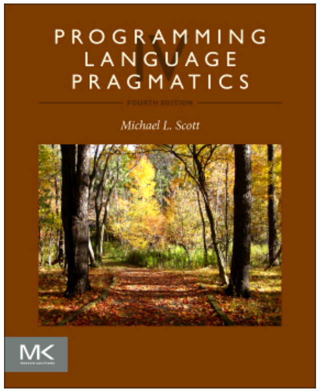
Types and Type Checking

17-363/17-663: Programming Language Pragmatics


PROGRAMMING
LANGUAGE
PRAGMATICS
FOURTH EDITION
Michael L. Scott
MK

Reading: PLP chapter 7


Ben Titzer


Jonathan Aldrich

Data Types

- What is a type? 3 views:
  - Denotational: a collection of values from a domain
    - e.g. the 32-bit integers (**int**), or the real numbers representable as IEEE single-precision floats (**float**)
  - Structural: a description of a data structure in terms of fundamental constructs
    - e.g. a point is a record made up of fields x and y, both of type **int**
  - Behavioral: the set of operations that can be applied to an object
    - e.g. a Stack has operations push(v) and pop()
    - Similar to structural, but the structure is a set of methods, not fields

Data Types

- What are types good for?
  - Documentation
    - What do I need to pass to this library function?
  - Implicit context for compilation
    - Is this + an integer add or a floating point add?
  - Checking - meaningless operations do not occur
    - e.g. `“hello, world” - 5` does not make sense
    - Type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful

Terminology

- Type safety
  - The language ensures that only type-appropriate operations are applied to an object
- Strong vs. weak typing
  - The degree to which the language enforces typing invariants and prevents accidental errors
- Static vs. dynamic typing
  - Whether types are checked at compile time or run time

Type Systems

- Examples
  - Java is **type safe**, **strongly** and **statically typed**
  - Common Lisp is **type safe**, **strongly** and **dynamically typed**
  - C and C++ are **statically** and **strongly typed**, but are **not (fully) type safe**
  - JavaScript is **type safe** and **dynamically typed**, but allows many implicit conversions between types, some of which are surprising. It would be considered more **weakly typed** than the above languages.

Fun with JavaScript

- What does it mean to be weakly typed?

```
[] == ![];

"b" + "a" + +"a" + "a";

null == 0;
null > 0;
null >= 0;
```

Type Examples and Terminology

- Discrete types – countable
  - integer
  - boolean
  - char
  - enumeration
  - subrange
- Scalar types - one-dimensional
  - All discrete types
  - real

Type Systems

- Composite types:
  - records
  - datatypes/unions
  - arrays
    - strings
  - sets
  - pointers
  - lists
  - files

Orthogonality in Type Systems

- Orthogonality is a desirable property
    - There are no restrictions on the way types can be combined
- Type theory typically studies orthogonal type constructs
    - e.g. we provide a grammar for types, they can be constructed in any way
- Most languages restrict orthogonality
    - Often for practical reasons, e.g. minimizing syntactic overhead or making type checking decidable
    - Example: ML only allows polymorphism at a **let**
    - Example: Java classes combine records with recursive types

Subtyping

- When one type can be safely used as another type
  - e.g. in most languages an integer can be used as a real
  - The “operational” definition of subtyping
- Other definitions
  - Intuitive: A<:B if A is a B
    - e.g. a StreetAddress is an Address
  - Denotational: A <: B if A describes a subset of the values that B describes
    - e.g. the integers are a subset of the reals
  - Structural: A <: B if A has all of the structure of B (and maybe more)
  - Behavioral: A <: B if A has all the operations that B does, and they behave as we’d expect for a B

Subtyping Rules

- Subsumption - a subtype can be treated as a supertype:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}\ \textit{T-subsume}$$

- Subtyping is reflexive and transitive:

$$\frac{}{\tau \leq \tau}\ \textit{S-reflexive}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \textit{S-transitive}$$

- We can capture some of Java's subtyping rules as follows:

$$\frac{}{\texttt{int} \leq \texttt{long}}\ \textit{S-int-long}$$

$$\frac{}{\texttt{long} \leq \texttt{float}}\ \textit{S-long-float}$$

$$\frac{}{\texttt{float} \leq \texttt{double}}\ \textit{S-float-double}$$

Subtyping Practice

- Show a derivation that types the expression 1 + 2.5

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2} \textit{ T-subsume}$$

$$\frac{}{\texttt{int} \leq \texttt{long}} \textit{ S-int-long}$$

$$\frac{}{\tau \leq \tau} \textit{ S-reflexive}$$

$$\frac{}{\texttt{long} \leq \texttt{float}} \textit{ S-long-float}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \textit{ S-transitive}$$

$$\frac{}{\texttt{float} \leq \texttt{double}} \textit{ S-float-double}$$

$$\frac{\Gamma \vdash e_1 : \texttt{double} \quad \Gamma \vdash e_2 : \texttt{double}}{\Gamma \vdash e_1 + e_2 : \texttt{double}} \textit{ T-add-double}$$

Type Checking

- A TYPE SYSTEM has rules for
  - type compatibility (when can a value of type A be used in a context that expects type B?)
    - Similar to the first definition of subtyping
      - But sometimes languages break this for convenience, e.g. allowing reals to be implicitly converted to integers, or integers to be implicitly truncated
    - Type equivalence: when two types are mutually compatible
  - type inference (what is the type of an expression, given the types of the operands?)

Structural vs. Name Equivalence

- Are these equivalent?

```
struct person {
  string name;
  string address;
}
struct school {
  string name;
  string address;
}
```

- Some languages let you choose. E.g. in Ada:

```
type Score is integer;               // structural equivalence; equiv to integer

type Fahrenheit is new integer;      // name equivalence
type Celsius is new integer;         // can’t assign Fahrenheit to Celsius
```

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name equivalence is based on declarations
    - Advantage: captures the programmer's intent
    - Typical in imperative & OO languages
  - Structural equivalence is based on some notion of meaning behind those declarations
    - Advantage: more flexible
    - Disadvantage: can "accidentally" equate types
    - Common in functional languages (but they usually have ways to support nominal equivalence also)

Type Checking

- Structural equivalence depends on simple comparison of type descriptions substitute out all names
  - expand all the way to built-in types
- Original types are equivalent if the expanded type descriptions are the same

Type Checking

- Coercion
  - When an expression of one type is used in a context where a different type is expected, one normally gets a type error
  - But what about

```
var a : integer; b, c : real;
     ...
c := a + b;
```

Type Checking

- Coercion
  - Many languages allow things like this, and COERCE an expression to be of the proper type
  - Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well

Type Checking

- C has lots of coercion, too, but with simpler rules:
  - all **`float`**s in expressions become **`double`**s
  - **`short`**, **`int`**, and **`char`** become **`int`** in expressions
  - if necessary, precision is removed when assigning into LHS

Coercion Rules

$$\frac{\Gamma \vdash e : \mathbf{int}}{\Gamma \vdash e \leadsto \mathtt{float}(e) : \mathbf{real}}\ \textit{coerce-real}$$

$$\frac{\Gamma \vdash e : \mathbf{real}}{\Gamma \vdash (\mathbf{int})e \leadsto \mathtt{trunc}(e) : \mathbf{int}}\ \textit{convert-int}$$

- Coercion and conversions can be added in an *elaboration* pass within the compiler
  - –Elaboration makes implicit things explicit
- Coercions are inserted when subsumption is used but the types have different representions
- Conversions are inserted where the user adds casts

Type Checking

- Make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - in C and derived languages, the word 'cast' is often used for conversions

Implementing Type Checkers

```
function typecheck_expr(scope : Scope, a : AST) : Type
case a of
    int_lit(n) : return integer
    real_lit(r) : return real
    var(x) : return symbol_table.get_type(x, scope, a)
    float(a1) :
        typ : Type := typecheck_expr(scope, a1)
        if typ $\notin$ {integer, error_type} then error(‘‘already a real’’, a)
        return float
    trunc(a1) :
        typ : Type := typecheck_expr(scope, a1)
        if typ $\notin$ {real, error_type} then error(‘‘already an integer’’, a)
        return integer
    bin_op(a1, op, a2) :
        typ1 : Type := typecheck_expr(scope, a1)
        typ2 : Type := typecheck_expr(scope, a2)
        if typ1 = typ2 then return typ1
        else if typ1 = error_type then return typ2
        else if typ2 = error_type then return typ1
        else error(‘‘mismatched types’’, a); return error_type
```

if x is not found, get_type will call error(“variable not declared”, a) and add x to scope with error_type, to avoid cascading messages

Implementing Type Checkers

```
function typecheck_stmt(scope : Scope, a : AST)
case a of
    int_decl(x, s) :
        symbol_table.add(x, integer, scope, a)
        typecheck_stmt(scope, s)
    real_decl(x, s) : . . . — analogous to int_decl
    assign(x, e, s) :
        typ_expr := typecheck_expr(scope, e)
        typ_x := symbol_table.get_type(x, scope, a) — see notes on get_type on prior slide
        if typ_expr ≠ typ_x and type_expr ≠ error_type and type_x ≠ error type
            error(‘‘mismatched types’’)
        typecheck_stmt(scope, s)
    read(x, s) :
        typ_x := symbol_table.get_type(x, scope, a) — see notes on get_type on prior slide
        typecheck_stmt(scope, s)
    write(e, s) :
        typecheck_expr(scope, e)
        typecheck_stmt(scope, s)
    null : return
```

if x is already present and not of error_type, add willcall error(“variable already declared in scope”, a) and set the type of x to error_type if the two declarations differ

Polymorphism

- Polymorphism allows one piece of code to work with multiple types

- Example: Polymorphism in Java

```
static <T> bool isMember(T value, T[] array) {
      for (int i = 0; i < array.length; ++i)
             if (T[i].equals(value)) return true;
      return false;
}
Integer[] a1 = { 1, 2, 3 };
String[] a2 = { “hello”, “world” };
bool result = isMember(a1, 5); // returns false
bool result2 = isMember(a2, “hello”); // returns true
bool error = isMember(a2, 5); // type error
```

Thinking about Polymorphic Types

- Example: Polymorphism in Java

```
static <T> bool isMember(T value, T[] array) { ... }

// typing: isMember: ∀T(T, T[]) -> bool

bool result = isMember(a, 5)

// think: bool result = isMember[int](a, 5)
// (the compiler figures out the [int] part)
// so we substitute T with int and we have
// isMember[int] : (int, int[]) -> bool
```

Polymorphism Typing Rules

$$\begin{array}{lcl} e & ::= & \ldots \mid \Lambda T.e \mid e[\tau] \\ \tau & ::= & \ldots \mid \forall T.\tau \mid T \\ \Gamma & ::= & \ldots \mid \Gamma, T \end{array}$$

$$\frac{\Gamma, T \vdash e : \tau}{\Gamma \vdash \Lambda T.e : \forall T.\tau}\ \textit{T-type-abstract}$$

$$\frac{\Gamma \vdash e : \forall T.\tau}{\Gamma \vdash e[\tau'] : [\tau'/T]\tau}\ \textit{T-type-apply}$$

$$\frac{}{(\Lambda T.e)[\tau] \rightarrow [\tau/T]e}\ \textit{step-type-apply}$$

$$\frac{e \rightarrow e'}{e[\tau'] \rightarrow e'[\tau']}\ \textit{congruence-type-abstract}$$

```
<T> bool isMember(T value, T[] array) { … }
≈ ΛT . (value: T, array: T[]) => …

bool result = isMember(a, 5)
≈ isMember[int](a, 5)
≈ (body of isMember, where T is replaced with int)(a, 5)
```

Polymorphism Practice

$$\begin{array}{lcl} e & ::= & \ldots \mid \Lambda T.e \mid e[\tau] \\ \tau & ::= & \ldots \mid \forall T.\tau \mid T \\ \Gamma & ::= & \ldots \mid \Gamma, T \end{array}$$

$$\frac{}{(\Lambda T.e)[\tau] \rightarrow [\tau/T]e} \; \textit{step-type-apply}$$

$$\frac{e \rightarrow e'}{e[\tau'] \rightarrow e'[\tau']} \; \textit{congruence-type-abstract}$$

$$\frac{\Gamma, T \vdash e : \tau}{\Gamma \vdash \Lambda T.e : \forall T.\tau} \; \textit{T-type-abstract}$$

$$\frac{\Gamma \vdash e : \forall T.\tau}{\Gamma \vdash e[\tau'] : [\tau'/T]\tau} \; \textit{T-type-apply}$$

Show a typing derivation for the program:

let id = ΛT. x:T => x
in id[**int**](3)

Also show the steps this program takes in reducing:

Local Type Inference

- In C++ (and many other languages)

auto x = 3.5+1;

- x will have type double since the right-hand side expression has that type

Global Type Inference

```
1 -- fib :: int -> int
2 let fib n =
3   let rec helper f1 f2 i =
4     if i = n then f2
5     else helper f2 (f1 + f2) (i + 1) in
6   helper 0 1 0;;
```

- i is int, because it is added to 1 at line 5
- n is int, because it is compared to i at line 4
- all three args at line 6 are int consts, so f1 and f2 are int
- also, the 3rd argument is consistent with the known int type of i (good!)
- and the types of the arguments to the recursive call at line 5 are similarly consistent
- since helper returns f2 (known to be int) at line 4, the result of the call at line 6 will be int
- Since fib immediately returns this result as its own result, the return type of fib is int
- For more details re: type inference in ML, read about Algorithm W