CPSC 311: Definition of Programming Languages: Type Inference

24-type-inference

DRAFT

Joshua Dunfield
University of British Columbia

November 26, 2015

1 Type inference

When we started doing typing, we encountered the rule

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{lam}\ x\ e) : A \to B}\ \text{?Type-lam}$$

which we couldn't implement, because to make the recursive call to `typeof`, we needed to extend the typing context Γ (`tc`) with $x : A$, but we didn't know what A was.

As a simple workaround, we added the type A to the abstract syntax of lam:

$$\frac{x : A, \Gamma \vdash e : B}{\Gamma \vdash (\mathsf{lam}\ x\ A\ e) : A \to B}\ \text{?Type-lam}$$

Now we'll show how to do (a simple form of) *type inference*, which *infers* the type A without making the programmer write it.

The general idea is to use A as a placeholder, and update it once we know what A needs to be. On paper, this is not too difficult: we write A and B instead of actual types, and leave empty boxes off to the side of the derivation. (See the scanned page.) Initially, these boxes are blank because we don't yet know what A and B are, but from Type-add, we can figure out that since $(\mathsf{id}\ x)$ has type A, and Type-add needs A needs to be Num, then we should use Num. At this point, we write Num in the box labelled A.

From the conclusion of Type-add, we also see that B is Num.

Even though we wrote $A \to B$ in the conclusion, we know that $A = \mathsf{Num}$ and $B = \mathsf{Num}$, so we have really derived

$$\emptyset \vdash (\mathsf{lam}\ x\ (\mathsf{add}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))) : \mathsf{Num} \to \mathsf{Num}$$

In fact, everywhere we wrote A in the derivation, we should now interpret A as Num. By writing Num in the box for A, we have updated or "mutated" A. This suggests a way to implement this technique: represent A as a Racket box that is either "blank" or "filled in" with a type. To "fill in" the box, we can use Racket's `set-box!`.

Extending the **define-type** for `Type` (see type-inference.rkt), we have

```
(define-type Type
    [t/num]                                 ; Num

    [t/bool]                                ; Bool
    [t/-> (domain Type?) (range Type?)]     ; {-> domain range}
    [t/var (var box?)]
)
```

We will represent a blank box on paper by a box containing `#false`, and a filled-in box by a box containing a `Type`.

(I tried to use a Racket "box contract" `box/c` to specify this, but ran into trouble with "impersonators" and "chaperones". Yes, really.)

1.1 Equating types

The `type=?` function provides a starting point for a central mechanism of type inference, *unification*. Our extended version of `type=?` is called `type=?!`.

But unlike `type=?`, which is only asking "are these types equal?", the new function `type=?!` is asking "can these types *be made* equal?"

If a mathematician accosts you and asks, "Is x plus 1 equal to 5?" the correct answer is "I don't know". But if she asks you to solve the equation

$$x + 1 = 5$$

you should answer "$x = 4$". More generally, given an equation, you can try to solve all the variables in it.

If the equation has no variables, then you are just doing arithmetic. So, for types without any `t/var`s, this new function `type=?!` will work just like `type=?`: if the types are literally the same, it will return `#true`, otherwise `#false`.

The difference is in how `type=?!` works on variables `t/var`:

- If the first type is a variable whose box (call it α) contains `#false` (meaning "blank" or "unknown"), then we are trying to solve the equation

  $$\alpha = B$$

  where we don't have a solution for α. But the solution is right there: let $\alpha = B$. So in this case, we use `set-box!` to replace the `#false` inside the box α with B.

  Boxes are mutable and global, so this operation effectively "rewrites" any other occurrences of `t/var` α in the derivation.

- If the second type is a variable whose box (call it β) contains `#false`, then we have a situation symmetric to the one above: we are trying to solve

$$A = \beta$$

  So we use `set-box!` to put A inside β.

- If the first type is a variable that has been solved, its box α contains a type $A0$. That is, we know that $\alpha = A0$, and we want to try to make $\alpha = B$, so we try to make $A0 = B$.
- Similarly, if the second type is variable that has been solved, its box β contains a type $B0$. We know that $\beta = B0$, and we want to make $A = \beta$, so we try to make $A = B0$.

Since `type=?!` will be our mechanism for solving type variables, we need to update `typeof` (which is now called `infer`) to use `type=?!` more often. For example, the old `infer` sometimes used `type-case` with a `t/num` branch to check whether a type `A1` was a Num. Now the type might be a `t/var`, so instead, `infer` calls `(type=?! (t/num) A1)`.

1.1.1 The "occurs check"

As given, this technique works most of the time, but it will not work for this Fun expression:

$$(\mathsf{lam}\ x\ (\mathsf{app}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x)))$$

In the lam branch, we create a `t/var` for x, and recursively call `infer` to infer the type of the body $(\mathsf{app}\ (\mathsf{id}\ x)\ (\mathsf{id}\ x))$.

In the app branch of `infer`, we create a type `(t/-> A1 B)` where A1 and B are `t/vars`. Here, A1 will be made equal to A (the type of x). But we need the type of $(\mathsf{id}\ x)$ to be equal to $A1 \to B$, that is, equal to `(t/-> A1 B)`. The type of $(\mathsf{id}\ x)$ is A, so (since $A = A1$) we are trying to make this equation hold:

$$A \to B = A$$

where A is unsolved. So (the original version of) `type=?!` sets the contents of A's box to $A \to B$. This creates a cyclic "type" that makes something (I'm not sure what, and lack the patience to figure it out) loop forever.

But it should never be possible for $A \to B$ to equal A. For any such types without `t/vars`, `type=?` would have returned `#false`.

The solution is something called an "occurs check", which checks whether the `t/vars` occurs in the other type. For our example, that amounts to checking whether A occurs in $A \to B$. It does occur, so we return `#false`.

Adding the occurs check led to another problem, which is that A occurs in A. So I added another check, done before the occurs check, to see whether both A and B are the same type variable; if they are, `type=?!` returns `#true`. (A mathematician accosts you and asks if x equals x. You should say "yes". You should especially say "yes" if the mathematician is also an Objectivist, because "*A* is *A*".)

2015-11-25

TYPE INFERENCE:

(lam x ~~A~~ e)
(rec u ~~B~~ e)

$$\frac{x{:}A, \Gamma \vdash e : B}{\Gamma \vdash (\text{lam } x\ e) : A \to B}\ \text{Type-lam [?]}$$

$$\frac{x{:}A \in \Gamma}{\Gamma \vdash (\text{id } x) : A}\ \text{Type-id} \qquad \frac{\Gamma \vdash e1 : \text{Num} \quad \Gamma \vdash e2 : \text{Num}}{\Gamma \vdash (\text{add } e1\ e2) : \text{Num}}\ \text{Type-add} \qquad \frac{\Gamma \vdash e1 : A1 \quad (A1 = \text{Num}) \quad \Gamma \vdash e2 : A2 \quad (A2 = \text{Num})}{\Gamma \vdash (\text{add } e1\ e2) : \text{Num}}$$

A = Num
B = Num

$$\frac{\dfrac{(x{:}A) \in x{:}A, \emptyset}{x{:}A, \emptyset \vdash (\text{id } x) : A} \qquad \dfrac{(x{:}A) \in x{:}A, \emptyset}{x{:}A, \emptyset \vdash (\text{id } x) : A}}{x{:}A, \emptyset \vdash (\text{add } (\text{id } x)\ (\text{id } x)) : \text{Num}}$$

$$\emptyset \vdash (\text{lam } x\ (\text{add } (\text{id } x)\ (\text{id } x))) : A \to B$$

Num → Num

Types:

+ (t/var b)

where b is a box that contains either #false (meaning "unknown", like A=[ ]), or a type.

(The type might also be a t/var.)

For the above example:

$$\emptyset \vdash (\text{lam } x\ (\text{add } (\text{id } x)\ (\text{id } x))) : A \to B$$

(t/-> (t/var •) (t/var •))

A [(t/num)] B [t/num]