Semantics for Safe Programming Languages

David Walker

Summer School on Security
University of Oregon, June 2004

The Current State of Affairs

Software security flaws cost our economy $10-$30 billion/year* ....

* some unverified statistics I have read lately

The Current State of Affairs

Software security flaws cost our economy $10-$30 billion/year* ....

.... and Moore's law applies:

The cost of software security failures is doubling every year.*

* some unverified statistics I have read lately

The Current State of Affairs

- In 1998:
  - 85%* of all CERT advisories represent problems that cryptography can't fix
  - 30-50%* of recent software security problems are due to buffer overflow in languages like C and C++
    - problems that can be fixed with modern programming language technology (Java, ML, Modula, C#, Haskell, Scheme, ...)
    - perhaps many more of the remaining 35-55% may be addressed by programming language techniques

* more unverified stats; I've heard the numbers are even higher

The Current State of Affairs

New York Times (1998): The security flaw reported this week in Email programs written by two highly-respected software companies points to an industry-wide problem – the danger of programming languages whose greatest strength is also their greatest weakness.

More modern programming languages like the Java language developed by Sun Microsystems, have built-in safeguards that prevent programmers from making many common types of errors that could result in security loopholes

Security in Modern Programming Languages

- What do programming language designers have to contribute to security?
  - modern programming language features
    - objects, modules and interfaces for encapsulation
    - advanced access control mechanisms: stack inspection
  - automatic analysis of programs
    - basic type checking: client code respects system interfaces
      - access control code can't be circumvented
    - advanced type/model/proof checking:
      - data integrity, confidentiality, general safety and liveness properties

Security in Modern Programming Languages

- What have programming language designers done for us lately?
  - Development of secure byte code languages & platforms for distribution of untrusted mobile code
    - JVM and CLR
    - Proof-Carrying Code & Typed Assembly Language
  - Detecting program errors at run-time
    - eg: buffer overrun detection; making C safe
  - Static program analysis for security holes
    - Information flow, buffer-overruns, format string attacks
    - Type checking, model checking

These lectures

- Foundations key to recent advances:
  - techniques for giving precise definitions of programming language constructs:
    - without precise definitions, we can't say what programs do let alone whether or not they are secure
  - techniques for designing safe language features:
    - use of the features may cause programs to abort (stop) but do not lead to completely random, undefined program behavior that might allow an attacker to take over a machine
  - techniques for proving useful properties of all programs written in a language
    - certain kinds of errors can't happen in any program

These lectures

- Inductive definitions
  - the basis for defining all kinds of languages, logics and systems
- MinML (PCF)
  - Syntax
  - Type system
  - Operational semantics & safety
- Acknowledgement: Many of these slides come from lectures by Robert Harper (CMU) and ideas for the intro came from Martin Abadi

Reading & Study

- Robert Harper's *Programming Languages: Theory and Practice*
  - http://www-2.cs.cmu.edu/~rwh/plbook/
- Benjamin Pierce's *Types and Programming Languages*
  - available at your local bookstore
- Course notes, study materials and assignments
  - Andrew Myers: http://www.cs.cornell.edu/courses/cs611/2000fa/
  - David Walker: http://www.cs.princeton.edu/courses/archive/fall03/cs510/
  - Others...

Inductive Definitions

Inductive Definitions

Inductive definitions play a central role in the study of programming languages

They specify the following aspects of a language:

- Concrete syntax (via CFGs)
- Abstract syntax (via CFGs)
- Static semantics (via typing rules)
- Dynamic semantics (via evaluation rules)

Inductive Definitions

- An inductive definition consists of:
  - One or more judgments (ie: assertions)
  - A set of rules for deriving these judgments
- For example:
  - Judgment is “n nat”
  - Rules:
    - zero nat
    - if n nat, then succ(n) nat.

Inference Rule Notation

Inference rules are normally written as:

$$\frac{J1 \quad \ldots \quad Jn}{J}$$

where J and J1,..., Jn are judgements. (For axioms, n = 0.)

An example

For example, the rules for deriving n nat are usually written:

$$\frac{}{\text{zero nat}} \qquad \frac{\text{n nat}}{\text{succ(n) nat}}$$

Derivation of Judgments

- A judgment J is derivable iff either
  - there is an axiom

$$\frac{}{J}$$

  - or there is a rule

$$\frac{J1 \quad \ldots \quad Jn}{J}$$

  - such that J1, ..., Jn are derivable

Derivation of Judgments

- We may determine whether a judgment is derivable by working backwards.
- For example, the judgment

  succ(succ(zero)) nat

  is derivable as follows:

a derivation (ie: a proof)

$$\dfrac{\dfrac{\dfrac{}{\text{zero nat}}\,\text{(zero)}}{\text{succ(zero) nat}}\,\text{(succ)}}{\text{succ(succ(zero)) nat}}\,\text{(succ)}$$

optional: names of rules used at each step

Binary Trees

- Here is a set of rules defining the judgment t tree stating that t is a binary tree:

$$\frac{}{\text{empty tree}} \qquad \frac{\text{t1 tree} \quad \text{t2 tree}}{\text{node (t1, t2) tree}}$$

- Prove that the following is a valid judgment:

  node(empty, node(empty, empty)) tree

Rule Induction

- By definition, every derivable judgment
  - is the consequence of some rule...
  - whose premises are derivable
- That is, the rules are an exhaustive description of the derivable judgments
- Just like an ML datatype definition is an exhaustive description of all the objects in the type being defined

Rule Induction

- To show that every derivable judgment has a property P, it is enough to show that
  - For every rule,

$$\frac{J1 \quad \ldots \quad Jn}{J}$$

if J1, ..., Jn have the property P, then J has property P

This is the principal of rule induction.

Example: Natural Numbers

- Consider the rules for n nat

$$\frac{}{\text{zero nat}} \qquad \frac{\text{n nat}}{\text{succ(n) nat}}$$

- We can prove that the property P holds of every n such that n nat by rule induction:
  - Show that P holds of zero;
  - Assuming that P holds of n, show that P holds of succ(n).
- This is just ordinary mathematical induction....

Example: Binary Tree

- Similarly, we can prove that every binary tree t has a property P by showing that
  - empty has property P;
  - If t1 has property P and t2 has property P, then node(t1, t2) has property P.
- This might be called tree induction.

Example: The Height of a Tree

- Consider the following equations:
  - *hgt*(empty) = 0
  - *hgt*(node(t1, t2)) = 1 + max(*hgt*(t1), *hgt*(t2))
- **Claim**: for every binary tree t there exists a unique integer n such that *hgt*(t) = n.
- That is, the above equations define a function.

Example: The Height of a Tree

- We will prove the claim by rule induction:
  - If t is derivable by the axiom

$$\frac{}{\text{empty tree}}$$

  - then n = 0 is determined by the first equation:

hgt(empty) = 0

  - is it unique? Yes.

Example: The Height of a Tree

- If t is derivable by the rule

$$\frac{\text{t1 tree} \quad \text{t2 tree}}{\text{node (t1, t2) tree}}$$

then we may assume that:

- exists a unique n1 such that *hgt*(t1) = n1;
- exists a unique n2 such that *hgt*(t2) = n2;

Hence, there exists a unique n, namely

1+max(n1, n2)

such that *hgt*(t) = n.

Example: The Height of a Tree

This is awfully pedantic, but it is useful to see the details at least once.

- It is not obvious *a priori* that a tree has a well-defined height!
- Rule induction justified the existence of the function *hgt*.

A trick for studying programming languages

99% of the time, if you need to prove a fact, you will prove it by induction on *something*

The hard parts are

- setting up your basic language definitions in the first place
- figuring out what *something* to induct over

Inductive Definitions in PL

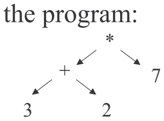
- We will be looking at inductive definitions that determine
  - abstract syntax
  - static semantics (typing)
  - dynamic semantics (evaluation)
  - other properties of programs and programming languages

Inductive Definitions

Syntax

Abstract vs Concrete Syntax

- the concrete syntax of a program is a string of characters:
  - '(' '3' '+' '2' ')' '*' '7'
- the abstract syntax of a program is a tree representing the computationally relevant portion of the program:

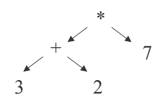


Abstract vs Concrete Syntax

- the concrete syntax of a program contains many elements necessary for parsing:
  - parentheses
  - delimiters for comments
  - rules for precedence of operators
- the abstract syntax of a program is much simpler; it does not contain these elements
  - precedence is given directly by the tree structure

Abstract vs Concrete Syntax

- parsing was a hard problem solved in the '70s
- since parsing is solved, we can work with simple abstract syntax rather than complex concrete syntax
- nevertheless, we need a notation for writing down abstract syntax trees
  - when we write (3 + 2) * 7, you should visualize the tree:



Arithmetic Expressions, Informally

- Informally, an arithmetic expression e is
  - a boolean value
  - an if statement (if e1 then e2 else e3)
  - the number zero
  - the successor of a number
  - the predecessor of a number
  - a test for zero (isZero e)

Arithmetic Expressions, Formally

- The arithmetic expressions are defined by the judgment e exp
  - a boolean value:

$$\frac{}{\text{true exp}} \qquad \frac{}{\text{false exp}}$$

  - an if statement (if e1 then e2 else e3):

$$\frac{\text{e1 exp} \quad \text{e2 exp} \quad \text{e3 exp}}{\text{if e1 then e2 else e3 exp}}$$

Arithmetic Expressions, formally

- An arithmetic expression e is
  - a boolean, an if statement, a zero, a successor, a predecessor or a 0 test:

$$\frac{}{\text{true exp}} \qquad \frac{}{\text{false exp}} \qquad \frac{\text{e1 exp} \quad \text{e2 exp} \quad \text{e3 exp}}{\text{if e1 then e2 else e3 exp}}$$

$$\frac{}{\text{zero exp}} \qquad \frac{\text{e exp}}{\text{succ e exp}} \qquad \frac{\text{e exp}}{\text{pred e exp}} \qquad \frac{\text{e exp}}{\text{iszero e exp}}$$

BNF

- Defining every bit of syntax by inductive definitions can be lengthy and tedious
- Syntactic definitions are an especially simple form of inductive definition:
  - context insensitive
  - unary predicates
- There is a very convenient abbreviation: BNF

Arithmetic Expressions, in BNF

e ::= true | false | if e then e else e

| 0 | succ e | pred e | iszero e

pick a new letter (Greek symbol/word) to represent any object in the set of objects being defined

separates alternatives

(7 alternatives implies 7 inductive rules)

subterm/ subobject is any "e" object

An alternative definition

b ::= true | false

e ::= b | if e then e else e

| 0 | succ e | pred e | iszero e

corresponds to two inductively defined judgements:

1. b bool
2. e exp

the key rule is an inclusion of booleans in expressions: $\dfrac{\text{b bool}}{\text{b exp}}$

Metavariables

b ::= true | false

e ::= b | if e then e else e

| 0 | succ e | pred e | iszero e

- b and e are called metavariables
- they stand for classes of objects, programs, and other things
- they must not be confused with program variables

2 Functions defined over Terms

constants(true) = {true}
constants (false) = {false}
constants (0) = {0}
constants(succ e) = constants(pred e) = constants(iszero e) = constants e
constants (if e1 then e2 else e3) = $U_{i=1-3}$ (constants ei)

size(true) = 1
size(false) = 1
size(0) = 1
size(succ e) = size(pred e) = size(iszero e) = size e + 1
size(if e1 then e2 else e3) = $+_{i=1-3}$ (size ei) +1

A Lemma

- The number of distinct constants in any expression e is no greater than the size of e:

  | constants e | ≤ size e
- How to prove it?

A Lemma

- The number of distinct constants in any expression e is no greater than the size of e:

  | constants e | ≤ size e
- How to prove it?
  - By rule induction on the rules for "e exp"
  - More commonly called induction on the structure of e
  - a form of "structural induction"

Structural Induction

- Suppose P is a predicate on expressions.
  - structural induction:
    - for each expression e, we assume P(e') holds for each subexpression e' of e and go on to prove P(e)
    - result: we know P(e) for all expressions e
  - if you study the theory of safe and secure programming languages, you'll use this idea for the rest of your life!

Back to the Lemma

- The number of distinct constants in any expression e is no greater than the size of e:

  | constants e | ≤ size e

- Proof:

By induction on the structure of e. — always state method first

case e is 0, true, false: ...

case e is succ e', pred e', iszero e': ...

case e is (if e1 then e2 else e3): ...

separate cases (1 case per rule)

The Lemma

- Lemma: | constants e | ≤ size e
- Proof: ...

2-column proof

case e is 0, true, false:

calculation	justification
| constants e | = |{e}|	(by def of constants)
= 1	(simple calculation)
= size e	(by def of size)

A Lemma

- Lemma: | constants e | ≤ size e

...

case e is pred e':

| constants e | = |constants e'|	(def of constants)
≤ size e'	(IH)
< size e	(by def of size)

A Lemma

- Lemma: | constants e | ≤ size e

...

case e is (if e1 then e2 else e3):

| constants e | = $\lvert U_{i=1..3}$ constants ei$\rvert$	(def of constants)
≤ $Sum_{i=1..3}$ |constants ei|	(property of sets)
≤ $Sum_{i=1..3}$ (size ei)	(IH on each ei)
< size e	(def of size)

A Lemma

- Lemma: | constants e | ≤ size e

...

other cases are similar. QED

this had better be true

use Latin to show off ☺

What is a proof?

- A proof is an easily-checked justification of a judgment (ie: a theorem)
  - different people have different ideas about what "easily-checked" means
  - the more formal a proof, the more "easily-checked"
  - when studying language safety and security, we often have a pretty high bar because hackers can often exploit even the tiniest flaw in our reasoning

MinML

Syntax & Static Semantics

MinML, The E. Coli of PL's

- We'll study MinML, a tiny fragment of ML
  - Integers and booleans.
  - Recursive functions.
- Rich enough to be Turing complete, but bare enough to support a thorough mathematical analysis of its properties.

Abstract Syntax of MinML

- The types of MinML are inductively defined by these rules:
  - t ::= int | bool | t → t

Abstract Syntax of MinML

- The expressions of MinML are inductively defined by these rules:
  - e ::= x | n | true | false | o(e,...,e) | if e then e else e
    | fun f (x:t):t = e | e e
- x ranges over a set of variables
- n ranges over the integers ...,-2,-1,0,1,2,...
- o ranges over operators +,-,...
  - sometimes I'll write operators infix: 2+x

Binding and Scope

- In the expression fun f (x:t1) : t2 = e the variables f and x are bound in the expression e
- We use standard conventions involving bound variables
  - Expressions differing only in names of bound variables are indistinguishable
    - fun f (x:int) : int = x + 3 same as fun g (z:int) : int = z + 3
  - We'll pick variables f and x to avoid clashes with other variables in context.

Free Variables and Substitution

- Variables that are not bound are called free.
  - eg: y is free in fun f (x:t1) : t2 = f y
- The capture-avoiding substitution e[e'/x] replaces all free occurrences of x with e' in e.
  - eg: (fun f (x:t1) : t2 = f y)[3/y] = (fun f (x:t1) : t2 = f 3)
- Rename bound variables during substitution to avoid "capturing" free variables
  - eg: (fun f (x:t1) : t2 = f y)[x/y] = (fun f (z:t1) : t2 = f x)

Static Semantics

- The static semantics, or type system, imposes context-sensitive restrictions on the formation of expressions.
  - Distinguishes well-typed from ill-typed expressions.
  - Well-typed programs have well-defined behavior; ill-typed programs have ill-defined behavior
  - If you can't say what your program does, you certainly can't say whether it is secure or not!

Typing Judgments

- A typing judgment, or typing assertion, is a triple G |-- e : t
  - A type context G that assigns types to a set of variables
  - An expression e whose free variables are given by G
  - A type t for the expression e

Type Assignments

- Formally, a type assignment is a finite function G : Variables → Types
- We write G,x:t for the function G' defined as follows:

  G'(y) = t   if x = y
  G'(y) = G(y)   if x ≠ y

Typing Rules

- A variable has whatever type G assigns to it:

$$\frac{}{\text{G |-- x : G(x)}}$$

- The constants have the evident types:

$$\frac{}{\text{G |-- n : int}}$$

$$\frac{}{\text{G |-- true : bool}} \qquad \frac{}{\text{G |-- false : bool}}$$

Typing Rules

- The primitive operations have the expected typing rules:

$$\frac{\text{G |-- e1 : int} \quad \text{G |-- e2 : int}}{\text{G |-- +(e1,e2) : int}}$$

$$\frac{\text{G |-- e1 : int} \quad \text{G |-- e2 : int}}{\text{G |-- =(e1,e2) : bool}}$$

Typing Rules

- Both "branches" of a conditional must have the same type!

$$\frac{G \mid\text{--}\ e : bool \quad G \mid\text{--}\ e1 : t \quad G \mid\text{--}\ e2 : t}{G \mid\text{--}\ \text{if } e \text{ then } e1 \text{ else } e2 : t}$$

- Intuitively, the type checker can't predict the outcome of the test (in general) so we must insist that both results have the same type. Otherwise, we could not assign a unique type to the conditional.

Typing Rules

- Functions may only be applied to arguments in their domain:

$$\frac{G \mid\text{--}\ e1 : t2 \rightarrow t \quad G \mid\text{--}\ e2 : t2}{G \mid\text{--}\ e1\ e2 : t}$$

- The result type of the co-domain (range) of the function.

Typing Rules

- Type checking recursive function:

$$\frac{G, f : t1 \rightarrow t2, x : t1 \mid\text{--}\ e : t2}{G \mid\text{--}\ \text{fun } f\,(x : t1) : t2 = e : t1 \rightarrow t2}$$

- We tacitly assume that $\{f,x\} \cap dom(G) = \{\}$. This is always possible by our conventions on binding operators.

Typing Rules

- Type checking a recursive function is tricky! We assume that:
  - The function has the specified domain and range types, and
  - The argument has the specified domain type.
- We then check that the body has the range type under these assumptions.
- If the assumptions are consistent, the function is type correct, otherwise not.

Well-Typed and Ill-Typed Expressions

- An expression e is well-typed in a context G iff there exists a type t such that G |-- e : t.

- If there is no t such that G |-- e : t, then e is ill-typed in context G.

Typing Example

- Consider the following expression e:

```
fun f (n:int) : int =
  if n=0 then 1 else n * f(n-1)
```

- Lemma: The expression e has type int → int.

To prove this, we must show that

{} |-- e : int → int

Typing Example

$$\{\}\ \text{|-- fun f (n:int):int = if n = 0 then 1 else n*f(n-1) : int} \rightarrow \text{int}$$

Typing Example

$$\frac{\text{G |-- if n = 0 then 1 else n*f(n-1) : int}}{\{\}\ \text{|-- fun f (n:int):int = if n = 0 then 1 else n*f(n-1) : int} \rightarrow \text{int}}$$

where G = f : int → int, n : int

Typing Example

$$\frac{\dfrac{\text{G |-- n=0 : bool} \qquad \text{G |-- 1 : int} \qquad \text{G |-- n*f(n-1) : int}}{\text{G |-- if n = 0 then 1 else n*f(n-1) : int}}}{\{\}\ \text{|-- fun f (n:int):int = if n = 0 then 1 else n*f(n-1) : int} \rightarrow \text{int}}$$

Typing Example

$$\frac{\dfrac{\dfrac{\dfrac{}{\text{G |-- n : int}} \quad \dfrac{}{\text{G |-- 0 : int}}}{\text{G |-- n=0 : bool}} \qquad \dfrac{}{\text{G |-- 1 : int}} \qquad \text{G |-- n*f(n-1) : int}}{\text{G |-- if n = 0 then 1 else n*f(n-1) : int}}}{\{\}\ \text{|-- fun f (n:int):int = if n = 0 then 1 else n*f(n-1) : int} \rightarrow \text{int}}$$

Typing Example

$$\text{Derivation D} = \quad \frac{\dfrac{}{\text{G |-- f : int} \rightarrow \text{int}} \qquad \dfrac{\dfrac{}{\text{G |-- n : int}} \quad \dfrac{}{\text{G |-- 1 : int}}}{\text{G |-- n-1 : int}}}{\text{G |-- f(n-1) : int}}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{\text{G |-- n : int}} \quad \dfrac{}{\text{G |-- 0 : int}}}{\text{G |-- n=0 : bool}} \qquad \dfrac{}{\text{G |-- 1 : int}} \qquad \dfrac{\dfrac{}{\text{G |-- n : int}} \quad \text{Derivation D}}{\text{G |-- n*f(n-1) : int}}}{\text{G |-- if n = 0 then 1 else n*f(n-1) : int}}}{\{\}\ \text{|-- fun f (n:int):int = if n = 0 then 1 else n*f(n-1) : int} \rightarrow \text{int}}$$

Typing Example

- Thank goodness that's over!
- The precise typing rules tell us when a program is well-typed and when it isn't.
- A type checker is a program that decides:
  - Given G, e, and t, is there a derivation of G |-- e : t according to the typing rules?

Type Checking

- How does the type checker find typing proofs?
- Important fact: the typing rules are syntax-directed --- there is one rule per expression form.
- Therefore the checker can invert the typing rules and work backwards toward the proof, just as we did above.
  - If the expression is a function, the only possible proof is one that applies the function typing rules. So we work backwards from there.

Type Checking

- Every expression has at most one type.
- To determine whether or not G |-- e : t, we
  - Compute the unique type t' (if any) of e in G.
  - Compare t' with t

Summary of Static Semantics

- The static semantics of MinML is specified by an inductive definition of typing judgment G |-- e : t.
- Properties of the type system may be proved by induction on typing derivations.

Properties of Typing

- Lemma (Inversion)
  - If G |-- x : t, then G(x) = t.
  - If G |-- n : t, then t = int.
  - If G |-- true : t, then t = bool, (similarly for false)
  - If G |-- if e then e1 else e2 : t, then G |-- e : bool, G |-- e1 : t and G |-- e2 : t.
  - etc...
- Proof: By induction on the typing rules

Induction on Typing

- To show that some property P(G, e, t) holds whenever G |-- e : t, it's enough to show the property holds for the conclusion of each rule given that it holds for the premises:
  - P(G, x, G(x))
  - P(G, n, int)
  - P(G, true, bool) and P(G, false, bool)
  - if P(G, e, bool), P(G, e1, t) and P(G, e2, t) then P(G, if e then e1 else e2)

  and similarly for functions and applications...

Properties of Typing

- Lemma (Weakening):

  If G |-- e : t and G' $\subseteq$ G, then G' |-- e : t.
- Proof: by induction on typing
- Intuitively, "junk" in the context doesn't matter.

Properties of Typing

- Lemma (Substitution):
  If G, x:t |-- e’ : t’ and G |-- e : t, then
  G |-- e’[e/x] : t’.
- Proof: ?

Properties of Typing

- Lemma (Substitution):
  If G, x:t |-- e’ : t’ and G |-- e : t, then
  G |-- e’[e/x] : t’.

$$\frac{\text{G, x:t |-- x : t} \quad \ldots \quad \text{G, x:t |-- x : t} \quad \ldots}{\text{G, x:t |-- e' : t'}} \quad \Rightarrow \quad \frac{\text{G |-- e : t} \quad \ldots \quad \text{G |-- e : t} \quad \ldots}{\text{G |-- e'[e/x] : t'}}$$

MinML

Dynamic Semantics

Dynamic Semantics

- Describes how a program executes
- At least three different ways:
  - Denotational: Compile into a language with a well understood semantics
  - Axiomatic: Given some preconditions P, state the (logical) properties Q that hold after execution of a statement
    - {P} e {Q} Hoare logic
  - Operational: Define execution directly by rewriting the program step-by-step
- We’ll concentrate on the operational approach

Dynamic Semantics of MinML

- Judgment: e $\rightarrow$ e’
  - A transition relation read:

    “expression e steps to e’ ”
  - A transition consists of execution of a single instruction.
  - Rules determine which instruction to execute next.
  - There are no transitions from values.

Values

- Values are defined as follows:
  - v ::= x | n | true | false | fun f (x : t1) : t2 = e
- Closed values include all values except variables (x).

Primitive Instructions

- First, we define the primitive instructions of MinML. These are the atomic transition steps.
  - Primitive operation on numbers (+,-,etc.)
  - Conditional branch when the test is either true or false.
  - Application of a recursive function to an argument value.

Primitive Instructions

- Addition of two numbers:

$$\frac{(n = n1 + n2)}{+(n1, n2) \rightarrow n}$$

- Equality test:

$$\frac{(n1 = n2)}{= (n1, n2) \rightarrow \text{true}} \qquad \frac{(n1 \neq n2)}{= (n1, n2) \rightarrow \text{false}}$$

Primitive Instructions

- Conditional branch:

$$\frac{}{\text{if true then e1 else e2} \rightarrow \text{e1}}$$

$$\frac{}{\text{if false then e1 else e2} \rightarrow \text{e2}}$$

Primitive Instructions

- Application of a recursive function:

$$\frac{(v = \text{fun } f\,(x : t1) : t2 = e)}{v\ v1 \rightarrow e[v/f]\ [v1/x]}$$

- Note: We substitute the entire function expression for f in e!

Search Rules

- Second, we specify the next instruction to execute by a set of search rules.
- These rules specify the order of evaluation of MinML expressions.
  - left-to-right
  - right-to-left

Search Rules

- We will choose a left-to-right evaluation order:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)}$$

$$\frac{e2 \rightarrow e2'}{+(v1, e2) \rightarrow +(v1, e2')}$$

Search Rules

- For conditionals we evaluate the instruction inside the test expression:

$$\frac{e \to e'}{\text{if } e \text{ then } e1 \text{ else } e2 \to \text{if } e' \text{ then } e1 \text{ else } e2}$$

Search Rules

- Applications are evaluated left-to-right: first the function then the argument.

$$\frac{e1 \to e1'}{e1\ e2 \to e1'\ e2}$$

$$\frac{e2 \to e2'}{v1\ e2 \to v1\ e2'}$$

Multi-step Evaluation

- The relation $e \to^* e'$ is inductively defined by the following rules:

$$\frac{}{e \to^* e} \qquad \frac{e \to e' \quad e' \to^* e''}{e \to^* e''}$$

- That is, $e \to^* e'$ iff
  $e = e0 \to e1 \to \ldots \to en = e'$ for some $n \geq 0$.

Example Execution

- Suppose that v is the function
  fun f (n:int) :int = if n=0 then 1 else n*f(n-1)
- Consider its evaluation:
  v 3 → if 3=0 then 1 else 3*v(3-1)
- We have substituted 3 for n and v for f in the body of the function.

Example Execution

v 3 → if 3=0 then 1 else 3*v(3-1)
→ if false then 1 else 3*v(3-1)
→ 3*v (3-1)
→ 3*v 2
→ 3*(if 2=0 then 1 else 2*v(2-1)
...
→ 3*(2*(1*1))
→ 3*(2*1)
→ 3*2
→ 6

where v = fun f (n:int) :int = if n=0 then 1 else n*f(n-1)

Induction on Evaluation

- To prove that $e \to e'$ implies P(e, e') for some property P, it suffices to prove
  - P(e, e') for each instruction axiom
  - Assuming P holds for each premise of a search rule, show that it holds for the conclusion as well.

Induction on Evaluation

- To show that e →* e' implies Q(e, e') it suffices to show
  - Q(e, e) (Q is reflexive)
  - If e → e' and Q(e', e'') then Q(e, e'')
    - Often this involves proving some property P of single-step evaluation by induction.

Properties of Evaluation

- Lemma (Values Irreducible)
  There is no e such that v → e.
- By inspection of the rules
  - No instruction rule has a value on the left
  - No search rule has a value on the left

Properties of Evaluation

- Lemma (Determinacy)
  For every e there exists at most one e' such that e → e'.
- By induction on the structure of e
  - Make use irreducibility of values
  - eg: application rules

$$\frac{e1 \to e1'}{e1\ e2 \to e1'\ e2} \qquad \frac{e2 \to e2'}{v1\ e2 \to v1\ e2'} \qquad \frac{(v = \text{fun } f\ (x : t1) : t2 = e)}{v\ v1 \to e[v/f]\ [v1/x]}$$

Properties of Evaluation

- Every expression evaluates to at most one value
- Lemma (Determinacy of values)
  For any e there exists at most one v such that e →* v.
- By induction on the length of the evaluation sequence using determinacy.

Stuck States

- Not every irreducible expression is a value!
  - (if 7 then 1 else 2) does not reduce
  - (true+false) does not reduce
  - (true 1) does not reduce
- If an expression is not a value but doesn't reduce, its meaning is ill-defined
  - Anything can happen next
- An expression e that is not a value, but for which there exists no e' such that e → e' is said to be stuck.
- Safety: no stuck states are reachable from well-typed programs. ie: evaluation of well-typed programs is well-defined.

Alternative Formulations of Operational Semantics

- We have given a "small-step" operational semantics
  - e → e'
- Some people like "big-step" operational semantics
  - e ⇓ v
- Another choice is a "context-based" "small-step" operational semantics

Context-based Semantics

- To avoid multiple search rules in the small-step semantics, we can define the set of "computational contexts" in which an instruction rule can be invoked
- Contexts E ::= [ ] | o(v,...,E,e,...) |
  | if E then e1 else e2
  | E e | v E

Context-based Semantics

- Any expression e that can take a step can be factored into two parts:
  - e = E[r]
  - r is a "redex" – the left-hand side of an instruction rule
  - r ::= o(v,...,v)
    | if true then e1 else e2
    | if false then e1 else e2
    | (fun f(x:t1):t2 = e) v

Context-based Semantics

- Now, we just need one rule to implement all of the search rules:

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']}$$

- Sometimes this makes the specification of the OS and proofs about it much more concise

Summary of Dynamic Semantics

- We define the operational semantics of MinML using a judgment e → e'
- Evaluation is deterministic
- Evaluation can get stuck...if expressions are not well-typed.

MinML

Type Safety

Type Safety

- Java and ML are type safe, or strongly typed, languages.
- C and C++ are often described as weakly typed languages.
- What does this mean? What do strong type systems do for us?

Type Safety

- A type system predicts at compile time the behavior of a program at run time.
  - eg: |-- e : int → int predicts that
    - the expression e will evaluate to a function value that requires an integer argument and returns an integer result, or does not terminate
    - the expression e will not get stuck during evaluation

Type Safety

- Type safety is a matter of coherence between the static and dynamic semantics.
  - The static semantics makes predictions about the execution behavior.
  - The dynamic semantics must comply with those predictions.
- Strongly typed languages always make valid predictions.
- Weakly typed languages get it wrong part of the time.

Type Safety

- Because they make valid predictions, strongly typed languages guarantee that certain errors never occur.
- The kinds of errors vary depending upon the predictions made by the type system.
  - MinML predicts the shapes of values (Is it a boolean? a function? an integer?)
  - MinML guarantees integers aren't applied to arguments.

Type Safety

- Demonstrating that a program is well-typed means proving a theorem about its behavior.
  - A type checker is therefore a theorem prover.
  - Non-computability theorems limit the strength of theorems that a mechanical type checker can prove.
  - Type checkers are always conservative --- a strong type system will rule out some good programs as well as all of the bad ones.

Type Safety

- Fundamentally there is a tension between
  - the expressivenes of the type system, and
  - the difficulty of proving that a program is well-typed.
- Therein lies the art of type system design.

Type Safety

- Two common misconceptions:
  - Type systems are only useful for checking simple decidable properties.
    - Not true: powerful type systems have been created to check for termination of programs for example
  - Anything that a type checker can do can also be done at run-time (perhaps at some small cost).
    - Not true: type systems prove properties for all runs of a program, not just the current run. This has many ramifications. See Francois' lectures for one example.

Formalization of Type Safety

- The coherence of the static and dynamic semantics is neatly summarized by two related properties:
  - Preservation: A well-typed program remains well-typed during execution.
  - Progress: Well-typed programs do not get stuck. If an expression is well-typed then it is either a value or there is a well-defined next instruction.

Formalization of Type Safety

- Preservation:

  If |-- e : t and e → e' then |-- e' : t
- Progress:

  If |-- e : t then either
  - e is a value, or
  - there exists e' such that e → e'
- Consequently we have Safety:

  If |-- e : t and e →* e' then e' is not stuck.

Formalization of Type Safety

- The type of a closed value determines its form.
- Canonical Forms Lemma: If |-- v : t then
  - If t = int then v = n for some integer n
  - If t = bool then v = true or v = false
  - If t = t1 → t2 then v = fun f (x : t1) : t2 = e for some f, x, and e.
- Proof by induction on typing rules.
- eg: If |-- e : int and e →* v then v = n for some integer n.

Proof of Preservation

- Theorem (Preservation)

  If |-- e : t and e → e' then |-- e' : t.
- Proof: The proof is by induction on evaluation.
  - For each operational rule we assume that the theorem holds for the premises; we show it is true for the conclusion.

Proof of Preservation

- Case addition:

  Given:

  $$\frac{(n = n1 + n2)}{+(n1,\ n2) \rightarrow n} \qquad |\text{-- } +(n1,n2) : t$$

  Proof:

Proof of Preservation

- Case addition:

  Given:

  $$\frac{(n = n1 + n2)}{+(n1,\ n2) \rightarrow n} \qquad |\text{-- } +(n1,n2) : t$$

  Proof:

  t = int (by inversion lemma)

Proof of Preservation

- Case addition:

Given:

$$\frac{(n = n1 + n2)}{+(n1, n2) \rightarrow n} \qquad |\text{-- } +(n1,n2) : t$$

Proof:

t = int (by inversion lemma)

|-- n : int (by typing rule for ints)

Proof of Preservation

- Case application:

Given:

$$\frac{(v = \text{fun } f\,(x : t1) : t2 = e)}{v\ v1 \rightarrow e[v/f]\ [v1/x]} \qquad |\text{-- } v\ v1 : t$$

Proof:

Proof of Preservation

- Case application:

Given:

$$\frac{(v = \text{fun } f\,(x : t1) : t2 = e)}{v\ v1 \rightarrow e[v/f]\ [v1/x]} \qquad |\text{-- } v\ v1 : t$$

Proof:

|-- v : t1→ t2; |-- v1 : t1; t = t2 (by inversion)

Proof of Preservation

- Case application:

Given:

$$\frac{(v = \text{fun } f\,(x : t1) : t2 = e)}{v\ v1 \rightarrow e[v/f]\ [v1/x]} \qquad |\text{-- } v\ v1 : t$$

Proof:

|-- v : t1→ t2; |-- v1 : t1; t = t2 (by inversion)

f: t1→ t2, x:t1|-- e : t2 (by inversion)

Proof of Preservation

- Case application:

Given:

$$\frac{(v = \text{fun } f\,(x : t1) : t2 = e)}{v\ v1 \rightarrow e[v/f]\ [v1/x]} \qquad |\text{-- } v\ v1 : t$$

Proof:

|-- v : t1→ t2; |-- v1 : t1; t = t2 (by inversion)

f: t1→ t2, x:t1|-- e : t2 (by inversion)

|-- e [v/f][v1/x] : t2 (by substitution)

Proof of Preservation

- Case addition search1:

Given:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)} \qquad |\text{-- } +(e1,e2) : t$$

Proof:

Proof of Preservation

- Case addition search1:

Given:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)} \qquad \text{|-- } +(e1,e2) : t$$

Proof:

|-- e1 : int (by inversion)

Proof of Preservation

- Case addition search1:

Given:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)} \qquad \text{|-- } +(e1,e2) : t$$

Proof:

|-- e1 : int (by inversion)

|-- e1' : int (by induction)

Proof of Preservation

- Case addition search1:

Given:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)} \qquad \text{|-- } +(e1,e2) : t$$

Proof:

|-- e1 : int (by inversion)

|-- e1' : int (by induction)

|-- e2 : int (by inversion)

Proof of Preservation

- Case addition search1:

Given:

$$\frac{e1 \rightarrow e1'}{+(e1, e2) \rightarrow +(e1', e2)} \qquad \text{|-- } +(e1,e2) : t$$

Proof:

|-- e1 : int (by inversion)

|-- e1' : int (by induction)

|-- e2 : int (by inversion)

|-- +(e1', e2) : int (by typing rule for +)

Proof of Preservation

- How might the proof have failed?
- Only if some instruction is mis-defined. eg:

$$\frac{(m = n)}{= (m, n) \rightarrow 1} \qquad \frac{(m \neq n)}{= (m, n) \rightarrow 0}$$

$$\frac{G \text{ |-- } e1 : int \quad G \text{ |-- } e2 : int}{G \text{ |-- } =(e1,e2) : bool}$$

- Preservation fails. The result of an equality test is not a boolean.

Proof of Preservation

- Notice that if an instruction is undefined, this does not disturb preservation!

$$\frac{(m = n)}{= (m, n) \rightarrow true}$$

$$\frac{G \text{ |-- } e1 : int \quad G \text{ |-- } e2 : int}{G \text{ |-- } =(e1,e2) : bool}$$

Proof of Progress

- Theorem (Progress)

  If |-- e : t then either e is a value or there exists e' such that e → e'.
- Proof is by induction on typing.

Proof of Progress

- Case variables:

  Given:

  G |-- x : G(x)

  Proof: This case does not apply since we are considering closed values (G is the empty context).

Proof of Progress

- Case integer:

  Given:

  |-- n : int

  Proof: Immediate (n is a value). Similar reasoning for all other values.

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  (1) e1 → e1', or (2) e1 = v1 (by induction)

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  (1) e1 → e1', or (2) e1 = v1 (by induction)

  +(e1,e2) → +(e1',e2) (by search rule, if 1)

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  Assuming (2) e1 = v1 (we've taken care of 1)

  (3) e2 → e2', or (4) e2 = v2 (by induction)

  +(v1,e2) → +(v1,e2') (by search rule, if 3)

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  Assuming (2) e1 = v1 (we've taken care of 1)

  Assuming (4) e2 = v2 (we've taken care of 3)

  .

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  Assuming (2) e1 = v1 (we've taken care of 1)

  Assuming (4) e2 = v2 (we've taken care of 3)

  v1 = n1 for some integer n1 (by canonical forms)

  v2 = n2 for some integer n1 (by canonical forms)

  .

Proof of Progress

- Case addition:

  Given:

  $$\frac{\text{|-- e1 : int} \quad \text{|-- e2 : int}}{\text{|-- +(e1,e2) : int}}$$

  Proof:

  Assuming (2) e1 = v1 (we've taken care of 1)

  Assuming (4) e2 = v2 (we've taken care of 3)

  v1 = n1 for some integer n1 (by canonical forms)

  v2 = n2 for some integer n1 (by canonical forms)

  +(n1,n2) = n where n is sum of n1 and n2 (by instruction rule)

  .

Proof of Progress

- Cases for if statements and function application are similar:
  - use induction hypothesis to generate multiple cases involving search rules
  - use canonical forms lemma to show that the instruction rules can be applied properly

  .

Proof of Progress

- How could the proof have failed?
  - Some operational rule was omitted

$$\frac{(m = n)}{= (m, n) \rightarrow \text{true}}$$

$$\frac{\text{G |-- e1 : int} \quad \text{G |-- e2 : int}}{\text{G |-- =(e1,e2) : bool}}$$

Extending the Language

- Suppose we add (immutable) arrays:
  - e ::= [e0,...,ek] | sub ea ei

Extending the Language

- Suppose we add (immutable) arrays:
  - e ::= [e0,...,ek] | sub ea ei

$$\frac{e1 \to e1'}{[v0,...,vj,e1,e2...,ek] \to [v0,...,vj,e1',e2...,ek]}$$

$$\frac{ea \to ea'}{sub\ ea\ ei \to sub\ ea'\ ei} \qquad \frac{ei \to ei'}{sub\ va\ ei \to sub\ va\ ei'} \qquad \frac{0 <= n <= k}{sub\ [v0,..,vk]\ n \to vn}$$

Extending the Language

- Suppose we add (immutable) arrays:
  - e ::= [e0,...,ek] | sub ea ei

$$\frac{e1 \to e1'}{[v0,...,vj,e1,e2...,ek] \to [v0,...,vj,e1',e2...,ek]}$$

$$\frac{ea \to ea'}{sub\ ea\ ei \to sub\ ea'\ ei} \qquad \frac{ei \to ei'}{sub\ va\ ei \to sub\ va\ ei'} \qquad \frac{0 <= n <= k}{sub\ [v0,..,vk]\ n \to vj}$$

$$\frac{G\ |\text{--}\ e0 : t\ \ ...\ \ G\ |\text{--}\ ek : t}{G\ |\text{--}\ [e0,...,ek] : t\ array} \qquad \frac{G\ |\text{--}\ ea : t\ array \quad G\ |\text{--}\ ei : int}{G\ |\text{--}\ sub\ ea\ ei : t}$$

Extending the Language

- Is the language still safe?
  - Preservation still holds: execution of each instruction preserves types
  - Progress fails:

    |-- sub [17,25,44] 9 : int

  but

    |-- sub [17,25,44] 9 : int → ???

Extending the Language

- How can we recover safety?
  - Strengthen the type system to rule out the offending case
  - Change the dynamic semantics to avoid "getting stuck" when we do an array subscript

Option 1

- Strengthen the type system by keeping track of array lengths and the values of integers:
  - types t ::= ... | t array(a) | int (a)
  - a ranges over arithmetic expressions that describe array lengths and specific integer values
- Pros: out-of-bounds errors detected at compile-time; facilitates debugging; no run-time overhead
- Cons: complex; limits type inference

Option 2

- Change the dynamic semantics to avoid "getting stuck" when we do an array subscript
  - Introduce rules to check for out-of-bounds
  - Introduce well-defined error transitions that are different from undefined stuck states
    - mimic raising an exception
  - Revise statement of safety to take error transitions into account

Option 2

- Changes to operational semantics:
  - Primitive operations yield "error" exception in well-defined places

$$\frac{n < 0 \;\text{ or }\; n > k}{\text{sub } [v0,..,vk]\ n \rightarrow \text{error}}$$

  - Search rules propagate errors once they arise

$$\frac{e1 \rightarrow \text{error}}{+(e1,\, e2) \rightarrow \text{error}} \qquad \frac{e2 \rightarrow \text{error}}{+(v1,\, e2) \rightarrow \text{error}}$$

(similarly with all other search rules)

Option 2

- Changes to statement of safety
  - Preservation: If |-- e : t and e → e' and
  e' ≠ error then |-- e' : t
  - Progress: If |-- e : t then either e is a value or
  e → e'
  - Stuck states: e is stuck if e is not a value, not error and there is no e' such that e → e'
  - Safety: If |-- e : t and e →* e' then e' is not stuck.

Weakly-typed Languages

- Languages like C and C++ are weakly typed:
  - They do not have a strong enough type system to ensure array accesses are in bounds at compile time.
  - They do not check for array out-of-bounds at run time.
  - They are unsafe.

Weakly-typed Languages

- Consequences:
  - Constructing secure software in C and C++ is extremely difficult.
  - Evidence:
    - Hackers break into C and C++ systems constantly.
      - It's costing us > $20 billion dollars per year and looks like it's doubling every year.
    - How are they doing it?
      - > 50% of attacks exploit buffer overruns, format string attacks, "double-free" attacks, none of which can happen in safe languages.
    - The single most effective defence against these hacks is to develop software infrastructure in safe languages.

Summary

- Type safety express the coherence of the static and dynamic semantics.
- Coherence is elegantly expressed as the conjunction of preservation and progress.
- When type safety fails programs might get stuck (behave in undefined and unpredictable ways).
  - Leads to security vulnerabilities
- Fix safety problems by:
  - Strengthening the type system, or
  - Adding dynamic checks to the operational semantics.
  - A type safety proof tells us whether we have a sound language design and where to fix problems.