

Compiler Construction

SMD163

Lecture 7: Semantic Analysis

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.
Contains material generously provided by Mark P. Jones



COMPUTER SCIENCE
AND ELECTRICAL ENGINEERING
LULEÅ UNIVERSITY OF TECHNOLOGY

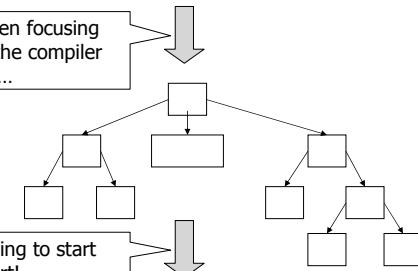
1

The Big Picture:

Flat input

So far, we've been focusing on the parts of the compiler that build trees ...

Structure

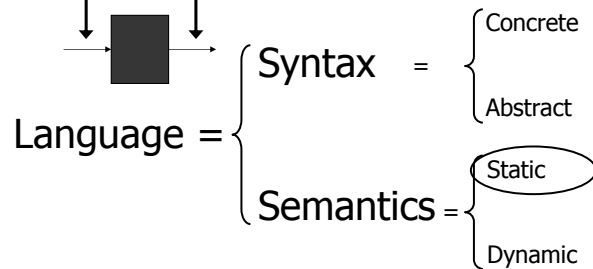


... Now we're going to start taking them apart!

Flat output

2

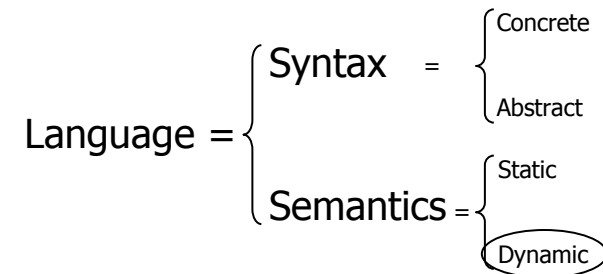
Structured representation Validated representation



Static Semantics: those aspects of a program's meaning that must be verified at compile time.

3

Not to be Confused With ...



Dynamic Semantics: the behavior of a program at run-time.

(But we will return to this subject soon...)

4

Uses of Static Analysis (1):

- ◆ To ensure validity of input programs:
 - Check that all variables and functions that are used in a program have been defined.
 - Check that the correct number and type of arguments are passed to functions, operators, etc...
 - Check that all variables are initialized before they are used.
- ◆ If we don't make these tests at compile-time, the program might malfunction at run-time.

5

Uses of Static Analysis (2):

- ◆ To clarify potential backend ambiguities:
 - Distinguish between different uses of the same variable name (e.g., global and local);
 - Distinguish between different uses of the same symbol (e.g., arithmetic operations on different numeric types).
- ◆ If we don't do these kinds of analysis, then it might be more difficult to compile input programs.

6

Uses of Static Analysis (3):

- ◆ To justify backend optimizations:
 - To allow run-time type checks to be omitted;
 - To identify redundant computations;
 - To identify repeated computations;
 - To make good use of the machine's registers or other resources;
- ◆ If we don't do these kinds of analysis, then we might not get the best performance for compiled programs.

7

Specifying Static Semantics:

- ◆ The static semantics of a language is part of the language specification.
- ◆ Some languages use very strict checking, others are more relaxed.
- ◆ But, in any interesting language, there are aspects of a program's behavior that cannot be determined at compile-time.
 - Unknown values;
 - Uncomputable problems.

8

Dealing with Unknown Values:

- ◆ Some values are not known at compile-time.
- ◆ In Java, we can get a representation of the current time and date using:
`Date today = new Date();`
- ◆ The actual result will depend on when we run the program, which isn't known at compile-time
- ◆ But we can be sure that the result will be a Date ... this is the result of type checking.

9

The Halting Problem:

- ◆ A compiler cannot, in general, distinguish programs that may loop infinitely from programs that are guaranteed to terminate.
- ◆ For example, write a function:
`boolean halts(Program p, Input i)`
;
which returns true if p halts on input i, and false if it doesn't ...
- ◆ ... it can't be done!

10

It can't be done!

- ◆ If we could write halts, then we could also write:

```
boolean loopIfHalts(Program p, Input i) {
    if (halts(p,i)) {
        while (true) ;
    } else {
        return true;
    }
}
boolean testSelf(Program p) {
    return loopIfHalts(p,p);
}
```

- ◆ What is the result of `testSelf(testSelf)`?

11

Static Approximates Dynamic:

- ◆ 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.

12

Some Examples:

- ◆ Given a program: `x = 6 * 7;`
 - We know that the result will be 42.
 - A static semantics might just ensure that the result will be an integer.
- ◆ Given a program: `(true ? 42 : "Hello")`
 - We know that the result will be 42.
 - A static semantics might suggest that the program "could" cause a type error ...
- ◆ Given a program: `int y; if (x*x>=0) y=x;`
 - We know that y will be initialized by this code.
 - A static semantics could suggest that y "might not" be initialized.

13

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.

14

Analysis at Program Points:

- ◆ Some kinds of semantic analysis work by trying to predict the state of a computation at individual program points:

```

int a=0, b=1, c=0;
for (;;) {
  c = a + b;
  a = b;
  b = c;
}

```

The sets at each program point here tell us which variables are "live"; i.e., which variables contain values that might be used later in the computation.

Observations: The initial assignment to c is not used. Three variables, but only two are live at any time.

15

Analysis on Program Variables:

- ◆ Many useful properties can be determined by using an environment to associate semantic information with program variables.
- ◆ Typical semantic properties include:
 - Does the program include a declaration for v?
 - Does the program include a definition for v?
 - What type of value is held in v?
 - Has v been properly initialized?
 - Does anybody ever use v?
 - Where will v be stored in the machine?

16

Environments (continued):

- ◆ Different environments will be needed at different points in a program, depending on which variables are in scope at each point.
- ◆ In some languages, a single variable might be associated with multiple, distinct sets of semantic information. (e.g., to accommodate overloading of a function name.)
- ◆ Some texts use the term "symbol table" instead of "environment".

17

Manipulating Environments:

- ◆ Useful operations on environments include:
 - A facility for looking up a variable v in an environment, E :
 - ◆ To see whether v is defined; and
 - ◆ If it is defined, to find the associated semantic information, $E(v)$.
 - A facility for extending an environment with information about new variables when they come into scope.

$$(E_1 \oplus E_2)(v) = \begin{cases} E_1(v), & \text{if } v \text{ defined in } E_1 \\ E_2(v), & \text{otherwise} \end{cases}$$

Information about v in E_1
masks information for v in E_2

18

For Example:

- ◆ We can use an environment to describe the types of variables in the following program fragment:

```

int x = 0;
if (x > 1) {
    int y = 1;
    f(x, y);
} else {
    boolean x = false;
    g(x);
}
    
```

$E_1 = \{(x, \text{int})\}$

$E_2 = \{(y, \text{int})\}$

$E_3 = \{(x, \text{boolean})\}$

uses $E_2 \oplus E_1$

uses $E_3 \oplus E_1$

19

Implementing Environments:

We can implement environments as association lists:

```

class VarEnv {
    private Id    id;
    private Type  type;
    private VarEnv next;

    public VarEnv(Id id, Type type, VarEnv next)
    {
        this.id    = id;
        this.type  = type;
        this.next  = next;
    }
    ...
}
    
```

20

Environment Lookup:

- ◆ We can implement environments as association lists:

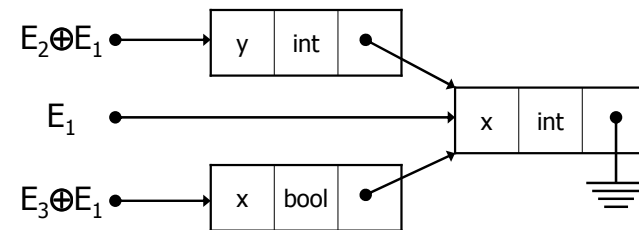
```
// Look for the entry corresponding to a
// particular identifier in a given
// environment.
```

```
public static VarEnv find(Id id, VarEnv env) {
    while (env != null && !env.id.equals(id)) {
        env = env.next;
    }
    return env;
}
```

returns the environment entry for the specified identifier, or null if that identifier is not defined.

21

Environment Linking:



- ◆ Tails of environments can be shared/preserved;
- ◆ Hiding of variables accounted for by taking the first entry for that variable;
- ◆ Access to an environment entry is linear in the size of the environment.

22

Improving on Efficiency:

- ◆ Appel's text discusses two more complex representations of environments, with the aim of improving lookup efficiency
- ◆ One is based on a hashtable, but requires an additional "undo" facility because of the destructive implementation of \oplus
- ◆ The other one uses binary trees to speed up a non-destructive, linked implementation of \oplus
- ◆ Efficiency is of course crucial in a real compiler...
- ◆ However, note that environment size is not dependent on the total number of variables in a program, only the number of variables in each possible nesting of scopes

23

What is a Type?

For now, we'll assume that there is an abstract datatype called `Type`, whose values represent the types of our programming language:

```
public abstract class Type {
    public abstract boolean equal(Type type);
    public static final Type INT = ...;
    public static final Type BOOLEAN = ...;
    public static final Type NULL = ...;
    ...
}
```

We must have a way to determine whether two types are equal.

These constants represent the primitive types of the language.

24

Type Checking Expressions:

Assume we have a class `Exp` in our abstract syntax representing generic expressions.

Now we can add a method to the `Exp` class to describe type checking:

```
public abstract Type typeOf(VarEnv env);  
...  
}
```

Our first semantic
analysis method!

25

Intuition:

- ◆ Given an `Exp e`, we will use the call `e.typeOf(env)` to calculate its type.
- ◆ The `env` parameter supplies the environment for the expression so that we can look up the types of any variables that are used.
- ◆ As part of finding a type for `e`, `e.typeOf(env)` will also compute types for any subexpressions of `e` (by recursively calling `typeOf!`)

26

Type Checking Multiplication:

◆ In English:

- A multiplication expects two integer arguments and returns an integer result.

◆ In Java, add to class `Times` (subclass of `Exp`):

```
public Type typeOf(VarEnv env) {  
    if (!e1.typeOf(env).equal(Type.INT) ||  
        !e2.typeOf(env).equal(Type.INT)) {  
        ... report type error ...  
    }  
    return Type.INT;  
}
```

27

Getting a Good Specification:

- ◆ Natural language is often too informal, and too open to misinterpretation to be used for precise specifications.
- ◆ Computer programs, on the other hand, are usually too specific; programs tend to be cluttered with details that make it harder to see the truly essential details.
- ◆ Is there a happy compromise?

28

Formal Semantics:

- ◆ Inference rules are widely used in formal specifications of programming language semantics:

$$\frac{\text{Hypothesis}_1 \dots \text{Hypothesis}_n}{\text{Conclusion}}$$

- ◆ Read this as a rule: If all of the hypotheses are true, then the conclusion will hold.

$$\frac{\text{Mice like cheese} \quad \text{The moon is made of cheese}}{\text{Mice like living on the moon}}$$

29

Type Checking Multiplication:

- ◆ For example, the rule for type checking multiplication is:

$$\frac{E \vdash e_1 : \text{int} \quad E \vdash e_2 : \text{int}}{E \vdash e_1 * e_2 : \text{int}}$$

- ◆ $E \vdash e : t$ means that, in environment E , the expression e has type t .

30

Checking Variables:

- ◆ To type check the occurrence of a variable, we have to check that the variable is defined in the current environment, and find the corresponding type:

$$\frac{E(v) = t}{E \vdash v : t}$$

31

Or, in Java:

- ◆ Add the following to class IdentifierExp:

```
public Type typeOf(VarEnv env) {
    VarEnv v = VarEnv.find(id, env);
    if (v==null) {
        ... report error for unbound variable ...
    }

    return v.getType();
}
```

(In practice, we might store the `VarEnv v` that we find for this variable as part of the `VarExpr` structure; that will allow static properties of the variable to be accessed during code generation.)

32

Rules, rules, rules ...

$$\frac{}{E \vdash \text{INTEGER} : \text{int}} \quad \frac{}{E \vdash \text{true} : \text{boolean}}$$

$$\frac{E \vdash e_1 : t \quad E \vdash e_2 : t \quad t \in \{\text{int}, \text{boolean}\}}{E \vdash e_1 == e_2 : \text{boolean}}$$

$$\frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \ \&\& \ e_2 : \text{boolean}}$$

33

Putting them Together:

- ◆ We can combine multiple rules to build up complete “proof trees”:

$$\frac{\frac{E(x) = \text{int}}{E \vdash x : \text{int}} \quad \frac{E \vdash 2 : \text{int} \quad \frac{E(y) = \text{int}}{E \vdash y : \text{int}}}{E \vdash 2 * y : \text{int}}}{E \vdash x == 2 * y : \text{boolean}}$$

- ◆ I.e.: In any environment where x and y are both defined with type `int`, the expression `x==2*y` has type `boolean`.

34

Checking Statements:

- ◆ Similar principles apply when we are dealing with statements:

$$\frac{E \vdash e : \text{boolean} \quad E \vdash s_1 \quad E \vdash s_2}{E \vdash \text{IF } e \text{ THEN } s_1 \text{ ELSE } s_2}$$

- ◆ Statements don't have a type, so we just write $E \vdash s$ to indicate that s is well-formed in environment E .
- ◆ In Java, add to class `If`:

```
void check(VarEnv env) {
    if (!e.typeOf(env).equal(Type.BOOLEAN))
        ... error ...
    s1.check(env);
    s2.check(env);
}
```

35

Extending the Environment:

- ◆ A method body with statement s , introducing a variable v of type t would be type-checked as:

$$\frac{\{(v,t)\} \oplus E \vdash s}{E \vdash t \ v; \ s}$$

- ◆ In Java, add to class `MethodDecl`:

```
void check(VarEnv env) {
    VarEnv env1 = new VarEnv(v, t, env);
    s.check(env1);
}
```

- ◆ Note: this is a simplified view of a method body!

36

Variable Declarations:

- ◆ In general, a method can contain an arbitrarily long list of variable declarations, subject to the restriction that all its variables are unique (note: must be checked!)
- ◆ In addition, formal arguments count as local variables as far as type-checking goes
- ◆ More complications: all class variables, as well as all methods of a class, should be in scope when type-checking each method of the class in question (here is where recursion is enabled!)

37

Serious Business!

- ◆ Inference rules are widely used for describing and reasoning about programming language semantics.
- ◆ The definition of Standard ML – a serious programming language – is completely specified using inference rules.
- ◆ In ongoing research, several groups are developing inference rules to describe the full Java language, and using mechanical theorem provers to establish formally that Java programs are “safe”.

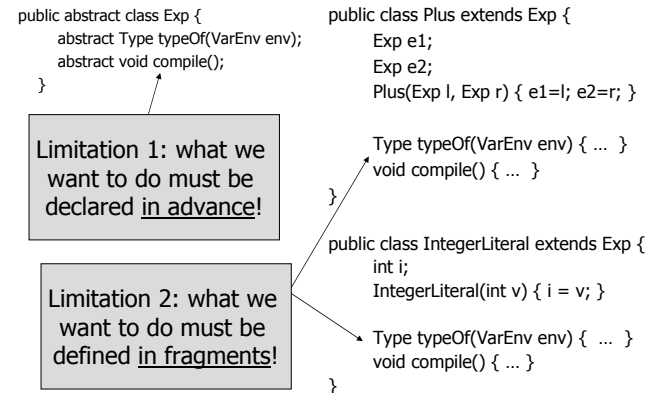
38

Summary:

- ◆ The static semantics of a language specifies properties of programs that must be verified at compile-time.
- ◆ Static properties are used both to validate source programs and to influence code generation.
- ◆ Environments can be used to record details about uses of variables.
- ◆ Static semantics can be specified in different ways, including the formal notation of inference rules.

39

Reflexion: Abstract syntax



40

Abstract Syntax in Java

- ◆ We might have a multitude of Exp subclasses (the MiniJava abstract syntax has 16 such classes !)
- ◆ Wouldn't it be nice if the type-checker could be defined in a single place, instead of being split into 16 different classes (files)?
- ◆ Wouldn't it be even nicer if a new operation on the abstract syntax (a method "optimize", say) could be added without modifying 16 existing classes?

41

The Java alternative: Visitors

- ◆ Idea: Define one method in Exp and every subclass that takes a Visitor object as its argument.

- ◆ The implementation of each such method is simple:

```
Type accept(Typevisitor v) {  
    v.visit(this);  
}
```

- ◆ The implementation of visit in the Visitor class can now be overloaded on its argument!

```
class Typevisitor extends Visitor {  
    Type visit ( Plus e ) { ... }  
    Type visit ( IntegerLiteral e ) { ... }  
    ...  
}
```

42

Drawbacks

- ◆ The definition of accept in each Exp subclass is tedious...
- ◆ We actually need a different accept for each combination of argument and result types our operation requires...

```
Type accept(Typevisitor v) {  
    v.visit(this);  
}  
  
Type accept(Typevisitor v, VarEnv env) {  
    v.visit(this, env);  
}  
  
void accept(Visitor v) {  
    v.visit(this);  
}
```

- ◆ To use or not to use – the choice is yours!

43