

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 types¹

```
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

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

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 accumulate 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 accumulate 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
	$(x : A) \rightarrow B$	function type
	$\{x : A\} \rightarrow B$	implicit function type
	$t\ u$	application
	$t\ \{u\}$	implicit application
	$\lambda x. t$	lambda abstraction
	$\lambda \{x\}. t$	implicit abstraction
	U	universe
	$\text{let } x : A = t \text{ in } u$	let-definition
	$-$	hole for inferred term

Fig. 1. Syntax of the surface 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 `let x : _ = U 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 $\Theta|\Gamma \vdash A : U$.

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 for simplicity of 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.

2.2.1 Example programs. 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 Booleans:

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 Manna, Rasmus Ejlers Møgelberg, Andrew M Pitts, and Bas Spitters. 2018. Modal dependent type theory and dependent right adjoints. *arXiv preprint arXiv:1804.05236* (2018).
- Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)* 9, 3 (2008), 1–49.
- Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.
- Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In *International Conference on Interactive Theorem Proving*. Springer, 359–374.
- Jonathan Sterling. 2019. Algebraic type theory and universe hierarchies. *arXiv preprint arXiv:1902.08848* (2019).

$\boxed{\Theta \vdash}$	<i>metacontext formation</i>		
$\boxed{\Theta \Gamma \vdash}$	<i>context formation</i>		
$\boxed{\Theta \Gamma \vdash t : A}$	<i>typing</i>		
$\boxed{\Theta \Gamma \vdash t \equiv u : A}$	<i>term equality</i>		
<hr/>			
METACON/EMPTY	METACON/BIND	CON/EMPTY	CON/BIND
$\frac{}{\bullet \vdash}$	$\frac{\Theta \vdash \quad \Theta \vdash A : U}{\Theta, \alpha : A \vdash}$	$\frac{\Theta \vdash}{\Theta \bullet \vdash}$	$\frac{\Theta \Gamma \vdash \quad \Theta \Gamma \vdash A : U}{\Theta \Gamma, x : A \vdash}$
<hr/>			
CON/DEFINE	TM/METAVAR		TM/BOUND-VAR
$\frac{\Theta \Gamma \vdash \quad \Theta \Gamma \vdash t : A}{\Theta \Gamma, x : A = t \vdash}$	$\frac{}{\Theta_0, \alpha : A, \Theta_1 \Gamma \vdash \alpha : A}$		$\frac{}{\Theta \Gamma, x : A, \Delta \vdash x : A}$
<hr/>			
TM/DEFINED-VAR	TY/U	LET	
$\frac{}{\Theta \Gamma, x : A = t, \Delta \vdash x : A}$	$\frac{}{\Theta \Gamma \vdash U : U}$	$\frac{\Theta \Gamma \vdash t : A \quad \Theta \Gamma, x : A = t \vdash u : B}{\Theta \Gamma \vdash \text{let } x : A = t \text{ in } u : B[x \mapsto t]}$	
<hr/>			
TY/FUN	TY/IMPLICIT-FUN		
$\frac{\Theta \Gamma \vdash A : U \quad \Theta \Gamma, x : A \vdash B : U}{\Theta \Gamma \vdash (x : A) \rightarrow B : U}$	$\frac{\Theta \Gamma \vdash A : U \quad \Theta \Gamma, x : A \vdash B : U}{\Theta \Gamma \vdash \{x : A\} \rightarrow B : U}$		
<hr/>			
TM/APP	TM/IMPLICIT-APP		
$\frac{\Theta \Gamma \vdash t : (x : A) \rightarrow B \quad \Theta \Gamma \vdash u : A}{\Theta \Gamma \vdash t u : B[x \mapsto u]}$	$\frac{\Theta \Gamma \vdash t : \{x : A\} \rightarrow B \quad \Theta \Gamma \vdash u : A}{\Theta \Gamma \vdash t \{u\} : B[x \mapsto u]}$		
<hr/>			
TM/LAM	TM/IMPLICIT-LAM		
$\frac{\Theta \Gamma, x : A \vdash t : B}{\Theta \Gamma \vdash \lambda x. t : (x : A) \rightarrow B}$	$\frac{\Theta \Gamma, x : A \vdash t : B}{\Theta \Gamma \vdash \lambda \{x\}. t : \{x : A\} \rightarrow B}$		
<hr/>			
FUN- β	IMPLICIT-FUN- β		
$\frac{\Theta \Gamma, x : A \vdash t : B \quad \Theta \Gamma \vdash u : A}{\Theta \Gamma \vdash (\lambda x. t) u \equiv t[x \mapsto u] : B[x \mapsto u]}$	$\frac{\Theta \Gamma, x : A \vdash t : B \quad \Theta \Gamma \vdash u : A}{\Theta \Gamma \vdash (\lambda \{x\}. t) \{u\} \equiv t[x \mapsto u] : B[x \mapsto u]}$		
<hr/>			
FUN- η	IMPLICIT-FUN- η		
$\frac{\Theta \Gamma \vdash t : (x : A) \rightarrow B}{\Theta \Gamma \vdash (\lambda x. t x) \equiv t : (x : A) \rightarrow B}$	$\frac{\Theta \Gamma \vdash t : \{x : A\} \rightarrow B}{\Theta \Gamma \vdash (\lambda \{x\}. t \{x\}) \equiv t : \{x : A\} \rightarrow B}$		
<hr/>			
DEFINITION			
$\frac{}{\Theta \Gamma, x : A = t, \Delta \vdash x \equiv t : A}$			

Fig. 2. Selected rules of the core language.