Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 6
The Implementation of Type Inference

CS4012

Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

Type Inference

As we know:

- Haskell is *strongly typed*. Incorrectly typed programs are rejected at compile time.
- The type system used is fairly sophisticated:
    - Types may be constructed as *sums* (e.g. choices between constructors) or *products* (e.g. combinations of types).
    - Types may be *polymorphic*, meaning type variables may be universally quantified over sets of types
    - The compiler is capable of *automatic type inference*, allowing it to discover the types of functions for itself.

For completeness…

We won't consider them today, but for completeness:

- Haskell functions can also take sets of types to provide a kind of structured overloading. This facility is provided via *type classes*
- And, of course, we know there are many extensions to the basic type system that people are experimenting with.
- We will ignore this today, but note that everything we discuss generalises (with varying levels of ease) to cover these

Haskell’s type system

Today we will introduce some of the ideas behind Haskell’s type inference mechanism.

The type system implementation must be able to:

- Determine whether the program is well typed
- If so, determine what the type of any expression in the program actually is.

The programmer (presumably) believes their program is well typed. They have an idea of the type of each *expression* in the program. They could, in fact, have labelled each expression with that type.

In a sense, the job of the type checker is to recover those “lost” type labels on each part of the program.

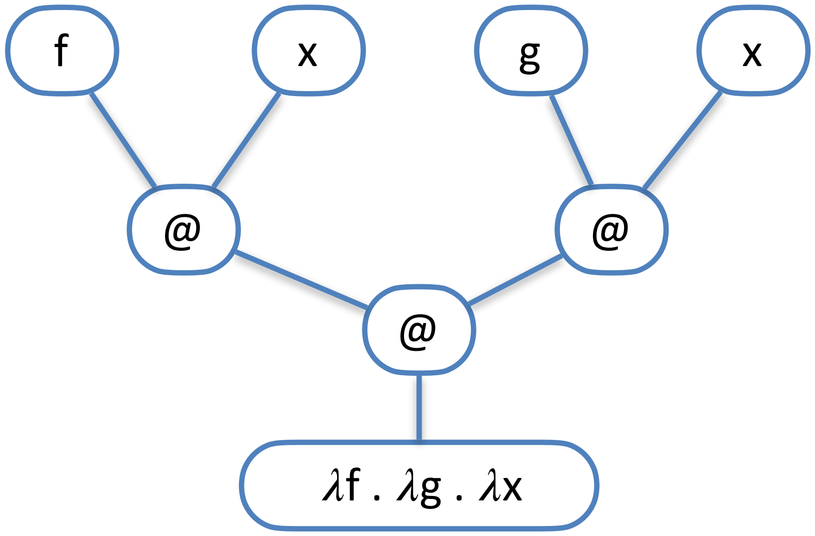
The type inference system

- **Typically we think of a type system as two separate but related things:**
- **A set of type, or *inference* rules which give the logical framework for the system. This is usually used to demonstrate that the system is sound.**
- **An inference *algorithm* which can be used to deduce the types of expressions. Determining a usable inference algorithm for a given set of rules is usually non-trivial.**

Example

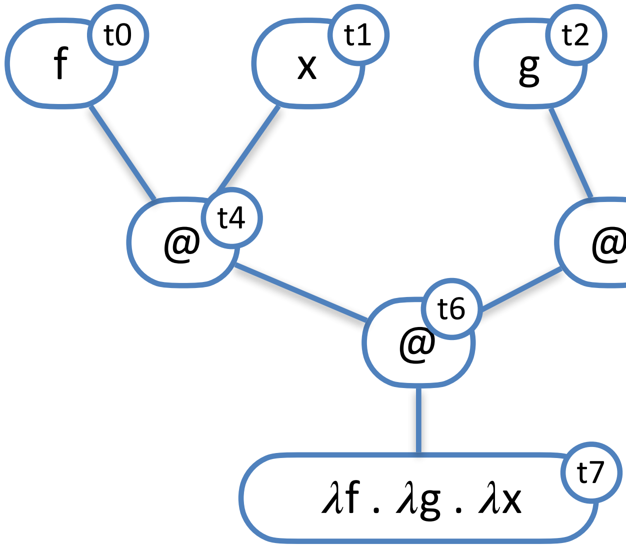
- **Take a simple expression, for example:**

```
\f g x -> f x (g x)
```



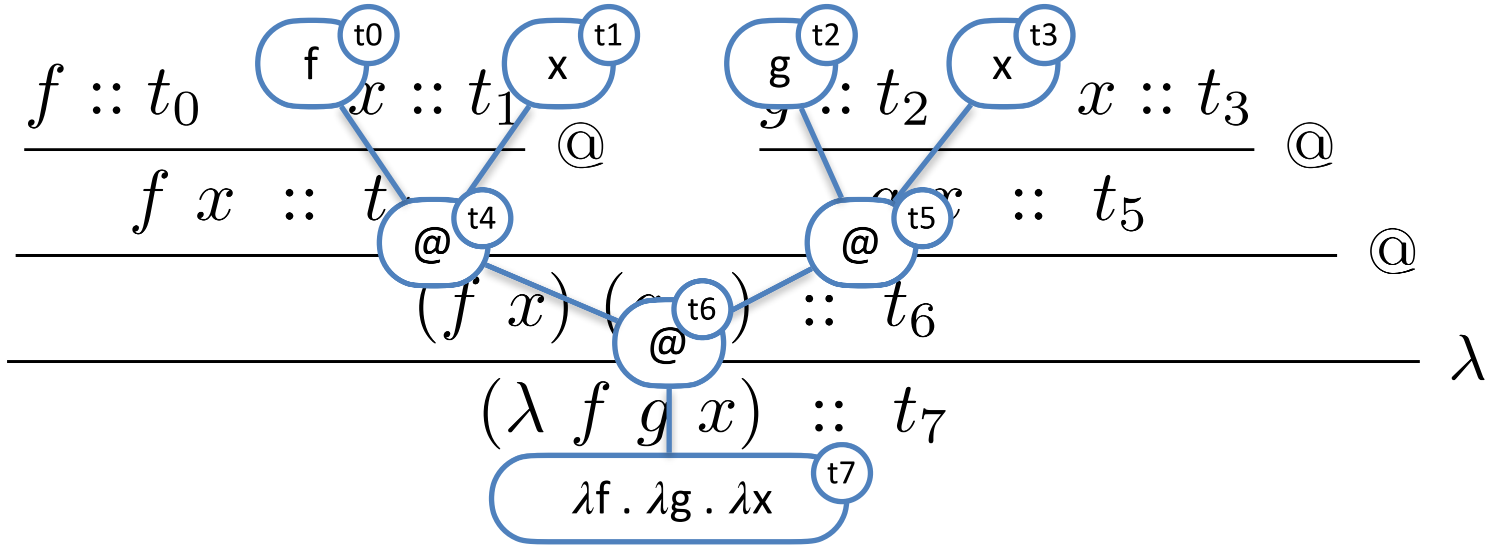
Example

- **Assigning type variables to each node of the tree**



Example

- I want to re-write this tree in a form that will emphasise the relationship this has to certain kinds of logical proofs.



Type rules

To capture my type system I will create some rules.

For example, when I have a function application I know that the the function itself should have an “arrow” type, and that type should relate the type of the argument to the type of the result.

These rules will be presented in a similar “tree” format; the thing we wish to prove goes on the bottom, the pre-conditions required go above.

Type rules

Application

So for any part of the tree that looks like this:

$$\frac{E_0 :: A \qquad E_1 :: B}{E_0 \; E_1 :: C} \; @$$

I should have a rule that says that the type "A" needs to be an arrow type that converts the type of the domain ("B") of the function to the type of the range ("C"):

$$\frac{E_0 :: t_0 \rightarrow t_1 \qquad E_1 :: t_0}{E_0 \; E_1 :: t_1} \; \mathrm{APP}$$

Type rules

Substitutions

$$\frac{E_0 :: t_0 \to t_1 \qquad E_1 :: t_0}{E_0\ E_1 :: t_1}\ \text{APP}$$

$$\cfrac{\cfrac{\cfrac{f :: t_0 \qquad x :: t_1}{f\ x\ ::\ t_4}\ @ \qquad \cfrac{g :: t_2 \qquad x :: t_3}{g\ x\ ::\ t_5}\ @}{(f\ x)\ (g\ x)\ ::\ t_6}\ @}{(\lambda\ f\ g\ x)\ ::\ t_7}\ \lambda$$

Using this rule I can come up with three equations (one for each application) that allow me to substitute one type for another in the tree:

$$t_0 = t_1 \to t_4$$
$$t_2 = t_3 \to t_5$$
$$t_4 = t_5 \to t_6$$

These could be substituted back into the tree.

$$\cfrac{\cfrac{\cfrac{f :: t_1 \to t_5 \to t_6 \qquad x :: t_1}{f\ x\ ::\ t_5 \to t_6}\ @ \qquad \cfrac{g :: t_1 \to t_5 \qquad x :: t_3}{g\ x\ ::\ t_5}\ @}{(f\ x)\ (g\ x)\ ::\ t_6}\ @}{(\lambda\ f\ g\ x)\ ::\ t_7}\ \lambda$$

Type rules

Lambda bound variables are monomorphic

To determine a type for the abstraction we can use a rule that looks like this

$$\frac{x :: A \qquad E :: B}{\lambda\, x.E \;::\; A \to B}\ \mathrm{ABS}$$

The final type is going to present a problem. What to do about "x" in the eventual result?

$$t_7 = t_0 \to t_2 \to t_1^3 \to t_6$$

If we assume that both types for x must be the same then we can add one more equation, and so:

$$t_1 = t_3$$
$$t_7 = t_0 \to t_2 \to t_1 \to t_6$$

Type rules

Finishing the inference

Substituting these two back in we get a final tree:

$$\dfrac{\dfrac{\dfrac{f :: t_1 \to t_5 \to t_6 \qquad x :: t_1}{f\ x\ ::\ t_5 \to t_6}\,@ \qquad \dfrac{g :: t_1 \to t_5 \qquad x :: t_1}{g\ x\ ::\ t_5}\,@}{(f\ x)\ (g\ x)\ ::\ t_6}\,@}{(\lambda\ f\ g\ x)\ ::\ (t_1 \to t_5 \to t_6) \to (t_1 \to t_5) \to t_1 \to t_6}\,\lambda$$

Type rules

How did we do?

Some variables no longer exist in this tree, so I'll do a quick renumbering to make the type variables sequential:

$$\dfrac{\dfrac{\dfrac{f :: t_1 \to t_2 \to t_3 \qquad x :: t_1}{f\ x\ ::\ t_2 \to t_3}\,@ \qquad \dfrac{g :: t_1 \to t_2 \qquad x :: t_1}{g\ x\ ::\ t_2}\,@}{(f\ x)\ (g\ x)\ ::\ t_3}\,@}{(\lambda\ f\ g\ x)\ ::\ (t_1 \to t_2 \to t_3) \to (t_1 \to t_2) \to t_1 \to t_3}\,\lambda$$

How did we do?

```
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :t (\f g x -> f x (g x))
(\f g x -> f x (g x)) :: (t1 -> t2 -> t3) -> (t1 -> t2) -> t1 -> t3
Prelude>
```

Type rules

Restricting the type of lambda-bound variables

The rule you might be questioning is the one about forcing all occurrences of “x” bound by a lambda to be of the same type.

Once an expression has been determined to be well typed it must be usable in any other well typed context.

It’s easy to come up with situations where allowing a variable like “x” to vary in different contexts would be OK, but it’s not *universally* true.

Type contexts

In order to type check expressions like this:

$$\dfrac{\dfrac{map :: T_0 \qquad f :: T_1}{map\ f\ ::\ T_3}\ @ \qquad \dfrac{}{[1,2,3]\ ::\ T_4}}{map\ f\ [1,2,3]\ ::\ T_5}\ @$$

We need a way to form opinions about pieces of syntax like “map” and “[1,2,3]”. We don’t deduce most of these from rules (we’d need to invent new rules all the time for new functions).

Instead we augment each rule with a way to “look up” known facts (like “map has the type (a->b)->[a]->[b]”).

Type contexts

The set of known facts about variables is represented by a mapping (written Γ) which we call the "type environment". We augment each rule with this mapping, and as we gather information about newly declared values we extend the environment.

In this way, when we encounter a name for a non-locally bound variable we can determine it's type by looking it up.

Type contexts

The key rules for a simple type system cover “lookup”, application, abstraction, and local naming:

$$\frac{}{\Gamma \cup \{x : t\} \vdash x : t}\ \textsc{Var}$$

$$\frac{\Gamma e : t' \rightarrow t \qquad \Gamma e' : t'}{\Gamma e\ e' : t}\ \textsc{App}$$

$$\frac{\Gamma \cup \{x : t'\} \vdash e : t}{\Gamma \vdash \lambda x.e : t' \rightarrow t}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash_p e : \sigma \qquad \Gamma \cup \{x : \sigma\} \vdash_p e' : \tau}{\Gamma \vdash_p \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau}\ \textsc{Let}$$

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/