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

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

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

* 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
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
    - e.g. buffer overrun detection; making C safe
  - Static program analysis for security holes
    - Information flow, buffer-oversruns, 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
- Benjamin Pierce’s Types and Programming Languages
  - Available at your local bookstore
- Course notes, study materials and assignments
  - Andrew Myoro: http://www.cs.cornell.edu/courses/ca611/2006fa/
  - David Walker: http://www.cs.princeton.edu/courses/archive/fall03/cs510/
  - Others...

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:

\[
\begin{align*}
J_1 & \ldots \quad J_n \\
\hline
  J
\end{align*}
\]

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

An example

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

\[
\begin{align*}
\text{zero nat} \\
\text{suc(n) nat}
\end{align*}
\]

Derivation of Judgments

- A judgment J is derivable iff either
  - there is an axiom
  \[
  \begin{align*}
  & J \hline \\
  \end{align*}
  \]
  - or there is a rule
  \[
  \begin{align*}
  J_1 & \ldots \quad J_n \\
  \hline
  J
  \end{align*}
  \]
  - such that J1, ..., Jn are derivable

Derivation of Judgments

- We may determine whether a judgment is derivable by working backwards.
- For example, the judgment
  \[
  \text{suc(suc(zero)) nat}
  \]
  is derivable as follows:

  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:

  \[
  \begin{align*}
  & \text{empty tree} \quad \text{t1 tree} \quad \text{t2 tree} \\
  \hline
  \text{node} (\text{empty}, \text{node} (\text{empty}, \text{empty})) \text{ tree}
  \end{align*}
  \]

- Prove that the following is a valid judgment:
  \[
  \text{node} (\text{empty}, \text{node} (\text{empty}, \text{empty})) \text{ tree}
  \]
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

Example: Natural Numbers

- Consider the rules for $n \text{ nat}$
  \[
  \begin{array}{c|c}
  \text{zero nat} & \text{n nat} \\
  \hline
  \text{succ(n) nat} & \text{succ(n) nat}
  \end{array}
  \]

- We can prove that the property $P$ holds of every $n$ such that $n \text{ nat}$ by rule induction:
  - Show that $P$ holds of zero;
  - Assuming that $P$ holds of $n$, show that $P$ holds of $\text{succ(n)}$.
- 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 $t_1$ has property $P$ and $t_2$ has property $P$, then $\text{node}(t_1, t_2)$ has property $P$.
- This might be called tree induction.

Example: The Height of a Tree

- Consider the following equations:
  - $hgt(\text{empty}) = 0$
  - $hgt(\text{node}(t_1, t_2)) = 1 + \max(hgt(t_1), hgt(t_2))$
- **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
    \[
    \text{empty tree}
    \]
  - then $n = 0$ is determined by the first equation:
    \[
    hgt(\text{empty}) = 0
    \]
  - is it unique? Yes.
Example: The Height of a Tree

- If $t$ is derivable by the rule
  \[
  \text{node}(t_1, t_2) \quad \text{tree} \\
\]
  then we may assume that:
  - exists a unique $n_1$ such that $hgt(t_1) = n_1$;
  - exists a unique $n_2$ such that $hgt(t_2) = n_2$;
  Hence, there exists a unique $n$, namely
  \[1 + \max(n_1, n_2)\]
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 \textit{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 \textit{something}.

The hard parts are
- setting up your basic language definitions in the first place
- figuring out what \textit{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 \textit{concrete syntax} of a program is a string of characters:
  \[
  \text{‘(‘ ‘3’ ‘+’ ‘2’ )’ ‘*’ ‘7’}
  \]
- the \textit{abstract syntax} of a program is a tree representing the \textit{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

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

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

\[
\begin{array}{c|c|c|c|c}
\text{true exp} & \text{false exp} & e1 exp & e2 exp & e3 exp \\
\hline
\text{zero exp} & \text{succ e exp} & \text{pred e exp} & \text{iszero e exp} &
\end{array}
\]

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 \((3 + 2) \times 7\), you should visualize the tree:

```
3 2 +
```

Arithmetic Expressions, Formally

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

\[
\begin{array}{c|c|c|c|c}
\text{true exp} & \text{false exp} & e1 exp & e2 exp & e3 exp \\
\hline
\text{if e1 then e2 else e3 exp}
\end{array}
\]

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

\[
\begin{array}{c|c|c|c|c}
\text{e1 exp} & \text{e2 exp} & \text{e3 exp} &
\end{array}
\]

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 ::= \text{true} \mid \text{false} \mid \text{if} \ e \ \text{then} \ e \ \text{else} \ e
\mid 0 \mid \text{suc} \ e \mid \text{pred} \ e \mid \text{iszero} \ e

- Pick a new letter (Greek symbol/word) to represent any object in the set of objects being defined
- Separates alternatives
- Subterm/ subobject is any "e" object

(7 alternatives implies 7 inductive rules)

An alternative definition

b ::= \text{true} \mid \text{false}

e ::= b \mid \text{if} \ e \ \text{then} \ e \ \text{else} \ e
\mid 0 \mid \text{suc} \ e \mid \text{pred} \ e \mid \text{iszero} \ e

corresponds to two inductively defined judgements:
1. \text{bool}
2. \text{exp}
The key rule is an inclusion of booleans in expressions:
\frac{\text{bool}}{\text{exp}}

2 Functions defined over Terms

\text{constants(true)} = \{\text{true}\}
\text{constants(false)} = \{\text{false}\}
\text{constants(0)} = \{0\}
\text{constants(suc} e) = \text{constants(pred} e) = \text{constants(iszero} e) = \text{constants} e
\text{constants(if} e_1 \ \text{then} \ e_2 \ \text{else} \ e_3) = \bigcup_{i=1}^{3} \text{constants} e_i

\begin{align*}
\text{size(true)} & = 1 \\
\text{size(false)} & = 1 \\
\text{size(0)} & = 1 \\
\text{size(suc} e) & = \text{size(pred} e) = \text{size(iszero} e) = \text{size} e + 1 \\
\text{size(if} e_1 \ \text{then} \ e_2 \ \text{else} \ e_3) & = \tau_{r+1} (\text{size} e) + 1
\end{align*}

Metavariabes

b ::= \text{true} \mid \text{false}

e ::= b \mid \text{if} \ e \ \text{then} \ e \ \text{else} \ e
\mid 0 \mid \text{suc} e \mid \text{pred} e \mid \text{iszero} e

- \text{b and e are called metavariables}
- They stand for classes of objects, programs, and other things
- They must not be confused with program variables

A Lemma

- The number of distinct constants in any expression e is no greater than the size of e:
  \mid \text{constants} e \mid \leq \text{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:
  \mid \text{constants} e \mid \leq \text{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:
  \[ | \text{constants e} | \leq \text{size e} \]

- Proof:
  By induction on the structure of e.
  case e is 0, true, false: ...
  case e is succ e', pred e', iszero e': ...
  case e is (if e1 then e2 else e3): ...

The Lemma

- Lemma: \[ | \text{constants e} | \leq \text{size e} \]
- Proof: ...
  case e is 0, true, false:
  \[ | \text{constants e}_i | = | \{e\} | \]
  \[ = 1 \]
  \[ = \text{size e} \]

  (by def of constants)
  (simple calculation)
  (by def of size)

A Lemma

- Lemma: \[ | \text{constants e} | \leq \text{size e} \]
- Proof: ...
  case e is pred e':
  \[ | \text{constants e} | = | \text{constants e'} | \]
  \[ \leq \text{size e'} \]
  \[ < \text{size e} \]

  (def of constants)
  (IH)
  (by def of size)

A Lemma

- Lemma: \[ | \text{constants e} | \leq \text{size e} \]
- Proof: ...
  case e is (if e1 then e2 else e3):
  \[ | \text{constants e} | = \bigcup_{i=1,3} | \text{constants e}_i | \]
  \[ \leq \sum_{i=1,3} | \text{constants e}_i | \]
  \[ \leq \sum_{i=1,3} \text{size e}_i \]
  \[ < \text{size e} \]

  (def of constants)
  (property of sets)
  (IH on each ei)
  (def of size)
What is a proof?

- A proof is an easily-checked justification of a judgment (i.e. 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:
  - \( t ::= \text{int} | \text{bool} | t \to t \)

Binding and Scope

- In the expression \( \text{fun } f (x : t_1) : t_2 = e \) 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
    - \( \text{fun } f (x : \text{int}) : \text{int} = x + 3 \) same as \( \text{fun } g (x : \text{int}) : \text{int} = x + 3 \)
  - We’ll pick variables \( f \) and \( x \) to avoid clashes with other variables in context.

Abstract Syntax of MinML

- The expressions of MinML are inductively defined by these rules:
  - \( e ::= x | n | \text{true} | \text{false} | o(e,...,e) | \text{if } e \text{ then } e \text{ else } e | \text{fun } f (x : t) : e = e | e e \)
  - \( x \) 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: \( 2 + x \)
Free Variables and Substitution

- Variables that are not bound are called free.
  - eg: y is free in \( \text{fun } (x : t_1) : t_2 = f_y \)
- The capture-avoiding substitution \( e[e'/x] \) replaces all free occurrences of \( x \) in \( e \).
  - eg: \( \text{fun } (x : t_1) : t_2 = f_y)(3/y) = (\text{fun } (x : t_1) : t_2 = f_3) \)
- Rename bound variables during substitution to avoid “capturing” free variables.
  - eg: \( (\text{fun } (x : t_1) : t_2 = f_y)(x/y) = (\text{fun } (x : t_1) : t_2 = f_x) \)

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 \( G \vdash e : t \)
  - 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 \( G : \text{Variables} \rightarrow \text{Types} \)
- We write \( G, x : t \) for the function \( G' \) defined as follows:
  \[ G'(y) = t \quad \text{if } x = y \]
  \[ G'(y) = G(y) \quad \text{if } x \neq y \]

Typing Rules

- A variable has whatever type \( G \) assigns to it:
  \[ G \vdash x : G(x) \]
- The constants have the evident types:
  \[ G \vdash \text{true} : \text{bool} \quad G \vdash \text{false} : \text{bool} \]

Typing Rules

- The primitive operations have the expected typing rules:
  \[
  \frac{G \vdash e_1 : \text{int}\quad G \vdash e_2 : \text{int}}{G \vdash \text{+}(e_1,e_2) : \text{int}}
  \]
  \[
  \frac{G \vdash e_1 : \text{int}\quad G \vdash e_2 : \text{int}}{G \vdash \text{--}(e_1,e_2) : \text{bool}}
  \]
Typing Rules

- Both “branches” of a conditional must have the same type!
  \[ G \vdash e : \text{bool} \quad G \vdash e_1 : t \quad G \vdash e_2 : t \]
  \[ G \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : t \]
- 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:
  \[ G \vdash e_1 : t_2 \rightarrow t \quad G \vdash e_2 : t_2 \]
  \[ G \vdash e_1 \ e_2 : t \]
- The result type of the co-domain (range) of the function.

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.

Typing Rules

- Type checking recursive function:
  \[ G, f : t_1 \rightarrow t_2, x : t_1 \vdash e : t_2 \]
  \[ G \vdash \text{fun } f(x : t_1) : t_2 = e : t_1 \rightarrow t_2 \]
- We tacitly assume that \( \{f,x\} \cap \text{dom}(G) = \{\} \). This is always possible by our conventions on binding operators.

Well-Typed and Ill-Typed Expressions

- An expression \( e \) is well-typed in a context \( G \) iff there exists a type \( t \) such that \( G \vdash e : t \).
- If there is no \( t \) such that \( G \vdash e : t \), then \( e \) is ill-typed in context \( G \).

Typing Example

- Consider the following expression \( e \):
  \[
  \text{fun } f(n : \text{int}) : \text{int} = \\
  \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n-1)
  \]
- Lemma: The expression \( e \) has type \( \text{int} \rightarrow \text{int} \).
  To prove this, we must show that
  \[ \{\} \vdash e : \text{int} \rightarrow \text{int} \]
Typing Example

\[
\{ \} \vdash \text{fun } f \left( \text{n : int} \right) \text{ : int} = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \rightarrow \text{int}
\]

Typing Example

\[
\begin{align*}
&G \vdash n = 0 \text{ : bool} & G \vdash 1 \text{ : int} & G \vdash n \cdot f \left( n - 1 \right) \text{ : int} \\
&G \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \\
&\{ \} \vdash \text{fun } f \left( \text{n : int} \right) \text{ : int} = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \rightarrow \text{int}
\end{align*}
\]

Typing Example

\[
\begin{align*}
&G \vdash n = 0 \text{ : int} & G \vdash 0 \text{ : int} & G \vdash 1 \text{ : int} & G \vdash n \cdot f \left( n - 1 \right) \text{ : int} \\
&G \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \\
&\{ \} \vdash \text{fun } f \left( \text{n : int} \right) \text{ : int} = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \rightarrow \text{int}
\end{align*}
\]

Typing Example

\[
\text{Derivation } D = \\
\begin{align*}
&G \vdash n \text{ : int} & G \vdash 1 \text{ : int} & G \vdash n \cdot f \left( n - 1 \right) \text{ : int} \\
&G \vdash \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \\
&\{ \} \vdash \text{fun } f \left( \text{n : int} \right) \text{ : int} = \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot f \left( n - 1 \right) \text{ : int} \rightarrow \text{int}
\end{align*}
\]

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 \( G \vdash e : t \) 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.

Summary of Static Semantics

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

Induction on Typing

- To show that some property \( \text{P}(G, e, t) \) holds whenever \( G \vdash e : t \), it’s enough to show the property holds for the conclusion of each rule given that it holds for the premises:
  - \( \text{P}(G, x, \text{G}(x)) \)
  - \( \text{P}(G, n, \text{int}) \)
  - \( \text{P}(G, \text{true}, \text{bool}) \) and \( \text{P}(G, \text{false}, \text{bool}) \)
  - if \( \text{P}(G, e, \text{bool}), \text{P}(G, e1, t) \) and \( \text{P}(G, e2, t) \) then \( \text{P}(G, \text{if } e \text{ then } e1 \text{ else } e2) \)
  - and similarly for functions and applications...

Properties of Typing

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

Properties of Typing

- **Lemma (Weakening):**
  If \( G \vdash e : t \) and \( G' \subseteq G \), then \( G' \vdash e : t \).
- **Proof**: by induction on typing
- Intuitively, “junk” in the context doesn’t matter.
Properties of Typing

- Lemma (Substitution):
  If $G, x : t' \vdash e' : t'$ and $G \vdash e : t$, then
  $G \vdash e'[e/x] : t'$.
- Proof: ?

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
    - ($P) \Rightarrow (Q)$: Hoare logic
    - Operational: Define execution directly by rewriting the program step-by-step
- We'll concentrate on the operational approach

MinML

Dynamic Semantics

Dynamic Semantics of MinML

- Judgment: $e \rightarrow e'$
  - A transition relation read: “$e$ 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:
  - $v ::= x \mid n \mid true \mid false \mid \text{fun}\ f \ (x : t_1) : t_2 = e$
- 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.

\[
\begin{align*}
(n = n_1 + n_2) & \implies (n_1, n_2) \rightarrow n \\
(n_1 \neq n_2) & \implies (n_1, n_2) \rightarrow \text{false}
\end{align*}
\]

Primitive Instructions

- Conditional branch:

\[
\begin{align*}
\text{if true then } e_1 \text{ else } e_2 & \rightarrow e_1 \\
\text{if false then } e_1 \text{ else } e_2 & \rightarrow e_2
\end{align*}
\]

Primitive Instructions

- Application of a recursive function:

\[
\begin{align*}
(v \text{ = fun } f(x : t1) : t2 \rightarrow e) \\
& \forall v \rightarrow e[v/f] [v1/x]
\end{align*}
\]

\[v1 \rightarrow e[v1/f] [v1/x]\]

- 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

\[
e_1 \rightarrow e'_1 \\
\tau(e_1, e_2) \rightarrow \tau(e'_1, e_2)
\]

\[
e_2 \rightarrow e'_2 \\
\tau(v_1, e_2) \rightarrow \tau(v_1, e'_2)
\]
Search Rules

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

\[
\text{if } e \text{ then } e_1 \text{ else } e_2 \rightarrow \text{if } e' \text{ then } e_1 \text{ else } e_2
\]

Multi-step Evaluation

• The relation \( e \rightarrow^* e' \) is inductively defined by the following rules:

\[
\begin{align*}
\frac{\text{ } & }{e \rightarrow^* e} \\
\frac{e \rightarrow e' \quad e' \rightarrow^* e''}{e \rightarrow^* e''}
\end{align*}
\]

• That is, \( e \rightarrow^* e' \) iff

\[
e = e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n = e' \text{ for some } n \geq 0.
\]

Example Execution

\[
v 3 \rightarrow \text{if } 3=0 \text{ then } 1 \text{ else } 3*v(3-1) \\
\rightarrow \text{if false then } 1 \text{ else } 3*v(3-1) \\
\rightarrow 3*v(3-1) \\
\rightarrow 3*v 2 \\
\rightarrow 3*v(2*0) \text{ if } 2=0 \text{ then } 1 \text{ else } 2*v(2-1) \\
\ldots \\
\rightarrow 3*(2*1) \\
\rightarrow 3*2 \\
\rightarrow 6
\]

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

Search Rules

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

\[
\begin{align*}
e_1 \rightarrow e_1' \\
e_1 e_2 \rightarrow e_1' e_2 \\
e_2 \rightarrow e_2'
\end{align*}
\]

Example Execution

• Suppose that \( v \) is the function

\[
\text{fun f (n:int) :int = if n=0 then 1 else n*f(n-1)}
\]

• Consider its evaluation:

\[
v 3 \rightarrow \text{if } 3=0 \text{ then } 1 \text{ else } 3*v(3-1)
\]

• We have substituted 3 for \( n \) and \( v \) for \( f \) in the body of the function.

Example Execution

\[
v 3 \rightarrow \text{if } 3=0 \text{ then } 1 \text{ else } 3*v(3-1) \\
\rightarrow \text{if false then } 1 \text{ else } 3*v(3-1) \\
\rightarrow 3*v(3-1) \\
\rightarrow 3*v 2 \\
\rightarrow 3*v(2*0) \text{ if } 2=0 \text{ then } 1 \text{ else } 2*v(2-1) \\
\ldots \\
\rightarrow 3*(2*1) \\
\rightarrow 3*2 \\
\rightarrow 6
\]

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

Induction on Evaluation

• To prove that \( e \rightarrow e' \) 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 $e \rightarrow^* e'$ implies $Q(e, e')$ it suffices to show
  - $Q(e, e)$ (Q is reflexive)
  - If $e \rightarrow e'$ 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 $v \rightarrow e$.
- 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 $e \rightarrow e'$.
- By induction on the structure of $e$
  - Make use irreducibility of values
  - eg: application rules
    - $e_1 \rightarrow e_1'$
    - $e_2 \rightarrow e_2'$
    - $v \rightarrow v_1 \rightarrow v_1 e_2 \rightarrow \ldots \\
    - $(v = \text{fun } f (x :: t1) : t2 = e)$
    - $v \rightarrow v_1 \rightarrow e[v_1[x] / x]$

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 $e \rightarrow^* v$.
- By induction on the length of the evaluation sequence using determinacy.

Stuck States

- Not every irreducible expression is a value!
  - $(\text{if } ? \text{ then } 1 \text{ else } 2)$ does not reduce
  - $(\text{true} \cdot \text{false})$ does not reduce
  - $(\text{true} \cdot 1)$ 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 $e \rightarrow e'$ 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
  - $e \rightarrow e'$
- Some people like “big-step” operational semantics
  - $e \Downarrow v$
- 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 $e_1$ else $e_2$
  | $E \ e \ \ve E$


textbox-contents: CONTEXT-BASED SEMANTICS

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 $e_1$ else $e_2$
  | if false then $e_1$ else $e_2$
  | $(\text{fun } f(x:1):x2 = e) \ v$


textbox-contents: CONTEXT-BASED SEMANTICS

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


textbox-contents: CONTEXT-BASED SEMANTICS

Summary of Dynamic Semantics

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


textbox-contents: SUMMARY OF DYNAMIC SEMANTICS

MinML

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?


textbox-contents: MINML

Type Safety
Type Safety

- A type system predicts at compile time the behavior of a program at run time.
  - eg: \( \texttt{e : int \rightarrow int} \) 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

- 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 the bad ones.

Type Safety

- Fundamentally there is a tension between
  - the expressiveness 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 François’ lectures for one example.
Formalization of Type Safety

- The coherence of the static and dynamic semantics is nearly 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

- The type of a closed value determines its form.
- **Canonical Forms Lemma**: If \( \vdash v : t \) then
  - If \( t = \text{int} \) then \( v = n \) for some integer \( n \)
  - If \( t = \text{bool} \) then \( v = \text{true} \) or \( v = \text{false} \)
  - If \( t = f : \tau \rightarrow t_1 \) then \( v = \text{fun } f(x : t_1) : t_2 = e \) for some \( f, x, \) and \( e \).
- Proof by induction on typing rules.
- eg: If \( \vdash e : \text{int} \) and \( e \rightarrow^* v \) then \( v = n \) for some integer \( n \).

Proof of Preservation

- **Theorem (Preservation)**
  \[
  \text{If } \vdash e : t \text{ and } e \rightarrow e' \text{ then } \vdash e' : t.
  \]
- **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**:
  \[
  \frac{(n = n_1 + n_2)}{(n, n_1, n_2) \rightarrow n} \vdash (n_1, n_2) : t
  \]
  **Proof**:
  \[
  t = \text{int} \quad \text{(by inversion lemma)}
  \]
Proof of Preservation

• Case addition:
  Given:
  \[(n = n1 + n2) \vdash (n1, n2) : t\]
  Proof:
  \[t = \text{int} \quad \text{(by inversion lemma)}\]
  \[\vdash n : \text{int} \quad \text{(by typing rule for ints)}\]

Proof of Preservation

• Case application:
  Given:
  \[(v = \text{fun } f \,(x : t1) : t2 = e) \vdash v : t1 \rightarrow t2; \vdash v1 : t1; \; t = t2 \quad \text{(by inversion)}\]

Proof of Preservation

• Case application:
  Given:
  \[(v = \text{fun } f \,(x : t1) : t2 = e) \vdash v : t1 \rightarrow e[v/f] [v1/x] \quad \text{(by inversion)}\]
  \[f : t1 \rightarrow t2, x:t1 \vdash e : t2 \quad \text{(by inversion)}\]

Proof of Preservation

• Case application:
  Given:
  \[(v = \text{fun } f \,(x : t1) : t2 = e) \vdash v : t1 \rightarrow e[v/f] [v1/x] \quad \text{(by inversion)}\]
  \[f : t1 \rightarrow t2, x:t1 \vdash e : t2 \quad \text{(by inversion)}\]

Proof of Preservation

• Case addition search1:
  Given:
  \[e1 \rightarrow e1' \quad \tau(e1, e2) \rightarrow \tau(e1', e2) \quad \vdash \tau(e1, e2) : t\]
  Proof:
Proof of Preservation

- Case addition search 1:
  Given:
  \[
  \frac{e_1 \rightarrow e_1'}{\tau(e_1, e_2) \rightarrow \tau(e_1', e_2)} \quad \vdash \tau(e_1, e_2) : t
  \]
  Proof:
  \[
  \vdash e_1 : \text{int} \quad \text{(by inversion)}
  \]

- Case addition search 1:
  Given:
  \[
  \frac{e_1 \rightarrow e_1'}{\tau(e_1, e_2) \rightarrow \tau(e_1', e_2)} \quad \vdash \tau(e_1, e_2) : t
  \]
  Proof:
  \[
  \vdash e_1 : \text{int} \quad \text{(by inversion)}
  \]
  \[
  \vdash e_1' : \text{int} \quad \text{(by induction)}
  \]
  \[
  \vdash e_2 : \text{int} \quad \text{(by inversion)}
  \]
  \[
  \vdash \tau(e_1', e_2) : \text{int} \quad \text{(by typing rule for +)}
  \]

Proof of Preservation

- How might the proof have failed?
- Only if some instruction is mis-defined. eg:
  \[
  \frac{(m = n)}{(m, n) \rightarrow t} \quad \frac{(m \neq n)}{(m, n) \rightarrow 0}
  \]
  \[
  G \vdash e_1 : \text{int} \quad G \vdash e_2 : \text{int}
  \]
  \[
  G \vdash \neg(e_1, e_2) : \text{bool}
  \]
- 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!
  \[
  \frac{(m = n)}{(m, n) \rightarrow \text{true}}
  \]
  \[
  G \vdash e_1 : \text{int} \quad G \vdash e_2 : \text{int}
  \]
  \[
  G \vdash \neg(e_1, e_2) : \text{bool}
  \]
Proof of Progress

- Theorem (Progress)
  If \( \vdash e : t \) then either \( e \) is a value or there exists \( e' \) such that \( e \rightarrow e' \).
- Proof is by induction on typing.

Proof of Progress

- Case variables:
  Given:
  \( G \vdash x : G(x) \)
  Proof: This case does not apply since we are considering closed values (\( G \) is the empty context).

Proof of Progress

- Case integer:
  Given:
  \( \vdash n : \text{int} \)
  Proof: Immediate (\( n \) is a value). Similar reasoning for all other values.

Proof of Progress

- Case addition:
  Given:
  \[
  \begin{align*}
  \vdash e_1 : \text{int} & \quad \vdash e_2 : \text{int} \\
  \vdash+ (e_1, e_2) : \text{int}
  \end{align*}
  \]
  Proof:
  (1) \( e_1 \rightarrow e_1' \), or (2) \( e_1 = v_1 \) (by induction)

Proof of Progress

- Case addition:
  Given:
  \[
  \begin{align*}
  \vdash e_1 : \text{int} & \quad \vdash e_2 : \text{int} \\
  \vdash+ (e_1, e_2) : \text{int}
  \end{align*}
  \]
  Proof:
  (1) \( e_1 \rightarrow e_1' \), or (2) \( e_1 = v_1 \) (by induction)
  \( + (e_1, e_2) \rightarrow + (e_1', e_2) \) (by search rule, if 1)
Proof of Progress

• Case addition:
  Given:
  \[ \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \]
  \[ \vdash +(e_1,e_2) : \text{int} \]
  Proof:
  Assuming (2) \( e_1 = v_1 \) (we’ve taken care of 1)
  (3) \( e_2 \to e_2' \), or (4) \( e_2 = v_2 \) (by induction)
  \[ +(v_1,e_2) \to +(v_1,e_2') \] (by search rule, if 3)

Proof of Progress

• Case addition:
  Given:
  \[ \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \]
  \[ \vdash +(e_1,e_2) : \text{int} \]
  Proof:
  Assuming (2) \( e_1 = v_1 \) (we’ve taken care of 1)
  Assuming (4) \( e_2 = v_2 \) (we’ve taken care of 3)
  \[ v_1 = n_1 \text{ for some integer } n_1 \] (by canonical forms)
  \[ v_2 = n_2 \text{ for some integer } n_1 \] (by canonical forms)
  \[ +(n_1,n_2) = n \text{ where } n \text{ is sum of } n_1 \text{ and } n_2 \] (by instruction rule)

Proof of Progress

• Case addition:
  Given:
  \[ \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \]
  \[ \vdash +(e_1,e_2) : \text{int} \]
  Proof:
  Assuming (2) \( e_1 = v_1 \) (we’ve taken care of 1)
  Assuming (4) \( e_2 = v_2 \) (we’ve taken care of 3)
  \[ v_1 = n_1 \text{ for some integer } n_1 \] (by canonical forms)
  \[ v_2 = n_2 \text{ for some integer } n_1 \] (by canonical forms)
  \[ +(n_1,n_2) = n \text{ where } n \text{ is sum of } n_1 \text{ and } n_2 \] (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
    \[
    \frac{m = n}{(m, n) \to \text{true}}
    \]
    \[
    \frac{G \vdash e_1 : \text{int}}{G \vdash \neg(e_1,e_2) : \text{bool}}
    \]
Extending the Language

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

\[
e_1 \rightarrow e_1' \\
[e_0,...,e_j,e_1,e_2,...,e_k] \rightarrow [e_0,...,e_j,e_1',e_2,...,e_k] \\
e_i \rightarrow e_i' \\
\text{sub ea ei} \rightarrow \text{sub ea' ei} \\
\text{sub va ei} \rightarrow \text{sub va' ei'} \\
0 <= n <= k \\
\text{sub } [e_0,...,e_k] n \rightarrow v_j
\]

G |- e0 : t _ _ G |- e k : t \\
G |- ea : t array  G |- ei : int \\
G |- [e_0,...,e_k] : t array  G |- sub ea ei : t

Extending the Language

• Is the language still safe?
  – Preservation still holds: execution of each instruction preserves types
  – Progress fails:
    \[ \vdash \text{sub } [17,25,44] 9 : \text{int} \]
    \[ \text{but} \]
    \[ \vdash \text{sub } [17,25,44] 9 : \text{int} \rightarrow ??? \]

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
    \[ n < 0 \text{ or } n > k \]
    \[ \text{sub [v0...vk]} n \rightarrow \text{error} \]
  - Search rules propagate errors once they arise
    \[ e_1 \rightarrow \text{error} \]
    \[ e_2 \rightarrow \text{error} \]
    \[ \text{+(e_1, e_2)} \rightarrow \text{error} \]
    (similarly with all other search rules)

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 defense 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:
  - 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.