Mechanizing an elaboration algorithm for the Hindley-Damas-Milner system

ROGER BOSMAN, KU Leuven, Belgium

1 INTRODUCTION

Let-polymorphism, as supported by the Hindley-Damas-Milner (HDM) system [2], lies at the center of the design of functional programming languages such as Haskell and OCaml. This is because it allows for sound, complete, and decidable type inference. Naturally, these properties have been thoroughly studied over the years and have been proven; some informally (on paper) and others formally (using proof assistants).

Most of this earlier work considers a type checking/inference (type checking with inference) algorithm like Damas and Milner's algorithm $\mathcal{W}$ [2]. However, in practice, the type systems of functional programming languages are extended with features such as type classes (originally introduced in Haskell [17]), implicits in Scala [8] and Agda [3], or intersection types [9, 10]. Often these features, including the ones listed here, are implemented using *elaboration*. This means that instead of adding explicit support to the entire pipeline, the feature is *desugared* into already-existing concepts by a source-to-source transformation. For example, type classes can be desugared into dictionaries, which themselves can be desugared into ordinary ADT's.

Since type checking algorithms do not necessarily synthesize all required information needed for elaboration, the existing type checking/inference algorithms (which includes algorithm $\mathcal{W}$) are no longer sufficient. Instead, an elaboration/inference (elaboration with inference) algorithm is needed. Somewhat surprisingly, elaboration has been studied in less detail, and to our knowledge, no end-to-end mechanization of an elaboration/inference algorithm for the HDM system exists. The (ongoing) work presented here addresses this by formulating an elaboration algorithm that is proven correct w.r.t. the original HDM specification in the Coq proof assistant [1]. Our elaboration algorithm targets System F [12].

Our main motivation for constructing this proof is to extend it with elaboration-based type system extensions, the correctness of which has been far less established, and subject to less study in general. Any such mechanization would have to mechanize an elaboration algorithm, which is why we consider this a good first step towards achieving these goals.

Additionally, given the significance of the HDM system, we feel an elaboration algorithm for said system deserves to be mechanized. On its own, this may not be enough reason for constructing this mechanization. However, as a side effect, it certainly is nice to have.

2 BACKGROUND

2.1 HDM type inference & let polymorphism

The Hindley-Damas-Milner type system offers decidable type inference for let-polymorphic programs. That is, given a let-polymorphic program, an algorithm exists that can determine its (most general) type without the need for typing annotations in the program.

Let-polymorphism allows for terms to be polymorphically typed. Consider the identity function $id = \lambda x.\ x$. Expressions in the simply-typed lambda calculus [11] can only have a single, *monomorphic* type. This means that the identity function has to be defined once for each type, all with the same implementation (returning the input). Polymorphism addresses this issue, allowing for expressions to be assigned a polytype that can be instantiated with various types.

Author's address: Roger Bosman, roger.bosman@kuleuven.be, KU Leuven, Celestijnenlaan 200A, Leuven, Belgium.

Ideally, one would infer a polymorphic type wherever possible. This would allow the following example, where the identity function is applied to both an integer and a boolean, to typecheck:

$$(\lambda f.\ (f\ 1, f\ \texttt{True}))\ (\lambda x.\ x)$$

Unfortunately, such type inference is known to be undecidable [18], although that has not stopped attempts at inferring such types in some of the situations [14]. Let-polymorphism, which *is* decidable, only infers polytypes for expressions that are bound to a name, which generally happens by means of a let expression, hence the name.

2.2 Declarative specification vs. algorithm

Often there is a distance between the specification of some algorithm and the algorithm itself. The sentence "let n and m be two primes such that $n * m = o$" tells you nothing about how to actually compute n and m.

The system as presented by Damas and Milner [2] is a declarative one. While it is syntax-directed, the inference rules given features existentially quantified types, without a clear way of computing said types. In essence, it allows for the construction of a proof that a term is correct if the types are already known beforehand, but does not offer a way of doing type inference.

Luckily, Damas and Milner did not only provide a specification, but also a type inference algorithm $\mathcal{W}$, which promises to implement the specification. In such a situation one might want to prove that this is actually true, to verify the algorithm "does what it says on the tin".

In the case of the HDM type system and algorithm $\mathcal{W}$ (and its adaptions), this question has been the topic of a great deal of previous work [5–7], some of which have been mechanized. Some proofs focus on either unification or inference [4, 13], or combine both in a single proof [16].

2.3 Type checking vs. Elaboration

On the surface, both type checking and elaboration aim to solve the same problem: they fold over an input program, infer types where needed, and produce the type of a program. However, there is one shortcut that a type checking algorithm can take that an elaboration algorithm cannot, which involves the tracking and generalization of unused type variables. Consider the following example.

$$\textbf{let}\ x = (\lambda x.\ \texttt{unit})\ (\lambda y.\ y)\ \textbf{in} \ldots$$

Algorithm $\mathcal{W}$ is able to correctly determine that the type for `x` is `Unit`. While running the algorithm, the type of `y` will be assigned some existential $\widehat{\alpha}$. As the applied function ignores its arguments, no further constraints are found, which means this existential is never solved. For type checking this is not an issue, because an unused type may be any type.

Yet when elaborating to something like System F, where terms have an explicit type, there is a problem. Even though the type of `y` is unused, it must be found, because it must be annotated in the output. What type is given to `y`? The output may not contain existential variables, because they are internal to the compiler. Instead, a type abstraction must be introduced, which then can be used for the type of `y`:

$$\textbf{let}\ x : (\forall a.\ \texttt{Unit}) = \Lambda a.(\lambda(x : a \to a).\ \texttt{unit})\ (\lambda(y : a).\ y)\ \textbf{in} \ldots$$

The problem is that type checking algorithms like algorithm $\mathcal{W}$ do not have the necessary information to perform the transformation described above, as they cannot determine which unsolved existentials should be generalized at what point.

This problem is far from insurmountable, and compilers such as GHC do incorporate such an elaboration algorithm. Yet, this means altering the algorithm, which means any guarantees given by (mechanical) proofs over the type checking algorithm no longer apply to the elaboration algorithm.

$\boxed{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}$ $\boxed{\Psi_{in} \vdash e : \sigma \rightsquigarrow t \dashv \Psi_{out}}$

$$\frac{\widehat{\alpha} \notin \Psi_{in} \qquad (\Psi_{in}; (\widehat{\alpha}); x : \widehat{\alpha}) \vdash e : [A_2]\tau_2 \rightsquigarrow t \dashv (\Psi_{out}; A_1; x : \tau_1)}{\Psi_{in} \vdash \lambda x.e : [A_1, A_2](\tau_1 \rightarrow \tau_2) \rightsquigarrow \lambda(x : \tau_1).t \dashv \Psi_{out}} \textsc{Abs}$$

$$\frac{\Psi_{in} \vdash e : [A]\tau \rightsquigarrow t \dashv \Psi_{out}}{\Psi_{in} \vdash e : \boldsymbol{\forall A.\tau} \rightsquigarrow \boldsymbol{\Lambda A.t} \dashv \Psi_{out}} \textsc{Gen}$$

Fig. 1. Examples of the algorithmic judgment for mono- and polytypes

3 APPROACH

Our main objective is to design a type inference algorithm that is also able to determine the set of (unsolved) existential variables in scope for some term, such that during generalization these can be generalized. One might try to formulate a judgment that simply inspects a derivation and collects the set of existential variables. Such a solution can work, but may be hard to mechanize. Ideally, all of our typing information is propagated bottom-up, without the need for inspecting sub-judgments.

Instead, we formulate our judgments as shown in figure 1. The left judgment infers the monotype of a term e under an input environment Ψ_{in}, and synthesizes an output environment Ψ_{out}, the type of the expression τ, an elaborated term t (the result of elaborating e), and most importantly, a list of existential variables A in scope for type τ.

Using this information, generalization is fairly straightforward. Displayed below is the monotype that is inferred for the right-hand side of the let binding we encountered before:

$$\Psi_{in} \vdash (\lambda x.\ \texttt{unit})\ (\lambda y.\ y) : [\widehat{\alpha}]\texttt{Unit} \rightsquigarrow (\lambda(x : \widehat{\alpha} \rightarrow \widehat{\alpha}).\ \texttt{unit})\ (\lambda(y : \widehat{\alpha}).\ y) \dashv \Psi_{out}$$

The elaborated term contains the unsolved existential $\widehat{\alpha}$, as indicated by $[\widehat{\alpha}]$ to the left of the type. Generalization, as displayed in figure 1, therefore is fairly straightforward. At the point of generalization, any leftover unsolved existential variables are free and can be converted to type abstractions. Each such $\widehat{\alpha}$ is mapped to a fresh type variable a, which can then be prepended with a $\forall$ in the type, and be substituted for $\widehat{\alpha}$ in the elaborated term. The bold terms $\boldsymbol{\forall A.\tau}$ and $\boldsymbol{\Lambda A.\tau}$ are functions that perform this operation, yielding the desired term.

4 RESULTS & CONTRIBUTIONS

As stated in the introduction, the work towards this mechanization is ongoing. The current phase of the project focuses on a simpler judgment than the one presented in section 3. Specifically, the judgment does not yet synthesize the elaborated term, but it *does* synthesize the set of existential variables. Essentially it is a type checking algorithm "with batteries included" for an extension to elaboration, which should be fairly straightforward.

Instead of proving this correct w.r.t. the original HDM system, we have formulated an adaptation of this system that is more alike the algorithmic version. This does require us to prove the equivalence of the adaptation and the original HDM system, but it does allow for a simpler correctness proof.

Currently, the systems have been formalized (using the Ott tool [15]), the algorithmic version has been proven to terminate, and work towards soundness and correctness is ongoing.

COLLABORATIVE WORK DISCLAIMER

The original version of rules for both the algorithmic version as the adaptation of the original HDM system have been formulated on paper by Georgios Karachalias (Tweag I/O), who at the time was a postdoctoral researcher working with the author's current PhD supervisor (Tom Schrijvers, KU Leuven). While he is still involved in the project in an advisory role, any work since then, including all of the formalization, modifications of the original rules, and writing this document, have been the work of the author.

REFERENCES

[1] Yves Bertot and Pierre Castéran. 2013. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Springer Science & Business Media.

[2] Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 207–212.

[3] Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. *ACM SIGPLAN Notices* 46, 9 (2011), 143–155.

[4] Catherine Dubois and Valerie Menissier-Morain. 1999. Certification of a Type Inference Tool for ML: Damas–Milner within Coq. *Journal of Automated Reasoning* 23, 3 (1999), 319–346.

[5] Adam Gundry, Conor McBride, and James McKinna. 2010. Type Inference in Context. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming.* 43–54.

[6] Conor McBride. 2003. First-Order Unification by Structural Recursion. *Journal of functional programming* 13, 6 (2003), 1061–1075.

[7] Wolfgang Naraschewski and Tobias Nipkow. 1999. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning* 23, 3 (1999), 299–318.

[8] Martin Odersky, Lex Spoon, and Bill Venners. 2008. Implicit Parameters and Conversions. In *Programming in Scala.* Artima Inc, Chapter 13.

[9] Bruno C d S Oliveira, Zhiyuan Shi, and Joao Alpuim. 2016. Disjoint Intersection Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming.* 364–377.

[10] Benjamin C Pierce. 1991. Programming with Intersection Types and Bounded Polymorphism. (1991).

[11] Benjamin C Pierce and C Benjamin. 2002. *Types and Programming Languages.* MIT press.

[12] John C Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium.* Springer, 408–425.

[13] Rodrigo Ribeiro and Carlos Camarao. 2015. A Mechanized Textbook Proof of a Type Unification Algorithm. In *Brazilian Symposium on Formal Methods.* Springer, 127–141.

[14] Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

[15] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, and Susmit Sarkar. 2010. Ott: Effective Tool Support for the Working Semanticist. *Journal of functional programming* 20, 1 (2010), 71–122.

[16] Rafael Castro G Silva, Cristiano Vasconcellos, and Karina Girardi Roggia. 2020. Monadic W in Coq. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity.* 25–32.

[17] Philip Wadler and Stephen Blott. 1989. How to Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 60–76.

[18] Joe B Wells. 1999. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic* 98, 1-3 (1999), 111–156.