(Advanced) topics in Programming Languages

Instructor: Mooly Sagiv

Dan David 203
http://www.cs.tau.ac.il/~msagiv/courses/apl17.html
Prerequisites

• Compilation course  or
• Programming Languages
Course Grade

• 10% Class notes
• 35% (Mainly Theoretical) Assignments
• 55% Home or (easier) class exam
Class Notes

• Prepared by two/three students
• First draft completed within one week
• Consumes a lot of time
• Use LaTeX template provided in the homepage
• Read supplementary material
• Correct course notes
  – Bonus for interesting corrections
• Add many more examples and elaborations
New PL in Industry

<table>
<thead>
<tr>
<th>PL</th>
<th>Company</th>
<th>Predecessor</th>
<th>Domain</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>Google</td>
<td>C</td>
<td>Systems</td>
<td>Channels Type Inference ML modules GC</td>
</tr>
<tr>
<td>Dart</td>
<td>Google</td>
<td>Javascript</td>
<td>Web</td>
<td>Types</td>
</tr>
<tr>
<td>Rust</td>
<td>Mozilla</td>
<td>C++</td>
<td>System Browser</td>
<td>Type Inference Ownership Linear Types</td>
</tr>
<tr>
<td>Hack</td>
<td>Facebook</td>
<td>PHP</td>
<td>Social Networks</td>
<td>Gradual Types</td>
</tr>
</tbody>
</table>
Course Themes

• Programming Language Concepts
  – A language is a “conceptual universe” (Perlis)
    • Framework for problem-solving
    • Useful concepts and programming methods
  – Understand the languages you use, by comparison
  – Appreciate history, diversity of ideas in programming
  – Be prepared for new programming methods, paradigms, tools

• Critical thought
  – Identify properties of language, not syntax or sales pitch

• Language and implementation
  – Every convenience has its cost
    • Recognize the cost of presenting an abstract view of machine
    • Understand trade-offs in programming language design
Language goals and trade-offs

Architect

Compiler, Runtime environment

Programmer

Testing

Programming Language

Diagnosti cTools
Instructor’s Background

• First programming language Pascal
• Soon switched to C (unix)
  • Efficient low level programming was the key
  • Small programs did amazing things
• Led an industrial project was written in common lisp
  • Semi-automatically port low level OS code between 16 and 32 bit architectures
• The programming setting has dramatically changed:
  • Object oriented
  • Garbage collection
  • Huge programs
  • Performance depends on many issues
  • Productivity is sometimes more importance than performance
  • Software reuse is a key
Other Lessons Learned

- Futuristic ideas may be useful problem-solving methods now, and may be part of languages you use in the future
  - Examples
    - Recursion
    - Object orientation
    - Garbage collection
    - High level concurrency support
    - Higher order functions
    - Pattern matching
More examples of practical use of futuristic ideas

- Function passing: pass functions in C by building your own closures, as in STL “function objects”
- Blocks are a nonstandard extension added by Apple to C that uses a lambda expression like syntax to create closures
- Continuations: used in web languages for workflow processing
- Monads: programming technique from functional programming
- Concurrency: atomicity instead of locking
- Decorators in Python to dynamically change the behavior of a function
What’s new in programming languages

• Commercial trend over past 5+ years
  – Increasing use of type-safe languages: Java, C#, ...
  – Scripting languages, other languages for web applications

• Teaching trends
  – Java replaced C as most common intro language
    • Less emphasis on how data, control represented in machine

• Research and development trends
  – Modularity
    • Java, C++: standardization of new module features
  – Program analysis
    • Automated error detection, programming env, compilation
  – Isolation and security
    • Sandboxing, language-based security, ...
  – Web 2.0
    • Increasing client-side functionality, mashup isolation problems
What’s worth studying?

• Dominant languages and paradigms
  – Leading languages for general systems programming
  – Explosion of programming technologies for the web
• Important implementation ideas
• Performance challenges
  – Concurrency
• Design tradeoffs
• Concepts that research community is exploring for new programming languages and tools
• Formal methods in practice
  • Grammars
  • Semantics
  • Domain theory
  • ...


Suggested Reading

- J. Mitchell. Concepts in Programming Languages
- B. Pierce. Types and Programming Languages
- J. Mitchell. Foundations for Programming Languages
- Glynn Winskel. Formal Semantics of Programming Languages
- Peter J. Landin. The next 700 programming languages
- ...
Related Courses

• Compilers
• Programming languages
• Semantics of programming languages
• Program analysis
Syntax vs. Semantics

• The pattern of formation of sentences or phrases in a language

• Examples
  – Regular expressions
  – Context free grammars

• The study or science of meaning in language

• Examples
  – Interpreter
  – Compiler
  – Better mechanisms will be given in the course
Syntax vs. Semantics

Propositional Logic

• Syntax: Boolean formulas over variables
  – \(<Formula> ::= \text{Variable} | <Formula> \text{‘|’} <Formula> | <Formula> \text{‘&’} <Formula> | ‘!’ <Formula> | ‘(‘ <Formula> ‘)’

• Semantics:
Who need formal semantics for PL?

- Language designers
- Compiler designers
- Programmers
Example C++

- Designed with a source to source compiler to C
- Many issues
  - Especially later
Type Safety

• A programming language is type safe if every well typed program has no undefined semantics
• No runtime surprise
• Is C type safe?
• How about Java?
void foo(s) {
  char c[100];
  strcpy(c, s);
}


A Pathological C Program

```c
a = malloc(...) ;
b = a;
free (a);
c = malloc (...);
if (b == c) printf(“unexpected equality”);
```
Another Pathological C Program

union {
    int x;
    int *p;
} mixed;
mixed.x = 1700;
free(mixed.p);
Alternative Formal Semantics

• Operational Semantics
  – The meaning of the program is described “operationally”
  – Natural Operational Semantics
  – Structural Operational Semantics

• Denotational Semantics
  – The meaning of the program is an input/output relation
  – Mathematically challenging but complicated

• Axiomatic Semantics
  – The meaning of the program are observed properties
Alternate Semantics

• What is the meaning of a program?
• “int f(int x) { int y=x; return y + 1 ;}“
• Operational
• Denotational
• Axiomatic
Semantics of Arithmetic Expressions

• Syntax
• Semantics
States

• Mapping of variables into values
• Var → Z
  – Can be partial
• Examples [x→1, y→7]
Example State Manipulations

• $[x \rightarrow 1, y \rightarrow 7] \ y =$
• $[x \rightarrow 1, y \rightarrow 7] \ t =$
Semantics of arithmetic expressions

• Assume that arithmetic expressions are side-effect free

• \( A[e] : \text{State} \rightarrow Z \)

• Defined by \textit{structural} induction on the syntax tree
  
  – \( A[e_{1} + e_{2}] s = A[e_{1}] s + A[e_{2}] s \)
  
  – \( A[e_{1} * e_{2}] s = A[e_{1}] s * A[e_{2}] s \)
  
  – \( A[(e_{1})] s = A[e_{1}] s \quad \text{--- not needed} \)
  
  – \( A[-e_{1}] s = -A[e_{1}] s \)
Properties of arithmetic expressions

• The semantics is **compositional**
  – $A[e_1 \text{ op } e_2] = \llbracket \text{ op } \rrbracket (A[e_1], A[e_2])$
  – Properties can be proved by structural induction

• **We say that** $e_1$ **is semantically equivalent** to $e_2$ ($e_1 \approx e_2$) **when**
  $A[e_1] = A[e_2]$
Commutativity of expressions

• Theorem: for every expressions $e_1, e_2$: $e_1 + e_2 \approx e_2 + e_1$

• Proof:
  
  \[
  \]
Program Analysis

- Automatically infer properties of programs
- Identify potential bugs
- Prove the absence of errors
A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program.

The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units.

The Yorktown was dead in the water for about two hours and 45 minutes.
A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program.

The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units.

The Yorktown was dead in the water for about two hours and 45 minutes.

Numeric static analysis can detect these errors when the ship is built!
\( x = 3; \)
\( y = 1/(x-3); \)

\( x = 3; \)
\( px = \&x; \)
\( y = 1/(px-3); \)

\[
\text{for (x = 5; x < y ; x++) { }
\text{y = 1/ (z - x) }}
\]

\text{need to track values other than 0}
\text{need to track pointers}
\text{Need to reason about loops}
Dynamic Allocation (Heap)

```c
x = 3;
p = (int*)malloc(sizeof(int));
*p = x;
q = p;
y = 1/(*q-3);
```

need to track heap-allocated storage
Why is Program Analysis Difficult?

• Undecidability
  – Checking if program point is reachable
    • The Halting Problem
  – Checking interesting program properties
    • Rice Theorem
  – Can the computer really perform inductive reasoning?
Why is Program Analysis Difficult?

• Complicated programming languages
  – Large/unbounded base types: int, float, string
  – Pointers/aliasing + unbounded #’s of heap-allocated cells
  – User-defined types/classes
  – Loops with unbounded number of iterations
  – Procedure calls/recursion/calls through pointers/dynamic method lookup/overloading
  – Concurrency + unbounded #’s of threads

• Conceptual
  – Which program to analyze?
  – Which properties to check?

• Scalability
Sidestepping Undecidability

Universe of States

Reachable States

Bad States
Sidestepping Undecidability

[Cousot & Cousot POPL77-79]

Overapproximate the reachable states

Universe of States

Reachable States

Bad States

False alarms
Abstract Interpretation

\[ x > 0 \]

\[ y := -2 \]

\[ y := -x \]
Infer Inductive Invariants via AI

```
x := 2;
y := 0;
x := x + y;
y := y + 1;
```
Infer Inductive Invariants via AI

\[
x := 2;
\]
\[
y := 0;
\]
\[
x := x + y;
\]
\[
y := y + 1;
\]
Infer Inductive Invariants via AI

x := 2;
y := 0;
x := x + y;
y := y + 1;
Infer Inductive Invariants via AI

\[ x := 2; \]
\[ y := 0; \]
\[ x := x + y; \]
\[ y := y + 1; \]
Infer Inductive Invariants via AI

\[
x := 2; \\
y := 0; \\
x := x + y; \\
y := y + 1;
\]
Infer Inductive Invariants via AI

\[\begin{align*}
x &:= 2; \\
y &:= 0; \\
x &:= x + y; \\
y &:= y + 1;
\end{align*}\]
AI Infers Inductive Invariants

\[ x := 2; \ y := 0; \]
\[ \text{while true do} \]
\[ \quad \text{assert } x > 0; \]
\[ \quad x := x + y; \]
\[ \quad y := y + 1 \]

Non-inductive

Inductive

\[ x \geq 1 \& y \geq 0 \]
Original Problem: **Shape Analysis**  
(Jones and Muchnick 1981)

- Characterize dynamically allocated data  
  - $x$ points to an acyclic list, cyclic list, tree, dag, etc.  
  - show that data-structure invariants hold
- Identify may-alias relationships
- Establish “disjointedness” properties  
  - $x$ and $y$ point to structures that do not share cells
- **Memory Safety**  
  - No null and dangling de-references  
  - No memory leaks
- In OO programming  
  - Everything is in the heap \( \Rightarrow \) requires shape analysis
Why Bother?

int *p, *q;
q = (int *) malloc();
p = q;
l1: *p = 5;
p = (int *) malloc();
l2: printf(*q); /* printf(5) */
Example: Concrete Interpretation

- `x = NULL`
- `t = malloc(..);`
- `t->next = x;`
- `x = t`
- `return x`
Example: Abstract Interpretation

```
x = NULL
```

```
F  T
```

```
t = malloc(..);
```

```
t -> next = x;
```

```
x = t
```

```
return x
```
Memory Leakage

List reverse(Element *head)
{
    List rev, ne;
    rev = NULL;
    while (head != NULL) {
        ne = head -> next;
        head -> next = rev;
        head = ne;
    }
    return rev;
}
Memory Leakage
Element* reverse(Element *head)
{
    Element *rev, *ne;
    rev = NULL;
    while (head != NULL) {
        ne = head -> next;
        head -> next = rev;
        rev = head;
        head = ne;
    }
    return rev;
}
No memory leaks
Mark and Sweep

\begin{align*}
\text{void Mark}(\text{Node root}) \{ \\
&\quad \text{if (root \neq \text{NULL})} \{ \\
&\quad\quad \text{pending} = \emptyset \\
&\quad\quad \text{pending} = \text{pending} \cup \{\text{root}\} \\
&\quad\quad \text{marked} = \emptyset \\
&\quad\quad \text{while (pending \neq \emptyset)} \{ \\
&\quad\quad\quad \text{x} = \text{SelectAndRemove}(\text{pending}) \\
&\quad\quad\quad \text{marked} = \text{marked} \cup \{\text{x}\} \\
&\quad\quad\quad \text{t} = \text{x} \rightarrow \text{left} \\
&\quad\quad\quad \text{if (t \neq \text{NULL})} \\
&\quad\quad\quad\quad \text{if (t \not\in \text{marked})} \\
&\quad\quad\quad\quad\quad \text{pending} = \text{pending} \cup \{\text{t}\} \\
&\quad\quad\quad \text{t} = \text{x} \rightarrow \text{right} \\
&\quad\quad\quad \text{if (t \neq \text{NULL})} \\
&\quad\quad\quad\quad \text{if (t \not\in \text{marked})} \\
&\quad\quad\quad\quad\quad \text{pending} = \text{pending} \cup \{\text{t}\} \\
&\quad\quad \} \\
&\quad \} \\
&\quad \text{assert(\text{marked} = = \text{Reachset}(\text{root}))} \\
&\} \\
&\forall v: \text{marked}(v) \iff \text{reach[root]}(v)
\end{align*}

\begin{align*}
\text{void Sweep}() \{ \\
&\quad \text{unexplored} = \text{Universe} \\
&\quad \text{collected} = \emptyset \\
&\quad \text{while (unexplored \neq \emptyset)} \{ \\
&\quad\quad \text{x} = \text{SelectAndRemove}(\text{unexplored}) \\
&\quad\quad \text{if (x \not\in \text{marked})} \\
&\quad\quad\quad \text{collected} = \text{collected} \cup \{\text{x}\} \\
&\quad\quad \} \\
&\quad \text{assert(\text{collected} = =} \\
&\quad\quad \text{\text{Universe} \setminus \text{Reachset}(\text{root})} \\
&\quad \) \\
&\} \\
\end{align*}
Example: Mark

```c
void Mark(Node root) {
    if (root != NULL) {
        pending = \emptyset;
        pending = pending \cup \{root\};
        marked = \emptyset;
        while (pending \neq \emptyset) {
            x = SelectAndRemove(pending);
            marked = marked \cup \{x\};
            t = x \rightarrow left;
            if (t \neq NULL) {
                if (t \not\in marked) {
                    pending = pending \cup \{t\};
                }
            /*
                t = x \rightarrow right
                if (t \neq NULL) {
                    if (t \not\in marked) {
                        pending = pending \cup \{t\};
                    }
                */
            }
        }
        assert(marked == Reachset(root));
    }
}```
Bug Found

• There may exist an individual that is reachable from the root, but not marked
## Properties Proved

<table>
<thead>
<tr>
<th>Program</th>
<th>Properties</th>
<th>#Graphs</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>LindstromScan</td>
<td>CL, DI</td>
<td>1285</td>
<td>8.2</td>
</tr>
<tr>
<td>LindstromScan</td>
<td>CL, DI, IS, TE</td>
<td>183564</td>
<td>2185</td>
</tr>
<tr>
<td>SetRemove</td>
<td>CL, DI, SO</td>
<td>13180</td>
<td>106</td>
</tr>
<tr>
<td>SetInsert</td>
<td>CL, DI, SO</td>
<td>299</td>
<td>1.75</td>
</tr>
<tr>
<td>DeleteSortedTree</td>
<td>CL, DI</td>
<td>2429</td>
<td>6.24</td>
</tr>
<tr>
<td>DeleteSortedTree</td>
<td>CL, DI, SO</td>
<td>30754</td>
<td>104</td>
</tr>
<tr>
<td>InsertSortedTree</td>
<td>CL, DI</td>
<td>177</td>
<td>0.85</td>
</tr>
<tr>
<td>InsertSortedTree</td>
<td>CL, DI, SO</td>
<td>1103</td>
<td>2.5</td>
</tr>
<tr>
<td>InsertAVLttree</td>
<td>CL, DI, SO</td>
<td>1855</td>
<td>27.4</td>
</tr>
<tr>
<td>RecQuickSot</td>
<td>CL, DI, SO</td>
<td>5585</td>
<td>9.2</td>
</tr>
</tbody>
</table>

CL=memory safety       DI=data structure invariant       TE=termination       SO=sorted
Success Story: The SLAM/SDV Project MSR

- Tool for finding possible bugs in Windows device drivers
- Complicated back-out protocols in driver APIs when events cancelled or interrupted

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability."

Bill Gates, April 18, 2002. [Keynote address at WinHec 2002](http://www.winhe.com/)

Type Checking

Benjamin Pierce. Types and Programming Languages
August 2005

As a Malaysia Airlines jetliner cruised from Perth, Australia, to Kuala Lumpur, Malaysia, one evening last August, it suddenly took on a mind of its own and zoomed 3,000 feet upward. The captain disconnected the autopilot and pointed the Boeing 777's nose down to avoid stalling, but was jerked into a steep dive. He throttled back sharply on both engines, trying to slow the plane. Instead, the jet raced into another climb. The crew eventually regained control and manually flew their 177 passengers safely back to Australia.

Investigators quickly discovered the reason for the plane's roller-coaster ride 38,000 feet above the Indian Ocean. A defective software program had provided incorrect data about the aircraft's speed and acceleration, confusing flight computers.
Error Detection

• Early error detection
  – Logical errors
  – Interface errors
  – Dimension analysis
  – Effectiveness also depends on the programmer
  – Can be used for code maintenance
Type Systems

• A tractable syntactic method for proving absence of certain program behaviors by classifying phrases according to the kinds they compute

• Examples
  – Whenever f is called, its argument must be integer
  – The arguments of f are not aliased
  – The types of dimensions must match
  – ...
What is a type

• A denotation of set of values
  – Int
  – Bool
  – ...
• A set of legal operations
Static Type Checking

• Performed at compile-time
• Conservative (sound but incomplete)
  – if <complex test> then 5 else <type error>
• Usually limited to simple properties
  – Prevents runtime errors
  – Enforce modularity
  – Protects user-defined abstractions
  – Allows tractable analysis
    • But worst case complexity can be high
• Properties beyond scope (usually)
  – Array out of bound
  – Division by zero
  – Non null reference
Abstraction

• Types define interface between different software components
• Enforces disciplined programming
• Ease software integration
• Other abstractions exist
Documentation

- Types are useful for reading programs
- Can be used by language tools
Language Safety

• A safe programming language protects its own abstraction
• Can be achieved by type safety
• Type safety for Java was formally proven
  – Recent bug in the soundness of generics
## Statically vs. Dynamically Checked Languages

<table>
<thead>
<tr>
<th>Statically Checked</th>
<th>Dynamically Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Safe</strong></td>
<td>ML, Haskel, Java, C#</td>
</tr>
<tr>
<td><strong>Unsafe</strong></td>
<td>C, C++</td>
</tr>
</tbody>
</table>
Efficiency

• Compilers can use types to optimize computations
• Pointer scope
• Region inference
Language Design

• Design the programming language with the type system
• But types incur some notational overhead
• Implicit vs. explicit types
  – The annotation overhead
• Designing libraries is challenging
  – Generics/Polymorphism help
Untyped Arithmetic Expressions

Chapter 3
Untyped Arithmetic Expressions

t ::= terms
   true constant true
   false constant false
   if t then t else t conditional
   0 constant zero
   succ t successor
   pred t predecessor
   iszero t zero test

if false then 0 else 1 1
iszero (pred (succ 0)) true
Untyped Arithmetic Expressions

t ::= terms
    true constant true
    false constant false
    if t then t else t conditional
    0 constant zero
    succ t successor
    pred t predecessor
    iszero t zero test
    succ true type error
    if 0 then 0 else 0 type error
Structural Operational Semantics (SOS)

- The mathematical meaning of programs
- A high level definition of interpreter
- Allow inductively proving program properties
- A binary relation on terms
  - \( t \rightarrow t' \)
    - One step of executing \( t \) may yield the value \( t' \)
- Inductive definitions of \( \rightarrow \)
  - Axioms
  - Inference rules
- The meaning of a program is a set of trees
- The actual interpreter can be automatically derived
SOS Axioms
for the Simple Language

\[ t ::= \]

- \text{true} \quad \text{constant true}
- \text{false} \quad \text{constant false}
- \text{if } t \text{ then } t \text{ else } t \quad \text{conditional}
- 0 \quad \text{constant zero}
- \text{succ } t \quad \text{successor}
- \text{pred } t \quad \text{predecessor}
- \text{iszero } t \quad \text{zero test}
SOS Inference for the Simple Language

t ::= terms
   true constant true
   false constant false
   if t then t else t conditional
   0 constant zero
   succ t successor
   pred t predecessor
   iszero t zero test
SOS rules for Untyped Arithmetic Expressions

\[
\text{if true then } t_1 \text{ else } t_2 \rightarrow t_1 \quad (E\text{-IFTRUE})
\]

\[
\text{if false then } t_1 \text{ else } t_2 \rightarrow t_2 \quad (E\text{-IFFALSE})
\]

\[
t_1 \rightarrow t'_1
\]

(E-IF)

\[
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\]

\[
t_1 \rightarrow t'_1
\]

(E-SUCC)

\[
succ \ t_1 \rightarrow succ \ t'_1
\]

\[
t_1 \rightarrow t'_1
\]

(E-PRED)

\[
pred \ t_1 \rightarrow pred \ t'_1
\]

\[
pred \ 0 \rightarrow 0 \quad (E\text{-PREDZERO})
\]

\[
pred \ (succ \ t) \rightarrow t \quad (E\text{-PREDSUCC})
\]

\[
t_1 \rightarrow t'_1
\]

(E-ISZERO)

\[
iszero \ t_1 \rightarrow iszero \ t'_1
\]

\[
iszero \ 0 \rightarrow true \quad (E\text{-ISZERZEROZERO})
\]

\[
iszero \ succ \ t \rightarrow false \quad (E\text{-ISZERONZERO})
\]
Examples

if false then 0 else 1

iszero (pred (succ 0))

succ true

if 0 then 0 else 0

if iszero (succ true) then 0 else 1
Properties of the semantics

• Partial

• Determinism

\[ t_1 \rightarrow t_2 \land t_1 \rightarrow t_3 \implies t_2 = t_3 \]

• Reflexive transitive closure

\[ t \rightarrow^* t' \text{ if either } t = t' \text{ or there exists } t_0, t_1, \ldots, t_n \text{ such that } t = t_0, t' = t_n \text{ and for every } 0 \leq i < n: t_i \rightarrow t_{i+1} \]

• Semantic meaning

\[ \llbracket \cdot \rrbracket : \text{Terms} \rightarrow \text{Nat} \cup \text{Bool} \]

\[ \llbracket t \rrbracket = t' \text{ if } t' \in \text{Nat} \cup \text{Bool} \land t \rightarrow^* t' \]
Typed Arithmetic Expressions

Chapter 8
Well Typed Programs

• A set of type rules conservatively define well typed programs
• The typing relation is the smallest binary relation between terms and types
  – in terms of inclusion
• A term $t$ is typable (well typed) if there exists some type $T$ such that $t : T$
• The type checking problem is to determine for a given term $t$ and type $T$ if $t : T$
• The type inference problem is to infer for a given term $t$ a type $T$ such that $t : T$
Type Safety

• Stuck terms: Undefined Semantics
  – $\neg \exists t': t \rightarrow t'$

• The goal of the type system is to ensure at compile-time that no stuck ever occurs at runtime

• Type Safety (soundness)
  – **Progress**: A well-typed term t never gets stuck
    • Either it has value or there exists t’ such that t $\rightarrow$ t’
  – **Preservation**: (subject reduction)
    • If well type term takes a step in evaluation, then the resulting term is also well typed
Typed Arithmetic Expressions

t ::= terms
  true constant true
  false constant false
  if t then t else t conditional
  0 constant zero
  succ t successor
  pred t predecessor
  iszero t zero test

v ::= values
  true true value
  false false value
  nv numeric value

nv ::= numeric values
  0 zero value
  succ nv successor value
Type Rules for Booleans

\[ T ::= \]
\[ \text{Bool} \quad \text{type of Boolean} \]

\[ t : T \]

\[ \text{true} : \text{Bool} \ (T-\text{TRUE}) \]

\[ \text{false} : \text{Bool} \ (T-\text{FALSE}) \]

\[ \begin{array}{c}
  t_1 : \text{Bool} \\
  t_2 : T \\
  t_3 : T
\end{array} \]

\[ \frac{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}{(T-\text{IF})} \]
Type Rules for Numbers

\[ T ::= \text{types} \]
\[ \text{Nat} \quad \text{type of Natural numbers} \]

\[ 0 : \text{Nat (T-ZERO)} \]

\[ t_1 : \text{Nat} \quad \text{T-SUCC} \]
\[ \text{succ}(t_1) : \text{Nat} \]

\[ t_1 : \text{Nat} \quad \text{T-PRED} \]
\[ \text{pred}(t_1) : \text{Nat} \]

\[ t_1 : \text{Nat} \quad \text{T-ISZERO} \]
\[ \text{iszero}(t_1) : \text{Bool} \]
Type Rules for Arithmetic Expressions

true : Bool (T-TRUE)
false : Bool (T-FALSE)

\[ \begin{align*}
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \\
\end{align*} \] (T-IF)

0 : Nat (T-ZERO)

\[ \begin{align*}
\text{succ}(t_1) : Nat \\
\text{pred}(t_1) : Nat \\
iszero(t_1) : Bool
\end{align*} \] (T-SUCC, T-PRED, T-ISZERO)
Examples

if false then 0 else 1
if false then succ true else 1
if iszero 0 then 0 else 1
iszero (pred (succ 0))
succ true
if 0 then 0 else 0
if iszero (succ true) then 0 else 1
Uniqueness of Types

• Each term $t$ has at most one type
  – If $t$ is typable then
    • its type is unique
    • There is a unique type derivation tree for $t$

• Does not hold for general languages
  – Need a partial order on types
  – Unique most general type
Type Safety

**Progress**: If $t$ is well typed then either $t$ is a value or for some $t': t \rightarrow t'\$

**Preservation**: if $t : T$ and $t \rightarrow t'$ then $t' : T$

$nv ::= \text{numeric values}$

- $0$ zero value
- $\text{succ } nv$ successor value
Language Restrictions so far

- Simple expression language
- Fixed number of types
- No loops/recursion
- No variables/states
- No memory allocation
Summary Type Systems

- Type systems provide a useful mechanism for conservatively enforcing certain safety properties
  - Can be combined with runtime systems and static program analysis
- Interacts with the programmer
- A lot of interesting theory
- Another alternative is static program analysis
  - Infer abstractions of values at every program point
Tentative Schedule

25/10  Overview
1/11   Inductive Definitions & Haskel
8/11   Monads and Continuation
15/11  Operational Semantics
22,27/11 Denotational Semantics
6, 13/12 Deductive Verification
27/12, 3/1 Abstract Interpretation
10, 17/1 Type Checking/Type inference