Eliminating bugs in type inference algorithms by describing them with precise types

An evaluation of Correct-by-Construction programming in Agda for bug-free type inference algorithms

Vincent Pikand[1]
Supervisor(s): Jesper Cockx[1]**, Sára Juhošová**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Vincent Pikand
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Sára Juhošová, Thomas Durieux

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Static type systems ensure code correctness by aligning implementations with defined type signatures. Despite their benefits in preventing common errors, complex type systems increase the likelihood of bugs within type checkers. Correct-by-construction (CbC) programming offers a solution by using precise types to create intrinsically verified type checkers, ensuring soundness and potentially completeness. This paper evaluates the use of CbC programming for implementing type inference for the simply typed λ-calculus, focusing on Hindley-Milner (HM) and bidirectional type inference. Implementations in Agda, a dependently typed language, reveal that while CbC programming can eliminate bugs, it introduces significant complexity. HM type inference proves challenging in terms of soundness and completeness, whereas bidirectional type inference is easier to implement but still complex to prove complete. The study highlights the trade-offs of CbC programming, suggesting it is more suited for research and in-depth understanding rather than pragmatic programming.

1 Introduction

Static type systems enable programmers to define their intentions within the type signatures of their programs. This allows type checkers to provide feedback on whether the implementation aligns with these intentions. By doing so, type checkers can prevent numerous common errors, such as type mismatches (e.g., providing a Boolean where an Integer is expected), missing arguments, or looking up data fields that don't exist.

As an efficient method for early error detection, type checkers are a crucial part of statically typed programming languages. However, as type systems become more complex it becomes increasingly more likely that the type checker itself contains bugs [4] [16]. Bugs can be classified into two categories: some result in a correct program being incorrectly rejected (a false negative), while others cause an incorrect program to be accepted (a false positive). The latter is arguably worse, as they can lead to crashes or breaches of safety assumptions.

Correct-by-construction (CbC) programming [3] is a style of programming that uses precise types to ensure that a program adheres to its specification. CbC programming can be used to create specifications for various type systems along with their corresponding type checkers. If done correctly, the result is an intrinsically verified type checker, which is free of false positive bugs. This means that the type checker is sound. For some type checkers, it is also possible to prove that they are free of false negative bugs. Those type checkers are complete.

This paper presents a qualitative evaluation of the advantages and disadvantages of using CbC programming to implement type inference for the simply typed λ-calculus (STLC). We explore two algorithms: Hindley-Milner (HM) type inference [11] [14] and bidirectional type inference [15]. For the HM algorithm, we provide an implementation in Agda [2], a dependently typed language designed for CbC programming, that returns a well-formed type, but not a proof of typing (Section 3.3 goes into more detail about this limitation). For bidirectional type inference, we use the implementation provided by Wadler, Kokke, and Siek [19]. These two implementations form the basis for our arguments.

2 Background

We explain what it means to write CbC programs in Agda and why as a result we get a bug-free program. We then present the type system (STLC) that we will be type checking. Finally, we explain type inference.

2.1 Agda

In Agda, types can *depend* on values, hence the name dependently typed language. This allows us to create expressive datatypes, encoding rich information such as mathematical properties or algorithms.

A simple example [5] of a dependent type is the type of vectors `Vec A n`. This type contains lists of exactly n elements of type A.

```
myVec1 : Vec Nat 5
myVec1 = 1 :: 2 :: 3 :: 4 :: 5 :: []

myVec2 : Vec (Bool → Bool) 2
myVec2 = not :: (λ x → x) :: []

myVec3 : Vec Nat 0
myVec3 = []
```

It might not be obvious yet why dependent types are useful let alone how we express mathematical rules. Before we proceed, let's consider the relation of less-than-or-equal ($\leq$) on natural numbers in Agda (example taken from Wadler, Kokke, and Siek [19]):

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n : ℕ}
      --------
    → zero ≤ n

  m≤n : ∀ {m n : ℕ}
    → m ≤ n
      --------------
    → m+1 ≤ n+1
```

The above is a datatype with two constructors, `z≤n` and `m≤n`. They can be read as deductions: for example assuming m $\leq$ n, we conclude m+1 $\leq$ n+1. In an ordinary programming language, their types would be the same, e.g. `lte` (just like a `string` or `int`). In contrast, here they have types dependent on values of $\mathbb{N}$, which accurately represent the mathematical idea of less-than-or-equal.

Datatypes are intrinsic proofs of soundness. We can *only* construct a value of type `m ≤ n` if m is actually less than or equal to n. This is the essence of correct-by-construction programming. It is also possible to create an "invalid" datatype, which incorrectly represents an idea. For example, we can write a datatype for $<$ with the same constructors as the ones defined for $\leq$. It is up to the people expressing these ideas to define what is correct and what isn't.

2.2 The Simply Typed λ-calculus

The STLC type checked in this paper has two types, the function type and the type variable. A type variable is a placeholder for a concrete type (e.g. int, bool). We don't have concrete types in our language. While unusual, for the purposes of the paper they are unnecessary. Type variables are denoted as "`# x`" where x is a de Bruijn index [9]. The function type is denoted as "`# x -> # y`". For example, the identity function has type:

```
# 0 -> # 0
```

The formalization of the STLC in Agda can be found in appendix A.

2.3 Type inference

Type inference deduces the types of terms without explicit type annotations from the programmer. As an example, consider the `map` function written in Haskell:

```
map :: ?
map f [] = []
map f (x:xs) = f x : map f xs
```

Looking at what the code does, we can deduce the type of the map function to be:

```
(# 0 -> # 1) -> [# 0] -> [# 1]
```

Type inference algorithms enable the compiler to perform this deduction automatically. There are two prevalent methods of type inference: HM and bidirectional type inference.

HM type inference [11] [14] is a classical approach to type inference that dates back to the 1960s. It is most notably used in the metalanguage family of languages, including Haskell [7] and OCaml. It always infers the most general type [8] (explained by figure 1) with no type annotations from the programmer.

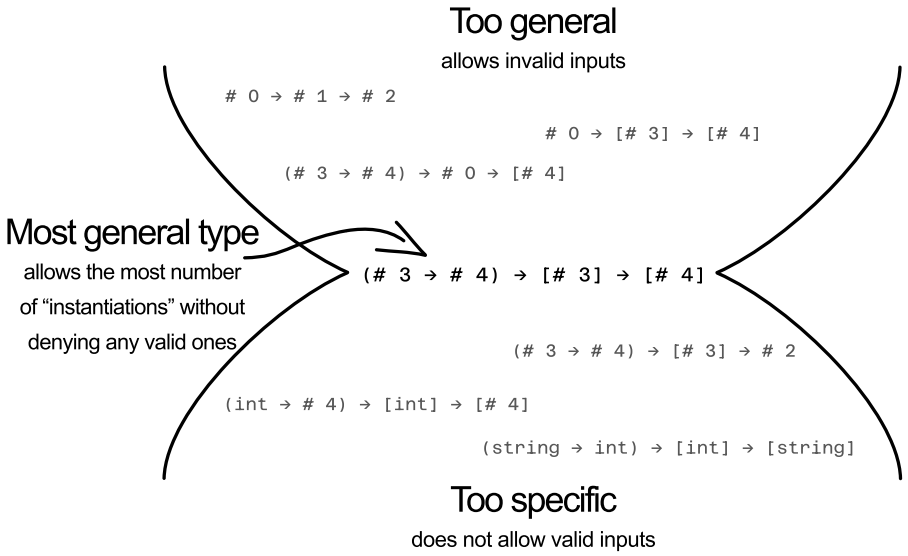

Figure 1: A visualization of different types of the map function.

In contrast, bidirectional type inference is a more recent approach that has gained popularity in languages with rich type systems such as Rust [18] and TypeScript [10]. This method is more flexible and can handle more complex type systems such as System F [17]. Bidirectional type inference requires some type annotations from the programmer.

3 Hindley-Milner type inference in Agda

Examples in this section are simplifications, which omit some details for the sake of brevity. For code examples, a reference to a corresponding appendix with the full implementation is given. Our implementation is built on top of the specifications of Abel [1]. We can break down the HM algorithm into 2 steps: constraint generation and unification.

3.1 Constraint generation

Constraints are equations between types, which describe how the types are related to each other within some term. A term can generate 0 or more constraints, which we call the system of constraint equations — akin to a system of linear equations. We now show an example that resembles constraint generation, but takes a few liberties for simplicity. Continuing with the `map` function from earlier:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

We first see that the function has 2 arguments, a function `f` and a list. Additionally, we must return some type. So, the *placeholder* type t of the `map` function is:

```
map :: # 0 -> # 1 -> # 2
```

Where `# 0` is the first argument's type, `# 1` is the second argument's type and `# 2` is the return type. The only goal of t is to ask the question "Is some substitution instance of t well typed?". The answer is given by unification, which either gives a substitution that yields the most general type or fails if the input was an ill-typed term. As previously mentioned, the first argument is a function `f`, so we generate the constraint

```
# 0 ≐ # 3 -> # 4
```

With `# 3` being `f`'s input type and `# 4` being the return type. The second argument is a list, so we generate the constraint

```
# 1 ≐ [# 3]
```

Finally, the return type is also a list, so we generate the constraint

```
# 2 ≐ [# 4]
```

As the example shows, the constraint generation algorithm may create new type variables as it traverses the term. Intuitively, it is essential that we avoid naming clashes. To that end we use renamings, denoted as m $\subseteq$ n. A renaming m $\subseteq$ n is a mapping from m type variables to n type variables. In other words, a renaming tells us the new value of an existing type variable after a constraint has been generated.

To generate constraints algorithmically, we define a function `generate` that takes as input an untyped term e alongside its context Γ and returns the constraint generation datatype denoted as $\Gamma \vdash e : A \dashv E \mid \eta$. It is dependent on 5 types:

- a context Γ, which maps variables to types
- a term e (variable, abstraction, application)
- a type A, which is the placeholder type of the expression
- a constraint E, the actual constraint generated
- a renaming η, which shows how constraint generation changed the type variables

The datatype can be read as "the term e has type A under the context Γ and it is accompanied by a constraint E and a renaming η". The full definition of the constraint generation datatype and `generate` can be found in appendix B. This is the function signature:

```
generate : Γ → e → Γ ⊢ e : A ⊣ E | η
```

Both the function and datatype are broken down into 3 cases: variable, abstraction, and application. The data constructor for variable constraints is:

```
var : Γ ⊢ var x : (lookup Γ x) ⊣ ϵ | id
```

It states that if we have a variable x and a context Γ, we can infer the type of x by looking it up in the context. `id` is the identity renaming (type variables are not changed in any way), and no constraints are generated (ϵ stands for empty constraint). Its corresponding case in the `generate` function is

```
generate Γ (var x) = var (Γ (var x) (lookup Γ x) ϵ id)
```

The function and constructor are virtually identical. Indeed, the same applies for the application and abstraction case in the sense that the constructor is so precise that the function is a direct translation.

There is one notable difference compared to a regular (non-CbC) implementation. We explicitly keep track of the number of type variables, denoted with Ξ. Contexts, renamings, types and constraints are all dependent on Ξ. Strictly speaking, this is necessary to implement a terminating unification algorithm. However, it also exposes nuanced details to us about what the individual pieces consist of and how they are changed. This aids our understanding of the algorithm and prevents bugs: it is not hard to imagine a scenario in a regular implementation, which has an off-by-one bug related to renamings. Simple bugs like these are impossible thanks to the types we use.

3.2 Unification

Once the constraints are generated, they must be solved by unification. The result of unification is a substitution, which makes both sides of an equation equal. Formally, a unification gives some substitution S, such that if $x = y$ is a constraint, then $S(x) = S(y)$. Note that unification can fail, in which case there is no equalising substitution. After solving a constraint, we record its resulting substitution and apply it to all other constraints. We generated 3 constraints for our `map` example:

```
map :: # 0 -> # 1 -> # 2
# 0 ≐ # 3 -> # 4
# 1 ≐ [# 3]
# 2 ≐ [# 4]
```

Solving the constraint `# 0 ≐ # 3 -> # 4` generates the substitution

```
# 3 -> # 4 for # 0
```

At this point we replace all occurrences of `# 0` with `# 3 -> # 4` in all other constraints. However, no constraints in our example contain `# 0`. Thus, we just record the substitution. Solving all the constraints gives us a composition of substitutions, which we can apply to the placeholder type to get the most general type for that term. For our example:

```
# 3 -> # 4 for # 0; [# 3] for # 1; [# 4] for # 2
```

applying it to the placeholder type `# 0 -> # 1 -> # 2`, we get the familiar type signature of the map function:

```
(# 3 -> # 4) -> [# 3] -> [# 4]
```

3.2.1 The occurs-check

Solving constraints involves replacing type variables with other types. However, we must be careful not to create infinite types. For example:

```
# 0 ≐ # 0 -> # 1
```

If we were to replace `# 0` with `# 0 -> # 1` in the other constraints, we wouldn't make any progress. Additionally, we wish to express the fact that if some `# x` does not occur in a type, the number of unique type variables has decreased. This is precisely what guarantees termination. Thus, a simple boolean check does not suffice. Instead, we map each type variable from a system of `suc`[1] `X` unique type variables to a system with X type variables while maintaining uniqueness. This mapping is called the `thick` function. Figure 2 (inspired by McBride [12], which introduced the algorithm) illustrates the point.

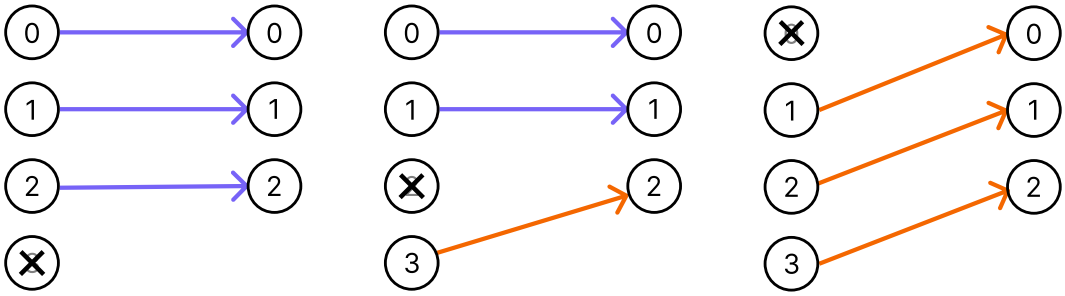

Figure 2: The `thick` function mapping. The colors highlight the two ways variables are mapped.

[1] suc stands for successor, e.g. if x = 1 then (suc x) = 2.

Our formalization of the unification algorithm requires a sound and complete implementation of the `thick` function. Thus, we define a datatype `StrTyVar` [1] that encompasses the idea of `thick`. As Figure 2 shows, there are 2 cases for soundness: the new type variable is smaller than the removed type variable (`suc-zero`), in which case the mapping is direct (shown in purple), or it is bigger (`zero-suc`), in which case the mapping is shifted by 1 (shown in orange) . The formalization is recursive; therefore, it also includes a `suc-suc` case.

```
-- X is the type variable that is removed
-- Y is the type variable we wish to map
-- Z is the new, mapped type variable
data StrTyVar : (X : TyVar (suc Ξ)) (Y : TyVar (suc Ξ)) (Z :
 TyVar Ξ)
  zero-suc : StrTyVar zero (suc Y) Y
  suc-zero : StrTyVar (suc X) zero zero
  suc-suc  : StrTyVar X Y Z → StrTyVar (suc X) (suc Y) (suc
 Z)
```

To prove that `thick` is complete, we must return evidence that we can not construct `StrTyVar` when $X = Y$. The explanation of the proof is greatly simplified. The chapter "Decidable" in Wadler, Kokke, and Siek [19] goes into great detail about the techniques we are using. We use the `Decidable` datatype, which is a type that can be either `yes <proof>` or `no <proof>`. We return Z and `StrTyVar` as a tuple.

```
thick : (X Y : TyVar Ξ) → Decidable (∃[ Z ] StrTyVar X Y Z)
thick zero zero = no ()
thick zero (suc Y) = yes (Y , zero-suc)
thick (suc X) zero = yes (zero , suc-zero)
thick (suc X) (suc Y) with thick X Y
... | yes (X' , proof) = yes (suc X' , suc-suc proof)
... | no ¬StrTyVarXYZ = no (¬StrTyVarXYZ → ¬(StrTyVar-sX-sY-
 sZ))
```

In the case of `thick zero zero`, Agda can infer that it is impossible to construct `StrTyVar` with the given arguments. In the recursive case, we essentially give the inverse of the `suc-suc` constructor. For the valid cases, we return the corresponding Z and constructor of `StrTyVar`.

On top of `thick`, we define the `check` function, which propagates `thick` through the structure of a type. The implementation of `check` is sound, but not complete. It is trivial to make it complete since `thick` is complete, but for our unification algorithm it is not necessary. The complete definition of both functions and their datatypes are given in Appendix C.

3.2.2 Unification Implementation in Agda

We introduce the Agda implementation piece by piece, explaining the individual cases of pattern matching. The full implementation can be found in appendix D. First, let's look at the type signature:

```
solve : E → Maybe (∃[ σ ] E ↘ σ)
```

The function takes a constraint E as input and outputs either nothing, in which case there is no unifying subsitution or the datatype `E` $\searrow$ σ, which states that the constraint E is solved by the substitution σ of type Subst Ξ Ξ_1 (from Ξ to Ξ_1 type variables). The `Subst` Ξ Ξ_1 type is well-scoped and well-typed [13].[2] The substitution σ and the corresponding constructor are returned as a tuple. Let's begin with the simplest case: the constraint is empty (ε).

```
solve ε = just (id , ε)
```

We return the identity substitution alongside the empty constructor.[3] Next, we examine the case where the equation has a type variable X on one side and *some* type A on the other side.

```
solve (tyvar X ≐ A) = do
  (A' , proof) ← check X A
  just (A' for X , X≐ proof)
```

We first check that the type variable X is not present in A using the `check` function. It returns A', which has 1 less type variable than A, and a proof of X's absence. Finally, we return the substitution that maps X to A' and leaves everything else unchanged. The constructor `X`$\doteq$ requires the proof as input to ensure correctness. This case is also defined the other way around, where the type variable is on the right-hand side. Functionally, everything stays the same.

The next case is when we have functions on both sides of the equation.

```
solve (A ⇒ B ≐ A' ⇒ B') = do
  (σ , proof) ← solve (A ≐ A' · B ≐ B')
  just (σ , ⇒E proof)
```

We break this case up to two separate cases based on the idea that their inputs must be equal and their outputs must be equal. Thus, we can break the constraint into two separate constraints and compose them to be solved individually.

Last but not least, we have the case where both sides of the equation are type variables. This case is to handle the edge case of the form `# x` $\doteq$ `# x`. This is a solvable constraint, but it would fail with the cases defined so far, as `# x` is present on both sides. Thus, we have to treat it separately.

```
solve (tyvar X ≐ tyvar Y) with thick X Y
... | yes (Z , proof)  = just (Z for X , X≐ proof)
... | no ¬proof = just (id , X≐X ¬proof)
```

If the function succeeds, `X` and `Y` were different and we can return the substitution that replaces `X` with `Z`. If it fails, `X` and `Y` must be equal and we return the identity substitution and the proof that they are the same.

[2]These properties are described in a draft paper, which forms the basis for the `Data.Fin.Substitution` module in the Agda standard library. This module, in turn, is what underpins the `Subst` Ξ Ξ_1 type we are using.

[3]Agda allows using the same name, in this case ε, for the constructors of different datatypes.

So far, we have closely followed McBride [12]: we are looking at cases with just one constraint. However, the algorithm presented by McBride is not for HM type inference, but for first-order logic. We will now deviate from it and go to the case of a constraint being a composition of constraints.

```
solve (E · F) = do
(σ , proof1) ← solve E
(τ , proof2) ← solve (subEqs σ F)
just (τ ∘ σ , proof1 ∘ proof2)
```

We solve E, which gives a substitution σ. We then solve F, but first apply the substitution σ to F. Solving the substituted F gives another substitution, τ. We return the composition of the subsitutions and the corresponding proof. However, there is a problem with this case. In the eyes of the Agda compiler, the function `subEqs` could extend F with additional constraints and thus lead to infinite recursion. To ensure termination, we'd have to [6]:

- define a small-step version of the unification algorithm that just does a single unification step and returns the new equations
- prove that the relation this defines on unification states is well-founded
- take the fixpoint of the solver, which uses the proof of well-foundedness (Agda's standard library defines this)

This is a major limitation – the above steps are non-trivial and require a deep understanding of certain proving techniques in Agda. To get around this, there are 2 options. The first option is to use either the `TERMINATING` or `NON-TERMINATING` pragma. The downside of these is obvious: there is no guarantee that the function terminates anymore. The second option is to include a fuel argument, which is a (arbitrary) natural number that decreases with each recursive call. If it reaches 0, we return nothing. The downside is that the algorithm can no longer be proven complete. Either way, both options lead to a situation where we can no longer prove that the algorithm is correct. For this paper, proving correctness is out of scope. Thus, we have decided to circumvent the problem by implementing the fuel argument.

There are a few differences compared to a regular implementation. Just as with constraint generation, we have to keep track of the number of type variables to create valid substitutions. The necessity of the fuel argument might seem like an inconvenience, but it is rather beneficial: an infinitely looping type checker would cause plenty of frustration to a programmer.

3.3 Combining constraint generation and unification

We now show the complete inference algorithm. It's a combination of everything we've introduced so far. It takes as input a context Γ and a term e and returns the most general type of the term. The full implementation can be found in appendix E.

```
infer : Γ → e → Maybe (∃[ Ξ₂ ] Ty Ξ₂)
infer Γ e = do
  let (_ , _ , A , E , _) = generate Γ e
  (Ξ₂ , σ , _) ← solve E 100000
  just (Ξ₂ , subTy σ A)
```

The context Γ has Ξ type variables. Constraint generation returns the placeholder type A with Ξ_1 type variables and the constraint E. We attempt to solve the constraint E with a fuel of 100000. If successful, it returns a substitution σ that maps Ξ_1 to Ξ_2 type variables. We then apply the substitution to the placeholder type A to get the most general type of the term.

While the individual components of the inference algorithm are sound, the function itself is not. For that, we'd have to return a well-typed term with the following signature [1]:

```
Tm (subCxt σ (wkCxt η Γ)) e (subTy σ A)
```

The context Γ is modified by first applying the renaming η and then the substitution σ. The term itself does not change and the type is substituted with σ. One might expect that since the individual components are sound, it would be trivial to combine them in a sound way. However, the crux of the problem is that constraint generation and unification compose renamings and subsitutions in a way that requires a multitude of lemmas to transform into the aforementioned structure. As an example, when inferring application, we'd have to prove the following transformation for contexts:

```
subCxt (renSub R.wk ∘ σ) (wkCxt η₂ (wkCxt η₁ Γ))
≡
subCxt σ (wkCxt (η₁ ⊙ (η₂ ⊙ R.wk)) Γ)
```

The exact mechanisms of the new elements (e.g. `R.wk` and `renSub`) are not important. Instead, we take away a lesson: it is not enough to have sound components. Complex datatypes transform types in complex ways and combining them in a generic way is difficult.

3.3.1 Example

An example can be shown via unit-testing. Since Agda has such a powerful type system, the tests are written in type signatures. Let's consider the Church numeral two as an example (in Haskell):

```
\x -> \y -> x (x y)
```

And in our type system:

```
two = λ (λ var 1 · (var 1 · var 0))
```

Let's input this term into the `infer` function with the empty context (since it is a closed-form term):

```
infer-two : Maybe (∃[ Ξ₂ ] Ty Ξ₂)
infer-two = infer [] two
```

We expect to infer the type `(# 0 -> # 0) -> (# 0 -> # 0)` with 1 unique type variable. We write this expectation as a test in Agda:

```
test-two : infer-two ≡ just (1 , (# 0 ⇒ # 0) ⇒ (# 0 ⇒ # 0))
test-two = refl
```

The type $x \equiv y$ has only one constructor, `refl` (short for reflexivity) and can only be instantiated if x and y are of the same type. The full implementation of the test can be found in appendix F.

4 Discussion

On the one hand, the basic implementation of the HM algorithm in Agda is relatively simple. The complexity does not lie in CbC programming, but rather in the algorithm itself. Still, a decent understanding of Agda is necessary. On the other hand, proving soundness and completeness is extremely difficult. The added complexity is intertwined with CbC programming. From the perspective of eliminating bugs, we *should* be proving those properties. Otherwise, our implementation is just as prone to bugs as a regular implementation.

Implementing bidirectional type inference in Agda is considerably simpler. Since bidirectional type inference doesn't require a global view of the program, the algorithm can directly deduce whether a well-typed term can be given or not. Thus, soundness is easy to achieve. Completeness is still difficult to prove and requires extensive knowledge of Agda. Figure 3 puts the comparison of HM and bidirectional type inference in a table.

	Basic	Sound	Complete
HM	moderate	hard	very hard
Bidir	easy	easy	hard

Figure 3: Subjective difficulty of implementing type inference in Agda.

Both methods scale relatively poorly. Soundness and completeness are not a one and done deal. As we extend our type checker, we must extend our proofs as well. For some features this is difficult with no existing implementations to draw inspiration from.

In many scenarios CbC programming "forces" the programmer to declare a multitude of bespoke datatypes for seemingly no reason. For example, when the familiar list would suffice for the constraints in a regular implementation, using CbC, we must define our own datatype. These datatypes add up and make it difficult for someone unfamiliar with the codebase to quickly grasp the main ideas. However, we argue that this is actually desirable. The types encode the ideas of the type checker. It is essentially impossible to contribute without deeply understanding them. We can think of them as puzzle pieces with an incredible level of fidelity. It is not possible to put them together in the wrong way and finding the right piece demonstrates understanding.

Dependent types are a powerful tool to describe our ideas and subsequently prevent bugs. However, as CbC type checkers quickly become complex (the datatype for constraint generation has five dependent types, each consisting of at least another dependent type), it is not obvious that the datatypes are correct. In a collaborative setting, it would be hugely beneficial if they were accompanied by a detailed explanation alongside an example or two.

In conclusion, we've shown an example implementation of the HM algorithm in Agda and discussed multiple trade-offs of using the CbC approach for implementing a type checker with type inference. Its value depends on the goals of a project. If the goal is to merely avoid some bugs, then the CbC approach *can* work, but probably adds too much additional complexity for pragmatic programmers. However, if the goal is to research type checkers, gain a deeper understanding of them, or to develop new features, then the CbC approach is invaluable.

5 Limitations and future work

The discussion did not go into great detail about proving techniques in Agda. Thus, a future evaluation could look into proving soundness and completeness of the HM algorithm in Agda and provide a more informative description of the proving methods while assessing their difficulty.

Another limitation of the paper is the simplicity of the language that we are type checking. The difficulty of extending it is discussed in a general manner. A future evaluation could look into extending the type checker with more advanced features and give a comparison of doing so with and without using CbC programming.

While this paper gives a general overview of the CbC style of programming for type inference, the preceding paragraphs show that there are many aspects left to be explored. A deeper dive into CbC programming would be beneficial to the community in a variety of ways. First of all, it would expose which parts of CbC programming are underrepresented in terms of learning material *and* which techniques require careful attention when explained. Secondly, it would aid beginners (and researchers) in deciding whether to use CbC programming to solve their problem at hand.

6 Responsible Research

This research focuses solely on logic and mathematical methods, avoiding any human or animal subjects. The results are derived from the created code, which is available for public audit on GitHub[4], ensuring transparency and verification. No external data was used, and the codebase is accessible for independent validation.

[4]https://github.com/VincentPikand/cbc-type-checker

References

[1] Andreas Abel. *Agda Formalization of Constraint-Based Type Inference for the Simply-Typed Lambda-Calculus*. https://github.com/andreasabel/constraint-based-type-inference.

[2] *Agda Programming Language*. https://github.com/agda/agda. Accessed: 2024-06-11. 2024.

[3] Tabea Bordis et al. "Correctness-by-Construction: An Overview of the CorC Ecosystem". In: *Ada Lett.* 42.2 (Apr. 2023), pp. 75–78. ISSN: 1094-3641. DOI: 10.1145/3591335.3591343. URL: https://doi.org/10.1145/3591335.3591343.

[4] Stefanos Chaliasos et al. "Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485500. URL: https://doi.org/10.1145/3485500.

[5] Jesper Cockx. *Agda Lecture Notes for the Functional Programming Course at TU Delft*. https://github.com/jespercockx/agda-lecture-notes/tree/master. 2024.

[6] Jesper Cockx. *Private Communication*. Personal email communication. June 2024.

[7] HaskellWiki contributors. *Type inference*. https://wiki.haskell.org/index.php?title=Type_inference&oldid=17047. Accessed: 2024-05-07. Nov. 2007.

[8] Luís Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 1982), pp. 207–212. DOI: 10.1145/582153.582176.

[9] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[10] *Documentation - Type Inference*. TypeScript Documentation. Accessed on 28 April 2024. URL: https://www.typescriptlang.org/docs/handbook/type-inference.html.

[11] R. Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 00029947. URL: http://www.jstor.org/stable/1995158 (visited on 04/27/2024).

[12] Conor McBride. "First-order unification by structural recursion". In: *Journal of Functional Programming* 13.6 (2003), pp. 1061–1075. DOI: 10.1017/S0956796803004957.

[13] Conor McBride. "Type-Preserving Renaming and Substitution". In: *Journal of Functional Programming* (Under consideration for publication). draft paper.

[14] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[15] Benjamin C. Pierce and David N. Turner. "Local type inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. URL: https://doi.org/10.1145/345099.345100.

[16] Akond Rahman et al. “Defect Categorization in Compilers: A Multi-vocal Literature Review”. In: *ACM Comput. Surv.* 56.4 (Nov. 2023). ISSN: 0360-0300. DOI: 10.1145/3626313. URL: https://doi.org/10.1145/3626313.

[17] John C. Reynolds. “Towards a theory of type structure”. In: *Symposium on Programming*. 1974. URL: https://api.semanticscholar.org/CorpusID:30450751.

[18] *Statements - The Rust Reference*. Rust Documentation. Accessed on 28 April 2024. URL: https://doc.rust-lang.org/reference/statements.html.

[19] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Aug. 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

A STLC specification

```
-- Well-scoped terms.

data Exp : ℕ → Set where
  var : (x : Fin n) → Exp n
  abs : (e : Exp (suc n)) → Exp n
  app : (f e : Exp n) → Exp n

-- Type variables.

TyCxt = ℕ

TyVar : (Ξ : TyCxt) → Set
TyVar = Fin

-- Types.

data Ty (Ξ : TyCxt) : Set where
  tyvar : (X : TyVar Ξ) → Ty Ξ
  _⇒_   : (A B : Ty Ξ) → Ty Ξ

-- Typing contexts.

Cxt : (Ξ : TyCxt) (n : ℕ) → Set
Cxt Ξ n = Vec (Ty Ξ) n

-- Well-typed terms.

data Tm (Γ : Cxt Ξ n) : (e : Exp n) → Ty Ξ → Set where
var :  Tm Γ (var x) (lookup Γ x)
app : (t : Tm Γ f (A ⇒ B)) (u : Tm Γ e A)
  → Tm Γ (app f e) B
abs : (t : Tm (A :: Γ) e B) → Tm Γ (abs e) (A ⇒ B)
```

B Constraint generation

The module `R` declares the subsitution operations for renamings. "wk" stands for weakening i.e. increasing the number of type variables in a type context.

```
data Inf : (Γ : Cxt Ξ n) (e : Exp n) (η : Ξ ⊆ Ξ₁) (A : Ty Ξ₁) (E
    : Eqs Ξ₁) → Set where
  var : Inf Γ (var x) R.id (lookup Γ x) ε

  abs : let X = tyvar zero
        in Inf (X :: wkCxt1 Γ) e η A E
        → Inf Γ (abs e) (R.wk R.∘ η) (wkTy η X ⇒ A) E

  app : Inf Γ f η₁ C E
      → Inf (wkCxt η₁ Γ) e η₂ A F
      → let X = tyvar zero
            η₂′ = η₂ R.∘ R.wk
        in Inf Γ (app f e) (η₁ R.∘ η₂′) X (wkEqs η₂′ E · (wkEqs1 F ·
    (wkTy η₂′ C ≐ wkTy1 A ⇒ X)))
----------------------------------------
generate : (Γ : Cxt Ξ n) → (e : Exp n) → Σ[ Ξ₁ ∈ TyCxt ] Σ[ η
   ∈ (Ξ ⊆ Ξ₁) ] Σ[ A ∈ (Ty Ξ₁) ] Σ[ E ∈ (Eqs Ξ₁) ]  Inf Γ e η A
   E
generate {Ξ} Γ (var x) = Ξ , R.id , (lookup Γ x , (ε , Inf.var)
   )
generate Γ (abs e) =
  let
    (Ξ₁ , η , A , E , e')  = generate (tyvar zero :: wkCxt1 Γ) e
  in
    Ξ₁ , R.wk R.∘ η , wkTy η (tyvar zero) ⇒ A , E , Inf.abs e'
generate Γ (app f e) =
  let
    (Ξ₁ , η₁ , C , E , f')  = generate Γ f
    (Ξ₂ , η₂ , A , F , e')  = generate (wkCxt η₁ Γ) e
    X = tyvar zero
    η₂′ = η₂ R.∘ R.wk
  in
    suc Ξ₂ , η₁ R.∘ η₂′ , X , wkEqs η₂′ E · wkEqs1 F · (wkTy η₂′ C ≐
   wkTy1 A ⇒ X) , app f' e'
```

C Occurs-check

The `delete` function is a helper function that removes a type variable from a type context.

```
delete : (X : TyVar Ξ) → TyCxt
delete {Ξ = suc Ξ} zero = Ξ
delete (suc X) = suc (delete X)

data StrTyVar : (X : TyVar Ξ) (Y : TyVar Ξ) (Z : TyVar (delete
   X)) → Set where
  zero-suc : StrTyVar zero (suc Y) Y
  suc-zero : StrTyVar (suc X) zero zero
  suc-suc  : StrTyVar X Y Z → StrTyVar (suc X) (suc Y) (suc Z)

data StrTy : (X : TyVar Ξ) (A : Ty Ξ) (A' : Ty (delete X)) →
   Set where
  tyvar : StrTyVar X Y Z
        → StrTy X (tyvar Y) (tyvar Z)

  _⇒_   : StrTy X A A'
        → StrTy X B B'
        → StrTy X (A ⇒ B) (A' ⇒ B')


thick : (x y : TyVar Ξ) → Dec (∃[ z ] (StrTyVar x y z))
thick zero zero = no  (λ ())
thick zero (suc y) = yes (y , zero-suc)
thick (suc x) zero = yes (zero , suc-zero)
thick (suc x) (suc y) with thick x y
... | yes (x' , p) = yes (suc x' , suc-suc p)
... | no ¬z = no λ{ (suc z , suc-suc snd) → ¬z (z , snd)}


check : (X : TyVar Ξ) → (A : Ty Ξ) → Maybe (Σ (Ty (delete X))
    (StrTy X A))
check X (tyvar Y) with thick X Y
... | yes (Z , proof) = just (tyvar  Z , tyvar proof)
... | no _ = nothing
check X (A ⇒ B) = do
  (A' , p₁) <- check X A
  (B' , p₂) <- check X B
  just (A' ⇒ B' , p₁ ⇒ p₂)
```

D Constraint solver

```
solve : (E : Eqs Ξ) → ℕ → Maybe (Σ[ Ξ₁ ∈ TyCxt ] Σ (Subst Ξ
   Ξ₁) (E ↘_))
solve _  zero = nothing
solve {Ξ} ε _ = just (Ξ , idSub , ε)
solve {Ξ} (tyvar X ≐ tyvar Y) _ with thick X Y
... | yes (y' , p) = just (delete X , sgSub X R.id (tyvar y') ,
    X≐ (tyvar p))
... | no neg-p = just (Ξ , idSub , X≐X neg-p)
solve (tyvar X ≐ A) _ = do
  (A' , p) <- check X A
  just (delete X , (sgSub X R.id A' , X≐ p))
solve (A ≐ tyvar X) _ = do
  (A' , p) <- check X A
  just (delete X , (sgSub X R.id A' , ≐X p))
solve (A ⇒ B ≐ A' ⇒ B') (suc n) = do
  (Ξ₁ , σ , p) <- solve (A ≐ A' · B ≐ B') n
  just (Ξ₁ , (σ , (⇒E p)))
solve (E · F) (suc n) = do
  (Ξ , σ , p₁) <- solve E n
  (Ξ₂ , τ , p₂) <- solve (subEqs σ F) n
  just (Ξ₂ , subSub τ σ , p₁ · p₂)
```

E Infer function

```
infer : (Γ : Cxt Ξ n) → Exp n → Maybe (∃[ Ξ₂ ] Ty Ξ₂)
infer Γ e = do
  let (Ξ₁ , η , A , E , e')  = generate Γ e
  (Ξ₂ , σ , p) <- solve E 100000
  just (Ξ₂ , subTy σ A)
```

F Example

```
two : Exp 0
two = abs (abs (app (var (suc zero)) (app (var (suc zero)) (var
    zero))))

two-tc : Maybe (∃[ Ξ₂ ] Ty Ξ₂)
two-tc = infer {zero} [] two

test-two : two-tc ≡ just (1 , ((tyvar zero) ⇒ (tyvar zero)) ⇒
    ((tyvar zero) ⇒ (tyvar zero)))
test-two = refl
```