Elaboration with First-Class Implicit Function Types

ANONYMOUS AUTHOR(S)

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

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

1 INTRODUCTION

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

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

```
-- Agda                     (* Coq *)
id : {A : Set} → A → A      Definition id {A : Type}(x : A) := x.
id x = x
```

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

```
id :: forall (a :: *). a -> a
id x = x
```

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

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

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

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

implicit function type, instead, implicitness is tied to particular *names*, and while we can write `List (forall {A : Type}, A -> A)`, the brackets here are simply ignored by Coq and we get a plain function type. For example, Coq accepts the following definition:

```
Definition poly : forall (f : forall {A : Type}, A -> A), bool * nat :=
  fun f => (f bool true, f nat 0).
```

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 -> A` is silently dropped.

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

```
poly :: (forall a. a -> a) -> (Bool, Int)
poly f = (f True, f 0)
```

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 -> 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} → A → A)` without issue. However, Agda's elaboration still has limitations when it comes to handling implicit functions. Assume that we have `[]` for the empty list and `_∷_` for list extension, and consider the following code:

```
polyList : List ({A : Set} → A → A)
polyList = (λ x → x) ∷ []
```

Agda 2.6.0.1 does not accept this. However, it does accept `polyList = (λ {A} x → x) ∷ []`. The issue is the following. Agda first *infers* a type for `(λ x → x) ∷ []`, then tries to unify the inferred type with the given `List ({A : Set} → A → A)` annotation. However, when Agda elaborates `λ x → 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 elects to not insert a lambda. This assumption is often correct, but sometimes — as in the current case — it is not.

There is significant literature on type inference in the presence of first-class polymorphic types, mainly in relation to GHC and ML-like languages [?]. The above issue in Agda is a specific instance of the challenges described in the mentioned works. However, none of the proposed algorithms have landed so far in production compilers, for reasons of complexity, fragility and interaction with other language features.

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

1.1 Contributions

- We propose an elaboration algorithm which translates from a small Agda-like surface language to a small Martin-Löf type theory extended with implicit function types, telescopes

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

1.1.1 *Note on Terminology.*

We prefer to avoid the term "impredicative inference" in order to avoid confusion with impredicativity in type theory. The two notions historically coincided, but has since diverged to the point of being orthogonal. In type theory, impredicativity is a property of a universe, i.e. closure of a universe under arbitrary products. In the type inference literature, impredicativity means the ability to instantiate type variables and metavariables to polymorphic types. In particular, we have that

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

2 BASIC BIDIRECTIONAL ELABORATION

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

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
	$\mid$	$\mathbf{let}\,x : A = t\,\mathbf{in}\,u$	let-definition
	$\mid$	$\_$	hole for inferred term

Fig. 1. Syntax of the surface language.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Selected rules of the core language.

2.1 Surface syntax

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

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

2.2 Core syntax

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

- 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. 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]. Also, typing is stable under substitution.
- Definitional equality is understood to be a congruence and an equivalence relation, which is respected by substitution and typing.
- We elide a number of well-formedness assumptions in rules. For instance, whenever a context appears in a rule, it is assumed to be well-formed. Likewise, whenever we have $\Theta|\Gamma \vdash t : A$, we assume that $\Theta|\Gamma \vdash A : \mathsf{U}$.

From now on, we will only consider well-formed core syntax, and only constructions which respect definitional equality. In other words, we quotient well-formed syntax by definitional equality.

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

how to refer to mathpartir rules from text?

Metacontexts are used to record metavariables which are created during elaboration. In our case, metacontexts are simply a context prefix, and we have variables pointing into it, but we do not support contextual modality e.g. as in [Nanevski et al. 2008]. The non-meta typing context additionally supports *defined variables*, which is used in the typing rule for *let-definitions*, and we have that any defined variable is equal to its definition. We mainly support this as a convenience feature in the implementation of our elaborator.

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

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

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

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

Example 2.1. The core syntax is quite expressive as a programming language, thanks to let-definitions and the type-in-type rule which allows Church-encodings of a large variety of inductive types. For example, the following term computes a list of types, by mapping over the list $\mathit{cons}\;\mathsf{U}\;(\mathit{cons}\;\mathsf{U}\;\mathit{nil})$.

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

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

2.3 Metasubstitutions

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

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

The following are admissible:

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

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

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

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

2.4 Fresh Metavariables

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

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

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

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

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

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

Definition 2.5 (Fresh meta creation). We specify fresh_A as follows:

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

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

2.5 Unification

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

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

Here, we do not require that unification returns most general unifiers, nor do we go into the details of how unification is implemented. Gundry describes unification in detail in [Gundry 2013, Chapter 4] for a similar syntax, with a similar (though more featureful) setup for metacontexts.

Note that our unification algorithm does not support *constraint postponing*, since we have not specified constraints. In our concrete implementation, unification supports basic higher-order pattern unification and metavariable pruning [**?**].

2.6 Elaboration

Elaboration consists of two (partial) functions, checking and inferring, which are defined by mutual induction on surface syntax. They are specified as follows:

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

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

In the following, we use a monadic pseudocode notation, where the side effect is failure.

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

REFERENCES

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

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

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

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

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

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

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

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