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

Supervisor
Sára Juhošová

Author
Vincent Pikand

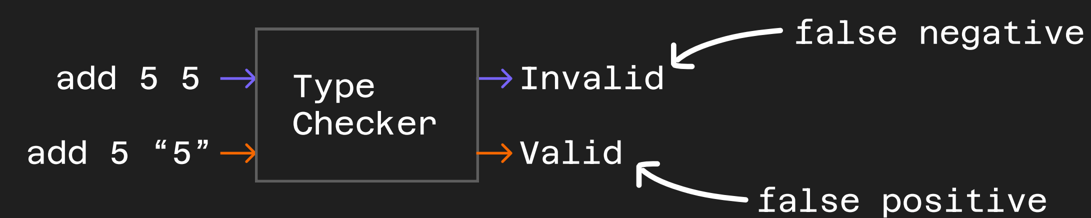
Responsible Professor
Jesper Cockx

Introduction

Static type systems enable programmers to define their intentions within the type signatures of their programs. By doing so, type checkers can prevent numerous common errors:

```
1 data Coordinate = { x :: Double, y :: Double }
2 c = Coordinate {5, 5}
3 print(c.z)
4
5   Error: z does not exist in record type Coordinate
6
```

However, as type systems become more complex it becomes increasingly more likely that the type checker itself contains bugs [0].



Correct-by-Construction programming

Correct-by-Construction is a style of programming that uses precise types to ensure that a program adheres to its specification. Agda [1] is a dependently typed programming language specifically designed for CbC programming. We present a qualitative evaluation of the advantages and disadvantages of using CbC programming to implement type inference for the simply typed λ-calculus (STLC).

Type Inference

Type inference deduces the types of terms without explicit type annotations from the programmer.

```
function f(x) {
  return x * 2;
}
```

```
function f(x : int) : int {
  return x * 2;
}
```

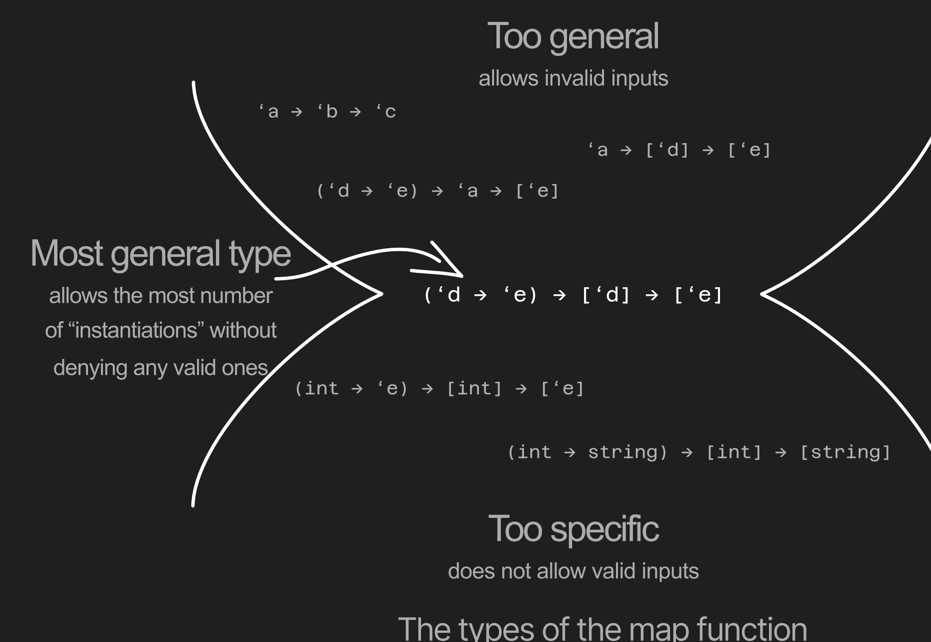
Agda

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.

```
1 data ≤ : ℕ → ℕ → Set where
2 z≤n : zero ≤ n
3 m≤n : m ≤ n → m+1 ≤ n+1
```

The Hindley-Milner (HM) Algorithm

The HM [2] algorithm is a type inference algorithm that requires *no* type annotations from the programmer. It always returns the most general type of a term. We can break down the HM algorithm into 2 steps: constraint generation and unification.



The types of the map function

Constraint Generation

Constraint generation describes the relationships between types. We implemented a sound constraint generation algorithm. The datatype for constraint generation is extremely precise – it exposes nuanced details that aid our understanding of the algorithm.
As types are meant to convey intention, using precise types elevates this purpose significantly. Thus, including a detailed description with motivating examples becomes immensely beneficial.

Unification

Unification is what gives meaning to the relationships described by constraint generation and tells us how to construct the most general type for a term.
We implemented a sound unification algorithm. As constraints can be a composition of constraints, e.g. C = A · B, it is difficult to show Agda that our algorithm terminates. To circumvent the issue, we use a fuel argument, which gives finite steps to solve the problem. This is a good thing – an infinitely running type checker would be quite tedious.

The complete inference algorithm

The combined algorithm takes an untyped term and returns a type. To return a well-typed term, which would imply soundness, we'd have to transform complex types in non-trivial ways, which is beyond the scope of the thesis.

HM and bidirectional type inference, compared

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

The basic implementation of the HM algorithm in Agda is relatively simple. Proving soundness and completeness is extremely difficult.
Implementing bidirectional type inference in Agda is considerably simpler. Since it doesn't require a global view of the program, the algorithm can directly deduce the type of a term. Thus, soundness is easy to achieve. Completeness is still difficult.
Both methods scale relatively poorly as proofs have to be extended with added features.

Conclusion

Dependent types are a powerful tool to describe our ideas and subsequently prevent bugs. Being able to use the datatypes is strong evidence of understanding the underlying concepts.
The value of CbC programming depends on the goal of a project. If we wish to avoid some bugs, then the CbC approach can work, but probably adds too much additional complexity for pragmatic programmers. 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.

Limitations

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.

References

[0] 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.
[1] Agda Programming Language. https://github.com/agda/agda. Accessed: 2024-06-11. 2024.
[2] 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.