CSE 505: Programming Languages

Professor: Alan Borning
TA: Josh Redstone

Goals:

- study of major concepts in programming languages
  - emphasis on nontraditional languages (i.e. other than imperative languages such as Pascal, C, Ada)
- programming experience in languages embodying concepts

Course outline

Functional languages (e.g. Scheme, ML, Haskell)

- side-effect-free programming
  (in Haskell, this is the *only* kind of programming)
- recursive first-class functions, recursive data structures
- first-class control (continuations)
- eager vs. lazy evaluation
- polymorphic static type inference (in ML and Haskell)

Object-oriented languages (e.g. Smalltalk, Cecil)

- inheritance, subclass polymorphism
- models of dynamic dispatching
- polymorphic static type systems

Declarative languages (e.g. CLP(R), Kaleidoscope)

- declarative programming, search algorithms
- constraints

Formal semantics

- operational & denotational semantics
- type deduction rules
- lambda calculi

Coursework

- Written homework and small programming projects
- Course term paper
  - 10-15 page paper on a language issue of student's choosing. Optional implementation component.
  - Topic negotiated with instructor (I'll ask for a project proposal in a couple of weeks)
- Midterm
- Final exam

Programming Language Design Principles & Goals

Programmers interact with program through language

- quickly learn programming language (tradeoff with language power)
- quickly express intent, model application domain
- read other people's code, understand intent
- reason about & ensure correctness, debug
- reason about performance trade-offs,
  ensure good performance
- modify/extend program

Tools interact with program through language

- compilers quickly analyze, optimize, and translate
- debuggers, program understanding tools
- machine program generators

Goals

Want language that makes these interactions easy as possible

W.r.t. programmers:

- learnable (simple, “natural”, no special cases)
- expressive, easy to write
  - for particular application domain? or in general?
- readable, understandable; no “clever” encodings
- simple, clear feature interactions;
  no undefined combinations;
  no error-prone features;
  early, extensive automatic error checking
- simple, clear, efficient implementation model
- what helps extensibility, modifiability?

W.r.t. tools:

- easy to automatically reason about & optimize
- easy to quickly extract important implementation info
- unambiguous
- regular enough to generate by machine

Design principles

Simplicity, regularity, orthogonality

- identify a small number of key concepts
- let them be combined in all combinations w/o restrictions

Provide abstraction mechanisms that enable
writing & checking (parameterized) code once,
reusing/instantiating code many times in wide range of ways

Be able to represent model of application as directly as possible

Fully define the semantics of programs, independent of underlying implementation environment

Avoid error-prone constructs,
provide automatic checks on consistency for allowed constructs

Type Safety; Static versus Dynamic Typing

Type safe. Type safe means that the language guarantees that a value of one type can’t be incorrectly used as if it were another type, in other words, that all expressions are guaranteed to be type consistent. Scheme, Haskell, Smalltalk, and Ada are examples of type safe languages.

Fortran and C are examples of languages that aren’t type safe.

Some languages for systems programming, for example Mesa from Xerox PARC, have a safe subset, although the language as a whole is not type safe.

An orthogonal issue is when the types of expressions are known and the statements in a program are checked for type safety. We can know the type of an expression at compile time or not until runtime, and thus we can check for type safety at compile time or at runtime. Many languages do a mixture of these.

Weakly typed. Weakly typed means “not type safe”. Fortran and C are examples of weakly typed languages.

Statically typed. Statically typed means that the type of every expression can be determined at compile time. ML and Ada are examples of statically typed languages. (Scheme is not statically typed though.)

Speaking more loosely, we may refer to a language as statically typed if the compiler can determine and check the types of almost all expressions at compile time.

Dynamically typed. The types of expressions are not be known until runtime.

Tradeoffs

Generally we want languages to be type safe.

An exception is a language used for some kinds of systems programming, for example writing a garbage collector. The "safe subset" approach is one way to deal with this problem.

Advantages of static typing:

- catch errors at compile time
- machine-checkable documentation
- potential for improved efficiency

Advantages of dynamic typing:

- flexibility
- rapid prototyping

More Terminology about Types

Unfortunately different authors mean different things by the terms "statically typed" and "strongly typed".

The previous definition of a statically typed language implies that such a language is also type safe. Other authors define "statically typed" to just mean that the compiler can statically assign a type to every expression -- but that typemight be wrong. By this definition Fortran is statically typed.

Strongly typed. We will equate strongly typed and type safe. For other authors, strongly typed implies type safe and statically typed. (Is Scheme strongly typed?)

To avoid misunderstanding, one can describe a language as e.g. "type safe and statically typed".

Typeless languages. A typeless language actually has just one type, the cell or word. Examples: Bliss, BCPL, assembler language.