Elaboration with First-Class Implicit Function Types[*]

ANDRÁS KOVÁCS, Eötvös Loránd University, Hungary

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 solution for elaborating first-class implicit functions, which is applicable to full dependent type theories and compares favorably to prior solutions in terms of power, generality and 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.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Polymorphism**.

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

ACM Reference Format:
András Kovács. 2020. Elaboration with First-Class Implicit Function Types. *Proc. ACM Program. Lang.* 4, ICFP, Article 101 (August 2020), 29 pages. https://doi.org/10.1145/3408983

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 systems such as Agda have gotten quite adept at this task. However, in practice, programmers still have to tell the compiler 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:

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

[*]The author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Author's address: András Kovács, Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary, kovacsandras@inf.elte.hu.

2475-1421/2020/8-ART101
https://doi.org/10.1145/3408983

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 [Eisenberg et al. 2016] and write *id* @*Bool True*.

Implicit function types 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 implicit function type, instead, implicitness is tied to particular *names*, and while we can write *list* (forall {A : 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 forall {A : 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 [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 : Set} $\to A \to A$) without issues. 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 *polyList* = (λ {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 : 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 λ {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

[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.

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. Currently none of the above solutions are supported 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 works support dependent types, which is a key point in our work.

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 accumulate information about implicit insertions. Our algorithm is based on Norell's bidirectional elaborator for Agda [Norell 2007, Chapter 3].
- In the System F fragment, the presented elaborator is comparable or superior to previous solutions for impredicative inference. However, it also supports full dependent type theory. Our inference is also global, i.e. it can consider the whole program and not just particular n-ary applications.
- We provide an executable implementation of the elaborator described in this paper.
- Our solution is simple: we implemented elaboration, evaluation and unification in about 680 lines of Haskell, of which about 230 lines implement the novel enhancements on the top of Norell's basic elaborator.
- The target theory of elaboration serves as a general platform for elaborating implicit function types: our concrete elaborator is a relatively simple one, and there is room to further develop it.

1.2 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 $(\mathsf{forall}\ (a :: *).\ a \to a) :: *$.

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. Our main goal with the elaborator is to have a setup which is as simple as possible, but sufficient to illustrate the enhancements in Section 5, which could be used to conservatively extend elaboration in existing dependently typed languages. Hence, we do not have more fine-grained control over metavariable insertion, nor the advanced unification algorithms used in Agda [Abel and Pientka 2011]; these are orthogonal improvements to the presented system.

t, u, v, A, B, C	$::=$	x	variable
	$\mid$	$(x : A) \to B$	function type
	$\mid$	$\{x : A\} \to B$	implicit function type
	$\mid$	$t\,u$	application
	$\mid$	$t\,\{u\}$	implicit application
	$\mid$	$\lambda\,x.\,t$	lambda abstraction
	$\mid$	$\lambda\,\{x\}.\,t$	implicit abstraction
	$\mid$	U	universe
	$\mid$	$\textbf{let}\,x : A = t\,\textbf{in}\,u$	let-definition
	$\mid$	$\_$	hole for inferred term

Fig. 1. Syntax of the surface language.

2.1 Surface Syntax

Figure 1 shows 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 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.

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

$\boxed{\Theta \vdash}$	*metacontext formation*
$\boxed{\Theta \vert \Gamma \vdash}$	*context formation*
$\boxed{\Theta \vert \Gamma \vdash t : A}$	*typing*
$\boxed{\Theta \vert \Gamma \vdash t \equiv u : A}$	*term equality*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Selected rules of the core language.

any defined variable is equal to its definition. We support **let** as a convenience feature which is important for defining more involved example programs[2].

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. Consistent universe setups are straightforward to devise, but they are orthogonal to the focus of this work, and we choose to avoid needless noise.

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 type.

We avoid a fully formal presentation in this paper. For examples of what such a presentation looks like, see [Abel et al. 2018] or [Wieczorek and Biernacki 2018]. Some notes on what is elided:

- 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. $t[\sigma \circ \delta] \equiv t[\sigma][\delta]$ and $t[\mathsf{id}] \equiv t$, 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, presented constructions on the core syntax 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. 2020]) do not precisely cover the current use case. We leave this to future work, along with the investigation of elaboration from an algebraic perspective.

Notations. We use Agda-like syntactic sugar both in the surface syntax and in core syntax.

- 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 $\lambda\,\{A\}\,\{B\}\,x\,y.\,x$.

Definition 2.1 (Spines). We 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.*

[2]In presentations of dependent type theories without explicit substitutions, the **let** rule is not derivable from function application, unlike in simple type theories.

$$\boxed{\theta : \Theta_0 \Rightarrow \Theta_1} \qquad \textit{metasubstitution formation}$$

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

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

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

Fig. 3. Admissible rules for metasubstitutions.

Example 2.2. The core syntax is quite expressive as a programming language, thanks to **let**-definitions and the type-in-type rule which allows Church encodings for 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 $\theta : \Theta_0 \Rightarrow \Theta_1$ assigns to each variable in Θ_1 a term in Θ_0, hence it is represented as a list of terms $(\alpha_1 \mapsto t_1,\ \dots\ \alpha_i \mapsto t_i)$.
- We define the action of a metasubstitution on contexts and terms by recursion; we write 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).
- We list some of the admissible rules in Figure 3, which are relevant in the rest of the paper.

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 $\mathsf{id} : (\Theta,\ \alpha : A) \Rightarrow (\Theta,\ \alpha : A)$.

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 2.3 (Closing type). For each $\Theta|\Gamma \vdash A : \mathsf{U}$, we define a term $\Gamma \Rightarrow A$ 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 2.4 (Contextualization). For each $\Theta|\bullet \vdash t : \Gamma \Rightarrow A$, we define the spine $\overline{\mathsf{vars}_\Gamma}$ such that that $\Theta|\Gamma \vdash t\,\overline{\mathsf{vars}_\Gamma} : A$, which is t applied to all bound variables in Γ.

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

Example 2.5. 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 2.6 (Fresh meta creation). We define $\mathsf{freshMeta}_{\,\Theta|\Gamma}\,A$ as follows:

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

In the definition above, 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 fresh meta applied to all bound variables.

2.5 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 [2013, Chapter 4] describes unification in detail 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.

2.6 Elaboration

Elaboration consists of two (partial) functions, checking and inferring, which are defined by mutual induction on surface syntax. We also have helper functions for implicit argument insertion, which can be used as a post-processing step after inference. First, about 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 unification. Patterns are matched in top-down order.
- We abbreviate $\theta_1 \circ \theta_2$ as θ_{12} and $\theta_1 \circ \theta_2 \circ \theta_3$ 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.

We present the specifications and definitions below, then we describe them in order.

$$\textsc{Insertion}\ \frac{\Theta_0|\Gamma \vdash \qquad (\Theta,\,\theta,\,t,\,A) \in \{(\Theta,\,\theta,\,t,\,A)\,|\,(\theta : \Theta \Rightarrow \Theta_0)\ \wedge\ (\Theta|\Gamma[\theta] \vdash t : A)\} \cup \{\mathsf{fail}\}}{\begin{array}{l}\mathsf{insert'}(\Theta,\,\theta,\,t,\,A) \in \{(\Theta,\,\theta,\,t,\,A)\,|\,(\theta : \Theta \Rightarrow \Theta_0)\ \wedge\ (\Theta|\Gamma[\theta] \vdash t : A)\} \cup \{\mathsf{fail}\}\\ \mathsf{insert}(\Theta,\,\theta,\,t,\,A)\ \in \{(\Theta,\,\theta,\,t,\,A)\,|\,(\theta : \Theta \Rightarrow \Theta_0)\ \wedge\ (\Theta|\Gamma[\theta] \vdash t : A)\} \cup \{\mathsf{fail}\}\end{array}}$$

```
$\mathsf{insert'}\ (\Theta_1,\,\theta_1,\,t,\,\{x : A\} \to B)$ :≡ $\mathbf{let}\ (\Theta_2,\,\theta_2,\,u) = \mathsf{freshMeta}_{\Theta_1|\Gamma[\theta_1]}\,A$
                                       $\mathbf{in}\ \mathsf{insert'}\ (\Theta_2,\,\theta_{12},\,(t[\theta_2])\,\{u\},\,B[\theta_2][x \mapsto u])$
$\mathsf{insert'}\ (\Theta_1,\,\theta_1,\,t,\,A)$            :≡ $\mathbf{return}\ (\Theta_1,\,\theta_1,\,t,\,A)$
$\mathsf{insert'}\ \mathsf{fail}$                        :≡ $\mathsf{fail}$
$\mathsf{insert}\ (\Theta,\,\theta,\,(\lambda\{x\}.t),\,A)$         :≡ $\mathbf{return}\ (\Theta,\,\theta,\,(\lambda\{x\}.t),\,A)$
$\mathsf{insert}\ (\Theta,\,\theta,\,t,\,A)$                :≡ $\mathsf{insert'}\ (\Theta,\,\theta,\,t,\,A)$
$\mathsf{insert}\ \mathsf{fail}$                         :≡ $\mathsf{fail}$
```

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

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

```
$[\![\lambda x.\,t]\!]\Downarrow_{\Theta|\Gamma} ((x : A) \to B)$ :≡ $\mathbf{do}$
    $(\Theta',\,\theta,\,t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\,x:A} B$
    $\mathbf{return}\ (\Theta',\,\theta,\,\lambda x.\,t')$
$[\![\lambda\,\{x\}.\,t]\!]\Downarrow_{\Theta|\Gamma} (\{x : A\} \to B)$ :≡ $\mathbf{do}$
    $(\Theta',\,\theta,\,t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\,x:A} B$
    $\mathbf{return}\ (\Theta',\,\theta,\,\lambda\,\{x\}.\,t')$
$[\![t]\!]\Downarrow_{\Theta|\Gamma} (\{x : A\} \to B)$ :≡ $\mathbf{do}$
    $(\Theta',\,\theta,\,t') \leftarrow [\![t]\!]\Downarrow_{\Theta|\Gamma,\,x:A} B$
```

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

$[\![x]\!]\Uparrow_{\,\Theta\,|\,\Gamma} :\equiv$ do
  if $(\Gamma \equiv (\Gamma_0,\ x : A,\ \Gamma_1))\ \vee\ (\Gamma \equiv (\Gamma_0,\ x : A = t,\ \Gamma_1))$
    then return $(\Theta,\ \mathsf{id},\ x,\ A)$
    else fail
$[\![\mathsf{U}]\!]\Uparrow_{\,\Theta\,|\,\Gamma} :\equiv$ do
  return $(\Theta,\ \mathsf{id},\ \mathsf{U},\ \mathsf{U})$
$[\![(x : A) \to B]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma} :\equiv$ do
  $(\Theta_1,\ \theta_1,\ A') \leftarrow [\![A]\!]\Downarrow_{\,\Theta_0\,|\,\Gamma}\,\mathsf{U}$
  $(\Theta_2,\ \theta_2,\ B') \leftarrow [\![B]\!]\Downarrow_{\,\Theta_1\,|\,\Gamma[\theta_1],\ x:A'}\,\mathsf{U}$
  return $(\Theta_2,\ \theta_{12},\ ((x : A'[\theta_2]) \to B'),\ \mathsf{U})$
$[\![\{x : A\} \to B]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma} :\equiv$ do
  $(\Theta_1,\ \theta_1,\ A') \leftarrow [\![A]\!]\Downarrow_{\,\Theta_1\,|\,\Gamma}\,\mathsf{U}$
  $(\Theta_2,\ \theta_2,\ B') \leftarrow [\![B]\!]\Downarrow_{\,\Theta_2\,|\,\Gamma[\theta_1],\ x:A'}\,\mathsf{U}$
  return $(\Theta_2,\ \theta_{12},\ (\{x : A'[\theta_2]\} \to B'),\ \mathsf{U})$
$[\![t\,u]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma} :\equiv$ do
  $(\Theta_1,\ \theta_1,\ t',\ A) \leftarrow \mathsf{insert}'\,([\![t]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma})$
  let $(\Theta_2,\ \theta_2, A_0) = \mathsf{freshMeta}_{\,\Theta_1\,|\,\Gamma[\theta_1]}\,\mathsf{U}$
  let $(\Theta_3,\ \theta_3, A_1) = \mathsf{freshMeta}_{\,\Theta_2\,|\,\Gamma[\theta_{12}],\ x:A_0}\,\mathsf{U}$
  $(\Theta_4,\ \theta_4) \leftarrow \mathsf{unify}\,(A[\theta_{23}])\,((x : A_0[\theta_3]) \to A_1)$
  $(\Theta_5,\ \theta_5,\ u') \leftarrow [\![u]\!]\Downarrow_{\,\Theta_4\,|\,\Gamma[\theta_{1234}]}\,(A_0[\theta_{34}])$
  return $(\Theta_5,\ \theta_{12345},\ (t'[\theta_{2345}])\,u',\ A_1[\theta_{45}][x \mapsto u'])$
$[\![t\,\{u\}]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma} :\equiv$ do
  $(\Theta_1,\ \theta_1,\ t',\ A) \leftarrow [\![t]\!]\Uparrow_{\,\Theta_0\,|\,\Gamma}$
  let $(\Theta_2,\ \theta_2, A_0) = \mathsf{freshMeta}_{\,\Theta_1\,|\,\Gamma[\theta_1]}\,\mathsf{U}$
```

```
    $\textbf{let}\,(\Theta_3,\ \theta_3, A_1) = \mathsf{freshMeta}_{\,\Theta_2\,|\,\Gamma[\theta_{12}],\ x:A_0}\ \mathsf{U}$
    $(\Theta_4,\ \theta_4) \leftarrow \mathsf{unify}\ (A[\theta_{23}])\ (\{x : A_0[\theta_3]\} \to A_1)$
    $(\Theta_5,\ \theta_5,\ u') \leftarrow [\![u]\!]\Downarrow_{\ \Theta_4\,|\,\Gamma[\theta_{1234}]}\ (A_0[\theta_{34}])$
    $\textbf{return}\ (\Theta_5,\ \theta_{12345},\ (t'[\theta_{2345}])\ \{u'\},\ A_1[\theta_{45}][x \mapsto u'])$
$[\![\lambda\, x.\, t]\!]\Uparrow_{\ \Theta_0\,|\,\Gamma}\ :\equiv \textbf{do}$
    $\textbf{let}\,(\Theta_1,\ \theta_1,\ A) = \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma}\ \mathsf{U}$
    $(\Theta_2,\ \theta_2,\ t',\ B) \leftarrow \mathsf{insert}\ ([\![t]\!]\Uparrow_{\ \Theta_1\,|\,\Gamma[\theta_1],\ x:A})$
    $\textbf{return}\ (\Theta_2,\ \theta_{12},\ \lambda\, x.\, t',\ (x : A[\theta_2]) \to B)$
$[\![\lambda\, \{x\}.\, t]\!]\Uparrow_{\ \Theta_0\,|\,\Gamma}\ :\equiv \textbf{do}$
    $\textbf{let}\,(\Theta_1,\ \theta_1,\ A) = \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma}\ \mathsf{U}$
    $(\Theta_2,\ \theta_2,\ t',\ B) \leftarrow \mathsf{insert}\ ([\![t]\!]\Uparrow_{\ \Theta_1\,|\,\Gamma[\theta_1],\ x:A})$
    $\textbf{return}\ (\Theta_2,\ \theta_{12},\ \lambda\, \{x\}.\, t',\ \{x : A[\theta_2]\} \to B)$
$[\![\textbf{let}\, x : A = t\ \textbf{in}\, u]\!]\Uparrow_{\ \Theta_0\,|\,\Gamma}\ :\equiv \textbf{do}$
    $(\Theta_1,\ \theta_1,\ A') \leftarrow [\![A]\!]\Downarrow_{\ \Theta_0\,|\,\Gamma}\ \mathsf{U}$
    $(\Theta_2,\ \theta_2,\ t') \leftarrow [\![t]\!]\Downarrow_{\ \Theta_1\,|\,\Gamma[\theta_1]}\ A'$
    $(\Theta_3,\ \theta_3,\ u',\ B) \leftarrow [\![u]\!]\Uparrow_{\ \Theta_2\,|\,\Gamma[\theta_{12}],\ x:A'[\theta_2] = t'}$
    $\textbf{return}\ (\Theta_3,\ \theta_{123},\ (\textbf{let}\, x : A'[\theta_{23}] = t'[\theta_3]\ \textbf{in}\, u'),\ B)$
$[\![\_]\!]\Uparrow_{\ \Theta\,|\,\Gamma}\ :\equiv \textbf{do}$
    $\textbf{let}\,(\Theta',\ \theta,\ A) = \mathsf{freshMeta}_{\,\Theta\,|\,\Gamma}\ \mathsf{U}$
    $\textbf{return}\ (\mathsf{freshMeta}_{\,\Theta'\,|\,\Gamma[\theta]}\ A)$
```

2.6.1 Implicit Argument Insertion. We define two functions, insert′ and insert which insert 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* occurrences to *id* $\{\alpha\}$, where α is a fresh metavariable. We define the insertion functions to take as input the output of $[\![-]\!]\Uparrow$, so that it is more convenient to use as an optional post-processing step after inference.

The difference between insert and insert′ is that the former avoids insertion to implicit λ-terms. We usually want to avoid that, as it creates a new implicit β-redex where the fresh metavariable is often under-constrained. We only create such β-redex when the surface program explicitly asks for it; we shall shortly see this in the discussion of the $[\![-]\!]\Uparrow$ case for explicit function application. With this, we largely follow the insertion behavior of Agda.

2.6.2 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:

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

Then, define addition using induction:

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

When the above is elaborated, the $\lambda\,f\,m.\,\mathit{suc}\,(f\,m)$ function is checked with the expected type $\{n : \mathit{Nat}\} \to (\mathit{Nat} \to \mathit{Nat}) \to (\mathit{Nat} \to \mathit{Nat})$, and the elaboration output is $\lambda\,\{n\}\,f\,m.\,\mathit{suc}\,(f\,m)$. 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. 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 (with implicit insertions) then unify the expected and inferred types.

2.6.3 Inferring. For $[\![x]\!] \Uparrow_{\Theta|\Gamma}$, we look up the type of x in Γ. 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.

The $[\![t\,u]\!] \Uparrow_{\Theta_0|\Gamma}$ and $[\![t\,\{u\}]\!] \Uparrow_{\Theta_0|\Gamma}$ cases are again interesting. Here, we first infer a type for t, 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 $\mathsf{insert}'\,([\![t]\!] \Uparrow_{\Theta_0|\Gamma})$, which inserts implicit applications. In the latter case we do no insertion. This 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) We change direction and infer U as type for U, and successfully unify U with U.

Also note that we use insert' here instead of insert. Thus, we get expected (Agda-like) insertions with β-redexes as well. For example $(\lambda\{A\}\,x.\,\mathit{id}\,\{A\}\,x)\ \mathsf{U}$ is elaborated to $(\lambda\{A\}\,x.\,\mathit{id}\,\{A\}\,x)\,\{\mathsf{U}\}\ \mathsf{U}$.

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.

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

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

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

THEOREM 2.7 (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. One 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, we target a far stronger theory, and to our knowledge no prior work has attempted to characterize inferable terms or show completeness for a comparably strong elaborator. Our experience in Agda is that it is not tractable in general to figure out which arguments are inferable, by looking at function types, and often we have to run elaboration to see what works. This does not imply that type inference with dependent types must necessarily be an unwieldy affair. While we lose completeness, we gain the ability to infer more parts of programs by following richer type dependencies, as well as more power to refine and synthesize terms when types are known beforehand.

We can still say something about the behavior of our elaborator. For this, we consider a translation from core terms to surface terms, the evident 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, but we shall only use this evil notion in the following statement.

THEOREM 2.8 (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 with core expressions.*
- *Having extra implicit λ-s and implicit applications inserted.*

This follows by straightforward induction on surface syntax. □

Remark. It is *not* the case that for every term in the core syntax there exists a surface term which elaborates to it. The main reason is that λ-s in the surface syntax are not annotated, and it is easy to find core λ expressions with uninferable domain types. We skip optional surface λ domain type annotations for the sake of brevity.

2.6.6 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] and [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:

$$\mathit{List} : \mathsf{U} \to \mathsf{U} \qquad \mathit{nil} : \{A : \mathsf{U}\} \to \mathit{List}\,A \qquad \mathit{cons} : \{A : \mathsf{U}\} \to A \to \mathit{List}\,A \to \mathit{List}\,A$$

Below, 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 mark recursive calls by indentation. We also omit some checking, inference, implicit insertion and unification calls which are not essential for illustration.

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

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 $(\Theta,\ \theta,\ t,\ A)$ can insert 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 type, but at that point we have already missed our shot at implicit insertion.

There is a potential naive 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 by [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 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, our 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 4 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 4. 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 $\mathsf{Rec}\, \epsilon$.

We naturally want to make sure that the extended core syntax is still sensible. In relation to type theories, the following three metatheoretical properties are commonly considered.

- A theory is *consistent* if not every closed type has a closed inhabitant. An inconsistent theory cannot be used as a *logic*, from the propositions-as-types perspective [Univalent Foundations Program 2013, Chapter 1], because there every proposition is provable, so proofs do not distinguish truth from falsehood.
- A theory has the *canonicity* property, if every closed term is definitionally equal to a canonical term. For example, every closed natural number is a numeral, every Bool is true or false and so on. A canonical theory is useful as a *total programming language*; if canonicity fails, then some closed programs diverge or compute to stuck terms which cannot be viewed as

values. In non-total languages, progress and type preservation constitute a weaker analogue of canonicity [Harper 2016, Chapter 6].

- A theory has *decidable conversion*, if definitional equality is decidable. Undecidable conversion usually implies undecidable type checking for type theories. Often, decidable conversion follows from a normalization procedure for open terms: we first normalize, then decide equality of normal forms.

The properties are listed roughly in the order of increasing strength. Canonicity often implies consistency (because empty types have no canonical terms), and normalization implies canonicity. Now, our current core theory has *none* of the above properties, because the $\mathsf{U} : \mathsf{U}$ rule precludes them. However, as we mentioned in Section 2.2, this is easily fixed by stratified universes.

In any case, a *reduction* of new features to old features is a common and convenient strategy to reduce metatheoretical properties of extended theories to that of smaller or better-known theories. See Boulier et al. [2017] for a collection of examples for this. In the simplest case, new features are definable from old features. Here, we can either assume that new features are merely definitional shorthands, or give an evident injective translation from the new theory to the old one. Both imply that the desired properties transfer from the old theory to the new one.

Telescopes and records are derivable from natural numbers, the unit type and Σ-types. We write $\top$ for the unit type, and we have $\mathsf{tt} : \top$. 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:

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

From the above, ϵ, $- \triangleright -$, $- :: -$ and $[]$ are evident, and all expected equalities hold definitionally. Hence, it does not matter whether telescopes are primitive or derived. In the rest of the paper we never need to look inside the representation of telescopes, and we only use the "interface" from Figure 4.

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 write 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.

$$\textsc{tel}\ \frac{}{\Theta|\Gamma \vdash \mathsf{Tel} : \mathsf{U}} \qquad \textsc{empty-tel}\ \frac{}{\Theta|\Gamma \vdash \epsilon : \mathsf{Tel}} \qquad \textsc{nonempty-tel}\ \frac{\Theta|\Gamma \vdash A : \mathsf{U} \qquad \Theta|\Gamma,\, x : A \vdash B : \mathsf{Tel}}{\Theta|\Gamma \vdash (x : A) \triangleright B : \mathsf{Tel}} \qquad \textsc{record-type}\ \frac{\Theta|\Gamma \vdash A : \mathsf{Tel}}{\Theta|\Gamma \vdash \mathsf{Rec}\, A : \mathsf{U}}$$

$$\textsc{empty-record}\ \frac{}{\Theta|\Gamma \vdash [\,] : \mathsf{Rec}\, \epsilon} \qquad \textsc{nonempty-record}\ \frac{\Theta|\Gamma \vdash t : A \qquad \Theta|\Gamma \vdash u : \mathsf{Rec}\,(B[x \mapsto t])}{\Theta|\Gamma \vdash t :: u : \mathsf{Rec}\,((x : A) \triangleright B)}$$

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

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

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

Fig. 4. Rules for telescopes and strictly curried functions.

Curried functions are *mostly* derivable from Nat, ⊤ 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 4 hold definitionally, except $\textsc{curried-}\beta$ and $\textsc{curried-}\eta$. 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 arbitrary (possibly neutral) telescopes. $\textsc{curried-}\beta$ is still provable as a propositional equality, and assuming function extensionality $\textsc{curried-}\eta$ is provable as well. For details, see our Agda formalization of these definitions, which is included alongside the prototype implementation.

Hence, we can derive a somewhat weaker version of curried functions, with propositional β and η. From this, we still get consistency and canonicity very cheaply. This is because models proving consistency or canonicity for the base theory with Nat, $\top$ and Σ usually support equality reflection, i.e. that propositional equality implies definitional equality. For consistency, standard set-theoretical models and models in extensional type theory have this property, for canonicity, glued models (e.g. as in [Kaposi et al. 2019] and [Sterling 2019]) also have this property.

In contrast, showing normalization and decidability of conversion would require some extra work. We leave this to future work, but we expect that it is not difficult 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) insert $(\Theta,\ \theta,\ t,\ A)$ does not insert an implicit application when A is meta-headed.

In the following, we shall only enhance λ-insertions (1). This allows a simple implementation which only requires small changes to unification, and which is already remarkably powerful. It seems that enhancing implicit application insertions in (2) requires extending unification; we discuss this in Section 7.3. 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[3]. We prepend the following clause to Definition 2.3:

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

Definition 5.2 (Contextualization). We extend spine notation to applications of curried functions. For example, we may have a spine $\overline{t} \equiv (\{x : \overline{A}\},\ \{y : \overline{B}\})$. We accordingly revise Definition 2.4 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 and elaboration, we have to address a curious issue. Assuming *Bool* : U, *true* : *Bool* and *false* : *Bool*, consider the following surface expression:

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

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

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

However, there are infinitely many core terms which are conservative over the surface expression in the sense of Theorem 2.8. 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

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

[3]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. This issue does not seem to come up in realistic programs in the prototype implementation; the first-order treatment of uncertain insertions seem to leave little remaining "higher-order" uncertainty on the table.

METACON/CONSTANCY

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

CONSTANCY-≡

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

METASUB/CONSTANCY

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

METASUB/WEAKEN-CONSTANCY

$$\frac{}{\mathsf{p} : (\Theta,\ \mathsf{constancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\, B) \Rightarrow \Theta}$$

Fig. 5. Rules for constancy constraints

This is clearly undesirable. With the type $\{y : \mathit{Bool}\} \to \mathit{Bool}$, 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 $x\,\{\mathit{true}\}$ or $x\,\{\mathit{false}\}$ 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 **let** $x : \_ = \mathit{true}$, the old elaborator checks *true* with a fresh meta, and just assumes that the meta does not stand for an implicit function type.

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 5.

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 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 2.6, 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 implement 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 inside 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 METASUB/CONSTANCY rule 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 $A[\theta]$ 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.

Also, when we create a new constancy constraint, we immediately review it as described above.

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 $A[\theta]$.

5.2 Unification For Strictly Curried Function Types

Although we omit most details of unification, we shall discuss it for curried function types, as it is essential in the extended elaboration algorithm. We seek to solve meta-telescopes in a way such that iterated implicit function types on the two sides of a unification problem become definitionally equal. First we define a helper function which counts leading implicit function types:

$$\begin{aligned} &\mathsf{implArity}\,(\{x : A\} \to B) :\equiv 1 + \mathsf{implArity}\,B \\ &\mathsf{implArity}\,A \qquad\qquad\quad\; :\equiv 0 \end{aligned}$$

Then, we extend unification. The most interesting case is when we unify a curried function type with an implicit function type. In this case, if there are "missing" implicit arguments on the side of the curried function type (note the implArity condition on the pattern), we refine the A telescope 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.

```
$\mathsf{unify}_{\Theta_0\,|\,\Gamma}(\{x : \overline{A}\} \to B)\,(\{x_0 : A_0\} \to B') \mid \mathsf{implArity}\,B < \mathsf{implArity}\,B' + 1 :\equiv \mathbf{do}$
    $\mathbf{let}\,(\Theta_1,\, \theta_1,\, A_1) = \mathsf{freshMeta}_{\,\Theta_0\,|\,\Gamma,\, x_0 : A_0}\,\mathsf{Tel}$
    $(\Theta_2,\, \theta_2) \leftarrow \mathsf{unify}_{\Theta_1\,|\,\Gamma[\theta_1]}\,(A[\theta_1])\,((x : A_0[\theta_1]) \triangleright A_1)$
    $\mathbf{let}\,(\Theta_3,\, \theta_3) = \mathsf{newConstancy}_{\Theta_2\,|\,\Gamma[\theta_{12}],\, x_0 : A_0[\theta_{12}],\, x_1 : \mathsf{Rec}\,(A_1[\theta_2])}\,(B[\theta_{12}][x \mapsto (x_0 :: x_1)])$
    $\mathsf{unify}_{\Theta_3\,|\,\Gamma[\theta_{123}],\, x_0 : A_0[\theta_{123}]}\,(\{x_1 : A_1[\theta_{23}]\} \to B[\theta_{123}][x \mapsto (x_0 :: x_1)])\,B'$
```

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.

Below, let us assume that B' is not meta-headed, and it also does not match the pattern above. In this case we solve the domain telescope to be empty.

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

Again, we also have the symmetric case. Some examples for unification behavior:

- If B is not meta-headed, $\mathsf{unify}\,(\{x : \overline{\alpha}\} \to B)\,B$ causes α to be solved as ϵ, for the following reason: if B is an implicit function type, the arity condition fails, otherwise the pattern match fails, so either way we fall through the first rule and solve α to ϵ.
- If B is again not meta-headed, $\mathsf{unify}\,(\{x : \overline{\alpha}\} \to B)\,(\{x : A\} \to B)$ causes α to be solved as $(x : A) \triangleright \epsilon$.

Note that this algorithm matches one curried function type at a time, and does not recognize that multiple possible solutions exist e.g. in the case of $\mathsf{unify}\,(\{x : \overline{\alpha}\}\{y : \overline{\beta}\} \to C)\,(\{x : A\}\{y : B\} \to C)$, where we could solve α to ϵ, or $(x : A) \triangleright \epsilon$, or $(x : A) \triangleright (y : B) \triangleright \epsilon$. We leave such more sophisticated n-ary matching to future work.

For telescopes, Rec, inhabitants of Rec, curried abstraction and application, unification is structural, and other cases remain the same as in the basic elaborator of Section 2.

5.3 Elaboration

In the definition of checking, we insert a new clause after $[\![t]\!] \Downarrow_{\,\Theta|\Gamma} (\{x : A\} \to B)$:

```
$[\![t]\!] \Downarrow_{\,\Theta_0|\Gamma} (\alpha\,\overline{u}) :\equiv \textbf{do}$
  $\textbf{let}\ (\Theta_1,\ \theta_1,\ A) = \mathsf{freshMeta}_{\,\Theta_0|\Gamma}\ \mathsf{Tel}$
  $(\Theta_2,\ \theta_2,\ t',\ B) \leftarrow \mathsf{insert}\ ([\![t]\!] \Uparrow_{\,\Theta_1|\Gamma[\theta_1],\ x:\mathsf{Rec}\,A})$
  $\textbf{let}\ (\Theta_3,\ \theta_3) = \mathsf{newConstancy}_{\Theta_2|\Gamma[\theta_{12}],\ x:\mathsf{Rec}\,(A[\theta_2])}\ B$
  $(\Theta_4,\ \theta_4) \leftarrow \mathsf{unify}\ ((\alpha\,\overline{u})[\theta_{123}])\ (\{x : \overline{A[\theta_{23}]}\} \to B[\theta_3])$
  $\textbf{return}\ (\Theta_4,\ \theta_{1234},\ (\lambda\,\{x : \overline{A[\theta_{234}]}\}.\ t'[\theta_{34}]))$
```

Hence, when checking a term with a meta-headed type, we create a fresh meta with Tel type, infer a type for the term and wrap the result in a strictly curried λ. We again need to create a new constancy constraint. This way, if B does not depend on x : Rec A, the strictly curried λ in the output immediately computes away, since A is solved to ϵ.

This concludes the definition of the extended elaborator. The new algorithm is sound with respect to the extended core syntax, and it also has the conservativity property from Theorem 2.8 if we additionally allow elaboration to insert λ-s for strictly curried functions. We present some examples of the algorithm in action.

Example 5.4. We return to the *polyList* example from Section 3. We trace elaboration using the extended algorithm.

```
0   $[\![\mathit{cons}\,(\lambda\,x.\,x)\ \mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A))$
1     $[\![\mathit{cons}\,(\lambda\,x.\,x)\ \mathit{nil}]\!] \Uparrow$
2       $[\![\mathit{cons}\,(\lambda\,x.\,x)]\!] \Uparrow$
3         $[\![\mathit{cons}]\!] \Uparrow$
4         $= \mathit{cons}\,\{\alpha_0\}\ :\ \alpha_0 \to \mathit{List}\,\alpha_0 \to \mathit{List}\,\alpha_0$
5         $[\![\lambda\,x.\,x]\!] \Downarrow\ \alpha_0$
6         $= \lambda\,\{y : \overline{\alpha_1}\}.\,\lambda\,x.\,x$
7       $= \mathit{cons}\,\{\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}\}\ (\lambda\,\{y : \overline{\alpha_1}\}.\,\lambda\,x.\,x)$
8       $[\![\mathit{nil}]\!] \Downarrow\ (\mathit{List}\,(\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}))$
9       $= \mathit{nil}\,\{\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}\}$
10    $= \mathit{cons}\,\{\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}\}\ (\lambda\,\{y : \overline{\alpha_1}\}.\,\lambda\,x.\,x)$
            $(\mathit{nil}\,\{\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}\})$
        $: \mathit{List}\,(\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\})$
11    $\mathsf{unify}\ (\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A))\ (\mathit{List}\,(\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}))$
12      $\mathsf{unify}\ (\{A : \mathsf{U}\} \to A \to A)\ (\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\})$
13        $\mathsf{unify}\ \alpha_1\ ((A : \mathsf{U}) \triangleright (\alpha_3\,A))$
14        $\mathsf{unify}\ (A \to A)\ (\{z : \overline{\alpha_3\,A}\} \to \alpha_2\,\{A\}\,\{z : \overline{\alpha_3\,A}\} \to \alpha_2\,\{A\}\,\{z : \overline{\alpha_3\,A}\})$
15          $\mathsf{unify}\ (\alpha_3\,A)\ \epsilon$
```

```
16              $\mathsf{unify}\,(A \to A)\,(\alpha_2\,\{A\} \to \alpha_2\,\{A\})$
17                $\mathsf{unify}\,A\,(\alpha_2\,\{A\})$
18                $\mathsf{unify}\,A\,A$
19        $= \mathit{cons}\,\{\{A : \mathsf{U}\} \to A \to A\}\,(\lambda\,\{A\}\,x.\,x)\,(\mathit{nil}\,\{\{A : \mathsf{U}\} \to A \to A\})$
```

We diverge from the previous attempt at line 5. Here, we check $\lambda\,x.\,x$ with the meta α_0, so we create a fresh $\alpha_1 : \mathsf{Tel}$ meta and wrap the result as $\lambda\,\{y : \overline{\alpha_1}\}.\,\lambda\,x.\,x$. This result has the inferred type $\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\} \to \alpha_2\,\{y : \overline{\alpha_1}\}$, and we promptly unify α_0 (which stands for the list element type) with it. On line 10, we return the elaborated *cons* expression, where the list element type is made explicit. It only remains to unify the inferred and expected types. On line 12 we have the case where a curried function type is matched with an implicit function type, so on line 13 we refine α_1 to a non-empty telescope, and proceed unifying the domains. Note that on line 14 we have $\alpha_2\,\{A\}\,\{z : \overline{\alpha_3\,A}\}$, which results from applying the substitution $y \mapsto (A :: z)$, and computing $\alpha_2\,\{(A :: z) : \overline{(A : \mathsf{U}) \triangleright (\alpha_3\,A)}\}$ further using the APP-$\triangleright$ rule from Figure 4.

On line 14 we have the case when a curried function type is matched with a type which is not a curried function, implicit function or a meta-headed type. We accordingly unify domain telescope $\alpha_3\,A$ with ϵ, which causes α_3 to be solved as $\lambda\,A.\,\epsilon$ according to standard pattern unification. Now, $\alpha_2\,\{A\}\,\{z : \overline{\alpha_3\,A}\}$ computes to $\alpha_2\,\{A\}\,\{z : \epsilon\}$, which further computes to $\alpha_2\,\{A\}$ by APP-ϵ. From here, unification finishes with success. On the last line, the expected output is returned: since the α_1 telescope meta is now solved, curried function types and abstractions are computed away.

Computing $[\![\mathit{cons}\,(\lambda\,\{A\}\,x.\,x)\;\mathit{nil}]\!] \Downarrow (\mathit{List}\,(\{A : \mathsf{U}\} \to A \to A))$ also succeeds, as in Agda and our basic elaborator. In this case elaboration also inserts a curried λ around $(\lambda\,\{A\}\,x.\,x)$, but its domain is then solved to ϵ according to the rules for curried function type unification.

Example 5.5. We illustrate now the action of constancy constraints, using the example from Section 5.1. Again, assume $\mathit{Bool} : \mathsf{U}$ and $\mathit{true} : \mathit{Bool}$.

```
0    $[\![\mathbf{let}\,x : \_ = \mathit{true}\,\mathbf{in}\,x]\!] \Uparrow$
1      $[\![\_]\!] \Downarrow \mathsf{U} = \alpha_0$
2      $[\![\mathit{true}]\!] \Downarrow \alpha_0$
3        $\mathsf{freshMeta}\,\mathsf{Tel} = \alpha_1$
4        $[\![\mathit{true}]\!] \Uparrow = \mathit{true} : \mathit{Bool}$
5        $\mathsf{newConstancy}_{\bullet,\,x:\mathsf{Rec}\,\alpha_1}\,\mathit{Bool}$
6          $\mathsf{unify}\,\alpha_1\,\epsilon$
7        $\mathsf{unify}\,\alpha_0\,\mathit{Bool}$
8       $= \mathit{true}$
9      $[\![x]\!] \Uparrow \mathit{true} = \mathit{true} : \mathit{Bool}$
10    $= (\mathbf{let}\,x : \mathit{Bool} = \mathit{true}\,\mathbf{in}\,x) : \mathit{Bool}$
```

First we create a fresh meta α_0 for the hole on line 1, then we check *true* with it. Since we are checking with a meta-headed type, we create a fresh telescope meta α_1, then infer *Bool* for *true*. On line 5 we create a constancy constraint and immediately try to solve it, as described in Section 5.1.2. Since *Bool* does not depend on the domain, we solve the constraint and thereby solve α_1 as ϵ. Hence, we simply return *true* on line 8, since $\lambda\,\{x : \overline{\epsilon}\}.\,\mathit{true}$ computes to that, and elaboration succeeds with the expected output.

6 IMPLEMENTATION

We provide an implementation of the elaborator from Section 5. It is a standalone Haskell program which reads a surface expression from standard input, and outputs the result of elaboration, or optionally the type or the normal form of the result. It is implemented in about 1100 lines of Haskell. Of this, 680 lines constitute the core syntax, evaluation, unification and elaboration. Of these 680 lines, about 450 lines implement the basic elaborator of Section 2 and 230 lines implement the extended elaborator.

Elaboration is implemented in the style of Coquand's algorithm [Coquand 1996], where elaboration is interleaved with normalization-by-evaluation. Hence, we do not perform any substitution operations on core syntax, instead we evaluate core terms into the semantic domain, and perform unification, strengthening and occurrence checking on semantic values. There is also a quoting (or readback) operation which yields normalized core terms from values.

Metacontexts are a key point in the implementation. We avoid the tedious (and inefficient) threading of composed metasubstitutions, instead we have a mutable reference which stores the current metasubstitution. We have a "forcing" operation which computes a semantic value to a head normal form with respect to the current metasubstitution. Hence, instead of constantly updating every term and type by performing metasubstitution, we only force them whenever we need to pattern match on the shape of types, for example when we are inserting implicit arguments based on types. In this paper we still stick with the "threaded metasubstitution" presentation because it is more conventional, and also more straightforward to formalize.

We use normalization-by-evaluation in Abel's style (see e.g. [Abel 2013, Chapter 3]), where semantic values use de Bruijn levels, and the core syntax uses de Bruijn indices. This is practically very favorable, because the evaluator never has to perform weakening on values. The implementation of curried functions presents a bit of a complication, because the computation rules are type-directed, so we have to annotate applications and abstractions with telescopes (just as in our notation for the core syntax). We implement constraints in a fairly optimized way: we keep track of relevant "blocking" metas for each constraint and upon solving a meta we only review constraints which were blocked on the meta.

7 RELATED WORKS AND EVALUATION

In this section we examine how the elaborator fares in practice, its limitations, and how it compares to related works. We refer to our elaborator as FCIF when comparing it to others.

7.1 Related Works

MLF [Botlan and Rémy 2003] extends System F with polymorphic subtyping bounds, and supports strong inference for first class polymorphism. HML [Leijen 2009]) is a simplified variant of MLF. These systems rely critically on subtyping and thus diverge markedly from (fragments of) Martin-Löf type theory. HMF [Leijen 2008] is a relatively simple system with first-class polymorphism; however, it is also weaker than others in terms of inferable annotations, and its more powerful variant (extended with n-ary application handling) is more complex.

There have been several attempts in the context of GHC. Boxy types [Vytiniotis et al. 2006] and FPH [Vytiniotis et al. 2008] were two early iterations which suffered from complex specification, complex implementation or fragility. More recent works are guarded impredicativity (GI) [Serrano et al. 2018] and quick look impredicativity (QL) [Serrano et al. 2020]. They both work by examining n-ary applications to find metavariable instantiations which arise from occurrences guarded by rigid type constructors. GI's use of *generalization constraints* anticipates our use of telescopes, but overall GI is also burdened by a great deal of complexity. QL streamlines GI by eschewing

$IdTy$	$:\equiv \{A : \mathsf{U}\} \to A \to A$	inc	$: Int \to Int$
$single$	$: \{A : \mathsf{U}\} \to A \to List\,A$	$auto$	$: IdTy \to IdTy$
id	$: IdTy$	$auto'$	$: \{B : \mathsf{U}\} \to IdTy \to B \to B$
ids	$: List\,IdTy$	$choose$	$: \{A : \mathsf{U}\} \to A \to A \to A$
nil	$: \{A : \mathsf{U}\} \to List\,A$	app	$: \{A\,B : \mathsf{U}\} \to (A \to B) \to A \to B$
$cons$	$: \{A : \mathsf{U}\} \to A \to List\,A \to List\,A$	$revapp$	$: \{A\,B : \mathsf{U}\} \to A \to (A \to B) \to B$
$head$	$: \{A : \mathsf{U}\} \to List\,A \to A$	$runST$	$: \{A : \mathsf{U}\} \to (\{S : \mathsf{U}\} \to ST\ S\,A) \to A$
$tail$	$: \{A : \mathsf{U}\} \to List\,A \to List\,A$	$argST$	$: \{S : \mathsf{U}\} \to ST\ S\,Int$
map	$: \{A\,B : \mathsf{U}\} \to (A \to B) \to List\,A \to List\,B$	$poly$	$: IdTy \to Pair\,Int\,Bool$

Fig. 6. Types used in Figure 7.

constraints in favor of an eager preprocessing pass on neutral expressions which finds polymorphic instantiations. QL also takes advantage of bidirectional type propagation. Although QL seems to be practically the most favorable so far, it is far from being elegant: it rechecks expressions multiple times, and the restriction of preprocessing to (nested) neutral applications is rather ad-hoc.

7.2 Evaluation

We borrow a very useful collection of inference benchmarks from the GI [Serrano et al. 2018] and QL [Serrano et al. 2020] papers: see Figure 6 and Figure 7. We also include a source file in the implementation which reproduces these results. In ibid. a comparison is presented between multiple systems, but here we only include FCIF. The relative performance of FCIF in this benchmark is easy to remember: it handles exactly the same cases as QL (which is slightly more than what GI covers). We mark some cases as "Yes*", where FCIF successfully infers a type, but since FCIF does no let-generalization, the inferred types are not fully constrained without extra contextual information.

In the $A1$ case, FCIF inserts only a single curried λ on the outside, which is why the inferred type is not unifiable with $\{A : \mathsf{U}\} \to A \to \{B : \mathsf{U}\} \to B \to A$. However, this could be easily remedied by adding an extra curried λ insertion in the definition of $[\![\lambda\,x.\,t]\!]\Uparrow_{\Theta|\Gamma}$.

In $A8$, the failure is intentional: the types of *id* and *auto'* are not unifiable because of the mismatched order of implicit and explicit arguments. We do not float out or reorder implicit arguments in any way, because we want to support arbitrary mixing of implicit and explicit arguments in the surface language, and such reordering would be problematic anyway in the presence of dependent types. The same situation arises in $E1$, where we cannot unify $\{A : \mathsf{U}\} \to Int \to A \to A$ with $Int \to \{A : \mathsf{U}\} \to A \to A$.

Some general comments on the comparison of FCIF to prior works. First, FCIF supports dependent types and prior solutions do not. It is also unclear whether prior solutions can scale to dependent types. MLF and HML rely on subtyping, and solutions which work one neutral spine at a time (HMF, GI, QL) also face issues with dependently typed spines. With such spines, we cannot simply process later arguments when previous ones are not yet elaborated, as return types depend on argument values, and skipping over arguments may clog up type computation in an unacceptable way.

Secondly, FCIF supports global inference. For an example for this, **let** x : _ = *single id* **in** *cons* $\{List\,IdTy\}$ *x nil* works in FCIF, MLF and HML but does not work in GI, QL and HMF. MLF and HML work here by immediately giving a principal type to *single id* which involves generalization

A	POLYMORPHIC INSTANTIATION	
A1	$\lambda\, x\, y.\, y$ we infer a type which may be solved to $\{A\, B : \mathsf{U}\} \to A \to B \to A$ but not to $\{A : \mathsf{U}\} \to A \to \{B : \mathsf{U}\} \to B \to A$	Yes*
A2	*choose id*	Yes*
A3	*choose nil ids*	Yes
A4	$\lambda\, (x : \{A : \mathsf{U}\} \to A \to A).\, x\, x$	Yes*
A5	*id auto*	Yes
A6	*id auto′*	Yes*
A7	*choose id auto*	Yes
A8	*choose id auto′*	No
A9	$\lambda\, (f : \{A : \mathsf{U}\} \to (A \to A) \to \mathit{List}\, A \to A).\, f\, (\mathit{choose}\ \mathit{id})\ \mathit{ids}$	Yes*
A10	*poly id*	Yes
A11	$\mathit{poly}\, (\lambda\, x.\, x)$	Yes
B	INFERENCE OF POLYMORPHIC ARGUMENTS	
B1	$\lambda\, f.\, \mathit{pair}\, (f\ \mathit{zero})\, (f\ \mathit{true})$	No
B2	$\lambda\, \mathit{xs}.\, \mathit{poly}\, (\mathit{head}\ \mathit{xs})$	No
C	FUNCTIONS ON POLYMORPHIC LISTS	
C1	*length ids*	Yes
C2	*tail ids*	Yes
C3	*head ids*	Yes*
C4	*single id*	Yes*
C5	*cons id ids*	Yes
C6	$\mathit{cons}\, (\lambda\, x.\, x)\ \mathit{ids}$	Yes
C7	*append (single inc) (single id)*	Yes
C8	$\lambda\, (g : \{A : \mathsf{U}\} \to \mathit{List}\, A \to \mathit{List}\, A \to A).\, g\, (\mathit{single}\ \mathit{id})\ \mathit{ids}$	Yes*
C9	*map poly (single id)*	Yes
C10	*map head (single ids)*	Yes
D	APPLICATION FUNCTIONS	
D1	*app poly id*	Yes
D2	*revapp id poly*	Yes
D3	*runST argST*	Yes
D4	*app runST argST*	Yes
D5	*revapp argST runST*	Yes
E	η-EXPANSION	
	assuming $k : \{A : \mathsf{U}\} \to A \to \mathit{List}\, A \to A$, $h : \mathit{Int} \to \{A : \mathsf{U}\} \to A \to A$ and $\mathit{lst} : \mathit{List}\, (\{A : \mathsf{U}\} \to \mathit{Int} \to A \to A)$	
E1	*k h lst*	No
E2	$k\, (\lambda\, x.\, h\, x)\ \mathit{lst}$	Yes*
E3	$\lambda\, (r : (\{A : \mathsf{U}\} \to A \to \{B : \mathsf{U}\} \to B \to B) \to \mathit{Int}).\, r\, (\lambda\, x\, y.\, y)$	Yes

"Yes*" means that our system can infer a type for the expression, but the lack of let-generalization yields unsolved metas in the type.

Fig. 7. Elaboration benchmark from [Serrano et al. 2018].

with a subtyping bound. In contrast, FCIF makes no promises about principal typing — which is not feasible with dependent types — and does no generalization, but it can still infer a type for *id* which can be later constrained to *IdTy*.

7.3 Polymorphic Argument Inference and Curried Application Insertion

FCIF cannot infer the $B1$ and $B2$ cases on Figure 7, which would require inferring polymorphic types for function arguments. In very simple cases, FCIF is able to infer (ambiguous) polymorphic argument types. For example, it can infer a type for $(\lambda\, f.\, f\{zero\})$. Assuming $Pair : \mathsf{U} \to \mathsf{U} \to \mathsf{U}$ and $pair : \{A\, B : \mathsf{U}\} \to A \to B \to Pair\, A\, B$, it can also infer type for the following:

$$\lambda\, f\, x.\, pair\ (f\ \{x\}\, x)\ (f\ x)$$

Here, FCIF can proceed because it first encounters $(f\ \{x\}\, x)$, where it learns that f should have two arguments, where the first one is implicit. Armed with this knowledge, it applies f to a fresh meta when elaborating $(f\ x)$. The following fails though:

$$\lambda\, f\, x.\, pair\ (f\ x)\ (f\ \{x\}\, x)$$

Here, FCIF decides upon elaborating $(f\ x)$ that f is a function whose first argument is explicit, and fails at $f\ \{x\}$. This is similar to the implicit λ insertion issue, but here we would have to refine implicit applications, i.e. cover issue (2).

We can try inserting an unknown curried application after the first f, which could be later refined. Below we present an extension of insert which accomplishes this. We omit metacontexts and metasubstitutions for brevity.

```
$\mathsf{insert}\ (t,\ \alpha\, \overline{u}) :\equiv \mathbf{do}$
    $A \leftarrow \mathsf{freshMeta}_{\Gamma}\ \mathsf{Tel}$
    $B \leftarrow \mathsf{freshMeta}_{\Gamma,\, x:\mathsf{Rec}\, A}\ \mathsf{U}$
    $\mathsf{unify}\ (\alpha\, \overline{u})\ (\{x : \overline{A}\} \to B)$
    $\mathsf{newConstancy}_{\Gamma,\, x:\mathsf{Rec}\, A}\ B$
    $u \leftarrow \mathsf{freshMeta}_{\Gamma}\ (\mathsf{Rec}\, A)$
    $\mathbf{return}\ (t\ \{u : \overline{A}\},\ B[x \mapsto u])$
```

However, this alone does not work: in $t\ \{u : \overline{A}\}$, we return a term applied to a record with *unknown telescope type*, which later may give rise to unification problems where metas are applied to such unknown records. Our current unification algorithm does not handle these cases meaningfully, and we are not aware of any applicable algorithm in prior literature.

A robust system for this would give a solution for polymorphic argument inference. We do not expect that this is easy. The only prior work which supports this feature is MLF, which does not support dependent types, but whose type inference algorithms are already very complex.

We nevertheless sketch a solution which could be fleshed out in future work. We work with the following example:

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

Plain FCIF will attempt to elaborate $f\ x$ to $f\ (\lambda\{A\}.\, x)$, but fails to infer a type for x, because A is not in the scope of x's type. FCIF with curried application insertion instead returns $f\ (\lambda\ \{A\}.\, x\ \{\alpha_2\, x\, A : \overline{\alpha_0}\})$ from the same expression, and we also have $x : \{y : \overline{\alpha_0}\} \to \alpha_1\ \{y : \overline{\alpha_0}\}$. However, at this point FCIF still needs to unify expected and inferred types, so it needs to compute

$$\mathsf{unify}\ (\alpha_1\ \{\alpha_2\, x\, A : \overline{\alpha_0}\})\ (A \to A)$$

This does not fall into any pattern fragment, and has no most general solution. However, we may make several assumptions about unknown telescopes in spines. We can view this as an extension of Miller's pattern unification [Miller 1991] with unknown spine slices: $\alpha_2\, x\, A$ stands for a part of the spine which may contain bound variables x and A. We make the following assumption: α_2

may be only solved with *order-preserving embeddings*, so the four potential solutions are $(\lambda\, x\, A.\, [])$, $(\lambda\, x\, A.\, (A :: []))$, $(\lambda\, x\, A.\, (x :: []))$ and $(\lambda\, x\, A.\, (x :: A :: []))$. On the one hand, this rules out solutions which contain arbitrary terms or non-linear variable occurrences, which would yield unsolvable spines[4]. On the other hand, we probably do not throw away many useful solutions, because all curried applications arise from elaboration, and we do not have to deal with arbitrary such applications coming from the surface language. In short, we try to keep the invariant that metas returning in record types are only ever solved with order-preserving embeddings.

Out of the four possible embeddings, $(\lambda\, x\, A.\, (x :: A :: []))$ and $(\lambda\, x\, A.\, (x :: []))$ are ruled out because they would require a cyclic solution for α_0, and $(\lambda\, x\, A.\, [])$ is ruled out because it would yield the unsolvable unify $\alpha_1\, (A \to A)$ problem. Thus, $(\lambda\, x\, A.\, (A :: []))$ is the unique order-preserving embedding solution in this case, which does yield the expected elaboration output. We note though that "unique order-preserving embedding" does not yet provide an efficient algorithmic implementation, and we would need to be smarter in considering particular unification problems, to avoid backtracking through possible solutions.

8 CONCLUSION AND FUTURE WORK

In type theory, it is a common endeavor to search for theories and features which confer the greatest amount of expressive power for the least amount of formal and conceptual complexity. In relation to elaboration and inference algorithms, we would similarly like to use tools and concepts with high power-to-weight ratio. Sometimes core theories need to be extended to allow more powerful elaboration. Certainly, type inference would be cumbersome without the notion of metavariables. Likewise, contextual metavariables are essential for elaboration in the presence of type dependencies.

In this paper we propose the concept of strictly curried functions, which supports elaboration of first-class implicit functions. It seems that trying to solve this problem by fiddling with postponing and heuristics, without extending the core theory, does not really cut it. In a dependently typed spine $t\, u\, v$, when elaborating v we already want to have an elaborated version of u with computational behavior. While contextual metavariables yield a nice modal type theory which computes in the presence of *unknown terms*, our curried functions extend this to a system which also computes in the presence of unknown *implicit insertions*. Since implicit insertions are in a way also just unknown terms, we can reuse most of the infrastructure of contextual/crisp type theory, and only add modest extensions.

Often it is the case that constructions on dependent type theory are forced to be more principled and structured than constructions on less powerful theories, because the many interactions and intricacies leave less room for whims. In our case, we believe that focusing on dependent type theory helped us home in on the essential parts of the problem, and it is likely that restrictions of our solution would be also favorably simple and powerful in simpler settings such as System F.

In future work, we would like to

- Investigate extending elaboration with curried application insertions, along the lines of Section 7.3.
- Formalize elaboration and unification from an algebraic/categorical perspective, in particular give algebraic definitions for the core theories.
- Investigate implementing features of our elaborator in production systems such as Agda. Also, investigate simplifying our algorithm to non-dependent (e.g. System F) settings.
- Investigate completeness and properties related to inferability.

[4]Technically, we could consider permutations such as $\lambda\, x\, A.\, (A :: x :: [])$ too, but these are superfluous, since nothing in our system depends on the argument ordering of metavariable solutions.

REFERENCES

Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Ph.D. Dissertation. Ludwig-Maximilians-Universität München. Habilitation thesis.

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. https://doi.org/10.1145/3158111

Andreas Abel and Brigitte Pientka. 2011. Higher-Order Dynamic Pattern Unification for Dependent Types and Records. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6690)*, C.-H. Luke Ong (Ed.). Springer, 10–26. https://doi.org/10.1007/978-3-642-21691-6_5

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. 2020. Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197

Didier Le Botlan and Didier Rémy. 2003. ML^{F}: raising ML to the power of system F. *SIGPLAN Notices* 38, 9 (2003), 27–38. https://doi.org/10.1145/944746.944709

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs* (Paris, France) *(CPP 2017)*. ACM, New York, NY, USA, 182–194. https://doi.org/10.1145/3018610.3018620

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Sci. Comput. Program.* 26, 1-3 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. https://doi.org/10.1145/2500365.2500582

Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. *CoRR* abs/1610.07978 (2016). arXiv:1610.07978 http://arxiv.org/abs/1610.07978

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 229–254. https://doi.org/10.1007/978-3-662-49498-1_10

Adam Michael Gundry. 2013. *Type inference, Haskell and dependent types.* Ph.D. Dissertation. University of Strathclyde, Glasgow, UK. http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22728

Robert Harper. 2016. *Practical foundations for programming languages.* Cambridge University Press.

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. Chalmers University of Technology.

Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for Type Theory. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 25:1–25:19. https://doi.org/10.4230/LIPIcs.FSCD.2019.25

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 283–294. https://doi.org/10.1145/1411204.1411245

Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 66–77. https://doi.org/10.1145/1480881.1480891

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22

Dale Miller. 1991. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *J. Log. Comput.* 1, 4 (1991), 497–536. https://doi.org/10.1093/logcom/1.4.497

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

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 *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9236)*, Christian Urban and Xingyuan Zhang (Eds.). Springer, 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

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, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 783–796. https://doi.org/10.1145/3192366.3192389

Jonathan Sterling. 2019. Algebraic Type Theory and Universe Hierarchies. *CoRR* abs/1902.08848 (2019). arXiv:1902.08848 http://arxiv.org/abs/1902.08848

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book, Institute for Advanced Study.

Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 251–262. https://doi.org/10.1145/1159803.1159838

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 295–306. https://doi.org/10.1145/1411204.1411246

Pawel Wieczorek and Dariusz Biernacki. 2018. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 266–279. https://doi.org/10.1145/3167091