CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Types

Typed λ-Calculus

In the pure λ-calculus, every value denotes a function that can be applied to any other value. Nothing can **go wrong**.

But in an applied λ-calculus, some applications may be nonsense, e.g.,

```
3 x
plus true
```

Types are used to filter out nonsense. By defining a **typed** λ-calculus, and permitting only **well-typed** expressions in the calculus, we can ensure that things don't **go wrong** when expressions are evaluated.

For simplicity, we'll talk mainly about the **simply-typed** λ-calculus, in which each expression has a single, well-defined type.

Most real languages have more sophisticated type systems, in which expressions (in particular functions) may take on different types at different points.

• ML and other strongly-typed FL's use a particular form of **polymorphic** λ-calculus that allows flexible use of polymorphic functions together with the security of compile-time type checking.

• Typed object-oriented languages use a different kind of flexibility based on **sub-typing** to support re-use of functions on different sorts of values.

Syntax

Type expressions are defined by the grammar

$$\tau ::= b \mid \tau_1 \to \tau_2 \qquad (b \in B)$$

where B is a set of **base types**, e.g., $B = \{\texttt{int}, \texttt{bool}\}$, and $\to$ is a **type constructor**.

In general a type constructor takes one or more types and "constructs" a new type; in this case, $\tau_1 \to \tau_2$ is the type of functions with doman τ_1 and range τ_2.

Examples of types

```
int
int→bool
int→(int→int)
(int→bool)→(int→bool)
```

Notational convention: $\to$ associates to the **right**, so, e.g.,

```
int→int→int ≡ int→(int→int)
```

Typed λ-terms

Syntax is just as for the ordinary calculus, except that we add an explicit type annotation to each λ-bound variable.

$$\begin{array}{lcll} M & := & c & \text{Built-in constants} \\ & \mid & v & \text{Variable names} \\ & \mid & (M_1 M_2) & \text{Function applications} \\ & \mid & (\lambda v : \tau . M) & \text{Function abstractions} \end{array}$$

We assume that we know the type of every constant in our particular applied calculus, e.g., there's a function `TypeOf(`c`)`.

Examples

```
3
xy
λx:int.λy:bool.x
λx:int→bool.λy:int.xy
λx:int.λy:bool.xy
```

Note that the last of these is syntactically legal but (intuitively) **not** well-typed.

We could actually add type annotations to **each term**, but this clutters up the syntax and is not very useful, as we'll see.

Typing Assertions

We relate expressions to types via assertions of the form:

$$E \triangleright M : \tau$$

where M is a typed λ-term, τ is a type, and E is a **context** (a.k.a. type assignment, type environment) mapping variables to types.

The role of E is to specify the types of the **free variables** of M, upon which the overall type of M obviously should depend. We write

$$E = \{\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n\}$$

where all the $\mathtt{x}_i$ are distinct. We write $E + \{\mathtt{x} : \tau\}$ to mean the result of extending or updating the mapping E to make x map to τ.

Examples

$\emptyset \triangleright \mathtt{3} : \mathtt{int}$
$\{\mathtt{z{:}bool{\to}int,\ w{:}bool}\} \triangleright \mathtt{zw} : \mathtt{int}$
$\{\mathtt{x{:}int{\to}bool}\} \triangleright \lambda\mathtt{y{:}int.xy} : \mathtt{int{\to}bool}$
$\{\mathtt{x{:}int{\to}bool}\} \triangleright \lambda\mathtt{y{:}int.xy} : \mathtt{bool}$

Note that the last example is a syntactically valid but (intuitively) **false** assertion.

Typing Rules

We want to distinguish a class of assertions that are **true**, i.e., that, if relied upon, will not lead to things going wrong at runtime.

Formally, we do this by establishing a deductive formal system, called a **type system**, consisting of axioms and rules of inference. We will claim that any typing assertion that can be **derived** within our formal system is **true** in the sense just given, i.e., that the formal system is **sound**.

Axiom Schemes

(CONST) $\mathtt{E} \rhd \mathtt{c} : \tau$ $(\forall \mathrm{c}, \tau \text{ s.t. } \mathtt{TypeOf(c)} = \tau)$

(VAR) $\mathtt{E} \rhd \mathtt{x} : \tau$ $(\text{if } \mathtt{E(x)} = \tau)$

Rules of Inference

$$\text{(ABS)}\ \frac{\mathtt{E} + \{\mathtt{x}:\tau_1\} \rhd \mathtt{M} : \tau_2}{\mathtt{E} \rhd (\lambda \mathtt{x}:\tau_1.\mathtt{M}) : \tau_1 \rightarrow \tau_2}$$

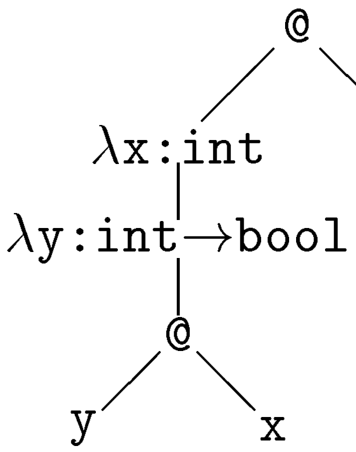
$$\text{(APP)}\ \frac{\mathtt{E} \rhd \mathtt{M} : \tau_1 \rightarrow \tau_2 \qquad \mathtt{E} \rhd \mathtt{N} : \tau_1}{\mathtt{E} \rhd (\mathtt{MN}) : \tau_2}$$

If an assertion is derivable in this system, we write $\vdash E \rhd M : \tau$.

If $\exists E, \tau$ such that $\vdash E \rhd M : \tau$, we say that M is **well-typed**.

Derivations

Example: $(\lambda\mathtt{x:int}.\lambda\mathtt{y:int}\to\mathtt{bool.yx})\ \mathtt{3}$



$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{x:int},\mathtt{y:int}\to\mathtt{bool}\}\ \triangleright\ \mathtt{y:int}\to\mathtt{bool}\qquad \{\mathtt{x:int},\mathtt{y:int}\to\mathtt{bool}\}\ \triangleright\ \mathtt{x:int}}{\{\mathtt{x:int},\mathtt{y:int}\to\mathtt{bool}\}\ \triangleright\ \mathtt{yx:bool}}}{\{\mathtt{x:int}\}\ \triangleright\ \lambda\mathtt{y:int}\to\mathtt{bool.yx:(int}\to\mathtt{bool)}\to\mathtt{bool}}}{\emptyset\ \triangleright\ \lambda\mathtt{x:int}.\lambda\mathtt{y:int}\to\mathtt{bool.yx:int}\to\mathtt{(int}\to\mathtt{bool)}\to\mathtt{bool}}\qquad \emptyset\ \triangleright\ \mathtt{3:int}}{\emptyset\ \triangleright\ (\lambda\mathtt{x:int}.\lambda\mathtt{y:int}\to\mathtt{bool.yx)3:(int}\to\mathtt{bool)}\to\mathtt{bool}}$$

Notice that for each syntactic form of expression, there is exactly one applicable typing rule. Thus, given an expresion, the **form** of any typing derivation tree for that expression has the same shape as the syntax tree for the expression, and is **independent** of the type we're attempting to derive.

So it should not be a surprise that the type of any expression in a given context is **unique**. The proof is by "structural induction" on the form of the expression. We can always compute this (unique) type of an typed term in a given context, based only on the types specified at λ-bindings. This is why we don't bother to annotate all subexpressions with types.

Soundness

A type system is only useful if it accurately predicts the behavior of "runtime" execution. In particular, a correct system should guarantee that execution does not **go wrong** by attempting to do something nonsensical. Such a type system is said to be **sound**.

To prove soundness, we must say formally what it means to go wrong, in the context of a specific **operational semantics** for the language.

For the simply-typed λ-calculus, we can take the ordinary β-reduction rule as the (sole) rule of an operational semantics. We just make one modification: β-reduction can only occur if the redex is well-typed. This doesn't change much; for example, the resulting calculus is still Church-Rosser.

Moreover, we can prove the following **subject reduction** theorem: if $E \rhd M : \tau$ and $M \rightarrow_\beta N$, then $E \rhd N : \tau$. In other words, β-reduction preserves typing.

In particular, if we start out with a well-typed term we can execute (i.e., perform reductions) as much as we like, and always end up with a well-typed term, i.e., things don't go wrong at runtime.

Termination Properties

Perhaps surprisingly, we can also prove (fairly easily) the following result about the simply-typed λ-calculus:

Strong Normalization Theorem. Every reduction sequence is finite (and terminates with a normal form).

This implies that non-terminating expressions in the ordinary λ-calculus must not have well-typed equivalents in the simply-typed calculus.

Indeed, for example, there is no typed equivalent for $\lambda\texttt{x.xx}$. This is easy to see if we try to give `x` some type τ_0. Then the (APP) rule requires that `x` also have type $\tau_0 \to \tau_1$, for some type τ_1, but clearly this is impossible.

These facts imply that the simply-typed λ-calculus is computationally rather weak, because we can't define a recursive Y-combinator **within** the calculus.

Of course, we can still add a Y-combinator as a constant, with an extra axiom schema to describe its (possible) types.

Losing some expressiveness in return for type safety is a theme we'll see again...

Type Checking

Given an expression in the simply-typed λ-calculus and an initial environment, we can easily determine the (unique) type for the expression in that environment, if it has one.

Here's the algorithm:

$$
\begin{array}{l}
\mathcal{T}(E, M) = \\
\quad \text{case } M \text{ of} \\
\quad\quad c : \text{Typeof}(c) \\
\quad\quad v : \text{if } v \in Dom(E) \text{ then } E(v) \\
\quad\quad\quad\quad \text{else error ``untyped free variable''} \\
\quad\quad \lambda\, x : \tau.M_1 : \\
\quad\quad\quad\quad \tau \rightarrow \mathcal{T}(E + \{x : \tau\}, M_1) \\
\quad\quad (M_1\ M_2): \\
\quad\quad\quad\quad \text{case } \mathcal{T}(E, M_1) \text{ of} \\
\quad\quad\quad\quad\quad \tau_1 \rightarrow \tau_2 : \\
\quad\quad\quad\quad\quad\quad \text{if } \tau_1 = \mathcal{T}(E, M_2) \text{ then } \tau_2 \\
\quad\quad\quad\quad\quad\quad \text{else error ``function-argument mismatch''} \\
\quad\quad\quad\quad\quad \text{other : error ``operator not a function''}
\end{array}
$$

Note that all the types used in the algorithm either come from the abstraction-binding annotations or appear in the initial environment.

We can regard this algorithm as a **type checker** for expressions; if no error is raised, the original expression is properly typed in the given environment.

Type Inference

Type inference is the problem of producing a well-typed term from an **untyped term**. Having a type inference mechanism for a language can be very useful, since it saves the programmer from having to declare the types of all identifiers (though it does not **prevent** such declarations).

Type inference is **not** possible for every language and type system, but there **is** a type inference algorithm relating the untyped and simply-typed λ-calculi.

Formally, we define a function **erase**, which takes a typed term, removes all the type annotations from the λ-bindings, and returns the resulting untyped term.

An untyped term M is **typable** if there exists a (simply-)typed term M', a type τ and an environment E (with FV(M) $\in$ Dom(E)) such that

$\vdash E \; \triangleright M' : \tau$ and $M = \text{erase}(M')$

As mentioned before, not all untyped terms are typable, e.g., λ`x.xx`.

A (typable) untyped term may be the erasure of more than one typed term. Example: λ`x.x` is the erasure of both λ`x:int.x` and λ`x:bool.x`.

Type Schemes

To make the type inference algorithm precise, it is helpful to extend the language of type expressions to include **type variables**, denoted by $\{\alpha, \beta, \gamma, \ldots\}$.

The resulting grammar for **type schemes** is:

$$\tau ::= b \mid \alpha \mid \tau_1 \to \tau_2 \qquad (b \in B)$$

They are called schemes because each expression can be thought of as a schematic representation of a whole family of (ordinary) types, produced by **instantiating** the type variables with ordinary types.

Examples

α
$\alpha \to \beta$
$\alpha \to (\texttt{int} \to \alpha)$
`bool`

An ordinary type is just a type scheme with no type variables. They are also called **ground types** or **monotypes**.

We can modify our type system so that typing rules, environments, λ-binding annotations, assertions, etc., all refer to type **schemes** where they used to refer to ground types. All this requires is a suitable change in the interpretation of the metavariables τ.

Inferring Principal Type Schemes

A **principal** type scheme for a term is a scheme τ such that every valid type scheme for the term is an instantiation of τ. It represents the largest (most general) possible family of valid types for the term.

Again, not all languages and type systems have principal types, but the simply-typed lambda calculus and many of its extensions do.

We normally want automated inference procedures to produce a principal type when that is possible.