

Last Time

- ◆ Dataflow framework details

Today

- ◆ Norrish's semantics for C

Why do we care?

- ◆ Can't construct a sound program analysis without a firm understanding of what programs mean
- ◆ Formal semantics provides a precise definition of programs' meaning

What's a formal PL semantics?

1. Model for a language telling us what programs in that language mean
 2. Specification for the language's virtual machine
- ◆ No free lunch
 - > Doesn't directly help us find bugs in programs
 - > Doesn't directly help us find bugs in compilers
 - > Doesn't tell us if a program halts
 - > ...

- ◆ Most language standards are in English
 - > What happens if users and implementors disagree on the meaning?
- ◆ A formal semantics is written down in mathematics
 - > Programs need not be translated into machine code to have meaning
- ◆ A mechanized semantics is one that can be manipulated by tools
 - > E.g. to partially automate proof construction and to ensure proof validity

Who benefits?

- ◆ Compiler / interpreter writers need unambiguous specification for how to translate programs
 - > Same for people creating program analysis tools
- ◆ Programmers need unambiguous specification of what the language will do
 - > Same for people creating tools that generate code
- ◆ Formal semantics is the contract between the language user and language implementor
- ◆ Note: For purposes of the C standard, there are only C implementations
 - > Irrelevant whether it's really a compiler, interpreter, JIT, ...

The Future: Divide and Conquer using a Verification Stack

Programmers and tool developers prove properties in terms of the C semantics

————— C semantics

Compiler writers (provably) preserve the meaning of a program across the translation from C to x86

————— x86 semantics

Architects (provably) implement the semantics

Developers prove properties of their Matlab / Simulink / Stateflow models

————— Matlab semantics

Code generator (provably) preserves the meaning of the Matlab program across the translation from Matlab to C

————— C semantics

Compiler writers (provably) preserve the meaning of a program across the translation from C to x86

————— x86 semantics

Architects (provably) implement the semantics

Programmers and tool developers prove properties in terms of the C semantics

————— C semantics

Compiler writers (provably) preserve the meaning of a program across the translation from C to LLVM

————— LLVM semantics

LLVM backend writers (provably) preserve the meaning of a program across the translation from LLVM to object code

————— x86

————— ARM

Architects (provably) implement the x86 semantics

Architects (provably) implement the ARM semantics

◆ Some erroneous program behavior is undefined by the C specification

- Undefined means a valid C implementation may take any action

◆ Examples

- Null pointer dereference
- Divide by zero
- Use of uninitialized memory
- Shift greater than bitwidth

◆ In practice programs with undefined behavior often continue to execute

◆ Formal semantics for C models undefined behavior by transitioning to an error state

What's a Safe Language?

- ◆ A safe language permits no untrapped errors [Cardelli]
 - Note that "error" is subject to definition
 - Many machine languages are safe under this criterion
- ◆ Defining the behavior of dynamically erroneous programs is useful
 - Permits us to reason about these programs' behavior
 - Prevents silent errors that are so common in C
 - Why does C have untrapped erroneous behavior?
 - What are some ways to define the behavior of erroneous programs?
- ◆ Under Cardelli's definition, statically safe languages are impossible

C is Underspecified

- ◆ Implementation defined constructs
 - Chosen once-and-for-all by each C implementation
 - Must be documented
 - Users of this implementation may rely on these
- ◆ Examples
 - Byte ordering
 - Number of bits in a byte
 - Sizes of numeric datatypes
 - Representation of signed quantities
- ◆ Formal semantics model these as fixed, but not at any particular value

More C is Underspecified

- ◆ **Unspecified behavior**
 - Standard gives a choice of behavior with no requirement for consistency or documentation
- ◆ **Examples**
 - Order of evaluation of function arguments
 - Order of evaluation of subexpressions
- ◆ **Formal semantics model these using non-deterministic choice**
 - All possible behaviors are allowed
 - Programs with nondeterminism can still have well-defined meanings

Operational Semantics

- ◆ Can think of this as a language interpreter written in logic
- ◆ Can use all of math (sets, quantifiers, etc.)
- ◆ Rules look like this:

$$\frac{\text{Premise}_1 \text{ Premise}_2 \dots \text{Premise}_n}{\text{start} \rightarrow \text{result}}$$
- ◆ Means that if the premises hold, then start can evaluate to result
 - Say "can" instead of "must" to allow for nondeterminism
 - In other words, sometimes multiple rules could fire
- ◆ Keep applying these rules until all the C is gone and only logic is left
 - Then we can prove things about the residual logic
 - Also: Can prove things about the semantics itself

- ◆ Rule for addition:

$$\frac{e_1 \rightarrow v_1 \quad e_2 \rightarrow v_2}{e_1 + e_2 \rightarrow v_1 + v_2}$$

- ◆ Note that there are two completely different meanings for + here
 - What's the left-hand one?
 - What's the right-hand one?
- ◆ Adding other operators is simple
- ◆ But what are we sweeping under the rug?

Addition Complications

- ◆ C expressions may have side effects
 - Order in which side effects occur is unspecified
- ◆ Addition is polymorphic
 - In the bitwidth of the operands
 - In signed / unsigned
 - In int / float
- ◆ What about operands with different types?
- ◆ What is the C addition function for 8-bit unsigned numbers?
 - $x+y = (x+y)\%256$
- ◆ What is the C addition function for 8-bit signed numbers?
 - Fixme...

- ◆ Rule for if:

$$\frac{G \rightarrow n \quad n \neq 0 \quad e_1 \rightarrow v_1}{\text{if } G \text{ then } e_1 \text{ else } e_2 \rightarrow v_1}$$

$$\frac{G \rightarrow n \quad n = 0 \quad e_2 \rightarrow v_2}{\text{if } G \text{ then } e_1 \text{ else } e_2 \rightarrow v_2}$$

- ◆ No Booleans here
 - In C, guards are just numbers
- ◆ Side effects not handled yet

- ◆ Rules can refer to each other
- ◆ Start with a rule for unary negation

$$\frac{e \rightarrow v}{-e \rightarrow -v}$$

- ◆ Define binary subtraction in terms of unary negation

$$\frac{e_1 + -e_2 \rightarrow v}{e_1 - e_2 \rightarrow v}$$

- ◆ (again note the different uses of +, -)

- ◆ C has variables
- ◆ Represent these as a map from names to values
 - > Called an environment or state
- ◆ Now rules look like this

$$\frac{\dots}{\langle \text{something}, \sigma_0 \rangle \rightarrow \text{result}}$$

- ◆ σ_0 is the environment before the rule fires
- ◆ Result will probably include a new environment indicating changes to the state of memory

- ◆ Rule for sequential composition

$$\frac{\langle \text{stmt}_1, \sigma_0 \rangle \rightarrow \sigma_1 \quad \langle \text{stmt}_2, \sigma_1 \rangle \rightarrow \sigma_2}{\langle \text{stmt}_1; \text{stmt}_2, \sigma_0 \rangle \rightarrow \sigma_2}$$

- ◆ A loop rule

$$\frac{\langle \text{if } G \text{ then } (\text{body}; \text{while } G \text{ do body}), \sigma_0 \rangle \rightarrow \sigma}{\langle \text{while } G \text{ do body}, \sigma_0 \rangle \rightarrow \sigma}$$

- ◆ What about expressions with side effects?

$$\frac{\dots}{\langle e, \sigma_0 \rangle \rightarrow \langle v, \sigma \rangle}$$

- ◆ Rule for a post-increment operator:

$$\frac{\text{state } \sigma \text{ maps var to value } v}{\langle \text{var} ++, \sigma \rangle \rightarrow \langle v, \sigma[\text{var} \mapsto v+1] \rangle}$$

- ◆ The real rules for side-effecting expressions are a lot worse than this example...

Validation

- ◆ Impossible to prove that a formal semantics for C is correct
 - > C standard is in English
- ◆ So, how to validate the formal semantics?
 - > Show that it proves some expected properties
 - $(0 ? s1 : s2) = s2$
 - $\text{for } (i=0; i<10; i++); = i=10;$
 - > Use it as an interpreter
 - I.e., prove that test programs have expected behavior
- ◆ Like any other large, complex software artifact a formal semantics needs to be tested, inspected, and used in real applications before we gain confidence in it

Summary

- ◆ Operational semantics translates AST for a C program into logic
- ◆ Operational semantics is
 - > Well-understood
 - > Extensible
 - > Modular
 - > Mechanizable
 - > Capable of dealing with real languages