CS321 Languages and Compiler Design I

Winter 2012

Lecture 13

Static Semantics

Static Semantics are those aspects of a program's meaning that can be studied at at compile time (i.e., without running the program).

Contrasts with **Dynamic Semantics**, which describe how the program behaves at run-time.

Uses for static semantics include:

• Checking validity of input programs to prevent malfunctions at run-time

— e.g., all variables and functions that are used in a program have been defined; correct number and type of arguments are passed to functions, operators, etc.

• Clarifying potential ambiguities about what code does at run-time

— e.g., distinguish between different uses of the same variable name (e.g., global and local) or same symbol (e.g., arithmetic operations on different numeric types).

• Justifying optimizations in code generator

Static/Semantic Analysis

(Static) Semantic analysis refers to the phases of a compiler that are responsible for checking that input programs satisfy the static semantics.

• Static analysis comes after parsing; the structure of a program must be understood before it can be analyzed.

• Static analysis comes before code generation; there's no point generating code for a "bad" program.

Static Approximates Dynamic

In any interesting language, there are aspects of a program's behavior that cannot be determined at compile-time, because of

• Unknown values (e.g. run-time user input)

• Uncomputable problems (e.g. implied by the halting problem)

If we want to check the static semantics of a program at compile-time, then we will have to accept a **conservative approximation**.

Conservative: rejecting programs that might actually be ok, rather than allowing programs that might actually fail.

Approximation: static semantics can tell us something about the result of a computation, but doesn't guarantee to predict every detail.

Approximation Examples

Consider these program fragments:

- `y = 6; x = y * 7;`

We know that the result of `x` will be 42, but static semantics might ensure only that `x` will be an integer.

- `x = (true ?  40 :  "Hello") + 2;`

We know that the result of `x` will be 42, but static semantics might suggest that the program "could" cause a type error

- `int y; if (x*x>=0) y=x;`

We know that `y` will be initialized by this code, but static semantics could suggest that `y` "might not" be initialized.

Types and Types Safety

We focus on the approximation of run-time values by compile-time **types**.

In particular, a static **type system** associates a type with each program identifier (variable, parameter, function, etc.) and expression. The static semantics checks that programs are formed in ways that make sense based on types, e.g.

```
x := x + y;   /* valid if x,y both numeric types */
x := y;       /* valid if x, y have same type */
x := y[10];   /* valid if y is an array containing
                 values of type x */
```

In a **type-safe language**, programs that have no type errors are guaranteed not to have (certain kinds of) runtime errors. This is a huge benefit of using types.

Type information is typically also used by the compiler to figure out how to generate code. For example, the code generated for the above statements may be different depending on whether `x,y` are integers, floats, strings, etc.

Values and Types

Values are the entities or objects manipulated by programs.

We divide the universe of values according to **types**.

We characterize types by:

- A set of values.
- A set of operations defined on those values
- How values are **represented** and operations are **implemented**.
- Whether values are **mutable** or **immutable**.
- In what **contexts** values are permitted.
- How literal values are described.

Examples of Types

Integers with the usual arithmetic operations.

Booleans with operators `and,or,not` and valid as arguments to conditional operations.

Arrays with operations like fetch and store.

Functions with the "call" operation.

Sets with operations like membership testing, union, intersection, etc.

Object-oriented classes with constructor operations and member methods.

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.
- **Booleans** with boolean and conditional operations. (Usually just a special view of integers.)
- **Floating point** numbers of various sizes, with standard arithmetic operations.
- **Pointers** to values stored in memory.
- **Instructions**, i.e., code, which can be executed.

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

Atomic Values and Types

The representation of **atomic** values is opaque to user-defined code; programs cannot "see inside" them.

Many of the **primitive** types that are built into languages are atomic, e.g. integers, floats, characters, booleans, etc.

Usually closely allied to hardware types.

Example: booleans. Note that in most languages (except C/C++), this is a **different** type from integers, even though boolean values may be **represented** internally by 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 is really fixed-word arithmetic which can silently overflow.

Atomic types might also be user-defined, e.g. enumerations.

Composite Values

Composite values are **constructed** from more primitive values, which can usually later be **selected** back from the composite, and perhaps selectively **updated**.

Example: Records (C Syntax)

```
struct emp {
    char *name;
    int age;
};

struct emp e = {"Andrew",99};

if (strcmp(e.name,"Fred")) ...;

e.age = 88;
```

It is typically necessary to declare new composite types (e.g., `struct emp`) before defining composite values (e.g., `e`). The type definition indicates how the type is constructed from more primitive types, using one of a few predefined **type constructors**.

Type Constructors

Programmers usually define composite types in order to implement **data structures** appropriate to an application and/or algorithm.

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

- Cartesian product ($S_1 \times S_2$)
- Disjoint union ($S_1 + S_2$)
- Mapping (by explicit enumeration or by formula) ($S_1 \rightarrow S_2$)
- Set ($\mathcal{P}^S$)
- Sequence (S^*)
- Recursive structures (lists, trees, etc.)

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.

Representation of Data Structures

Historically, most languages provide direct representations only for a few data structures, usually those whose values can be represented **efficiently** on a conventional computer. Often, they are restricted so that all values will be of **fixed size**.

For conventional languages, this is the short list:

- **Records.**
- **Unions.**
- **Arrays.**

Many languages also support manipulation of **pointers** to values of these types, in order to allow moving data "by reference" and to support recursive structures; more later.

Records = Cartesian Products

Records, tuples, "structures", etc. Nearly every language has them.

"Take a bunch of existing types and choose one value from each."

Examples (Ada Syntax)

```
type EMP is
  record
    NAME : STRING;
    AGE : INTEGER;
  end record;

E: EMP := (NAME => "ANDREW", AGE => 99);
```

(ML syntax):

```
type emp = string * int  (unlabeled fields)
val e : emp = ("ANDREW",99);

type emp =
   {name: string, age: int}  (labeled fields)
val e : emp = {name="ANDREW",age=99};
```


Records (continued)

Standard operations: construction, selection, selective update.

Representation: Usually as described above. Because records may be large, they are often manipulated by reference, i.e., represented by a pointer. The fields within a record may also be represented this way.

Allowed contexts: In many languages, treated like primitive values, e.g., can be assigned as a unit, passed to or returned by functions, etc. But since they may be large, some languages add restrictions.

Literals: Most languages allow a literal record to be specified by specifying each component, either by position or by name. (But C doesn't permit literals except as initializers.) Some languages require components to be initialized after creation.

Disjoint Unions

Variant records, discriminated records, unions, etc.

"Take a bunch of existing types and choose one value from one type."

Pascal Example:

```
type RESULT = record
                case found : Boolean of
                  true: (value:integer);
                  false: (error:STRING)
                end;

function search (...) : RESULT;
...
```

Generally behave like records, with **tag** as an additional field.

Represented by the variant's representation, usually plus a tag (thus forming a record). Size typically equals the size of the largest variant plus tag size.

Variant Insecurities

Pascal variant records are **insecure** because it is possible to manipulate the tag independently from the variant contents.

```
tr.value := 101;
write tr.error;

if (tr.found) then begin
   ...
   tr := tr1;
   x := tr.value
```

These problems were fixed in Ada by requiring tag and variant contents to be set simultaneously, and inserting a runtime check on the tag before any read of the variant contents.

Non-discriminated unions

C unions don't even have a tag mechanism: the programmer must provide the tag separately:

```
union resunion {
    int value;
    char *error;
};
struct result {
  int found;  /* boolean tag */
  union resunion u;
}

struct result search (...);
```

Since the connection between tag and union value is informal, this is completely unsafe.

Disjoint Unions Done Properly

ML has very clean approach to building and inspecting disjoint unions:

```
datatype result =  FOUND of integer |  NOTFOUND of string

fun search (..) : result  =
   if ... then FOUND 10 else NOTFOUND "problem"

case search(...) of
  FOUND x =>
    print ("Found it : " ^ (Int.toString x))
| NOTFOUND s =>
    print ("Couldn't find it : " ^ s)
```

Here `FOUND` and `NOTFOUND` tags are **not** ordinary fields. Case **combines** inspection of tag and extraction of values into one operation.

Object-oriented languages like Java don't support disjoint unions directly, but subclasses provide a (somewhat awkward) way to achieve the same effect. (More later.)

Arrays and Mappings

Basic implementation idea: a **table** laid out in adjacent memory locations permitting **indexed access** to any element using the hardware's ability to compute memory addresses.

Mathematically: A finite **mapping** from **index set** to **component set**.

Index set is nearly always a set of integers $0..n$, where n is small enough to allow space for the entire array, *or* some other small discrete set isomorphic to them.

Pascal Example:

```
type day = (Sunday, Monday, ..., Saturday);
var workday = array[day] of boolean;
workday[Saturday] := false;
```

Associative Arrays

Arrays with arbitrary index sets are sometimes called **associative arrays**.

More general index sets are seldom supported directly by language because of the lack of a single, uniform, good implementation.

Awk Example:

```
workday["Saturday"] = false;
workday["Sunday"] = false;
```

How might this be implemented?

Array Size

Is the size of an array part of its type? Some older languages (e.g. Fortran) took this attitude, but most modern languages are more flexible, and allow the size to be set independently for each array value when the array is first created:

- as a local variable, e.g., in Ada:

```
function fred(size:integer);
  var bill: array(0..size) of real;
```

- or on the heap, e.g., in Java:

```
int[] bill = new int[size];
```

Arrays are often large, and hence better manipulated by reference.

Major security issue for arrays is **bounds checking** of index values. In general, it's not possible to check all bounds at compile time (though often possible in particular cases).

Runtime checks are always possible, although they may be costly. But they are essential for type safety, which is a huge benefit!

Functions and Mappings

Mathematical mappings can also be represented by an algorithmic **formula**.

A **function** gives a "recipe" for computing a **result** value from an **argument** value.

A program function can describe an infinite mapping.

But differs from mathematical function in that:

- it must be specified by an explicit algorithm
- executing the function may have side-effects on variables.

It can be very handy to manipulate functions as first-class values. But most languages put severe limitations on what can be done with functions.

How does one represent a function as a first-class value? In some languages, can just use a **code pointer**. In others, representation must include values of **free variables**, which can affect runtime peroformance. More on this later.

Sequences

What about data structures of essentially **unbounded** size, such as **sequences** (or **lists**)?

"Take an arbitrary number of values of some type."

Such data structures require special treatment: they are typically represented by small segments of data linked by pointers, and dynamic storage allocation (and deallocation) is required.

The basic operations on a sequence include

- **concatenation** (especially concatenating a single element onto the head or tail of an existing sequence); and
- **extraction** of elements (especially the head).

An important example is the (unbounded) **string**, a sequence of chars.

Best representation depends heavily on what nature and frequency of various operations. Hard to give single, uniformly efficient implementation. So many older languages don't support directly. But so useful that newer languages increasingly do (esp. strings).

Defining Sequences

Unless the programming language supports sequences directly, the programmer must define them using a **recursive** definition.

For example, a list of integers is either

- **empty**, or
- has a **head** which is an integer and **tail** which is itself a list of integers.

ML has particularly clean mechanisms for describing recursive types.

```
datatype intlist =  EMPTY | CELL of int * intlist
```

Internally, the non-empty case can be represented by a two-element heap-allocated record, containing an integer and a **pointer** to another list. (Obviously, the tail list itself cannot be embedded in the record, since it's size is unknown.) The empty case is conveniently represented by a `null` pointer. Corresponds directly to C representation:

```
typedef struct intlist *Intlist;
struct intlist { int val; Intlist next; };
```


Processing Sequences: Iteration or Recursion

```
/* Iterative version */
int inlist(Intlist list, int i) {
  while (list) {
    if (list->val == i)
      return 1;
    else
      list = list->next;
  };
  return 0;
}

/* Recursive Version */
int inlist(Intlist list, int i) {
  if (list) {
    if (list->val == i)
      return 1;
    else
      return inlist(list->rest,i);
  } else
    return 0;
}
```


Recursive Types

Recursion can be used to define and operate on more complex types, in which the type being defined appears more than once in the definition.

ML Example: binary trees with integer labels (only) at the leaves.

```
datatype tree =
  INTERNAL of {left:tree,right:tree}
| LEAF of {contents:int}
```

Now we **must** use recursion (not iteration) to process the full tree:

```
fun sum(tree: tree) =
  case tree of
    INTERNAL{left,right} =>
        sum(left) + sum(right)
  | LEAF{contents} => contents
```


Values and References

What does the assignment `x := y` actually do at runtime?

If `x,y` are integers, they are probably stored as 32-bit words, and assignment simply copies the word from one memory location (or machine register) to another.

If `x,y` belong to a constructed type (record, array), they are probably stored as a block of words. Assignment might still be defined as copying...

For example, C defines assignment for `structs` this way, so the program

```
struct emp { char* name; int age; }
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
printf("%d %d", e1.age, e2.age);
```

prints 18, 91.

Values and References (2)

But copying large numbers of words is expensive. Moreover, it is difficult to compile such copies if the size of the value is not statically known.

This is particularly true for recursive structures, which naturally grow without fixed bound (and are commonly allocated on the **heap**).

So many languages represent and manipulate (at least some) composite types by **reference**. That is, a value of such a type is actually a **pointer** to the data, and assignment is just a **shallow copy** of the pointer.

For example, a similar Java program:

```
class emp {  String name;  int age; }
emp e1;
e1.age = 91;
emp e2 = e1;
e1.age = 18;
System.out.print(e1.age + " " + e2.age);
```

prints 18, 18.

Explicit Pointers

Many older languages have **pointer types** to enable programmers to construct recursive data structures, e.g., in C:

```
typedef struct intlist *Intlist;
struct intlist {
  int head;
  Intlist tail;
}
Intlist mylist = (Intlist) malloc(sizeof(struct intlist));
...free(mylist)...
```

Note that programmers must make explicit `malloc` (C++: `new`) and `free` calls to manage heap values, and must explicitly manipulate pointers.

Lots of opportunity for **dangling pointer bugs** (failing to realize that a pointer is no longer valid after freeing) and **memory leaks** (failing to free a pointer when it is no longer needed)!

A language that allows dangling pointers cannot have a safe type system!

(Full type safety story in C/C++ is even worse...)

Automated Heap Management

Many modern languages, such as Java and ML, **implicitly** allocate space for composite values (records, disjoint unions, arrays) on the heap.

All such values are represented by references (pointers) into the heap, but programmers can't manipulate the pointers directly. In particular, they can't explicitly deallocate records (or objects) from the heap.

Instead, these languages provide automatic **garbage collection** of unreachable heap values, thus avoiding both **dangling pointer** and **memory leak** bugs.

Garbage collection may add some overhead over manual memory management (though not necessarily). But it allows these languages to be **type-safe**, which is a huge benefit.

Dynamic Typing

Some high-level languages take a different approach to catching bad uses of values: instead of detecting them statically, they check for them explicitly at **runtime**.

For example, before performing an operation like `x+y`, the program will check to make sure that `x` and `y` contain numeric values. If not, the runtime system will issue an informative error message and halt the program (or at least raise an exception). (Note: this is **not** the same as crashing with no warning!)

This so-called **dynamic typing** occurs in Lisp, Scheme, Smalltalk, VB, JavaScript, etc.

- Values carry tags corresponding to their (current) type.
- The type associated with identifiers can vary dynamically as the program runs.
- Correctness of operations can't generally be checked until runtime.
- Program incurs overhead for tagged representations and checking.

FLEXIBILITY OF DYNAMIC TYPING

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

So why ever settle for dynamic typing?

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

• Flexibility. Dynamic typing allows **container** types, like lists or arrays, to contain mixtures of values of arbitrary types.

Note: Some statically-typed languages offer alternative ways to achieve these aims, via **type inference** and **polymorphic typing**. (More later.)