with a constraint. The constancy-≡ rule expresses that, assuming we have a constancy constraint for A and B in context, if B does not depend on the $x : \mathsf{Rec}\, A$ domain variable, then A is equal to the the empty telescope ϵ.

The metasub/constancy rule defines metasubstitutions whose codomains are extended with constraints. Intuitively, while the metasub/extended rule from Section 2.3 can be used to solve a metavariable (by mapping it to a term), metasub/constancy solves a constraint. We can only extend $\theta : \Theta_0 \Rightarrow \Theta_1$ to map into an additional constraint if θ forces the constraint to hold. In metasub/weaken-constancy, we overload p for the weakening substitution which drops a constraint.

Definition 5.3 (Creating a new constraint). We do this similarly to Definition 2.5, by simply returning a weakening substitution.

$$\mathsf{newConstancy}_{\Theta|\Gamma,\, x:\mathsf{Rec}\, A}\, B :\equiv ((\Theta,\, \mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B),\, \mathsf{p})$$

5.1.2 Algorithmic implementation of constraint solving. The above specification for constancy constraints is compact but not particularly algorithmic: we just magically get new definitional equalities whenever we have constraints in contexts. In our prototype implementation, we augment the unification procedure with eager removal of solvable constraints.

After solving a meta α during unification, which yields a unifying θ substitution, we review all $(\mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B)$ constraints in the context, such that x occurs in B inside a $\bar{t}$ spine of some $\alpha\, \bar{t}$ term. In other words, we review constraints where the new meta solution might make a difference.

(1) If we have $x \in \mathsf{FreeVars}(B[\theta])$, where x occurs rigidly in $B[\theta]$, i.e. the occurrence is not in a spine of a meta, then no metasubstitution can possibly remove this occurrence. In this case the constraint holds vacuously, so we can use the rule metasub/constancy to return a θ' substitution which also solves the constraint.
(2) If we have $x \notin \mathsf{FreeVars}(B[\theta])$, we recursively unify $A[\theta]$ with ϵ. If that succeeds, we get a θ' which unifies A and ϵ and thus forces the constraint to hold, so we can again use metasub/constancy to solve the constraint.
(3) In any other case we simply return θ and keep the constraint around.

Remark. In the case with $x \in \mathsf{FreeVars}(B[\theta])$, it would be also sound to solve the constraint when the occurrence is not rigid. However, this way we could lose potential non-ϵ solutions of A.

5.2 Unification For Strictly Curried Functions

Although we omit most details of unification, we shall discuss it for curried functions, as it is essential in the extended elaboration algorithm. The most interesting case is when we unify a curried function type with an implicit function type. In this case, we learn that the domain of the curried function is non-empty, so we refine the A domain to an extended $(x_0 : A_0) \triangleright A_1$ telescope. Since we invent a fresh A_1 domain for a curried function type, we need to add a constancy constraint for it as well.

$$
\begin{aligned}
&\mathsf{unify}_{\Theta_0\,|\,\Gamma}(\{x : \overline{A}\} \to B)\,(\{x_0 : A_0\} \to B') :\equiv \mathbf{do}\\
&\quad (\Theta_1,\, \theta_1,\, A_1) \leftarrow \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma,\, x_0 : A_0}\, \mathsf{Tel}\\
&\quad (\Theta_2,\, \theta_2) \leftarrow \mathsf{unify}_{\Theta_1\,|\,\Gamma[\theta_1]}\,(A[\theta_1])\,((x : A_0[\theta_1]) \triangleright A_1)\\
&\quad (\Theta_3,\, \theta_3) \leftarrow \mathsf{newConstancy}_{\Theta_2\,|\,\Gamma[\theta_{12}],\, x_0 : A_0[\theta_{12}],\, x_1 : \mathsf{Rec}\,(A_1[\theta_2])}\,(B[\theta_{12}][x \mapsto (x_0 :: x_1)])\\
&\quad \mathsf{unify}_{\Theta_3\,|\,\Gamma[\theta_{123}],\, x_0 : A_0[\theta_{123}]}\,(\{x_1 : A_1[\theta_{23}]\} \to B[\theta_{123}][x \mapsto (x_0 :: x_1)])\,B'
\end{aligned}
$$

We have the symmetric $\mathsf{unify}_{\Theta_0\,|\,\Gamma}(\{x_0 : A_0\} \to B')\,(\{x : \overline{A}\} \to B)$ case the same way as above.

Now, let us assume that B' is not an implicit function type and not meta-headed. Then, we have the following case, where we solve a telescope domain to be empty.

$$
\begin{aligned}
&\mathsf{unify}_{\Theta_0\,|\,\Gamma}(\{x : \overline{A}\} \to B)\,B' :\equiv \mathbf{do}\\
&\quad (\Theta_1,\, \theta_1) \leftarrow \mathsf{unify}_{\Theta_0\,|\,\Gamma}\,A\,\epsilon\\
&\quad \mathsf{unify}_{\Theta_1\,|\,\Gamma[\theta_1]}\,(B[\theta_1][x \mapsto []])\,(B'[\theta_1])
\end{aligned}
$$

Again, we also have the symmetric case. For $\lambda\,\{x : \overline{A}\}.\,t$ and $t\,\{u : \overline{A}\}$, unification is structural, and other cases remain the same as in the the basic elaborator of Section 2.

5.3 Elaboration

REFERENCES

Andreas Abel and Brigitte Pientka. 2011. Higher-order dynamic pattern unification for dependent types and records. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 10–26.

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 18–29. https://doi.org/10.1145/2837614.2837638

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M Pitts, and Bas Spitters. 2018. Modal dependent type theory and dependent right adjoints. *arXiv preprint arXiv:1804.05236* (2018).

Joshua Dunfield and Neelakantan R Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. *ACM SIGPLAN Notices* 48, 9 (2013), 429–442.

Peter Dybjer. 1995. Internal type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 120–134.

Richard A Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice*. University of Pennsylvania.

Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types*. Ph.D. Dissertation. University of Strathclyde.

Marcus Johansson and Jesper Lloyd. 2015. *Eliminating the problems of hidden-lambda insertion-Restricting implicit arguments for increased predictability of type checking in a functional programming language with depending types*. Master's thesis.

Daan Leijen. 2008. HMF: Simple type inference for first-class polymorphism. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 283–294.

Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 66–77.

Daniel R Licata, Ian Orton, Andrew M Pitts, and Bas Spitters. 2018. Internal universes in models of homotopy type theory. *arXiv preprint arXiv:1801.07664* (2018).

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 1–49.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *International Conference on Interactive Theorem Proving*. Springer, 359–374.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. (January 2020). https://www.microsoft.com/en-us/research/publication/a-quick-look-at-impredicativity/ In submission.

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 783–796.

Jonathan Sterling. 2019. Algebraic type theory and universe hierarchies. *arXiv preprint arXiv:1902.08848* (2019).

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn (X) Modular type inference with local assumptions. *Journal of functional programming* 21, 4-5 (2011), 333–412.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*. 251–262.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-class polymorphism for Haskell. *ACM Sigplan Notices* 43, 9 (2008), 295–306.