36  Simple types

The language we have looked at so far has no notion of type – any term can be applied to any other term, regardless of whether it makes any sense. For example, \((\lambda x.xxx)(0)\) reduces to 0 0 0 – what does it mean to apply 0 to 0? If 0 is a primitive symbol for zero, added to the language as suggested in the previous section, it means nothing – the expression does not reduce. If 0 is an abbreviation for the Church numeral \(\lambda f, x.x\) (see tutorial sheet 5), then \(0 \beta \rightarrow \lambda x.x\), and so \((0 0) \beta \rightarrow 0\), which also makes very little sense!

Similarly, the other example on the slide \((+ (\lambda x.xx)42)\) makes no sense either.

Simply typed lambda-calculus is the lambda-calculus extended with a set of types, and rules for assigning types to terms.

Definition 81 The types are defined thus:

- Fix some set of base types, e.g. nat, bool, for the constants. (In the language without any extensions, there is a single base type, usually called o.)
- If \(\sigma, \tau\) are types, then \(\sigma \rightarrow \tau\) is a type, the function type of abstractions that take an argument of type \(\sigma\) and return a value of type \(\tau\). Convention: \(\rightarrow\) associates to the right, so \(\sigma \rightarrow \tau \rightarrow \upsilon\) means \(\sigma \rightarrow (\tau \rightarrow \upsilon)\).

Simply typed lambda-calculus has the same syntax as the pure lambda-calculus, except that the syntax (Definition 74) of abstractions is changed to:

\((3')\) If \(x\) is a variable, \(t\) is a term and \(\sigma\) is a type, then \((\lambda x: \sigma. t)\) is a term.

Note that in this language, the user/programmer is responsible for saying what type every variable has. We will see later how to recover the behaviour of advanced languages like ML and Haskell, where the compiler does much of this work for the programmer.

It is, of course, still possible to write down nonsensical expressions such as \(\lambda x: \text{nat}. xx\). The way that we rule these out of consideration is to say that we are interested only in well-typed terms, that is, terms in which the types make sense as far as function application goes.

Definition 82 For a term \(t\) and type \(\tau\), the judgement \(t : \tau\), read as ‘\(t\) has type \(\tau\)’, is defined by induction on the structure of terms. If \(t\) has a type, we say it is well typed.

- Constants (individuals or functions) are well typed with their assigned type.
- If \(t : \tau\) under the assumption that \(x : \sigma\), then \((\lambda x: \sigma. t) : \sigma \rightarrow \tau\).
- If \(t : \sigma \rightarrow \tau\) and \(s : \sigma\), then \((ts) : \tau\).

For example, the term \(\lambda x: \text{nat}. + x 42\) has type \(\text{nat} \rightarrow \text{nat}\), by the following reasoning: 42 is a constant, and will have type \(\text{nat}\) assigned in its definition; + is a constant, and will have type \(\text{nat} \rightarrow \text{nat}\) assigned in its definition. Then if we assume that \(x : \text{nat}\) (as specified in its binding), we see that \((+ x 42)\) will have type \(\text{nat}\), and so the whole term is \(\text{nat} \rightarrow \text{nat}\).

On the other hand, if we try to work out a type for \(\lambda x: \text{nat}. xx\), we fail, and so the term is not well typed.

37  Proofs for types

The process of using the definition to work out the type of a term is called type inference. It is common to present type inferences as proof trees, as we do with proofs in propositional
or predicate logic. This is partly for humans, so that we have a formal system with which we can carefully check details, and partly for machines, so that we can write programs to do type inference for us.

From now on, we assume that all formulae are $\alpha$-converted to avoid clashes of bound variables.

**Definition 83** $\Gamma \vdash t : \tau$ is a judgement that $t$ has type $\tau$ under the set of assumptions $\Gamma$, where an assumption is a judgement $x : \sigma$ for some variable $x$ and type $\sigma$. Assumption sets are written as comma-separated lists of assumptions; the notation $\Gamma, x : \sigma$ means $\Gamma \cup \{x : \sigma\}$.

The judgement $\Gamma \vdash t : \tau$ is true iff it can be proved from the following rules and axioms, where in each rule the premises appear above, and the consequence below.

- For any set of assumptions $\Gamma$, and for every constant $c$, there is an axiom
  \[ \Gamma \vdash c : \tau \]
  where $\tau$ is the type assigned to $c$ by its definition.

- The Assumption Rule
  \[ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \]
  transfers assumptions on to the right-hand side of the judgement.

- The Abstraction Rule
  \[ \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma.t : \sigma \to \tau} \]
  formalizes the typing of abstractions.

- The Application Rule
  \[ \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash ts : \tau} \]
  formalizes the typing of function applications.

Usually we apply these rules in a goal-directed manner: we start with the judgement we want to prove, and work backwards. The procedure is entirely mechanical, as illustrated by the examples on the slides, which I will not repeat here. It should also be fairly obvious that the computational complexity of this procedure is small, at most quadratic in the size of the term, and that a given term only has one type.

**Theorem 84 (Uniqueness of types)** In a given consistent set of assumptions (i.e., one that does not contain two different assumptions for the same variable), any simply typed lambda term has at most one type (and this is decidable, in small polynomial time).

**Proof.** (Informal) By induction on the structure of terms. The base case is variables and constants, which have a unique type because the assumption set is consistent by hypothesis.

For the abstraction case, given assumptions $\Gamma$ and $\lambda x : \sigma.t$, the Abstraction Rule adds $x : \sigma$ to $\Gamma$, which yields a consistent set $\Gamma'$ of assumptions because we are assuming that all variables are distinct. Then by the induction hypothesis, $t$ has a unique type $\tau$ under $\Gamma'$, and then the Abstraction Rule says that $\lambda x : \sigma.t : \sigma \to \tau$, and there is no other way to infer a type for it.

For applications, again the two sub-terms have unique types by hypothesis, and the rule gives a unique type for the application.
Properties of simple types

The types and typing rules have several other desirable properties. In a slightly deeper course, we would prove them, with the basic technique again being induction on the structure of terms. However, as is often the case with such proofs, they can get fiddly to do properly, and particularly to take care of substitution and \( \alpha \)-conversion. Therefore, in this course we just state the results; proofs can be found in the textbook.

**Theorem 85 (Type safety)** The type of a term remains unchanged under \( \alpha, \beta, \eta \)-conversion.

**Theorem 86 (Strong normalization)** A well typed term evaluates under \( \beta \)-reduction (and a given evaluation strategy) in finitely many steps to a unique irreducible term. If the type is a base type, then the irreducible term is a constant.

This is a powerful theorem, as it tells us that ‘computation’ only yields one answer – even if our evaluation strategy is ‘choose a \( \beta \)-redex at random and reduce it’. It is quite intricate to prove, taking a few pages of working in full.

Of course, this also means that computation in the simply-typed lambda-calculus always terminates, and hence:

**Corollary 87** Simply typed lambda calculus is not Turing-complete.

Why is this? Simply because we have no way to achieve recursion any more. The amazing magical \( Y \) combinator is not available to us in the simply typed world:

**Proposition 88** Any term containing a self-application (e.g. \( xx \)), such as \( Y \), cannot be given a simple type.

**Proof.** Consider \( xx \): if \( x : \tau \), then we must also have \( x : \tau \rightarrow \sigma \) for some \( \sigma \), and so \( \tau = \tau \rightarrow \sigma \), which is impossible in the simple type system.

Recursion for simple types

The \( Y \) combinator is a term that takes a function (e.g. \( \lambda f.\lambda x.\ldots \)) and gives back a fixpoint: a function \( g = \lambda x.\ldots \) such that feeding \( g \) to \( \lambda f.\lambda x.\ldots \) gives back (something equivalent to) \( g \).

We cannot reproduce \( Y \) in a simply-typed world, so instead we have to extend the language with a fixpoint constructor. This is a new way of making term, like the symbol \( \lambda \). We then extend the rules for \( \beta \)-reduction to make this constructor behave as \( Y \) does.

**Definition 89** The syntax (Definition 74) of the lambda-calculus is extended thus:

(4) If \( t \) is a term, then \( \text{fix} \ t \) is a term. We will adopt the bracketing convention that the argument of \( \text{fix} \) is the immediately following term, i.e. we treat it like an application.

The \( \beta \)-reduction rules are extended thus:

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

(often called the unfolding rule).

A new typing rule is added:

\[
\frac{\Gamma \vdash t : \tau \rightarrow \sigma}{\Gamma \vdash \text{fix} \ t : \tau}
\]

\( \text{fix} \) can now be used just as \( Y \) was used. For example, the factorial function is just \( \text{fix}(\lambda f.\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x - 1)) \).
40  Adding structured datatypes

Even with primitive booleans and integers added, our language is still not very expressive compared to usual programming languages. It is possible, and indeed easy, albeit tedious, to add your favorite structured datatype, by again extending the syntax, typing rules, and evaluation rules.

For example, suppose we wish to add lists. We will need a separate list type for every element type, in order for typing still to work. So we will add a type constructor \texttt{list}, which is traditionally written after its argument, e.g. \texttt{nat list}. Then for every type \( \tau \), we add list \texttt{constructors} \texttt{nil} \( : \tau \texttt{list} \) and \texttt{cons} \( : \tau \to \tau \texttt{list} 	o \tau \texttt{list} \), and \texttt{destructors} \texttt{hd} \( : \tau \texttt{list} \to \tau \) and \texttt{tl} \( : \tau \texttt{list} \to \tau \texttt{list} \).

For the gory details of this approach to extending the language, refer to the textbook.

41  let and letrec

The \texttt{fix} constructor enables recursive definitions, but it is not particularly easy to use. It would be nice if we could define functions – and indeed ‘local’ variables – in a way more familiar from programming languages. We can do this without extending the core language, simply by adding some ‘syntactic sugar’ that presents \( \lambda \) and \texttt{fix} in a more intuitive way. First, we consider a non-recursive \texttt{let} that gives a more attractive way to write local variables.

\textbf{Definition 90} \( (\texttt{let } x: \tau = t \texttt{ in } t') \) is an abbreviation for \( (\lambda x: \tau.t')t \).

The term \( (\lambda x: \tau.t')t \), when evaluated, binds the term \( t \) to the variable \( x \) and evaluates \( t' \), which is exactly how \texttt{let} is expected to behave. However, this doesn’t allow defining recursive local functions, because any occurrences of \( x \) in \( t \) are free, and therefore not bound by the \( \lambda x \) (\( x \) will have been \( \alpha \)-converted to something else, by our convention that all variables should be distinct). The recursive version is

\textbf{Definition 91} \( (\texttt{letrec } x: \tau = t \texttt{ in } t') \) abbreviates \( (\texttt{let } x: \tau = \texttt{fix}(\lambda x: \tau.t) \texttt{ in } t') \).

For example, if we write

\texttt{letrec}
\begin{align*}
\text{fac:} \tau & \to \tau = \lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \times \text{fac}(x - 1) \\
\text{in}
\text{fac}(3)
\end{align*}

we can see that by expanding the abbreviation we end up with \texttt{fac} bound to the original fixpoint factorial function definition when we evaluate \texttt{fac}(3). (You should write this out in full.)