CS558 Programming Languages

Fall 2023
Lecture 7a

Andrew Tolmach
Portland State University

Values and Types

- We divide the universe of values according to types
- A type is a set of values and a set of operations on them
- How is each value represented? How is each operation implemented?

Integers with +,-,etc.
machine integer
HW instruction

Booleans with &,|,~
machine bit or byte
HW instruction/sequence

Arrays with read,update
contiguous block of memory
address arithmetic
+ indirect addressing

Functions with apply
closures
jsr instruction

Atomic vs Constructed Types

- Atomic (or primitive) types are those whose values cannot be taken apart or constructed by user code
  - Can only manipulate using built-in language operators
  - Typically includes the types that have direct hardware support e.g., integers, floats, pointers, instructions
- Composite types are built from other types using type constructors
  - User code can construct values, inspect/modify internals
  - E.g. arrays, records, unions, function

Static Type Checking

- High-level languages differ from machine code in that explicit types appear and type violations are usually caught at some point
- Static type checking is probably most common: FORTRAN, Algol, Pascal, C/C++, Java, Scala, etc.
- Types are associated with identifiers (variables, parameters, functions)
- Every use of an identifier can be checked for type-correctness before program is run
- “Well-typed programs don’t go wrong” (if type system is sound)
- Compilers can generate efficient code because it knows how each value is represented
- Type declarations provide useful documentation for code

Dynamic Type Checking

- Dynamic type checking occurs in LISP, Smalltalk, Python, JavaScript, many other scripting languages
- 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 run time
- Type violations become checked run-time errors
- Generating optimized code and value representations is hard
- Programs can be harder to read

Static Type Systems

- 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 reals, or take the square root of a string
- More formally, we can give a set of typing rules (sometimes called static semantics) from which we can derive typing judgments about programs
- Program is well-typed if-and-only-if we can derive a typing judgment for it

Typing Judgments

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.

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.

Rules for a simple language

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

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

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

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

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

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

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

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

Assumes just two types: `Int` and `Bool`

Static Type Checking

- We can turn the typing rules into a recursive type-checking algorithm

- A type checker 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 there is exactly one rule for each expression form)

  - It calls itself recursively on sub-expressions

Static Type Checking (2)

- But there are some differences:
  - Type checker returns a type, not a value
  - It must examine every possible execution path, but just once
    - e.g. it examines both arms of a conditional expression (not just one)
    - e.g. if our language has functions, it processes the body of each function only once, no matter how many places the function is called from
  - Most languages require the types of function parameters and return values to be declared explicitly. The type checker can use this info to check separately that applications of the function are correctly typed and that the body of the function is correctly typed.

Type Compatibility

- Rules so far assume that type checking is based on syntactic equality comparisons between types

- But we can often achieve a sound type system without requiring exact equality

- And we need a way to handle languages in which types can be named

- Leads to general questions about type compatibility or equivalence

Structural equivalence and subtyping

- For example, suppose `x` has type t_1 and `e` has type t_2. For which t_1 and t_2 is it sound to allow the assignment `x := e` ?
- If we think of types as sets of values, then the assignment is sound if every value in set t_2 is also a value in set t_1.
  - That way, any expectations the code might have about `x` will be met by the value of `e`.
- In this case, we say t_2 is a structural subtype of t_1, written $t_2 <: t_1$
- As a important special case, if t_1 and t_2 describe exactly the same set of values, we say they are structurally equivalent, written $t_1 \equiv t_2$

Defining Structural Equivalence

- Structural equivalence is defined inductively:
  - Atomic types are equivalent if they are identical
  - Constructed types are equivalent if they are the same kind of construction and their components are pairwise structurally equivalent.
    - e.g. record types $\{a : t_1, b : t_2\} \equiv \{a : t_3, b : t_4\}$ iff $t_1 \equiv t_3$ and $t_2 \equiv t_4$
    - *similarly for unions, arrays, functions, etc.*

Structural Subtyping

- More generally, a given language might support structural subtyping for atomic or constructed values, e.g. (in imaginary language)

```
char <: int32
```
(if no representation change is needed)

```
{a:int, b:bool} <: {a:int}
```
(records)

```
(int32 => {a:int,b:bool}) <:
(char => {a:int})
```
(functions)

- Depends on details of each type construction in specific language

Incorporating Subtyping

- To add subtyping to language's type system we add a subsumption rule

$$\frac{TE \vdash e : t' \quad t' <: t}{TE \vdash e : t} \text{ (Sub)}$$

- We also extend the <: relation to a preorder

$$\frac{}{t <: t} \text{ (Reflexive)}$$

$$\frac{t'' <: t' \quad t' <: t}{t'' <: t} \text{ (Transitive)}$$

Type Names

- Many languages let us define names for types
- This is a convenient shorthand to avoid repeating long type expressions

```
fun f(r:{x:int,y:bool,z:real}) : {x:int,y:bool,z:real} = …
type t = {x:int,y:bool,z:real}
fun f(r:t) : t = …
```

Standard ML

- For structural type equivalence, we just unfold the names before comparing types
  - But special care is needed for recursive type names

```
type t1 = {a:int,b:t1}      type t2 = {a:int,b:t2}
```

Standard ML syntax

Name equivalence

- A more powerful use of type names is to harness the type-checker to help enforce program correctness by defining sets of values according to their program-specific meaning, e.g.

```
type polar = { r:real, a:real }
type rect = { x:real, y:real }
function polar_add(x:polar,y:polar) : polar = ...
function rect_add(x:rect,y:rect) : rect = ...
var a:polar; c:rect;
a := (150.0,30.0) (* ok *)
polar_add(a,a)  (* ok *)
c := a  (* type error *)
rect_add(a,c) (* type error *)
```

made-up language

- To get the desired feedback from the type checker, we say types are equivalent only if they have the same name

- Name equivalence implies structural equivalence but not vice-versa (that's the whole point!)

Pure Name Equivalence

- Treating all named types as distinct is too restrictive

```
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

made-up language

- Different type names now seem too distinct; cannot even convert from one form of real to another.

- So most languages used mixed equivalence models

C Type Equivalence

- C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types, e.g.

```
char a[100];
void f(char b[]);
f(a); /* ok */

struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; /* type error */
```

C

- A type defined by a `typedef` declaration is just an abbreviation for an existing type.

```
typedef struct polar Polar;
Polar c = a;    /* ok */
```

C

Java Type Equivalence

- Java uses nearly strict name equivalence, where names are either:
  - One of 8 built-in primitive types (`int`,`float`,`boolean`,…), or
  - Declared classes or interfaces (reference types)
- The only non-trivial type expressions that can appear in a source program are array types, which are compared structurally, using name equivalence for the ultimate element type.
- Java has no mechanism for naming type abbreviations.

Java Subtyping

- Java types form a name-based subtyping hierarchy
  - If class A extends class B, then A <: B
  - If class A implements interface I, then A <: I
  - If numeric type t can be coerced to numeric type u without loss of precision, then t <: u
    - This may require the compiler to insert a run time coercion from t to u