CS 320: Principles of Programming Languages

Mark P Jones, Portland State University

Spring 2019

Weeks 9: Static Semantics and Static Analysis

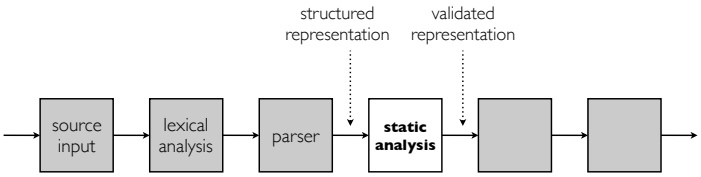
Eliminating errors at compile-time

- Languages can be designed to ensure that certain common classes of error:
  - either cannot occur at all;
  - or else can be detected automatically at compile-time.
- These guarantees are typically reflected in the *static semantics* of a language, and enforced by the use of *static analysis* in its implementations.
- How does this work?
- How far can it take us?

language = { syntax = { concrete, abstract }, semantics = { static, dynamic } }

static semantics: those aspects of a program's behavior/meaning that must be verified at compile time

Static analysis



- Check that the program is reasonable:
  - no references to unbound variables
  - no type inconsistencies
  - etc...

Uses of static analysis (1)

- *To ensure validity of input programs:*
  - Check that all variables and functions that are used in the 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
- *Why?*
  - If we don't make these tests at compile-time, the program might malfunction at run-time

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 operators on different numeric types)
- *Why?*
  - If we don't do these kinds of analysis, then it might be difficult to run, compile, or analyze input programs

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 and other resources
- *Why?*
  - If we don't do these kinds of analysis, then we might not get the best performance for compiled programs

Specifying static semantics

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

Dealing with unknown values

- Some values are not know at compile-time
- In Java, we can find 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

Uncomputable problems

- A compiler cannot, in general, distinguish programs that may loop indefinitely from programs that are guaranteed to terminate
- Exercise: write a function

```
boolean halts(Program p, Input i)
```

that returns `true` if `p` halts on input `i`, and `false` if it doesn't

- This task is known as the "**halting problem**"
- Extra credit homework assignment?
- Don't try too hard: it can't be done!

If you were able to write `halts()` then you could use it to write this code

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

What is `test(test)`?

false? ✘	true? ✘	loops? ✘
Can't occur because `twist()` only loops or returns `true`	So `twist(test,test)` returns `true`	So `twist(test,test)` loops
	So `halts(test,test)` returns `false`	So `halts(test,test)` returns `true`
	So `test(test)` loops	So `test(test)` terminates

✘ conclusion: there is no way to write `halts`!

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 risk 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

Some examples

- Given a program: `x = 6 * 7;`
  We know that the result will be `42`
  Static semantics can confirm that the result is an integer
- Given a program: `(true ? 16 : "Hello")`
  We know that the result will be `16`
  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
  Static semantics could suggest that `y` might not be initialized by this code … and it might be right too!

Static analysis

- Static analysis refers to the phase(s) 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:
  We need static analysis results to enable code generation
  There is no point generating code for a bad program

What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...

    int f(S p, ...) {
        ...
    }

    void g() {
        V y;
        ...
    }
}
```

static analysis does **NOT** check for syntax errors, correct grammar, …

The preceding "syntax analysis" phases do that

but some "syntactic" properties are most easily checked after parsing

`return` statements here must specify a result expression…

`return` statements here must NOT specify a result expression …

examples

What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...

    f(S p, ...) {
    ...
    }

    void g() {
        V y;
        ...
    }
}
```

is there a definition for `D`?

is `D` defined as a subclass of `C`?

is there another definition for `C`?

is there a definition for `T`?

is there a definition for `S`?

valid types?

is there a definition for `V`?

What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...

    int f(S p, ...) {
        ...
    }

    ) {
    ;
        ...
    }
}
```

are all the variables mentioned in `e` defined?

does `e` produce a value of type `T`?

is there another field called `x` in this class?

are there multiple parameters called `p`?

does this code return a value of type `int`?

is there another method called `f` in this class?

valid definitions?

does this have the right access modifiers, if we are overriding a method from `D`?

What exactly does static analysis check?

```
class C extends D {
    T x = e;
    ...

    int f(S p, ...) {
        ...
    }

    void g() {
        V y;
        ...
    }
}
```

valid statements and expressions?

are all the variables used here defined?

do all the expressions here produce results of appropriate types?

is `y` initialized before it is used?

is `y` actually used?

What exactly does static analysis check?

Is `p.q.r` a reference to:

- the package `p.q.r`?
- the class `r` in package `p.q`?
- the class `q.r` in package `p`?
- the inner class `r` in the class `q` in package `p`?
- the static field `r` in the class `q` in package `p`?
- the field `r` in the static field `q` of the class `p`?
- The `r` field of the `q` field in the object referenced by the local variable/parameter `p`?
- ...

the answer depends on the context in which `p.q.r` appears

static analysis can handle this using environments to keep track of which local variables, parameters, classes, packages, are in scope

One additional possible answer: `p.q.r` is not valid!

What exactly does static analysis check?

- In general ... it depends:
  - Static analysis will typically check many properties, as determined by the language definition (imagine the previous examples had been written in Python ...)
  - Individually, many of these are easy to implement
  - But there are usually a lot of checks, and you may have to be careful about the order in which they are performed
- Two of the most common uses of static analysis:
  - Scope resolution: matching every use of an identifier with the corresponding definition
  - Type checking: ensuring that every expression will produce a value of the appropriate type.

Bindings, Scope, and Lifetime

Bindings and scope

- A **binding** is an association between a name and a value/ object/thing/…

  early binding (e.g., during language design, language implementation, program development, compilation time, …) typically results in greater efficiency

  late binding (e.g., link time, load time, run time, …) typically results in greater flexibility.

- The **scope** of a binding is the section of a program in which the binding is active

  static (or **lexical**) **scoping** determined by program structure

  dynamic scoping determined by program execution

Example

Bindings and scope in a typical, block-structured language:

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

creates binding for `x`

scope for `x`

a **hole** in the scope for `x`; the same name is used for a different variable in this scope.

the inner binding **hides** or **shadows** the outer one.

Binding times

- Examples of **early binding**:

  language design time (e.g., the value of pi)
  language implementation time (e.g., number of bits in a word)
  program development (e.g., algorithms/data structures)
  compilation time (e.g., size of an internal buffer)

- Examples of **late binding**:

  link time (e.g., number of modules in a program)
  load time (e.g., the user's home directory)
  run time (e.g., the current time, user input, etc…)

Object lifetimes

- The period of time between the creation and destruction of on object is known as the object's **lifetime**

```
int n = 2;

Point f(int x) {
   int   p = m*x;
   Point q = new Point(p, n);
   n       = n+1;
   return q;
}
```

static lifetime; variable `n` exists throughout program execution

stack lifetime; variables `x`, `p`, `q` exist from introduction to end of function

heap lifetime; this `Point` object exists from point of creation, beyond execution of the function (`q` no longer in scope)

- Object lifetimes do not necessarily coincide with the time when a binding is in scope.

Common lifetime bugs in C/C++ code (1)

- Returning a pointer to a local variable:

```
int* danglingptr() {
   int n = 0;
   return &n;
}
```

the lifetime of this pointer extends beyond the lifetime of the object that it points to!

- How to fix this problem?
  - C/C++: Programmer discipline
  - Java: Language design - no explicit pointers or & operator
  - Rust: Static analysis to track the lifetime of each value and flag an error in situations like this

Common lifetime bugs in C/C++ code (2)

- Failing to free allocated memory:

```
int spaceleak() {
   void* p = malloc(1000);
   return 0;
}
```

the object that `p` points to lasts to the end of the program

`p`'s lifetime extends to the end of the function

- How to fix this problem?
  - C/C++: Programmer discipline
  - Java: Automatic "garbage collection" recovers allocated memory when it becomes unreachable
  - Rust: Automatically frees memory for local variables when they go out of scope

Common lifetime bugs in C/C++ code (3)

- Using memory after free:

```
int useafterfree(int* p) {
   free(p);
   ...
   *p = 0;
}
```

signals that memory pointed to by `p` can be reused for other purposes

`p` is still in scope, but the memory that it points to should not be accessible

- How to fix this problem?
  - C/C++: Programmer discipline
  - Java: No free function (relies on garbage collection)
  - Rust: No free function (the caller owns the data that is pointed to / the callee "borrows" it from the caller)

"These problems don't occur in practice"

- If only that were true!
- Example: https://betanews.com/2017/02/23/red-hat-use-after-free-vulnerability/
  - "Red Hat Product Security has published details of an important security vulnerability in the Linux kernel. The IPv6 implementation of the DCCP protocol means that it is possible for a local, unprivileged user to alter kernel memory and escalate their privileges."
  - (DCCP = Datagram Congestion Control Protocol, as used in streaming, multiplayer games, Internet telephony, …)
- Good programming language design can prevent certain classes of bugs and improve the reliability/safety/security of computer systems in meaningful ways!

Static vs dynamic scoping

- So far, we have assumed "static scoping"
  - The definition (if any) that is referenced by a use of an identifier can be determined without running the program
  - With "dynamic scoping", the definition that is referenced may depend on the runtime behavior of the program
- Example:

```
int n = 1;
void f() { n = 0; }
void g() { int n = 2; f(); }
void h() { g(); print n; f(); print n; }
```

with **static scoping**: `h()` prints 0 and 0

with **dynamic scoping**: `h()` prints 1 and 0

- Example: early versions of LISP and Perl used dynamic scoping
- Not widely adopted in modern languages (it can make programs harder to understand, and more difficult to execute)

Scoping without declarations

- Python identifiers can be defined in
  - the global scope (e.g., top-level assignments, imports)
  - the scope of a function body (e.g., parameters, local vars)
- As in many languages, uses of an identifier x resolve to the nearest enclosing scope that binds x
- But because variables are not explicitly declared it is not obvious where binding should occur
- Unusual Python feature: a variable that is written to anywhere in a function is treated as local to that function ...
  - ... unless a global declaration is used

Examples

```
a = 10
def g(b):
  c = a + b
  return c
print (g(1), a)
```

11 10

```
a = 10
def g(b):
  a = 20
  c = a + b
  return c
print (g(1), a)
```

21 10

```
a = 10
def g(b):
  global a
  a = 20
  c = a + b
  return c
print (g(1), a)
```

21 20

```
a = 10
def g(b):
  c = a + b
  a = 20
  return c
print (g(1),a)
```

UnboundLocalError: local variable 'a' referenced before assignment

Environments

Environments

- Environments are data structures that describe sets of bindings: i.e., functions that map variable names to associated semantic information
- In an interpreter, we might use an environment that maps variable names to the current values of those variables
- In static analysis, we might use an environment that maps variables names to information that specifies:

  What type of value does the variable hold?
  Has the variable been initialized?
  Has the variable been used?
  Where was the variable defined?
  Where will the variable be stored?
  ...

Example: tracking types

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 = true;
    g(x);
}
```

{(x, int)}

{(x, int), (y, int)}

{(x, boolean)}

Alternative notation

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 = true;
    g(x);
}
```

x:int

x:int, y:int

x:boolean

you might also see examples using notation like { x ⟼ int } instead of { x : int } or { (x, int) }

A compositional approach *

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 = true;
    g(x);
}
```

E_1 = x:int

E_2 = y:int

E_3 = x:boolean

E_1 — $E_2 \oplus E_1$ — $E_3 \oplus E_1$

* Building new environments by combining/composing smaller environments

Manipulating environments

- Useful operations on environments include:
  - A facility for extending an environment with information about new variables when they come in to scope

$$(E_2 \oplus E_1)\ v = E_2(v), \quad \text{if } v \text{ is defined in } E_2$$
$$= E_1(v), \quad \text{otherwise}$$

Information in E_2 masks information in E_1

  - A facility for looking up a variable v in an environment E
    - to see whether v is defined/in scope; and
    - if it is defined, find the associated semantic information

Implementing environments

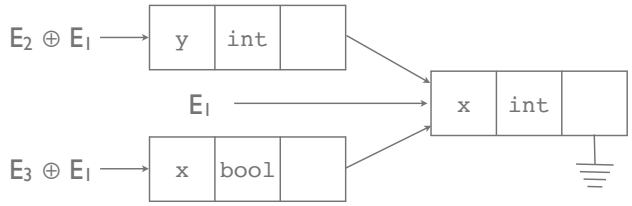
We can implement environments as association lists:

```
/** Represents a variable environment, mapping identifiers
 *  to corresponding types.
 */
public class Env {
    private Id   id;
    private Type type;
    private Env  next;

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

`new Env(id, type, next)` corresponds to $\{ \texttt{id} \longmapsto \texttt{type} \} \oplus \texttt{next}$

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

Dynamic scoping revisited

- Recall the earlier example:

```
int n = 1;
void f() { n = 0; }
void g() { int n = 2; f(); }
void h() { g(); print n; f(); print n; }
```

with **dynamic scoping**: h( ) prints 1 and 0

- One way to implement dynamic scoping:
  - Use an environment **at runtime** to store variable values
  - **Push** an environment entry when a variable is declared
  - **Pop** the environment entry when it goes out of scope
  - Any **use** of a variable refers to the first (most recently defined) occurrence of that name in the environment

Type Checking as a Static Analysis

What is a Type?

- A type is a set of values with an associated set of operations
- For type checking, we work instead with a language of **type expressions**: these are syntactic objects that denote types
- In (simplified) Java:

  `T = int | boolean | char | ClassName | T[] | …`

- In (simplified) Haskell:

  `T = Int | Bool | Char | DataName | T -> T | [T] | (T, T) | …`

- Confusingly enough, people often refer to "type expressions" as "types" (and vice versa) … beware!

What is type checking?

- Type checking is a form of static analysis that attempts to check that every part of a program has the expected type
- Uses an environment to track the type of each variable
- High-level goal for a **type safe language** is to verify that:

  If an expression e has type t …

  … then the meaning (denotation) of e should be a member of the meaning (denotation) of t

- Examples in C (C is not a type safe language)

✓ (1 + 3 * 2) has type int, and 7 is a valid integer

✘ (int*)0 has type "int pointer", but does **not** point to an int

Type checking multiplication

- In English:

  A multiplication expects two numeric arguments of the same type and returns a result with that same type.

- In Java: (in the class for binary arithmetic expressions)

```
/** Return the type of value that will be produced when this
 *  expression is evaluated.
 */
public Type typeOf(Context ctxt, VarEnv env) throws Failure {
    // Covers +, -, *, /
    Type lt = left.typeOf(ctxt, env);
    Type rt = right.typeOf(ctxt, env);
    if (!lt.equal(rt)) {
        throw new Failure("Arithmetic operands have different types");
    }
    if (!lt.equal(Type.INT) && !lt.equal(Type.DOUBLE)) {
        throw new Failure("Invalid operand types for arithmetic operation");
    }
    return lt;
}
```


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, cluttered with implementation details that make it harder to see the parts that are truly essential
- Is there a happy compromise?

Inference rules! (remember CS251?)

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

$$\frac{\text{Hypothesis}_1 \quad \ldots \quad \text{Hypothesis}_n}{\text{Conclusion}}$$

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

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

$$\frac{x > y \qquad y > z}{x > z} \qquad \frac{P(0) \qquad \forall n.P(n) \Longrightarrow P(n+1)}{\forall n.P(n)}$$

Type checking multiplication

- For example, the following rule applies for type checking a multiplication:

$$\frac{t \in \{\text{int, double}\} \qquad env \vdash e_1 : t \qquad env \vdash e_2 : t}{env \vdash e_1 * e_2 : t}$$

- The judgement "env ⊢ e : t" means that, in an environment described by env, the expression e has type t.
- The above rule is equivalent to the following pair of rules:

$$\frac{env \vdash e_1 : \text{double} \qquad env \vdash e_2 : \text{double}}{env \vdash e_1 * e_2 : \text{double}} \qquad \frac{env \vdash e_1 : \text{int} \qquad env \vdash e_2 : \text{int}}{env \vdash e_1 * e_2 : \text{int}}$$

Checking variables

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

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

"the environment function env maps v to type t"

"in environment env, the expression v has type t"

Rules, rules, rules, ...

$$\frac{}{env \vdash \text{INTLIT} : \text{int}} \qquad \frac{}{env \vdash \text{true} : \text{boolean}}$$

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

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

$$\frac{t\ f(t_1, ..., t_n) \quad env \vdash e_1 : t_1 \quad ... \quad env \vdash e_n : t_n}{env \vdash f(e_1, ..., e_n) : t}$$

Putting them together

- We can combine multiple rules to build up complete "proof trees"

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

- In any environment where x and y are both defined with type int, the expression x == 2*y has type boolean

Checking statements

- Similar principles apply with statements. For example:

$$\frac{env \vdash e : \text{boolean} \quad env \vdash s_1 \quad env \vdash s_2}{env \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

- Statements don't have a type, so we just write env ⊢ s to assert that s is well-formed in environment env
- Quiz: What is wrong with each of the following variants?

$$\frac{env \vdash s_1 \quad env \vdash s_2}{env \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2} \qquad \frac{env \vdash e : \text{boolean}}{env \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

$$\frac{env \vdash e{=}{=}\text{true} \quad env \vdash s_1}{env \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2}$$

Serious business!

- The goal is to use inference rules to focus on the essential details, avoiding clutter from implementation details
- Notations like this are widely used by researchers for describing and reasoning about programming language semantics
- The definition of Standard ML, a general purpose programming language, is completely specified using inference rules (including both its static and dynamic semantics)
- Although they are not "official standards", similar formulations have been developed for versions of Java and C, and have been used, in conjunction with automated proof assistants, to reason about the correctness of nontrivial software systems

Summary

- The static semantics of a language specifies properties 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 natural language, type checker implementations, or the formal notation of inference rules