CS558 Programming Languages

Fall 2015

Lecture 7

VALUES AND TYPES

We divide the universe of values according to **types**; a **type** is:

- a **set** of values; and
- a collection of **operations** defined on those values.

In practice, important to know how values are **represented** and how operations are **implemented** on real hardware.

Examples:

Integers (represented by machine integers) with the usual arithmetic operations (implemented by corresponding hardware instructions).

Booleans (represented by machine bits or bytes) with operators `and`,`or`,`not` (implemented by hardware instructions or code sequences).

Arrays (represented by contiguous blocks of machine addresses) with operations like fetch and update (implemented by address arithmetic and indirect addressing).

Strings (represented how?) with operations like concatenation, substring extraction, etc. (implemented how?)

HARDWARE TYPES

Machine language doesn't distinguish types; all values are just bit patterns until **used**. As such they can be loaded, stored, moved, etc.

But certain **operations** are supported directly by hardware; the operands are thus implicitly typed.

Typical hardware types:

- **Integers** of various sizes, signedness, etc. with standard arithmetic operations.
- **Floating point** numbers of various sizes, with standard arithmetic ops.
- **Booleans** with conditional branch operations.
- **Pointers** to values stored in memory.
- **Instructions**, i.e., code, which can be executed.
- Many others are possible, e.g., binary coded decimal.

Details of behavior (e.g., numeric range) are **machine-dependent**, though often subject to **standards** (e.g., IEEE floating point, Unicode characters, etc.).

Language Primitive Types

Primitive types of a language are those whose values cannot be further broken down by user-defined code; they can be managed only via operators built into the language.

Usually includes hardware types plus others that can easily mapped to a hardware type.

Example: **enumeration** types are usually mapped to integers.

Numeric types only **approximate** behavior of true numbers. Also, they often inherit machine-dependent aspects of machine types, causing serious **portability** problems.

Example: Integer arithmetic in most languages.

Composite Types

Composite types are built from existing types using **type constructors**.

Typical composite types include **records**, **unions**, **arrays**, **functions**, etc.

Abstractly, such type constructors can be seen as mathematical operators on underlying **sets** of simpler values. A small number of set operators suffices to describe most useful type constructors:

Cartesian product ($S_1 \times S_2$)
- records, tuples, C structs

Sum or (disjoint) union ($S_1 \oplus S_2$)
- enumerations, Pascal variant records, C unions

Mapping ($S_1 \to S_2$)
- arrays, association lists, functions

In addition, we often need a way to represent **recursive structures** such as lists and trees.

Composite Type Representation

Concretely, each language defines the internal **representation** of values of the composite type, based on the type constructor and the types used in the construction.

Example: The fields of a record might occupy successive memory addresses (perhaps with some alignment restrictions). The total size of the record is (roughly) the sum of the field sizes.

Often a range of representations are possible, from highly packed to highly indirected. There's often a tradeoff between space and access time.

Example: Arrays of booleans can be efficiently packed using one bit per element, but this makes it more complicated to read or set an element.

STATIC TYPECHECKING

High-level languages differ from machine language in that explicit types appear and type violations are ordinarily caught at some point.

Static typechecking is most common: FORTRAN, Algol, Pascal, C/C++, Java, Scala, etc.

• Types are associated with identifiers (esp. variables, parameters, functions).

• Every use of an identifier can be checked for type-correctness at compile time.

• "Well-typed programs don't go wrong." (If type system is **sound**; often not true.)

• Compiler can optimize generated code because it knows about value representations.

DYNAMIC TYPECHECKING

Dynamic typechecking occurs in Lisp, Scheme, Smalltalk, Python, many other scripting languages, etc.

- Types are attached to values (usually as explicit tags).
- The type associated with an identifier can vary.
- Correctness of operations can't (in general) be checked until runtime.
- Type violations become checked runtime errors.
- Optimized representation hard.

STATIC TYPE SYSTEMS

The main goal of a type system is to characterize programs that won't "go wrong" at runtime.

Informally, we want to avoid programs that confuse types, e.g., by trying to add booleans to integers, or take the square root of a string.

Formally, we can give a set of **typing rules** (sometimes called as **static semantics**) from which we can derive **typing judgments** about program fragments. (This should sound familiar!)

Each judgment has the form

$$TE \vdash e : t$$

Intuitively this says that expression e has type t, under the assumption that the type of each free variable in e is given by the *type environment* TE.

The key point is that an expression is well-typed **if-and-only-if** we can derive a typing judgment for it.

Typing Rules

Consider a variant of our usual simple imperative expression language (see Lecture 6), and suppose we have just two types, `Int` and `Bool`.

We write $TE(x)$ for the result of looking up x in TE, and $TE + \{x \mapsto t\}$ for the type environment obtained from TE by extending it with a new binding from x to t.

Here is a suitable set of typing rules:

$$\frac{TE(x) = t}{TE \vdash x : t} \text{ (Var)}$$

$$\frac{}{TE \vdash i : \texttt{Int}} \text{ (Int)}$$

$$\frac{TE \vdash e_1 : \texttt{Int} \quad TE \vdash e_2 : \texttt{Int}}{TE \vdash \texttt{(+ } e_1 \ e_2\texttt{)} : \texttt{Int}} \text{ (Add)}$$

TYPING RULES (2)

$$\frac{TE \vdash e_1 : \texttt{Int} \quad TE \vdash e_2 : \texttt{Int}}{TE \vdash \texttt{(<= } e_1 \ e_2\texttt{)} : \texttt{Bool}} \text{ (Leq)}$$

$$\frac{TE \vdash e_1 : t_1 \quad TE + \{x \mapsto t_1\} \vdash e_2 : t_2}{TE \vdash \texttt{(let } x \ e_1 \ e_2\texttt{)} : t_2} \text{ (Let)}$$

$$\frac{TE(x) = t \quad TE \vdash e : t}{TE \vdash \texttt{(:= } x \ e\texttt{)} : t} \text{ (Assgn)}$$

$$\frac{TE \vdash e_1 : \texttt{Bool} \quad TE \vdash e_2 : t \quad TE \vdash e_3 : t}{TE \vdash \texttt{(if } e_1 \ e_2 \ e_3\texttt{)} : t} \text{ (If)}$$

$$\frac{TE \vdash e_1 : \texttt{Bool} \quad TE \vdash e_2 : t}{TE \vdash \texttt{(while } e_1 \ e_2\texttt{)} : \texttt{Int}} \text{ (While)}$$

FORMALIZING TYPE SAFETY

The typing rules are just (another) formal system in which judgments can be derived. How do we connect this system with our slogan that "well-typed programs don't go wrong" ?

First we need an auxiliary judgment system assigning types to values, written $\models v : t$.

For example, we would have $\models i : \texttt{Int}$ for every integer i, $\models \texttt{true} : \texttt{Bool}$, and $\models \texttt{false} : \texttt{Bool}$

We also add a special value `error` which does not belong to any type: $\not\models \texttt{error} : t$

We extend this notation to environments and stores, and write
$\models E, S : TE$
iff *dom*$(E) =$ *dom*(TE) and $\models S(E(x)) : TE(x), \forall x \in dom(E)$.

Formalizing Type Safety (2)

Recall our formal **dynamic semantics** for our language, defined using $\Downarrow$ judgments. In our previous definition, expressions corresponding to runtime errors simply had no applicable rule (they were "stuck."). Let's change the system slightly by adding new rules so that all expressions corresponding to runtime errors evaluate to `error` instead.

Now, if everything has been defined correctly, we should be able to prove a **theorem** roughly like this:

If $TE \vdash e : t$ and $\models E, S : TE$ and $\langle e, E, S\rangle \Downarrow \langle v, S'\rangle$ then $\models v : t$.

In other words, well-typed programs evaluate to values of the expected type; so in particular, they can't evaluate to `error`, which belongs to no type.

STATIC TYPECHECKING

We can turn the typing rules into a recursive **typechecking algorithm**.

A typechecker is very similar to the **evaluators** we have already built:

- it is parameterized by a type environment;
- it dispatches according to the syntax of the expression being checked (note that there is exactly one rule for each form);
- it calls itself recursively on sub-expressions;
- it returns a type.

There are some differences, though. For example, a typechecker always examines **both** arms of a conditional (not just one). If we consider a language with **functions**, the typechecker processes the body of each function only once, no matter how many times the function is called.

Note that most languages require the types of function parameters and return values to be **declared** explicitly. The typechecker can use this declaration to check that applications of the function are correctly typed, and **separately** checks that the body of the function is correctly typed.

FLEXIBILITY OF DYNAMIC TYPECHECKING

Static typechecking offers the great advantage of **catching errors early**, and generally supports more efficient execution.

Why ever consider dynamic typechecking?

• Simplicity. For short or simple programs, it's nice to avoid the need for declaring the types of identifiers.

• Flexibility. Static typechecking is inherently more **conservative** about what programs it admits.

For example, suppose function `f` happens to always return `false`. Then

```
(if f() then "a" else 2) + 2
```

will never cause a runtime type error, but it will still be rejected by a static type system.

Perhaps more usefully, dynamic typing allows **container** data structures, to contain mixtures of values of arbitrary types, like this list:

```
List(2, true, 3.14)
```

TYPE INFERENCE

Some statically-typed languages, like ML (and to a lesser extend Scala), offer alternative ways to approach these goals, via **type inference** and **polymorphic typing**.

Type inference works like this:

- The types of identifiers are automatically inferred from the way they are **used**.
- The programmer is no longer required to declare the types of identifiers (although this is still permitted).
- Requires that the types of operators and literals are known.

INFERENCE EXAMPLES

```
(let f (fun (x) (+ x 2))
     (@ f y))
```

The type of `x` must be `int` because it is used as an arg to `+`. So the type of `f` must be `int` $\rightarrow$ `int` (i.e. the type of functions that expect an `int` argument and return an `int` result), and `y` must be an `int`.

```
(let f (fun (x) (cons x nil))
     (@ f true))
```

Suppose `x` has some type `t`. Then the type of `f` must be `t` $\rightarrow$ `(list t)`. Since `f` is applied to a `bool`, we must have `t = bool`.

(For the moment, we're assuming the `f` must be given a unique **monomorphic** type; we will relax this soon...)

SYSTEMATIC INFERENCE

A harder example:

```
(let f (fun (x) (if x p q))
     (+ 1 (@ f r)))
```

Can only infer types by looking at **both** the function's body and its applications.

In general, we can solve the inference task by extracting a collection of typing **constraints** from the program's AST, and then finding a simultaneous solution for the constraints using **unification**.

Extract constraints that tell us how types **must** be related if we are to be able to find a typing derivation. Each node generates one or more constraints.

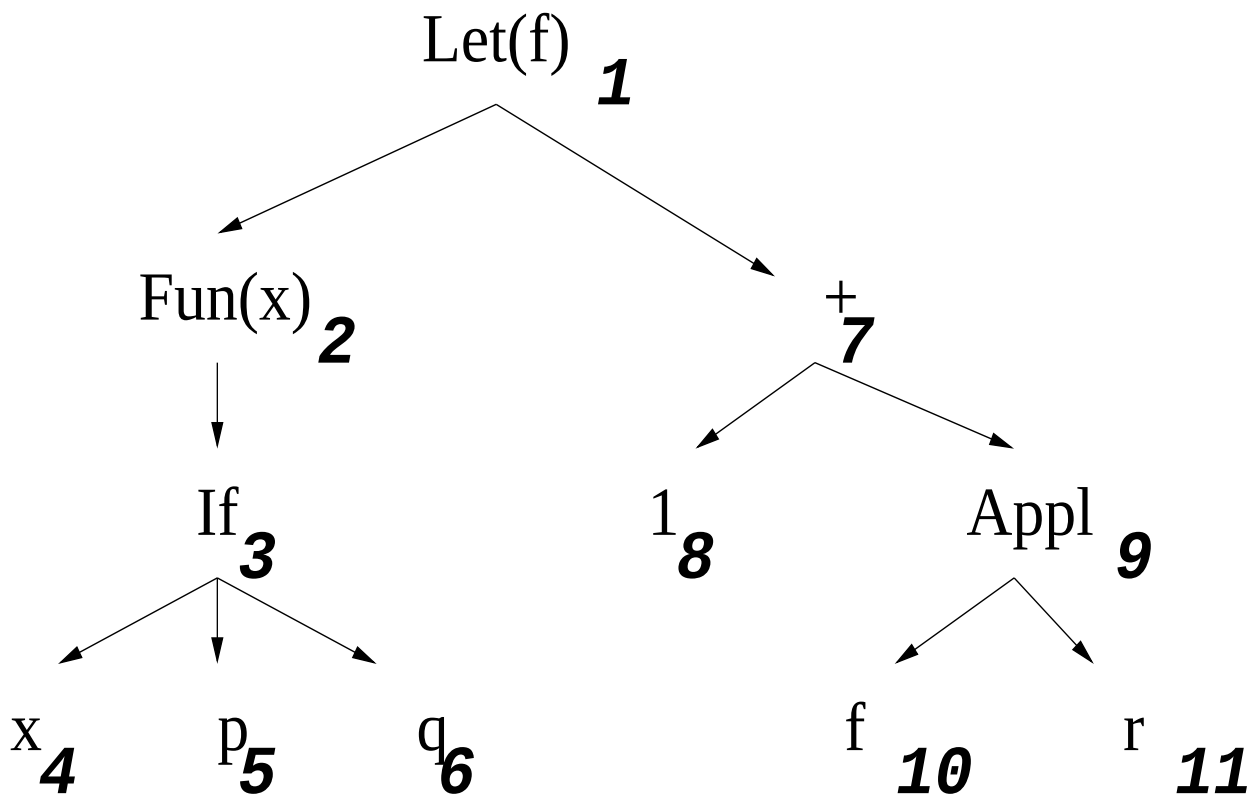
INFERENCE FOR FIRST-CLASS FUNCTIONS

We'll need some extra type judgment rules:

$$\frac{TE + \{x \mapsto t_1\} \vdash e : t_2}{TE \vdash \texttt{(fun (}x\texttt{) } e\texttt{)} : t_1 \to t_2} \text{ (Fn)}$$

$$\frac{TE \vdash e_1 : t_1 \to t_2 \quad TE \vdash e_2 : t_1}{TE \vdash \texttt{(@ } e_1 \ e_2\texttt{)} : t_2} \text{ (Appl)}$$

INFERENCE EXAMPLE



Solving Inference Constraints

Node	*Rule*	*Constraints*	
1	Let	$t_f = t_2$	$t_1 = t_7$
2	Fun	$t_2 = t_x \rightarrow t_3$	
3	If	$t_4 = \texttt{bool}$	$t_3 = t_5 = t_6$
4	Var	$t_4 = t_x$	
5	Var	$t_5 = t_p$	
6	Var	$t_5 = t_q$	
7	Add	$t_7 = t_8 = t_9 = \texttt{int}$	
8	Int	$t_8 = \texttt{int}$	
9	Appl	$t_{10} = t_{11} \rightarrow t_9$	
10	Var	$t_{10} = t_f$	
11	Var	$t_{11} = t_r$	

Solution : $t_1 = t_7 = t_8 = t_9 = t_3 = t_5 = t_p = t_6 = t_q = \texttt{int}$

$t_4 = t_x = t_{11} = t_r = \texttt{bool}$ $\quad$ $t_2 = t_f = t_{10} = \texttt{bool} \rightarrow \texttt{int}$

Drawbacks of Inference

Consider this variant program:

```
(let f (fun (x) (if x p false)
     (+ 1 (@ f r))))
```

Now the body of `f` return type `bool`, but it is used in a context expecting an `int`.

The corresponding extracted constraints will be **inconsistent**; no solution can be found. Can report this to the programmer.

But which is wrong, the definition of `f` or the use? Doesn't really work to associate the error message with a single program point. (In general, may need to consider an arbitrarily long chain of program points.)

POLYMORPHISM

Consider

```
(let snd (fun (l) (head (tail l)))
     (@ snd (cons 1 (cons 2 (cons 3 nil)))))
```

By extracting the constraints as above, and solving, we will conclude that `snd` has type `(list int)` $\rightarrow$ `int`.

It is also perfectly sensible to write:

```
(let snd (fun (l) (head (tail l)))
     (@ snd (cons true (cons false (cons true nil)))))
```

giving `snd` the type `(list bool)` $\rightarrow$ `bool`. Note that the definition of `snd` hasn't changed at all!

So reasonable to ask: why can't we write something like:

```
(let snd (fun (l) (head (tail l)))
     (seq (@ snd (cons 1 (cons 2 (cons 3 nil))))
          (@ snd (cons true (cons false (cons true nil))))))
```

Can do this if we treat the type of `snd` as **polymorphic**.

PARAMETRIC POLYMORPHISM

By default, languages like ML infer the most polymorphic possible type for every function. In this case, these languages would give `snd` the type `(list a)` $\rightarrow$ `a`, where `a` is a **type variable** that is implicitly universally quantified. Each use of `snd` occurs at a particular **instance** of `a` (first at `bool`, then at `int`).

This is called **parametric polymorphism** because the function definition is (implicitly) parameterized by the instantiating type.

Java **generics** and Scala **type parameterization** provide a form of parameteric polymorhism in which type abstraction and instantiation are (mostly) **explicit**.

In this model, the **behavior** of the polymorphic function is **independent** of the instantiating type. For example, an ML compiler typically generates just one piece of object code for each polymorphic function, shared by all instances. (More later.) An alternative is to generate type-specific versions of the code for each different instance.

PARAMETRIC POLYMORPHISM VS. OVERLOADING

Most languages provide some form of **overloading**, where the same symbol means different things depending on the types to which it is applied. E.g., overloading of arithmetic operators to work on either integers or reals is very common.

Although aim is to do "what we expect" rules can get quite complicated!

Some languages (e.g., Ada, C++) support **user-defined** overloading, normally for user-defined types (e.g. complex numbers).

In conventional languages, overloading is resolved **statically**; that is, the compiler selects the appropriate version of the operator once and for all at compiler time. (Different from object-oriented dynamic overriding; more later.)

Overloading is sometimes called **"ad-hoc polymorphism"**. It is fundamentally **different** from parametric polymorphism, because the implementation of the overloaded operator changes according to the underlying types.