Review

Last week

- Function Languages
- Lambda Calculus
- Scheme review
Outline

- Promises, promises, promises
- Types, round 1

Sources:

PLP, 7
Barrett. Lecture notes, Fall 2008.
Types

What is a type?

- A type consists of a set of values
- The compiler/interpreter defines a mapping of these values onto the underlying hardware.
Types

What purpose do types serve in programming languages?

- Implicit context for operations
  - Makes programs easier to read and write
  - Example: $a + b$ means different things depending on the types of $a$ and $b$.

- Constrain the set of correct programs
  - Type-checking catches many errors
A *type system* consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - *type equivalence*: when do two objects have the same type?
  - *type compatibility*: where can objects of a given type be used?
  - *type inference*: how do you determine the type of an expression from the types of its parts
Type Systems

A *type system* consists of:

- a mechanism for defining types and associating them with language constructs
- a set of rules for:
  - *type equivalence*: when do two objects have the same type?
  - *type compatibility*: where can objects of a given type be used?
  - *type inference*: how do you determine the type of an expression from the types of its parts

*What constructs are types associated with?*

- Constant values
- Names that can be bound to values
- Subroutines (sometimes)
- More complicated expressions built up from the above
Type Checking

*Type checking* is the process of ensuring that a program obeys the type system’s type compatibility rules.

A violation of the rules is called a *type clash*.

Languages differ in the way they implement type checking:

- strong vs weak
- static vs dynamic
Strong vs Weak Typing

- A strongly typed language does not allow variables to be used in a way inconsistent with their types (no loopholes)

- A weakly typed language allows many ways to bypass the type system (e.g., pointer arithmetic)

C is a poster child for the latter. Its motto is: “Trust the programmer”.
Static vs Dynamic Type Systems

Static vs Dynamic

- Static
  - Variables have types
  - Compiler ensures that type rules are obeyed at compile time
  - Ada, Pascal, ML

- Dynamic
  - Variables do not have types, values do
  - Compiler ensures that type rules are obeyed at run time
  - LISP, Scheme, Smalltalk, scripting languages

A language may have a mixture: Java has a mostly static type system with some runtime checks.

Pros and cons

- static: faster (dynamic typing requires run-time checks), easier to understand and maintain code, better error checking

- dynamic: more flexible, easier to write code
Polymorphism

Polymorphism allows a single piece of code to work with objects of multiple types.

- **Parametric polymorphism**: types can be thought of as additional parameters
  - *implicit*: often used with dynamic typing: code is typeless, types checked at run-time (LISP, SCHEME) - can also be used with static typing (ML)
  - *explicit*: templates in C++, generics in JAVA

- **Subtype polymorphism**: the ability to treat a value of a subtype as a value of a supertype

- **Class polymorphism**: the ability to treat a class as one of its superclasses (special case of subtype polymorphism)
Parametric polymorphism example

SCHEME

\[
\text{(define (length l)} \\
\text{ (cond} \\
\text{ ((null? l) 0)} \\
\text{ (#t (+ (length (cdr l)) 1))})
\]

The types are checked at run-time.

ML

\[
\text{fun length xs =} \\
\text{ if null xs} \\
\text{ then 0} \\
\text{ else 1 + length (tl xs)}
\]

How can ML be statically typed and allow polymorphism?

It uses \textit{type variables} for the unknown types. The type of this function is written \texttt{\'a list -> int}.
Assigning types

Programming languages support various methods for assigning types to program constructs:

- **determined by syntax**: the syntax of a variable determines its type (FORTRAN 77, ALGOL 60, BASIC)
- **no compile-time bindings**: dynamically typed languages
- **explicit type declarations**: most languages
Types: Points of View

**Denotational**

- type is a set $T$ of values
- value has type $T$ if it belongs to the set
- object has type $T$ if it is guaranteed to be bound to a value in $T$

**Constructive**

- type is either *built-in* (int, real, bool, char, etc.) or
- *constructed* using a *type-constructor* (record, array, set, etc.)

**Abstraction-based**

- Type is an *interface* consisting of a set of operations
Scalar Types Overview

- discrete types
  - integer types
    - often several sizes (e.g., 16 bit, 32 bit, 64 bit)
    - sometimes have signed and unsigned variants (e.g., C/C++, Ada, C#)
  - enumeration types

- floating-point types
  - typically 64 bit (double in C); sometimes 32 bit as well (float in C)

- rational types
  - used to represent exact fractions (Scheme, Lisp)

- complex
  - Fortran, Scheme, Lisp, C 99, C++ (in STL)
Other intrinsic types

- **boolean**
  Common type; C had no boolean until C 99

- **character, string**
  - some languages have no character data type (e.g., JAVASCRIPT)
  - internationalization support
    - **JAVA**: UTF-16
    - **C++**: 8 or 16 bit characters; semantics implementation dependent
  - string mutability
    most languages allow it, JAVA does not.

- **void, unit**
  Used as return type of procedures;
  **void**: (C, JAVA) represents the absence of a type
  **unit**: (ML, HASKELL) a type with one value: ( )
Enumeration types: abstraction at its best

- trivial and compact implementation: values are mapped to successive integers
- very common abstraction: list of names, properties
- expressive of real-world domain, hides machine representation

**Example in ADA:**

```ada
  type Suit is (Hearts, Diamonds, Spades, Clubs);
  type Direction is (East, West, North, South);
```

Order of list means that `Spades > Hearts`, etc.

Contrast this with C#:

```
  "arithmetics on enum numbers may produce results in the underlying representation type that do not correspond to any declared enum member; this is not an error"
```
Enumeration types and strong typing

ADA again:

type Fruit is (Apple, Orange, Grape, Apricot);
type Vendor is (Apple, IBM, HP, Dell);

My_PC : Vendor;
Dessert : Fruit;
...
My_PC := Apple;
Dessert := Apple;
Dessert := My_PC; -- error

Apple is overloaded. It can be of type Fruit or Vendor.

Overloading is allowed in C#, JAVA, ADA

Not allowed in PASCAL, C
Subranges

Ada and PASCAL allow types to be defined which are subranges of existing discrete types.

\begin{verbatim}
  type Sub is new Positive range 2 .. 5; -- Ada
  V: Sub;

  type sub = 2 .. 5; (* Pascal *)
  var v: sub;
\end{verbatim}

Assignments to these variables are checked at runtime:

\begin{verbatim}
  V := I + J;  -- runtime error if not in range
\end{verbatim}
Composite Types

- arrays
- records
- variant records, unions
- pointers, references
- function types
- lists
- sets
- maps
Composite Literals

Does the language support these?

- array aggregates

  \[ A := (1, 2, 3, 10); \quad \text{-- positional} \]
  \[ A := (1, \text{others} \Rightarrow 0); \quad \text{-- for default} \]
  \[ A := (1..3 \Rightarrow 1, 4 \Rightarrow -999); \quad \text{-- named} \]

- record aggregates

  \[ R := (\text{name} \Rightarrow "NYU", \text{zipcode} \Rightarrow 10012); \]
Type checking and inference

- **Type checking:**
  - Variables are declared with their type.
  - Compiler determines if variables are used in accordance with their type declarations.

- **Type inference: (ML, Haskell)**
  - Variables are declared, but not their type.
  - Compiler determines type of a variable from its usage/initialization.

In both cases, type inconsistencies are reported at compile time.

```plaintext
fun f x =
  if x = 5 (* There are two type errors here *)
    then hd x
  else tl x
```
Type equivalence

Name vs structural

- name equivalence
  two types are the same only if they have the same name (each type definition introduces a new type)
  - strict: aliases (i.e. declaring a type to be equal to another type) are distinct
  - loose: aliases are equivalent

- structural equivalence
  two types are equivalent if they have the same structure

Most languages have mixture, e.g., C: name equivalence for records (structs), structural equivalence for almost everything else.
Type equivalence example

Structural equivalence in ML:

```ml
type t1 = { a: int, b: real };      
type t2 = { b: real, a: int };      
(* t1 and t2 are equivalent types *)
```
Accidental structural equivalence

```typescript
type student = {
    name: string,
    address: string
}

type school = {
    name: string,
    address: string
}

type age = float;
type weight = float;
```

With structural equivalence, we can accidentally assign a `school` to a `student`, or an `age` to a `weight`. 
Type conversion

Sometimes, we want to convert between types:

- if types are structurally equivalent, conversion is trivial (even if language uses name equivalence)
- if types are different, but share a representation, conversion requires no run-time code
- if types are represented differently, conversion may require run-time code (from int to float in C)

A nonconverting type cast changes the type without running any conversion code. These are dangerous but sometimes necessary in low-level code:

- unchecked_conversion in ADA
- reinterpret_cast in C++