Properties of simple types

- **Uniqueness of types.** In a given context (types for free variables), any simply typed lambda term has at most one type (and this is decidable, in small polynomial time). (This should be fairly obvious from what we’ve just done; formal proof takes a little work, but not much.)
- **Type safety.** The type of a term remains unchanged under $\alpha, \beta, \eta$-conversion. (Also straightforward, though with some lemmas (e.g. to handle substitution.).)
- **Strong normalization.** A well typed term evaluates under $\beta$-reduction in finitely many steps to a unique irreducible term. If the type is a base type, then the irreducible term is a constant. (A bit harder to prove.)
- **Corollary:** Simply typed lambda calculus cannot be Turing-complete! What have we lost?

Recursion re-gained

If your tool-kit doesn’t have a gadget: make one and put it in the kit!

- We add the term constructor **fix** to the language: if $t$ is a term, then $(\text{fix } t)$ is a term.
- We extend $\beta$-reduction to say that:

$$\text{fix}(\lambda x : \tau. t) \xrightarrow{\beta} t[\text{fix}(\lambda x : \tau. t)/x]$$

(often called the unfolding rule).
- We add a new typing rule

| \( \Gamma \) | \( t : \tau \rightarrow \tau \) | \( \Gamma \vdash \text{fix } t : \tau \) |

Now we can use **fix** just as we used $Y$.

**Exercise:** Define addition of (base type nat) numbers using **fix**, the successor and predecessor functions $\text{suc}, \text{prd} : \text{nat} \rightarrow \text{nat}$, and the equality of numbers function $= : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$.

Recursion lost

Just as $\lambda x.xx$ can’t be typed, nor can $Y$, nor any other potentially non-terminating term. (**Exercise:** convince yourself that $Y$ can’t be given a simple type.)

What can we do? General recursion is bound to allow non-termination.

More bells and whistles

We can further extend the syntax of simply typed lambda with constructors and types for the familiar apparatus of programming languages:

- tuples (products)
- records (named products)
- sums (union types)
- lists, trees etc. (N.B. one list type for each base type!)

See Pierce ch. 11 for gory (rather uninteresting) details.

More interesting is the **let** statement. Simple (non-recursive) **lets** can be done by

$$\text{let } x : \tau = t \text{ in } t' \equiv (\lambda x : \tau. t)t$$

and fully recursive **let** definitions by

$$\text{letrec } x : \tau = t \text{ in } t' \equiv \text{let } x : \tau = \text{fix}(\lambda x : \tau. t) \text{ in } t'$$

**Exercise:** Think carefully about the last! How does it work?
Type reconstruction

It’s rather boring having to type all these types! Can we leave them out, write in the untyped language, and let the computer work out the types? Essentially we program up the inference we did ‘in our heads’ to make type proofs.

Give every (ordinary) variable \( x \) a different type variable, so \( x : \alpha \). Then see what relations have to hold between the \( \alpha \).

Before, we were using meta-variables \( \sigma, \tau \) to do human reasoning; now we’re going to promote these to actual variables ranging over types.

This is a constraint solving problem: e.g. we have \( \alpha, \beta \) and we know that \( \alpha = \beta \to \beta \).

Where do the constraints come from? From the typing rules.

First, some formalities . . .

Reconstruction example

Find types for \( \lambda f . \lambda x . \lambda y . \text{succ}(f(+ x y)) \).

First annotate: \( \lambda f : \gamma . \lambda x : \alpha . \lambda y : \beta . \text{succ}(f(+ x y)) \).

Now make the proof tree, using additional type variables and recording equations instead of working things out:

\[
\begin{align*}
[\alpha \to \theta \to \eta = \text{nat} \to \text{nat} \to \text{nat}] \\
\ldots \vdash + : \text{nat} \to \text{nat} \to \text{nat} \\
\ldots \vdash (\theta \to \eta) \to \eta \\
\ldots \vdash x : \alpha \\
\ldots \vdash y : \beta \\
[\gamma = \eta \to \text{nat}] \\
\ldots \vdash f : \gamma \\
\ldots \vdash (+ x) : \theta \to \eta \\
\ldots \vdash (+ y) : \eta \\
\ldots \vdash \text{succ} : \text{nat} \to \text{nat} \\
\ldots \vdash f(,+ x y) : \text{nat} \\
[\zeta = \text{nat}] \\
\end{align*}
\]

Solve the equations to get \( \alpha = \beta = \text{nat} \), and \( \gamma = \text{nat} \to \text{nat} \to \text{nat} \).

Type variables

We add type variables to our typed language. **Convention:** I use the start of the Greek alphabet for type variables. Now types include:

- every type variable \( \alpha \) is a type.

Eventually, type variables need to turn into actual types: consider \( (\lambda x : \alpha . x)1 \).

A type substitution gives a substitution of types for type variables, e.g. \([\text{nat}/\alpha, \text{bool}/\beta, (\alpha \to \beta)/\gamma]\). We’ll write type substitution before a term, e.g. \([\text{nat}/\alpha](\lambda x : \alpha . x) = \lambda x : \text{nat} . x\).

Substitutions apply to terms, contexts, etc. in the obvious way.

- Theorem: type substitution preserves typing: if \( \Gamma \vdash t : \tau \), and \( \xi \) is a type substitution, then \( \xi \Gamma \vdash \xi t : \xi \tau \)
- The reverse is not true: \( x : \alpha \not\vdash x : \text{nat} \), but \([\text{nat}/\alpha](x : \alpha) \vdash [\text{nat}/\alpha]x : [\text{nat}/\alpha]\text{nat} \).

Solving the type equations

The equations are very simple: just equalities involving variables and \( \to \).

We can solve them by unification (technique from logic programming). Standard algorithm: look it up in Pierce for this case, or anywhere for the general case.

**Theorem:** if an untyped \( \lambda \)-term can be simply typed, this procedure will find a simple type for it, or return ‘fail’ otherwise (if the equations have no solution).

Proof by a somewhat technically involved induction on terms. (Full details in Pierce.)

So type inference is decidable (with reasonable polynomial complexity – in all practical cases, it’s even linear).