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 GHC 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 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 new 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:

$$id : \{A : \mathsf{Set}\} \to A \to A \qquad \mathsf{Definition}\ id\ \{A : \mathsf{Type}\}(x : A) := x.$$
$$id\ x = x$$

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

$$id :: \mathsf{forall}\ (a :: *).\ a \to a$$
$$id\ x = x$$

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 $list\,(\mathsf{forall}\,\{A : \mathsf{Type}\},\, A \to A)$ for a list type of polymorphic functions, the brackets here are simply ignored by Coq and we get a plain function type. For example, Coq accepts the following definition:

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

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:

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

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 $List\,(\{A : \mathsf{Set}\} \to A \to A)$ 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:

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

Agda 2.6.0.1 does not accept this. However, it does accept $polyList = (\lambda\, \{A\}\, x \to x) :: []$. 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 $List\,(\{A : \mathsf{Set}\} \to A \to A)$ 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 [**?**]. The above issue in Agda is a specific instance of the challenges described in the mentioned works. However, none of the proposed algorithms have landed so far in production compilers, for reasons of complexity, fragility and interaction with other language features.

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. The extensions to the target theory are modest and are derivable from W-types. 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 historically coincided, but has since diverged to the point of being 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 $(\mathsf{forall}\ (a :: *).\ a \to a) :: *$.

2 BASIC 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 illustrate the key issues, and we extend it in Section TODO with additional rules.

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 minimal 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 $\mathbf{let}\, x : \_ = \mathsf{U}\, \mathbf{in}\, x$.

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 only constructions which respect definitional equality. In other words, we quotient well-formed syntax by 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], TODO) do not precisely cover the current use case. We leave this to future work, along with the investigation of elaboration from an algebraic perspective.

how to refer to mathpartir rules from text?

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, but we do not support contextual modality e.g. as in [Nanevski et al. 2008]. The non-meta typing context additionally supports *defined variables*, which is used in the typing rule for *let-definitions*, and we have

$\boxed{\Theta \vdash}$ *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
$$\frac{\Theta | \Gamma \vdash \qquad \Theta | \Gamma \vdash t : A}{\Theta | \Gamma, x : A = t \vdash}$$

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

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

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

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

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

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

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

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

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

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

TM/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.

that any defined variable is equal to its definition. We mainly support this as a convenience feature in the implementation of our elaborator.

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 (Spines). We sometimes use a spine notation for neutral terms. A spine is a list of terms, noted as $\overline{t}$, 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 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 over the list $\mathit{cons}\,\mathsf{U}\,(\mathit{cons}\,\mathsf{U}\,\mathit{nil})$.

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

In the core syntax, all applications must be explicit, so using implicit function type for *map* above does not make much difference in notation.

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).

The following are admissible:

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

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

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

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 $\{A : \mathsf{U}\} \to A \to A$ occurring in the surface program, we usually want to expand *id* to $id\,\{\alpha\}$, where α is a fresh metavariable.

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

We have an additional $ins \in \{\mathsf{true}, \mathsf{false}\}$ 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.

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

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:

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

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.

example for meta solution, more references, describe at least pattern unif and pruning briefly

Note that our unification algorithm does not support *constraint postponing*, since we have not specified constraints. In our concrete implementation, unification supports basic higher-order 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 requires an evaluation/normalization procedure on core terms, but we already assume this in unification.

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 $ins \in \{\mathsf{true}, \mathsf{false}\}$ argument for inference, which toggles whether the output has inserted implicit arguments. We explain the cases of the definition in order.

CHECKING

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

INFERRING

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

$[\![\lambda\, x.\, t]\!]\Downarrow\ \Theta\,\Gamma\,((x : A) \to B) :\equiv \mathbf{do}$
  $(\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow\ \Theta\,(\Gamma, x : A)\,B$
  $\mathbf{return}\ (\Theta', \theta, \lambda\, x.\, t')$
$[\![\lambda\, \{x\}.\, t]\!]\Downarrow\ \Theta\,\Gamma\,(\{x : A\} \to B) :\equiv \mathbf{do}$
  $(\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow\ \Theta\,(\Gamma, x : A)\,B$
  $\mathbf{return}\ (\Theta', \theta, \lambda\, \{x\}.\, t')$
$[\![t]\!]\Downarrow\ \Theta\,\Gamma\,(\{x : A\} \to B) :\equiv \mathbf{do}$
  $(\Theta', \theta, t') \leftarrow [\![t]\!]\Downarrow\ \Theta\,(\Gamma, x : A)\,B$
  $(\Theta', \theta, \lambda\, \{x\}.\, t')$
$[\![\mathbf{let}\, x : A = t\, \mathbf{in}\, u]\!]\Downarrow\ \Theta_0\,\Gamma\,B :\equiv \mathbf{do}$
  $(\Theta_1, \theta_1, A') \leftarrow [\![A]\!]\Downarrow\ \Theta_0\,\Gamma\,\mathsf{U}$
  $(\Theta_2, \theta_2, t') \leftarrow [\![t]\!]\Downarrow\ \Theta_1\,(\Gamma[\theta_1])\,A'$
  $(\Theta_3, \theta_3, u') \leftarrow [\![u]\!]\Downarrow\ \Theta_2\,(\Gamma[\theta_1 \circ \theta_2])\,(B[\theta_1 \circ \theta_2])$
  $\mathbf{return}\ (\Theta_3, \theta_1 \circ \theta_2 \circ \theta_3, \mathbf{let}\, x : A'[\theta_2 \circ \theta_3] = t'[\theta_3]\, \mathbf{in}\, u')$
$[\![\_]\!]\Downarrow\ \Theta\,\Gamma\,A :\equiv \mathbf{do}$
  $\mathbf{return}\ (\mathsf{freshMeta}\,\Theta\,\Gamma\,A)$
$[\![t]\!]\Downarrow\ \Theta_0\,\Gamma\,A :\equiv \mathbf{do}$
  $(\Theta_1, \theta_1, t', B) \leftarrow [\![t]\!]\Uparrow\ \mathsf{true}\,\Theta_0\,\Gamma$
  $(\Theta_2, \theta_2) \leftarrow \mathsf{unify}\,(A[\theta_1])\,B$
  $\mathbf{return}\ (\Theta_2, \theta_1 \circ \theta_2, t'[\theta_2])$

$[\![x]\!]\Uparrow\ ins\,\Theta\,\Gamma :\equiv \mathbf{do}$
  $\mathbf{if}\ (x : A \in \Gamma)\ \vee\ (x : A = t \in \Gamma)$
    $\mathbf{then}\ \mathbf{return}\ (\mathsf{insert}\, ins\,\Theta\,\Gamma\,x\,A)$
    $\mathbf{else}\ \mathsf{fail}$
$[\![\mathsf{U}]\!]\Uparrow\ ins\,\Theta\,\Gamma :\equiv \mathbf{do}$
  $\mathbf{return}\ (\Theta, \mathsf{id}, \mathsf{U}, \mathsf{U})$
$[\![(x : A) \to B]\!]\Uparrow\ ins\,\Theta_0\,\Gamma :\equiv \mathbf{do}$
  $(\Theta_1, \theta_1, A') \leftarrow [\![A]\!]\Downarrow\ \Theta_1\,\Gamma\,\mathsf{U}$

$$
\begin{aligned}
&\quad (\Theta_2,\ \theta_2,\ B') \leftarrow [\![B]\!]\Downarrow\ \Theta_2\ (\Gamma[\theta_1],\ x : A')\ \mathsf{U} \\
&\quad \textbf{return}\ (\Theta_2,\ \theta_1 \circ \theta_2,\ ((x : A'[\theta_2]) \to B'),\ \mathsf{U}) \\
&[\![\{x : A\} \to B]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\equiv \textbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A') \leftarrow [\![A]\!]\Downarrow\ \Theta_1\ \Gamma\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ B') \leftarrow [\![B]\!]\Downarrow\ \Theta_2\ (\Gamma[\theta_1],\ x : A')\ \mathsf{U} \\
&\quad \textbf{return}\ (\Theta_2,\ \theta_1 \circ \theta_2,\ (\{x : A'[\theta_2]\} \to B'),\ \mathsf{U}) \\
&[\![\lambda\, x.\, t]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\equiv \textbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A) \leftarrow \mathsf{freshMeta}\ \Theta\ \Gamma\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ t',\ B) \leftarrow [\![t]\!]\Uparrow\ \mathsf{true}\ \Theta_1\ (\Gamma[\theta_1],\ x : A) \\
&\quad \textbf{return}\ (\Theta_2,\ \theta_1 \circ \theta_2,\ \lambda\, x.\, t',\ (x : A[\theta_2]) \to B) \\
&[\![\lambda\, \{x\}.\, t]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\equiv \textbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ A) \leftarrow \mathsf{freshMeta}\ \Theta\ \Gamma\ \mathsf{U} \\
&\quad (\Theta_2,\ \theta_2,\ t',\ B) \leftarrow [\![t]\!]\Uparrow\ \mathsf{true}\ \Theta_1\ (\Gamma[\theta_1],\ x : A) \\
&\quad \textbf{return}\ (\mathsf{insert}\ ins\ \Theta_2\ (\Gamma[\theta_1 \circ \theta_2])\ (\lambda\, \{x\}.\, t')\ (\{x : A[\theta_2]\} \to B)) \\
&[\![t\ u]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\equiv \textbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ t',\ A) \leftarrow [\![t]\!]\Uparrow\ \mathsf{true}\ \Theta_0\ \Gamma \\
&\quad \textbf{case}\ A\ \textbf{of} \\
&\quad\quad ((x : A) \to B) \Rightarrow \textbf{do} \\
&\quad\quad\quad (\Theta_2,\ \theta_2,\ u') \leftarrow [\![u]\!]\Downarrow\ \Theta_1\ (\Gamma[\theta_1])\ A \\
&\quad\quad\quad \textbf{return}\ (\mathsf{insert}\ ins\ \Theta_2\ (\Gamma[\theta_1 \circ \theta_2])\ (t'[\theta_2]\ u')\ (B[\theta_2][x \mapsto u'])) \\
&\quad\quad A \Rightarrow \mathsf{fail} \\
&[\![t\ \{u\}]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\equiv \textbf{do} \\
&\quad (\Theta_1,\ \theta_1,\ t',\ A) \leftarrow [\![t]\!]\Uparrow\ \mathsf{false}\ \Theta_0\ \Gamma \\
&\quad \textbf{case}\ A\ \textbf{of} \\
&\quad\quad (\{x : A\} \to B) \Rightarrow \textbf{do} \\
&\quad\quad\quad (\Theta_2,\ \theta_2,\ u') \leftarrow [\![u]\!]\Downarrow\ \Theta_1\ (\Gamma[\theta_1])\ A \\
&\quad\quad\quad \textbf{return}\ (\mathsf{insert}\ ins\ \Theta_2\ (\Gamma[\theta_1 \circ \theta_2])\ (t'[\theta_2]\ \{u'\})\ (B[\theta_2][x \mapsto u'])) \\
&\quad\quad A \Rightarrow \mathsf{fail} \\
&[\![\textbf{let}\ x : A = t\ \textbf{in}\ u]\!]\Uparrow\ ins\ \Theta_0\ \Gamma :\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',\ B) \leftarrow [\![u]\!]\Uparrow\ ins\ \Theta_2\ (\Gamma[\theta_1 \circ \theta_2]) \\
&\quad \textbf{return}\ (\Theta_3,\ \theta_1 \circ \theta_2 \circ \theta_3,\ (\textbf{let}\ x : A'[\theta_2 \circ \theta_3] = t'[\theta_3]\ \textbf{in}\ u'),\ B) \\
&[\![\_]\!]\Uparrow\ ins\ \Theta\ \Gamma :\equiv \textbf{do} \\
&\quad \textbf{let}\ (\Theta',\ \theta,\ A) = \mathsf{freshMeta}\ \Theta\ \Gamma\ \mathsf{U} \\
&\quad \textbf{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 expressions. Hence, we simply check under binders with $[\![t]\!] \Downarrow \Theta\,\Gamma\,(x : A)\,B$, and wrap the resulting term in the appropriate 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:

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

Then, define addition using induction:

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

When the above is elaborated, the $\lambda\,f\,m.\,\mathsf{suc}\,(f\,m)$ function is checked with the expected type $\{n : \mathsf{Nat}\} \to (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$, 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 $[\![\textbf{let}\ x : A = t\ \textbf{in}\ u]\!] \Downarrow \Theta_0\,\Gamma\,B$, 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 $[\![x]\!] \Uparrow \mathit{ins}\,\Theta\,\Gamma$, 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 the bodies. In the case of $[\![\lambda\,\{x\}.\,t]\!] \Uparrow \mathit{ins}\,\Theta_0\,\Gamma$, we additionally perform insert o n t he output.

explain this weird case

The $[\![t\,u]\!] \Uparrow \mathit{ins}\,\Theta_0\,\Gamma$ and $[\![t\,\{u\}]\!] \Uparrow \mathit{ins}\,\Theta_0\,\Gamma$ cases are again interesting. Here, we first infer a type for the term which is being applied, and only proceed if the type is an appropriate function. Note a difference between the explicit and implicit case. In the former case, we use $[\![t]\!] \Uparrow \mathsf{true}\,\Theta_0\,\Gamma$, which inserts implicit applications. In the latter case we do not insert implicit applications. This, together with the inference definition for variables, ensures that implicit applications in the surface syntax behave similarly as in Agda. For example, given $\mathit{id} : \{A : \mathsf{U}\} \to A \to A$ in scope, we elaborate $\mathit{id}\,\mathsf{U}$ as follows:

(1) The expression is an explicit application, so we infer id and insert implicit arguments, returning $\mathit{id}\,\{\alpha\}$, 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 $\mathit{id}\,\{\mathsf{U}\}\,\mathsf{U}$.

On the other hand, we elaborate $\mathit{id}\,\{\mathsf{U}\}$ 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.

explain that we fail if t has meta-headed type or handle it in elab

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

2.7.3 On soundness, completeness, and other properties. We briefly discuss what we can and cannot say about the presented elaborator. First, elaboration is *sound* in the sense that it never produces malformed core syntax.

Theorem 2.6 (Soundness of elaboration). *The definitions of* $[\![-]\!]\Downarrow$ *and* $[\![-]\!]\Uparrow$ *conform to the specification* **??**. *This follows by induction on surface syntax and analyzing the definitions, while also relying on properties of substitution, metasubstitution,* 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. □

Say something about termination of elab?

Also, note that we do not support *let-generalization*. This is an open research topic in settings with dependent types, and we make no attempt at tackling it. See [Eisenberg 2016] for a treatment in a proposed dependent version of Haskell.

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.

$$\begin{array}{ll} 0 & [\![\mathit{cons}\,(\lambda\, x.\, x)\,\mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A)) \\ 1 & \quad [\![\mathit{cons}\,(\lambda\, x.\, x)\,\mathit{nil}]\!] \Uparrow\ \mathsf{true} \\ 2 & \quad\quad [\![\mathit{cons}\,(\lambda\, x.\, x)]\!] \Uparrow\ \mathsf{true} \\ 3 & \quad\quad\quad [\![\mathit{cons}]\!] \Uparrow\ \mathsf{true} \\ 4 & \quad\quad\quad = \mathit{cons}\,\{\alpha_0\}\ :\ \alpha_0 \to \mathit{List}\,\alpha_0 \to \mathit{List}\,\alpha_0 \\ 5 & \quad\quad\quad [\![\lambda\, x.\, x]\!] \Downarrow\ \alpha_0 \\ 6 & \quad\quad\quad = \lambda\, x.\, x \\ 7 & \quad\quad = \mathit{cons}\,\{\alpha_1 \to \alpha_1\}\,(\lambda\, x.\, x) : \mathit{List}\,(\alpha_1 \to \alpha_1) \to \mathit{List}\,(\alpha_1 \to \alpha_1) \\ 8 & \quad\quad [\![\mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\alpha_1 \to \alpha_1)) \\ 9 & \quad\quad = \mathit{nil}\,\{\alpha_1 \to \alpha_1\} \\ 10 & \quad = \mathit{cons}\,\{\alpha_1 \to \alpha_1\}\,(\lambda\, x.\, x)\,(\mathit{nil}\,\{\alpha_1 \to \alpha_1\}) : \mathit{List}\,(\alpha_1 \to \alpha_1) \\ 11 & \quad \mathsf{unify}\,(\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A))\,(\mathit{List}\,(\alpha_1 \to \alpha_1)) \\ 12 & \quad\quad \mathsf{unify}\,(\{A : \mathsf{U}\} \to A \to A)\,(\alpha_1 \to \alpha_1) \\ 13 & \quad\quad = \mathsf{fail} \end{array}$$

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) insert true $\Theta\,\Gamma\,A$ 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 $Bool : \mathsf{U}$ and $true : Bool$, and try to infer type for the following surface term:

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

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

REFERENCES

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.

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.

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