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 in general 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 the programmer still has 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 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. In Agda, this is the case. For example, we have a list type where the type of elements is an implicit function,

[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/

as in `List ({A : Set} → A → A)`. That said, Agda's elaboration has limitations when it comes to handling such types, and in many cases programmers must write out implicit lambda abstractions.

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 : Set}, A -> A)`, the brackets here are simply ignored by Coq. For illustration, 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 somewhere between Agda and Coq. We can certainly 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 the 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.

There is significant literature on type inference in the presence of first-class polymorphic types, mainly in relation to GHC and ML-like languages [?]. However, none of the proposed algorithms have landed so far in production compilers, for reasons of complexity, fragility, interaction with other language features, or necessity to modify surface or core languages.

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.
- Our algorithm is a conservative extension of Norell's bidirectional elaborator for Agda [Norell 2007, Chapter 3.6]; 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 a standalone implementation of the elaborator described in this paper.

1.2 Note on Terminology

We prefer to avoid the term "impredicative inference" in order to avoid confusion with impredicativity in type theory, which is an orthogonal notion. 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 is type-theory-predicative, but implements type-inference-impredicative elaboration.
- Coq has type-theory-impredicative Prop universe, but implements type-inference-predicative elaboration.

- GHC is type-theory-impredicative with `RankNTypes` but without `ImpredicativeTypes`, as we have `(forall a. a -> a) :: *`.

2 BIDIRECTIONAL ELABORATION

- Limitations in Agda
- Insertion of implicits.
- First-class implicit functions vs Coq.
- Unique vs non-unique solutions.
- On the uniqueness of implicit insertion.

REFERENCES

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