Three Type-Inference Algorithms
ML-Club Talk

Bruce J. McAdam

26th February 1998

Title Slide

I will explain my new type-inference algorithm and review two recent papers on other type-checking algorithms. All three algorithms are designed to handle *incorrectly typed* programs better than the conventional algorithm.

My own algorithm, W' [McA98], uses *unification of substitutions* to type-check application expressions symmetrically. It is designed to be more bottom up than the conventional algorithm W and to reduce the bias towards detecting errors late in the program.

The second algorithm is from the paper *Debugging Type Errors* by Karen L. Bernstein and Eugene W. Stark [BS95]. It introduces the notion of *assumption environments* to handle unbound variables and to infer types for expressions in a context free manner.

In contrast, the third algorithm is highly context sensitive (a truly top-down type-inference algorithm). M was used in an early implementation of Caml and is analysed in the paper *Proofs about a Folklore Let-Polymorphic Type Inference Algorithm* by Oukseh Lee and Kwangkeun Yi [LY98].

This document alternates my slides and notes, the notes aren't really designed for anyone else to read. Please e-mail me at bjm@dcs.ed.ac.uk for an explanation of any points.

Introduction

We will look at some pragmatic issues about using Hindley-Milner type-systems.

- Why is type-inference difficult (especially when programmers have made mistakes)?
- What algorithm do we normally use?
- What problems does the usual algorithm have when programmers make mistakes?
- My algorithm unifies substitutions.
- My algorithm has several advantages.
- Bernstein and Stark have another algorithm.
- So do Lee and Yi.
- There is much more to be studied in this area.

Slide 1: Introduction

This slide just summarises the contents of the talk (one bullet per slide).

Terminology note: Programs are either typeable (we can infer a type for them), or untypeable (we can't). Analogous to "correct" and "incorrect". I have tried to use "infer" instead of "type-check", but sometimes it sounds funny (compare "type-inferer" with "type-checker").

Background

- We like strong polymorphic type-systems :-),
- but we make mistakes when using them :-(.
- The error messages compilers give are usually *vague*,
- because there is an *inherent difficulty* in locating the source of type inconsistencies.
- It is difficult to design type-inference algorithms as systems are formulated using nondeterministic rules,
- but we need to develop algorithms which handle both *typeable* and *untypeable* programs.

We will be looking at three algorithms designed to overcome these difficulties.

Slide 2: Background

Problems with error messages are

Vague reasons since it is difficult to relate a unification failure to the programmers syntactic construction.

Vague locations since the point of type-inference failure is generally not the place the programmer made a mistake.

Not only do implementations tell you very little, but they are very abrupt about telling you it.

Inherent Difficulty: *Implicit typing* means there is no standard to mark programs against (only inconsistencies between expressions). *Polymorphic typing* means that some mistakes can go unnoticed (having an unexpected type). So on an intuitive level, the question “what is my mistake?” is difficult to answer.

Nondeterminism: It us useful to formulate systems nondeterministically, but it makes the algorithms difficult to design/implement. Also, the sorts of data-structures in the systems are sometimes difficult to implement.

See [McA98] for more on this.

Algorithm W

The usual algorithm for Hindley-Milner type-inference is W. Takes type-environment, Γ, and expression, e, returns substitution, S, and type, τ.

$$
\begin{array}{rcl}
W(\Gamma, x) & = & (Id, \Gamma(x)) \\
W(\Gamma, e_0 e_1) & = & \textbf{let} \\
 & & \quad (S_0, \tau_0) = W(\Gamma, e_0) \\
 & & \quad (S_1, \tau_1) = W(S_0\Gamma, e_1) \\
 & & \quad V = U(V\tau_0, \tau_1 \to \beta) \\
 & & \textbf{in } (VS_1S_0, V\beta) \\
W(\Gamma, \lambda x.e) & = & \textbf{let } (S, \tau) = W(\{x : \alpha\}\Gamma_x, e) \\
 & & \textbf{in } (S, S\alpha \to \tau) \\
W(\Gamma, \textsf{let } x = e_0 \textsf{ in } e_1) & = & \textbf{let} \\
 & & \quad (S_0, \tau_0) = W(\Gamma, e_0) \\
 & & \quad (S_1, \tau_1) = W(S\Gamma_x \cup \{x : \tau_0\}, e_1) \\
 & & \quad \textbf{in}(S_1S_0, \tau_1)
\end{array}
$$

Most compilers use W in some optimised form.

Slide 3: Algorithm W

(This isn't one of the 'three algorithms').

I have used Γ for the type-environment.

Describe the algorithm. See [DM82].

Essentially, we derive the most general possible derivation for an expression's type, then apply appropriate substitutions to turn it into a more specific derivation. We have a lemma which tells us we can apply substitutions to types and assumptions in a conclusion.

Substitutions tell us how non-generic type-variables were used in the derivation.

Type-inference can fail with $e_0 e_1$ if U fails. This causes an error message to be generated.

Describe soundness and completeness theorems. Note that for the final result, the substitution can be discarded as we are finding the type of a let-bound top level value.

What's wrong with W

We have all seen error messages produced by compilers based on W. Usually the mistake we made occurs before the line highlighted by the compiler.

W has a *left–right bias*.

Part of the problem is

- When inferring a type for an application, the substitution from e_0 is used for e_1,
- If e_0 contains a mistake, then S_0 may be wrong,
- There appears to be a mistake *inside* e_1 — but there is actually an inconsistency *between* e_0 and e_1.

It is also difficult for the implementation to say *why* it failed, in terms the programmer can understand. The programmer makes more mistakes than just applying the wrong argument to a function.

Slide 4: What's wrong with W

The problems come from the way in which unification and substitutions are used.

Heuristics are typically used to get a message from a particular failure of unification.

The three new algorithms in this paper each try to overcome some of these problems in different ways. Each has a notably different way of using unification.

My algorithm — Symmetric inference with U_S

To eliminate the left–right bias, the recursive calls should be $W(\Gamma, e_0)$ and $W(\Gamma, e_1)$. This results in two substitutions which must be combined.

The resulting substitution is the *unification* of S_0 and S_1. I have an algorithm, U_S, for computing this (too complicated to fit on a slide).

W only changes in the case of applications:

$$
\begin{aligned}
W'(\Gamma, e_0 e_1) \quad = \quad & \textbf{let} \\
& \quad (S_0, \tau_0) = W'(\Gamma, e_0) \\
& \quad (S_1, \tau_1) = W(\Gamma, e_1) \\
& \quad S' = U_S(S_0, S_1) \\
& \quad V = U(S'\tau_0, \beta \to S'\tau_1) \\
& \textbf{in } (VS', V\beta)
\end{aligned}
$$

Slide 5: My algorithm — Symmetric inference with U_S

W' and U_S are described (and proven correct) in [McA98].

This eliminates left–right bias from applications (though not declarations).

U_S can be applied to anything using unification and applying substitutions to type-environments. Not just relevant to Hindley-Milner systems.

I will show what the results of U_S look like (on the blackboard).

The modified version of W is called W' in the tradition of uninspired algorithm names.

Advantages of my algorithm

W' distinguishes two different problems

- *Inconsistent subexpressions*. E.g. if e_0 uses i as and integer and e_1 uses it as a real.
- *Incompatible function and parameter*. E.g. if e_0 expects an integer, but e_1 is a real.

W gets these confused leading to misleading error messages. If the programmer knows which is the problem, he can apply different debugging techniques.

I further extended the algorithm to treat entire curried applications symmetrically. This means the inference looks at the program *in the same way* as the programmer.

U_S can be used on other algorithms (and other type-systems). It is applicable for type-checking as well as type-inference, and in polymorphic or monomorphic settings.

Less work need be performed if we modify the program and infer a new type.

Slide 6: Advantages of my algorithm

Explain the two types of inconsistency. W detects an inconsistency as if it is an incompatibility somewhere inside e_1.

Explain why U_S is generally useful.

Explain why curried expressions are important. Note that the same holds for any similar expressions (tuples, nested applications. . .).

Automated repair will be possible as we can modify the program and don't need to retraverse large areas of code (e.g. we can rearrange parameters). Discuss the reduction in re-computation required.

Bernstein & Stark

Use *assumption environments*: finite mappings from identifiers to *sets of types*, Δ.

Define inference using Δ. $\Delta \rhd e : \tau$.

Show how inference using assumption environment relates to inference using type environments.

Define an algorithm which simulates derivations in the new system (a rather natty idea).

Unification occurs at abstractions and declarations (when names are bound). Like in my algorithm, applications are traversed symmetrically.

Their implementation does not use the algorithm, it is slightly weaker (closer to W).

The programmer is supposed to debug programs by finding the assumption-environment relating to open expression. I think this is a poor idea.

I am planning to look as substitutions which map tyvars to sets of types to handle ambiguities in program. This will result in a similar algorithm.

Slide 7: Bernstein & Stark

Paper is [BS95]. I photocopied the derivation system and algorithm from it onto a slide.

This is a good paper. I like the technique of describing a new type-system (using assumption environments) then showing it to be equivalent to the usual type system.

Their debugging method is poor and uninspired. Too much like copy and pasting into a command line environment. The *breakpoint* idea is quite nice but should probably be explicitly added to the language.

The idea of having several types for a name is similar to an idea I had — to create a superposition of possible substitutions, in which a single type-variable maps to a number of types (these mappings can be labelled to show that if you believe one part of a program then a certain instantiation is required). (I will try to write about on this on the blackboard).

Lee & Yi

Algorithm M — *top-down* (upside-down) version of W.

Used in CamlLight 0.61. But this is the first paper giving proofs.

Context sensitive: the context tells us roughly what type to derive. From (Γ, e, ρ) derive S s.t. $S\Gamma \vdash e : S\rho$.

Unification occurs at identifier and abstraction expressions. There is a left–right bias in applications which could be removed by using my U_S algorithm.

The paper concentrates on showing that M terminates sooner than W when it has untypeable input.

I think that programs are better traversed in a bottom up manner to see whether parts are internally consistent, rather than whether they fit their context (since the context may have mistakes). However this is useful for bidirectional type-inference.

Slide 8: Lee & Yi

Paper is [LY98], I photocopied the algorithm from it onto a slide.

The algorithm was originally by Xavier Leroy, used in CamlLight 0.61. This can be combined with W, as it is in the Centaur SML system [RT97], also see Pierce and Turner's $\mathit{Local\ type\text{-}inference}$ [PT97].

Seems odd to me that quick failure is considered positive. Sure it is better than W's way of being vague, but I think we want context independence.

I don't like the context sensitivity as usually we are no more sure that the context is correct than any other part of the program. E.g. we may derive that e_0 expects an integer, but this could be wrong so it is wrong to call $M(\Gamma, e_1, \mathsf{int})$. This may however be useful for dealing with specifications such as type-constraints and signatures.

Conclusions

Summary

- W cannot give useful information about untypeable programs.
- There are a number of ways to infer types and deal with untypeable programs.
- U_S can remove left–right bias.
- Bernstein and Stark use assumption environments to infer types in a completely context-free way.
- Lee and Yi use a context-sensitive method. It can be used for bidirectional type-inference.

Future work

- Look at substitutions with multiple types (superposition).
- Apply a Duggan style labelling to U_S.
- Can we infer typeable programs from untypeable programs?

Slide 9: Conclusions

References

[BS95] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, Computer Science Department, 1995.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, (27):37–83, 1996.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual Symposium on Principles of Programming Languages*. Association of Computing Machinery, 1982.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM transactions on Programming Languages and Systems*, 1998. Accepted for publication.

[McA98] Bruce J. McAdam. Symetric type-inference. In progress, 1998.

[PT97] Benjamin C. Pierce and David N. Turner. Local type inference. Technical Report 493, CSCI, Indiana University, November 1997.

[RT97] Laurence Rideau and Laurent Théry. Interactive programming environment for ML. Technical Report 3139, Inria, March 1997.