Type Systems and Semantics

This material covered in Chapter 3 of the text

- Syntax versus semantics
- Types
- Formal descriptions of programming language semantics
  - Operational semantics
  - Axiomatic semantics (just skim this section in book)
  - Denotational semantics

CSE 341, Winter 2003 1

Type Systems

- Terms to learn:
  - Type
  - Type system
  - Statically typed language
  - Dynamically typed language
  - Type error
  - Strongly typed
  - Weakly typed
  - Type safe program
  - Type safe language

CSE 341, Winter 2003 2

Type – Definition

- Type: a set of values and operations on those values
- Java examples of types:
  - int
    - values = {-2^{31}, …, -2, -1, 0, 1, 2, … , $2^{31}-1$} (2^{31} = 2,147,483,648)
    - operations = {+, -, *, …}
  - boolean
    - values = {false, true}
    - operations = {&&, ||, !, …}
  - String
    - values = {"", "a", "b", … , "A", …, "$", … "Σ", … "aa", "ab", …}
    - operations = {+, trim(), equals(Object x), clone(), …}
  - Applet
    - values = [all possible applets]
    - operations = {init(), paint(Graphics g), clone(), …}

CSE 341, Winter 2003 3

Type System

- A well-defined system of associating types with variables and expressions in the language

CSE 341, Winter 2003 4

Statically Typed Languages

- **Statically typed**. Statically typed means that the type of every expression can be determined at compile time. Java and Haskell are examples of statically typed languages. (Scheme is not statically typed though.)
- Each variable has a single type throughout the lifetime of that variable at runtime.

CSE 341, Winter 2003 5

Dynamically Typed Languages

- **Dynamically typed**. The types of expressions are not known until runtime.
  - Example languages: Smalltalk, Scheme.
- Book adds this additional part to the definition:
  - The type of a variable can change dynamically during the execution of the program
  - This isn't a standard part of the definition of dynamically typed, but it's true for all the dynamically typed languages that I know of
- This is legal Smalltalk code:

```
x ← 3.5.
x ← true.
```

CSE 341, Winter 2003 6

Type error

- A type error is a runtime error that occurs when we attempt an operation on a value for which that operation is not defined.
- Examples:

```
boolean b, c;
b = c+1;


int i;
boolean b;
i = b;
```


Strongly Typed Language

- A language is **strongly typed** if the language guarantees that a value of one type can't be incorrectly used as if it were another type, in other words, that all expressions are guaranteed to be type consistent.
- This checking can be done at compile time, at run time, or a combination of both
- Java, Smalltalk, Scheme, Haskell, and Ada are examples of strongly typed languages.
- Fortran and C are examples of languages that aren't strongly typed.

Weakly Typed

- **Weakly typed**. Weakly typed means "not strongly typed".

Type Safety

- A program is **type safe** if it is known to be free of type errors.
  - However, the system is allowed to halt at runtime before performing an operation that would result in a type error. Unfortunately the book is sloppy about this.
- A language is **type safe** if all legal programs in that language are type safe.
  - So strongly typed language = type safe language.
- Some languages for systems programming, for example Mesa, have a safe subset, although the language as a whole is not type safe.

Tradeoffs

- Generally we want languages to be type safe.
- An exception is a language used for some kinds of systems programming, for example writing a garbage collector. The "safe subset" approach is one way to deal with this problem.
- Advantages of static typing:
  - catch errors at compile time
  - machine-checkable documentation
  - potential for improved efficiency
- Advantages of dynamic typing:
  - Flexibility
  - rapid prototyping

Terminology about Types - Problems

- Unfortunately different authors sometimes use different definitions for the terms "statically typed" and "strongly typed".
- **Statically typed**. The book defines "statically typed" to mean that the compiler can statically assign a type to every expression – but that type might be wrong.
  - By this definition C and Fortran are statically typed.
  - Other authors define "statically typed" to also imply "type safe".
- **Strongly typed**. The book equate strongly typed and type safe (sloppily …)
- For other authors, strongly typed implies type safe and statically typed. (Is Scheme strongly typed?)
- To avoid misunderstanding, one can describe a language as e.g. "type safe and statically typed".

Jay

- Jay is a toy language used in the text to illustrate language concepts. In its original form it has no procedures or functions, and no user-defined types
- Jay is statically and strongly typed

Type checking in Jay

- Informal summary of type checking in Jay:
  - Each variable must have a unique identifier
  - Each variable has a type (int or boolean)
  - Each variable in an expression must have been declared
  - Expression has a result type (details on next slide)
  - For an assignment, the type of the variable on the left must be the same as the type of the expression on the right
  - For a conditional or loop, the type of the expression must be boolean

Result type of an expression

- Let the result type be r. For an expression expr:
  - If expr is a variable, r is the type of the variable
  - If expr is a constant, r is the type of the constant
  - If expr has an arithmetic operator (+ - * /) at the top level, then r is int, and the types of the terms of the operator must be int
  - If expr has a relational operator at the top level, then r is bool, and the types of the terms must be int
  - If expr has a boolean operator at the top level, then r is bool, and the types of the terms must be bool

Jay – Type-checking Mini-Exercise #1

- Describe how the following Jay program would be type-checked.

```
Void main () {
  int j, k;
  boolean b;
  j = 2;
  k = 3;
  b = (j<k) && true;
  if (b) {
    j = j+10;
  }
}
```


Jay – Type-checking Mini-Exercise #2

- Describe how the following Jay program would be type-checked.

```
Void main () {
  int j;
  boolean b;
  j = 2;
  b = 3;
}
```


Semantic Domains

- The semantic domains for a language are sets with well-understood properties, which are independently understood.
- Examples:
  - N (the set of natural numbers)
  - I (the integers)
  - B (*true*, *false*)
- When we are speaking precisely, we distinguish the semantic domains from the types in the language itself (for example, int versus I)

Semantic Domains (2)

- Useful semantic domains for imperative languages:
  - environment γ
    pairs <variable, memory location>
  - memory μ
    pairs <location,value>
  - locations N
    natural numbers
  - state σ
    pairs <variable,value>
    (this is a simplified version, that leaves out memory locations)

Semantic Domains – Example

- Suppose we have variables b, stored at location 100, and k, stored at location 101. At a particular time, b contains false, and k contains 3.
  - environment γ = {<b,100>, <k,101>}
  - memory μ = {<100,false>, <101,3>}
  - locations N = {100, 101}
  - state σ_1 = {<b, false >, <k, 3>}

Semantic Domains – Assignments

- Suppose we have variables b and k (as on the previous slide). After an assignment k=4 we have:
  - environment γ = {<b,100>, <k,101>}
  - memory μ = {<100,false>, <101,4>}
  - locations N = {100, 101}
  - state σ_2 = {<b, false >, <k, 4>}

Semantic Domains – Overriding Union

σ_1 = {<x, 10 >, <y, 20>}

$\sigma_2 = \sigma_1 \overline{\cup}$ {<y, 30>, <z,40>}

So σ_2 = {<x, 10 >, <y, 30>, <z,40>}

Jay – Semantic Domains Mini-Exercise

- What is the state of the following Jay program initially, and after executing each statement?

```
Void main () {
  int j, k;
  boolean b;
  j = 2;
  k = 3;
  b = (j<k) && true;
  if (b) {
    j = j+10;
  }
}
```


Operational Semantics

- Define the meaning of a program by simulating it with a simple abstract machine
- $\sigma(e) \Rightarrow v$
  compute the value v of an expression e in state σ
- Execution rules have the form

$$\frac{\textit{premise}}{\textit{conclusion}}$$

(If the premise is true, then the conclusion is true)

Execution Rules - Examples

- Execution rule for addition:

$$\frac{\sigma(e_1) \Rightarrow v_1 \quad \sigma(e_2) \Rightarrow v_2}{\sigma(e_1 + e_2) \Rightarrow v_1 + v_2}$$

So if $\sigma = \{<x, 10>, <y, 20>\}$ then
$\sigma(x + y) \Rightarrow 30$

Execution Rule for Assignment

- Consider an assignment statement s.target = s.source;

$$\frac{\sigma(\text{s.source}) \Rightarrow v}{\sigma(\text{s.target = s.source;}) \Rightarrow \sigma\ \overline{\cup}\ <\text{s.target},v>\}}$$

So if $\sigma = \{<x, 10>, <y, 20>\}$ then after executing x=x+y
$\sigma = \{<x, 30>, <y, 20>\}$

Execution Rule for Statement Sequences

$$\frac{\sigma(s_1) \Rightarrow \sigma_1 \quad \sigma_1(s_2) \Rightarrow \sigma_2}{\sigma(s_1\ s_2) \Rightarrow \sigma_2}$$

So if $\sigma = \{<x, 10>, <y, 20>\}$
s1 is x=x+y;
s2 is y=0;
then
$\sigma(x=x+y;\ y=0;) \Rightarrow \{<x, 30>, <y, 0>\}$

Other Execution Rules

- Conditionals
- Loops (note that this is a recursive rule)

(see the text for definitions)

Operational Semantics Mini-Exercise #1

- Use the operational semantics rules to find the final state for this program:

```
Void main () {
  int j;
  j = 2;
  if (j<5) {
    j = j+10;
  }
}
```


Operational Semantics Mini-Exercise #2

- Use the operational semantics rules to find the final state for this program:

```
Void main () {
  int j;
  j = 2;
  while (j<5) {
    j = j+2;
  }
}
```

Axiomatic Semantics

- Uses the notion of an assertion: a predicate that describes the state of a program at some point in its execution
- Concepts:
  - Precondition
  - Postcondition
- {P}s{Q}
  - This means that if precondition P holds before executing s, then Q will hold after executing s (provided s halts)
- Other concept: loop invariant
- Expectation in the 60's and 70's: eventually programs would be routinely proved correct.
- This obviously hasn't happened, but the notion of preconditions, postconditions, and loop invariants are still useful

Denotational Semantics

- The denotational semantics of a language defines its meaning in terms of a "meaning function" M
  - As before let σ be a program state.
  - Let Σ be the set of all possible program states.
  - Let *Class* be a kind of element in the language (e.g. Assignment, Conditional, etc)
- Then:

  M: $Class \times \Sigma \to \Sigma$
- In other words, the meaning function M takes a language element and a state σ_1 and returns a new state σ_2

Meaning of Assignments

M: $Assignment \times \Sigma \to \Sigma$

M(*Assignment* a, *State* σ) =

$\sigma \; \overline{\cup} \; \{$<a.target, M(a.source, σ)>$\}$

Example:
M(j=x , {<x,5>, <j, undef>}) = {<x,5>, <j,5>}

Meaning of Expressions

M: Expression $\times$ State $\to$ Value

M(Expression e, State σ)

= e if e is a Value

= σ(e) if e is a Variable

= M(x_1,σ) + M(x_2,σ) if e = $x_1 + x_2$

= M(x_1,σ) - M(x_2,σ) if e = $x_1 - x_2$

More on Denotational Semantics (Optional, Extra Material)

- In most papers on denotational semantics, the meaning function M is written using double brackets:

  [[e]] σ

  rather than

  M(e,σ)
- Strictly speaking, the meaning function applied to a constant takes an element of the language into a semantic domain – these two domains are different:

  [[12]] σ = *12*
- There is a copy of a tutorial on denotational semantics by R.D. Tennent linked from the 341 website for those who would like to learn more.