Spring 2012

(Advanced) topics in Programming Languages

Instructor: Mooly Sagiv

TA: Shachar Itzhaky

http://www.cs.tau.ac.il/~msagiv/courses/apl12.html

Inspired by John Mitchell CS'242

Prerequisites

- Compilation course

Course Grade

- 20% Class notes
- 30% Assignments
- 50% Home or (easier) class exam

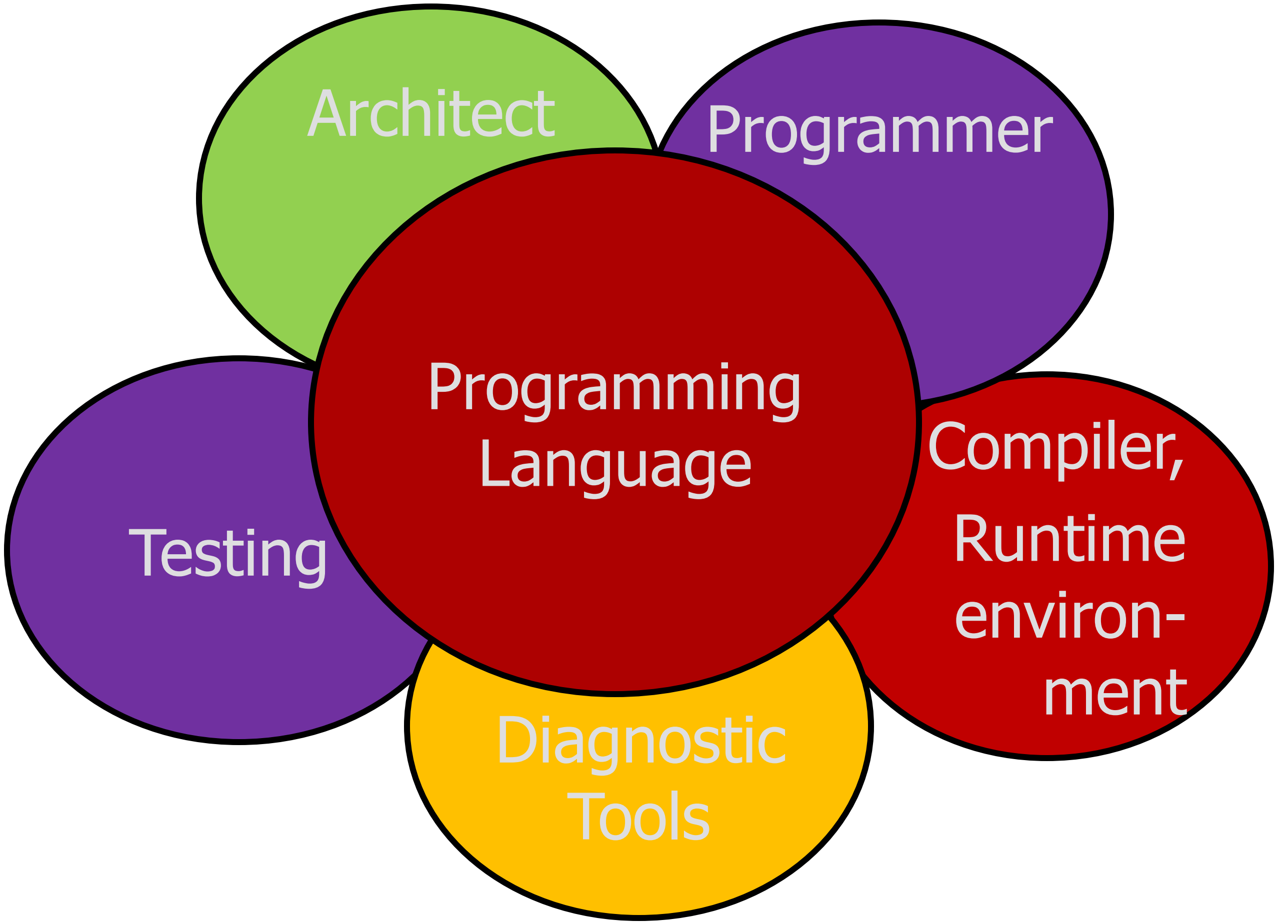
Class Notes

- Prepared by two 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

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



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
- C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576-580 and 583, October 1969
- Peter J. Landin. The next 700 programming languages
- …

Related Courses

- Compilers
- Programming languages
- Semantics of programming languages
- Program analysis

Tentative Schedule

6/3	introduction
13/3	javascript
20/3	Haskel
27/3	No class
3/4	Exception and continuation
17/4, 24/4, 2/5	Type Systems
8/5	Dependent types
15/5	IO Monads
22/5, 29/5, 5/6	Concurrency
12/6, 19/6	Domain Specific Languages
22/6	Summary class

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 o

zoomed 3,000 feet upward.
The captain disconnected the autopilot and pointed the Boe
777's nose down to avoid stalling, but was jerked into a ste
He throttled back sharply on both engines, trying to slow th
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. August 2005

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

Statically vs. Dynamically Checked Languages

	Statically Checked	Dynamically Checked
Safe	ML, Haskel, Java, C#	Lisp, Scheme, Perl, Python
Unsafe	C, C++	

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 rules for Untyped Arithmetic Expressions

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

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

$$\frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$$ (E-IF)

$$\frac{t_1 \rightarrow t'_1}{\text{succ } t_1 \rightarrow \text{succ } t'_1}$$ (E-SUCC)

$$\frac{t_1 \rightarrow t'_1}{\text{pred } t_1 \rightarrow \text{pred } t'_1}$$ (E-PRED)

$\text{pred } 0 \rightarrow 0$ (E-PREDZERO)

$\text{pred (succ } t) \rightarrow t$ (E-PREDSUCC)

$$\frac{t_1 \rightarrow t'_1}{\text{iszero } t_1 \rightarrow \text{iszero } t'_1}$$ (E-ISZERO)

$\text{iszero } 0 \rightarrow \text{true}$ (E-ISZEROZERO)

$\text{iszero succ } t \rightarrow \text{false}$ (E-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

- Determinism
  - $t_1 \to t_2 \;\wedge t_1 \to t_3 \Rightarrow t_2 = t_3$
- Reflexive transitive closure
  - $t \to^* t'$ if either $t = t'$ or there exists $t_0, t_1, \ldots, t_n$ such that $t=t_0$, $t' = t_n$ and for every $0 \leq i < n$: $t_i \to t_{i+1}$
- Semantic meaning
  - $[\![ ]\!]$: Terms $\rightharpoonup$ Nat $\cup$ Bool
  - $[\![t]\!] = t'$ if $t' \in \text{Nat} \cup \text{Bool} \wedge t \to^* t'$

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 ::=		types
	Bool	type of Boolean

$t : T$

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

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

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{(T-IF)}$$

Type Rules for Numbers

T ::= types
Nat type of Natural numbers

$t : T$

$0 : Nat$ (T-ZERO)

$$\frac{t_1 : Nat}{succ(t_1) : Nat} \text{ T-SUCC}$$

$$\frac{t_1 : Nat}{pred(t_1) : Nat} \text{ T-PRED}$$

$$\frac{t_1 : Nat}{iszero(t_1) : Bool} \text{ T-ISZERO}$$

Type Rules for Arithmetic Expressions

true : Bool (T-TRUE)

false : Bool (T-FALSE)

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{(T-IF)}$$

0 : Nat (T-ZERO)

$$\frac{t_1 : \text{Nat}}{\text{succ}(t_1) : \text{Nat}} \text{ T-SUCC}$$

$$\frac{t_1 : \text{Nat}}{\text{pred}(t_1) : \text{Nat}} \text{ T-PRED}$$

$$\frac{t_1 : \text{Nat}}{\text{iszero}(t_1) : \text{Bool}} \text{ T-ISZERO}$$

Examples

if false then 0 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

LEMMA: Inversion of the typing relation

true : R $\Rightarrow$ R = Bool

false : R $\Rightarrow$ R = Bool

if t_1 then t_2 else t_3 : R $\Rightarrow$ t_1: Bool, t_2 : R, t_3: R

0 : R $\Rightarrow$ R = Nat

succ t_1 : R $\Rightarrow$ R = Nat and t_1 : Nat

pred t_1 : R $\Rightarrow$ R = Nat and t_1 : Nat

iszero t1 : R $\Rightarrow$ R = Bool and t_1 : Nat

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

LEMMA 8.3.1: Canonical Forms:

If v is a value of type Boolean then v =true or v=false

If v is a value of type Nat then v belongs to nv

nv ::=	numeric values	
	0	zero value
	succ nv	successor value

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

Preservation: if t : T and $t \to t'$ then t' : T

Language Restrictions so far

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

Extensions

- Untyped lambda calculus (Chapter 5)
- Simple Typed Lambda Calculus (Chapter 9)
- Subtyping (Chapters 15-19)
  - Most general type
- Recursive Types (Chapters 20, 21)
  - NatList = <Nil: Unit, cons: {Nat, NatList}>
- Polymorphism (Chapters 22-28)
  - length:list $\alpha \to$ int
  - Append: list $\alpha \to \alpha \to$ list α
- Higher-order systems (Chapters 29-32)

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

Other Course Topics

- Dependent Types
- Monads
- Continuations
- Concurrency
- Domain specific languages
- …