Languages and Compilers
(SProg og Oversættere)

Lecture 12
Types

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to Simon Gay, Elsa Gunter and Elizabeth White whose slides this lecture is based on.

Learning goals

- Understand primitive and composit types
  - How implementations may affect types in languages
  - Pointer and references
  - Constructed datatypes:
    - Arrays
    - Records/structs
    - Unions or variant records
  - Structural and Name Equivalence
  - Recursive Types
  - E.g.: $\text{List} = \text{Unit} + (\text{Int} \times \text{List})$
  - Implicit versus explicit type conversions

- Understand some of the principles behind more advanced type systems
  - Polymorphism
  - Subtyping

Types revisited

- Fisher et al. and Sebesta, to some extent, may leave you with the impression that types in languages are simple and type checking is a minor part of the compiler
- However, type system design and type checking and/or inferencing algorithms is one of the hottest topics in programming language research at present!
- Types:
  - Have to be an integral part of the language design
    - Syntax
    - Contextual constraints (static type checking)
    - Code generation (space allocation and dynamic type checking)
  - Provides a precise criterion for safety and sanity of a design.
    - Language level
    - Program level
  - Close connections with logics and semantics.
    - The Curry–Howard correspondence

Typechecking

- Static typechecking
  - All type errors are detected at compile-time
  - Mini Triangle is *statically typed*
  - Most modern languages have a large emphasis on static typechecking
- Dynamic typechecking
  - Scripting languages such as JavaScript, PhP, Perl and Python do run-time typechecking
- Mix of Static and Dynamic
  - object-oriented programming requires some runtime typechecking: e.g. Java has a lot of compile-time typechecking but it is still necessary for some potential runtime type errors to be detected by the runtime system

- Static typechecking involves calculating or *inferring* the types of expressions (by using information about the types of their components) and checking that these types are what they should be (e.g. the condition in an *if* statement must have type *Boolean*).

Static Typechecking

- Static (compile-time) or dynamic (run-time)
  - *static is often desirable: finds errors sooner, doesn't degrade performance*
- Verifies that the programmer's intentions (expressed by declarations) are observed by the program
- A program which typechecks is guaranteed to behave well at run-time
  - *at least: never apply an operation to the wrong type of value*
  more: eg. security properties
- A program which typechecks respects the high-level abstractions
  - *eg: public/protected/private access in Java*

Why are Type declarations important?

- Organize data into high-level structures
  essential for high-level programming
- Document the program
  basic information about the meaning of variables and functions, procedures or methods
- Inform the compiler
  example: how much storage each value needs
- Specify simple aspects of the behaviour of functions
  "types as specifications" is an important idea

Why type systems are important

- Economy of execution
  - E.g. no null pointer checking is needed in SML
- Economy of small-scale development
  - A well-engineered type system can capture a large number of trivial programming errors thus eliminating a lot of debugging
- Economy of compiling
  - Type information can be organised into interfaces for program modules which therefore can be compiled separately
- Economy of large-scale development
  - Interfaces and modules have methodological advantages allowing separate teams to work on different parts of a large application without fear of code interference
- Economy of development and maintenance in security areas
  - If there is any way to cast an integer into a pointer type (or object type) the whole runtime system is compromised – most vira and worms use this method of attack
- Economy of language features
  - Typed constructs are naturally composed in an orthogonal way, thus type systems promote orthogonal programming language design and eliminate artificial restrictions

Why study type systems and programming languages?

The type system of a language has a strong effect on the "feel" of programming.

Examples:

- In original Pascal, the result type of a function cannot be an array type. In Java, an array is just an object and arrays can be used anywhere.
- In SML, programming with lists is very easy; in Java it is much less natural.

To understand a language fully, we need to understand its type system. The underlying typing concepts appearing in different languages in different ways, help us to compare and understand language features.

Java Example

Type definitions and declarations are essential aspects of high-level programming languages.

```
class Example {
      int a;
      void set(int x) {a=x;}
      int  get() {return a;}
}

Example e = new Example();
```

Where are the type definitions and declarations in the above code?

SML example

Type definitions and declarations are essential aspects of high-level programming languages.

```
datatype 'a tree =
   INTERNAL of {left:'a tree,right:'a tree}
 | LEAF of {contents:'a}

fun sum(tree: int tree) =
  case tree of
    INTERNAL{left,right} => sum(left) + sum(right)
  | LEAF{contents} => contents
```

Where are the type definitions and declarations in the above code?

Types

- Types are either primitive or constructed.
- Primitive types are atomic with no internal structure as far as the program is concerned
  - Integers, float, char, …
- Arrays, unions, structures, functions, … can be treated as constructor types
- Pointers (or references) and String are treated as basic types in some languages and as constructed types in other languages

Specification of Primitive Data Types

- Basic attributes of a primitive type usually used by the compiler and then discarded
- Some partial type information may occur in data object
- Values usually match with hardware types:
  - 8 bits, 16 bits, 32 bits, 64 bits
- Operations: primitive operations with hardware support, and user-defined/library operations built from primitive ones
- But there are design choices to be made!

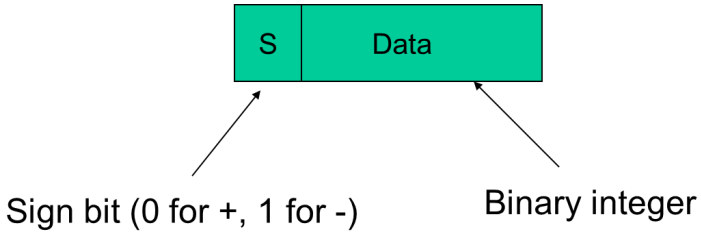
Integers – Specification

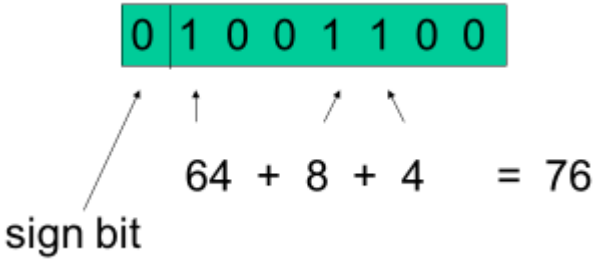
- The set of values of type *Integer* is a finite set
  - {-*maxint* … *maxint* }
  - typically -2^31 through 2^31 – 1
  - –2^30 through 2^30 - 1
  - not the mathematical set of integers (as operations may overflow).
- Standard collection of operators:
  - +, -, *, /, mod, ~ (negation)
- Standard relational operators:
  - =, <, >, <=, >=, =/=
- The language designer has to decide
  - which representation to use
  - The collection of operators and relations

Integers - Implementation

- Implementation:
  - Binary representation in 2's complement arithmetic
  - Three different standard representations:

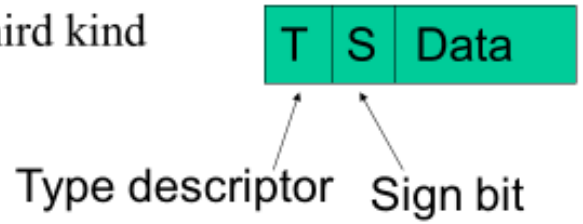
- First kind:





- Second kind

- Third kind



Floating Points

- IEEE standard 754 specifies both a 32- and 64-bit standard
- At least one supported by most hardware
- Some hardware also has proprietary representations
- Numbers consist of three fields:
  - S (sign), E (exponent), M (mantissa)

- Every non-zero number may be uniquely written as

$$(-1)^S * 2^E * M$$

where $1 \leq M < 2$ and S is either 0 or 1

Language design issue

- Should my language support floating points?
- Should it support IEEE standard 754
  - 32 bit, 64 bits or both
- Should my language support native floating points?
- Should floating points be the only number representation in my language?

Other Primitive Data

- Short integers (C) - 16 bit, 8 bit
- Long integers (C) - 64 bit
- Boolean or logical - 1 bit with value true or false (often stored as bytes)
- Byte - 8 bits
- Java has
  - byte, short, int, long, float, double, char, boolean
- C# also has
  - sbyte, ushort, uint, ulong

Characters

- Character - Single 8-bit byte - 256 characters
- ASCII is a 7 bit 128 character code
- Unicode is a 16-bit character code (Java)
- In C, a char variable is simply 8-bit integer numeric data

Enumerations

- Motivation: Type for case analysis over a small number of symbolic values
- Example: (Ada)

  Type DAYS **is** {Mon, Tues, Wed, Thu, Fri, Sat, Sun}
- Implementation: Mon → 0; … Sun → 6
- Treated as ordered type (Mon < Wed)
- In C, always implicitly coerced to integers
- Java didn't have enum until Java 1.5

Java Type-safe enum

Remember

```
public class Token {
   byte kind; String spelling;
   final static byte
      IDENTIFIER = 0; INTLITERAL = 1; OPERATOR   = 2;
      BEGIN      = 3; CONST      = 4; ...
      ...
...
}
```

```
private void parseSingleCommand() {
   switch (currentToken.kind) {
     case Token.IDENTIFIER : ...
     case Token.IF : ...
     ... more cases ...
     default: report a syntax error
   }
}
```

Java Type-safe enum

Can now be written as

```
public class Token {
   String spelling;
   enum kind {IDENTIFIER, INTLITERAL, OPERATOR,
      BEGIN, CONST, ... }
      ...
...
}
```

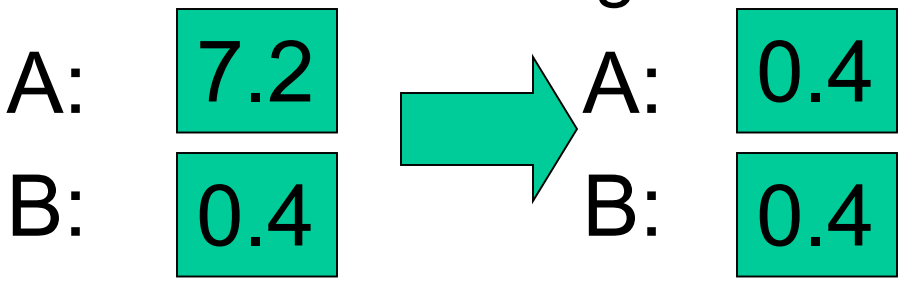
```
private void parseSingleCommand() {
   switch (currentToken.kind) {
     case IDENTIFIER : ...
     case IF : ...
     ... more cases ...
     default: report a syntax error
   }
}
```

Pointers

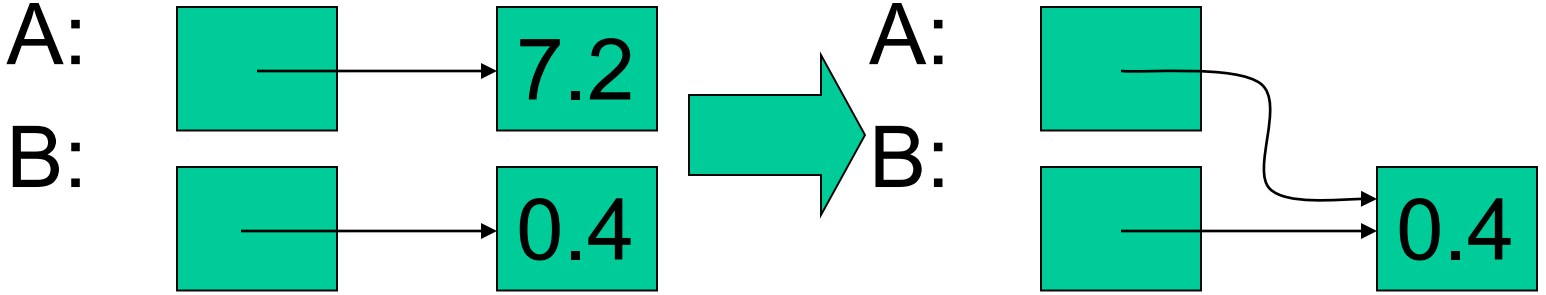
- A *pointer type* is a type in which the range of values consists of memory addresses and a special value, nil (or null)
- Each pointer can point to an object of another data structure
  - Its l-value is its address; its r-value is the address of another object
- Accessing r-value of r-value of pointer called *dereferencing*
- Use of pointers to create arbitrary data structures

Pointer Aliasing

- A:= B
  - Numeric assignment

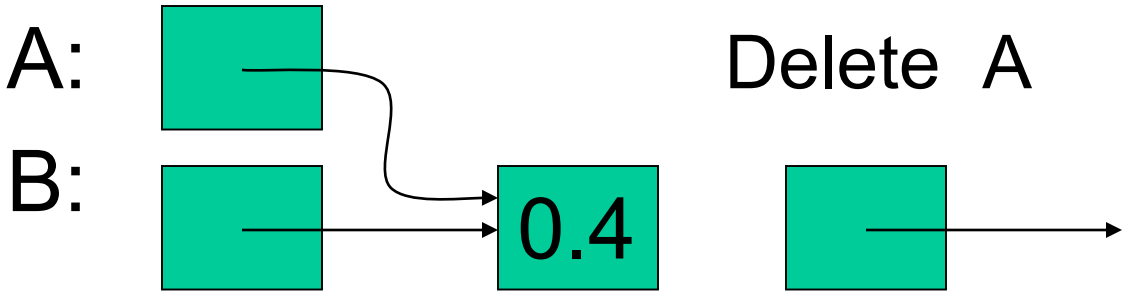


  - Pointer assignment

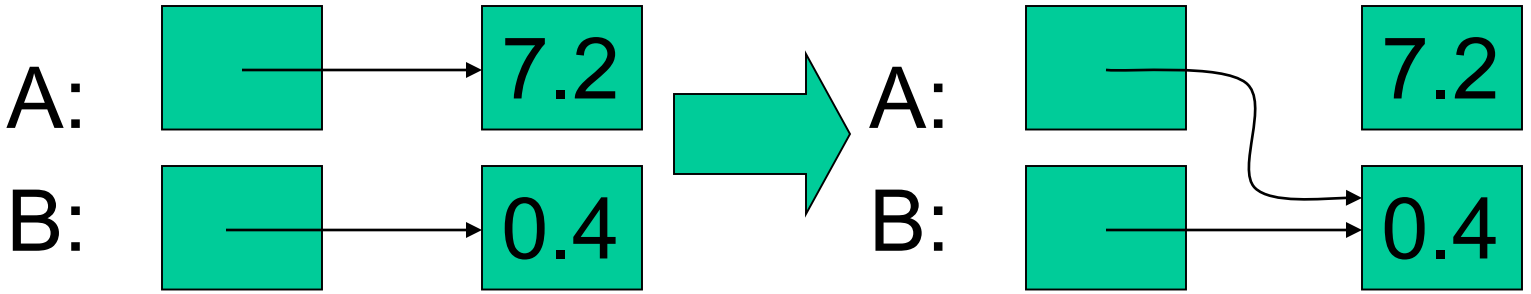


Problems with Pointers

- Dangling Pointer



- Garbage (lost heap-dynamic variables)



SML references

- An alternative to allowing pointers directly
- References in SML can be typed
- … but they introduce some abnormalities

- SML reference cells
  - Different types for location and contents

    x : int non-assignable integer value
    y : int ref location whose contents must be integer
    !y the contents of location y
    ref x expression creating new cell initialized to x
  - SML assignment

    operator := applied to memory cell and new contents
  - Examples

    y := x+3 place value of x+3 in cell y; requires x:int
    y := !y + 3 add 3 to contents of y and store in location y
-

References in Java and C#

- Similar to SML both Java and C# use references to heap allocated objects

```
class Point {
  int x,y;
  public Point(int x, int y) {
     this.x=x; this.y=y;
  }

  public void move(int dx, int dy) {
     x=x+dx; y=y+dy;
  }
}
…
Point p = new Point(2,3);
p.move(5,6);
Point q = new Point(0,0);
p = q;
p.move(3,7);
q = null;
```

Nullable Types in C#

- T? same as System.Nullable<T>

```
int? x = 123;
double? y = 1.25;
```

- null literal conversions

```
int? x = null;
double? y = null;
```

- Nullable conversions

```
int i = 123;
int? x = i;             // int --> int?
double? y = x;          // int? --> double?
int? z = (int?)y;       // double? --> int?
int j = (int)z;         // int? --> int
```

Strings

- Can be implemented as
  - a primitive type as in SML
  - an object as in Java
  - an array of characters (as in C and C++)
- If primitive, operations are built in
- If object or array of characters, string operations provided through a library

- String implementations:
  - Fixed declared length
  - Variable length with declared maximum
  - Unbounded length
    - Linked list of fixed length strings
    - null terminated contiguous array

Arrays

An array is a collection of values, all of the same type, indexed by a range of integers (or sometimes a range within an enumerated type).

In Ada: a : array (1..50) of Float; (static arrays)
In Java: float[] a; (dynamic arrays)

Most languages check at runtime that array indices are within the bounds of the array: a(51) is an error. (In C you get the contents of the memory location just after the end of the array!)

If the bounds of an array are viewed as part of its type, then array bounds checking can be viewed as typechecking, but in general it is impossible to do it statically: consider a(f(1)) for an arbitrary function f.

Static typechecking is a compromise between *expressiveness* and *computational feasibility*. More about this later

Array Layout and Component Access

- Component access through subscripting, both for lookup (r-value) and for update (l-value)

- Component access should take constant time (ie. looking up the 5^{th} element takes same time as looking up 100^{th} element)

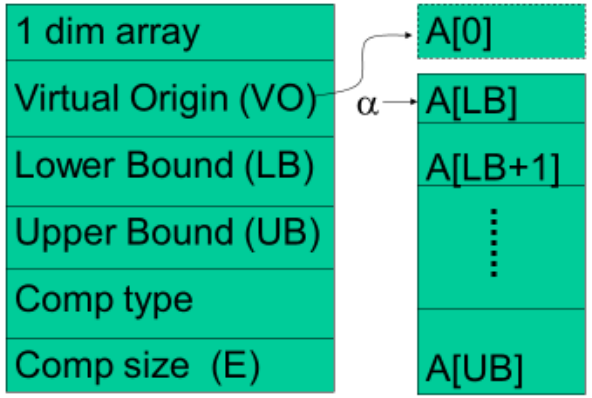
- L-value of A[i] = VO + (E * i)
  $= \alpha + (E * (i - LB))$

- Computed at compile time
- VO = α - (E * LB)

- More complicated for multiple dimensions

Array Layout

- Assume one dimension



Pause

Composite Data Types

- Composite data types are sets of data objects built from data objects of other types
- Data type constructors are arrays, structures, unions, lists, …
- It is useful to consider the structure of types and type constructors independently of the form which they take in particular languages.

Products and Records

If T and U are types, then $T \times U$ (written (T * U) in SML) is the type whose values are pairs (t,u) where t has type T and u has type U.

Mathematically this corresponds to the *cartesian product* of sets. More generally we have *tuple* types with any number of components. The components can be extracted by means of *projection functions*.

Product types more often appear as *record types*, which attach a label or *field name* to each component. Example in Ada and C:

```
type T is
record
   x : Integer;
   y : Float
end record
```

```
struct T {
   int x;
   float y;
}
```

Products and Records

If *v* is a value of type *T* then *v* contains an Integer and a Float. Writing *v.x* and *v.y* can be more readable than *fst(v)* and *snd(v)*.

```
type T is
record
   x : Integer;
   y : Float
end record
```

Record types are mathematically equivalent to products.

An object can be thought of as a record in which some fields are functions, and a class definition as a record type definition in which some fields have function types. Object-oriented languages also provide *inheritance*, leading to *subtyping* relationships between object types.

Variant Records

In Pascal, the value of one field of a record can determine the presence or absence of other fields. Example:

```
type T = record
           x : integer;
           case b : boolean of
             false : (y : integer);
             true : (z : boolean)
         end
```

It is not possible for static type checking to eliminate all type errors from programs which use variant records in Pascal: the compiler cannot check consistency between the *tag field* and the data which is stored in the record. The following code passes the type checker in Pascal:

```
var r : T, a : integer;
begin
  r.x := 1; r.b := true; r.z := false;
  a := r.y * 5
end
```

Variant Records in Ada

Ada handles variant records safely. Instead of a tag field, the type definition has a parameter, which is set when a particular record is created and then cannot be changed.

```
type T(b : Boolean) is record
   x : Integer;
   case b is
      when False => y : Integer;
      when True  => z : Boolean
   end case
end record;

declare r : T(True), a : Integer;
begin
   r.x := 1; r.z := False;
   a := r.y * 5;
end;
```

r does not have field y, and never will

this type error can be detected statically

Disjoint Unions

The mathematical concept underlying variant record types is the *disjoint union*. A value of type $T+U$ is either a value of type T or a value of type U, tagged to indicate which type it belongs to:

$$T+U = \{ \mathit{left}(x) \mid x \in T \} \cup \{ \mathit{right}(x) \mid x \in U \}$$

SML and other functional languages support disjoint unions by means of *algebraic datatypes*, e.g.

```
datatype X = Alpha String | Numeric Int
```

The *constructors* Alpha and Numeric can be used as functions to build values of type X, and pattern-matching can be used on a value of type X to extract a String or an Int as appropriate.

An enumerated type is a disjoint union of copies of the *unit* type (which has just one value). Algebraic datatypes unify enumerations and disjoint unions (and recursive types) into a convenient programming feature.

Variant Records and Disjoint Unions

The Ada type:

```
type T(b : Boolean) is record
   x : Integer;
   case b is
      when False => y : Integer;
      when True  => z : Boolean
   end case
end record;
```

can be interpreted as

$$(Integer \times Integer) + (Integer \times Boolean)$$

where the Boolean parameter b plays the role of the *left* or *right* tag.

Note C also has union types

but they are unsafe as no check is performed on field selection

Functions

In a language which allows functions to be treated as values, we need to be able to describe the type of a function, independently of its definition.

In Ada, defining `function f(x : Float) return Integer is …`

produces a function f whose type is

`function (x : Float) return Integer`

the name of the parameter is insignificant (it is a *bound name*) so this is the same type as `function (y : Float) return Integer`

In SML this type is written `Float → Int`

In Scala this type is written `Float => Int`

Functions and Procedures

A function with several parameters can be viewed as a function with one parameter which has a product type:

```
function (x : Float, y : Integer) return Integer
```

$$\text{Float} \times \text{Int} \rightarrow \text{Int}$$

In Ada, procedure types are different from function types:

```
procedure (x : Float, y : Integer)
```

whereas in Java a procedure is simply a function whose result type is *void*. In SML, a function with no interesting result could be given a type such as $\text{Int} \rightarrow ( )$ where $( )$ is the empty product type (also known as the *unit* type) although in a purely functional language there is no point in defining such a function.

Structural and Name Equivalence

At various points during type checking, it is necessary to check that two types are the same. What does this mean?

structural equivalence: two types are the same if they have the same structure: e.g. arrays of the same size and type, records with the same fields.

name equivalence: two types are the same if they have the same name.

Example: if we define

```
type A = array 1..10 of Integer;
type B = array 1..10 of Integer;
function f(x : A) return Integer is …
var b : B;
```

then f(b) is correct in a language which uses structural equivalence, but incorrect in a language which uses name equivalence.

Structural and Name Equivalence

Different languages take different approaches, and some use both kinds.

Ada uses name equivalence.
Triangle uses structural equivalence.
Haskell uses structural equivalence for types defined by *type* (these are viewed as new names for existing types) and name equivalence for types defined by *data* (these are algebraic datatypes; they are genuinely new types).

Structural equivalence is sometimes convenient for programming, but does not protect the programmer against incorrect use of values whose types accidentally have the same structure but are logically distinct.

Name equivalence is easier to implement in general, especially in a language with recursive types.

Recursive Types

Example: a list is either empty, or consists of a value (the *head*) and a list (the *tail*)

SML:

```
datatype List = Nil
              | Cons (Int * List)
```

```
Cons 2 (Cons 3 (Cons 4 Nil))
```

represents [2,3,4]

Abstractly: List = Unit + (Int × List)

In SML, the implementation uses pointers, but the programmer does not have to think in terms of pointers.

Recursive Types

Java:

```
class List {
   int head;
   List tail;
}
```

The Java definition does not mention pointers,
but we use the explicit null pointer **null** to represent the empty list.

Equivalence of Recursive Types

In the presence of recursive types, defining structural equivalence is more difficult.

We expect $\text{List} = \text{Unit} + (\text{Int} \times \text{List})$

and $\text{NewList} = \text{Unit} + (\text{Int} \times \text{NewList})$

to be equivalent, but complications arise from the (reasonable) requirement that $\text{List} = \text{Unit} + (\text{Int} \times \text{List})$

and $\text{NewList} = \text{Unit} + (\text{Int} \times (\text{Unit} + (\text{Int} \times \text{NewList})))$

should be equivalent.

It is usual for languages to avoid this issue by using name equivalence for recursive types, but recent research on co-inductive types show it is Possible and (sometimes) useful to have structural equivalence on recursive types

Other Practical Type System Issues

- Implicit versus explicit type conversions
  - Explicit ➔ user indicates (Ada, SML)
  - Implicit ➔ built-in (C int/char) -- coercions
- Overloading – meaning based on context
  - Built-in
  - Extracting meaning – parameters/context
- Polymorphism
- Subtyping

Coercions Versus Conversions

- When A has type **real** and B has type **int**, many languages allow coercion implicit in

  A := B

- In the other direction, often no coercion allowed; must use explicit conversion:
  - B := round(A); Go to integer nearest B
  - B := trunc(A); Delete fractional part of B

Explicit vs. Implicit conversion Autoboxing/Unboxing

- In Java 1.4 you had to write:

```
Integer x = Integer.valueOf(6);
Integer y = Integer.valueOf(2 * x.IntValue);
```

- In Java 1.5 you can write:

```
Integer x = 6;              //6 is boxed
Integer y = 2*x + 3;        //x is unboxed, 15 is boxed
```

  - Autoboxing wrap ints into Integers
  - Unboxing extract ints from Integers

Explicit vs. Implicit conversion Autoboxing/Unboxing

- Extending a language can imply difficult design compromises. In Java 1.5 we can write:

- Integer x = 3; (an integer object)
- int y = 3; (an integer)
- Integer z = 3; (an integer)
- .. x==y .. (true due to auto unboxing)
- .. y == z .. (true due to auto unboxing)
- .. x == z .. (false due to object comparisson)

- I.e. the convenience of autoboxing/unboxing leads to the == operator no longer being transitive
- Note: Not a problem in C# as autoboxing/unboxing is handled by the run-time system.

Polymorphism

Polymorphism describes the situation in which a particular operator or function can be applied to values of several different types. There is a fundamental distinction between:

- *ad hoc polymorphism*, usually called *overloading*, in which a single name refers to a number of unrelated operations.
    - Examples: + and static overloading of methods
- *bounded or Subtype polymorphism (inheritance polymorphism)*
- *parametric polymorphism (generics)*, in which the same computation can be applied to a range of different types which have structural similarities.

Most languages have some support for overloading.

Parametric polymorphism is familiar from functional programming, but less common (or less well developed) in imperative languages. Generics (or Parametric Polymorphism) has recently had a lot of attention in OO languages.

Parametric polymorphism (generics)

```
datatype 'a tree =
    INTERNAL of {left:'a tree,right:'a tree}
  | LEAF of {contents:'a}

fun tw(tree: `a tree, comb: `a*`a->'a) =
  case tree of
    INTERNAL{left,right} => comb(tw(left),tw(right))
  | LEAF{contents} => contents
```

Parametric polymorphism (generics)

```
public class List<ItemType>
{
   private ItemType[] elements;
   private int count;

   public void Add(ItemType element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public ItemType this[int index] {
      get { return elements[index]; }
      set { elements[index] = value; }
   }

   publ
      g
   }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);             // No boxing
intList.Add(2);             // No boxing
intList.Add("Three");       // Compile-time error

int i = intList[0];         // No cast required
```

Implementing generic types

- Type erasure, e.g:
  - <T extends Addable> T add(T a, T b) { … }
  - can be compiled, type-checked, and called the same way as:
  - Addable add(Addable a, Addable b) { … }

- Template:
- Apply the template to the provided template arguments. E.g calling template
  - <class T> T add(T a, T b) { … }
  - as add<int>(1, 2)
  - actual function int __add__T_int(int a, int b)

The Hindley-Milner Type inference Algorithm

Algorithm $\mathscr{W}$

$$\mathscr{W}(\bar{p}, f) = (T, \bar{f}), \text{ where}$$

(i) If f is x, then:

if λx_σ or *fix* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\sigma$;
if *let* x_σ is active in $\bar{p}$ then
$T = I, \bar{f} = x_\tau$
where $\tau = [\beta_i/\alpha_i]\sigma$, α_i are the generic variables of σ, and β_i are new variables.

(ii) If f is (de), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$, and $(S, \bar{e}_\sigma) = \mathscr{W}(R\bar{p}, e)$;
let $U = \mathscr{U}(S\rho, \sigma \to \beta)$, β new;
then $T = USR$, and $\bar{f} = U(((S\bar{d})\bar{e})_\beta)$.

(iii) If f is (*if* d *then* e *else* e'), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$ and $U_0 = \mathscr{U}(\rho, \iota_0)$;
let $(S, \bar{e}_\sigma) = \mathscr{W}(U_0R\bar{p}, e)$, and $(S', \bar{e}'_{\sigma'}) = \mathscr{W}(SU_0R\bar{p}, e')$;
let $U = \mathscr{U}(S'\sigma, \sigma')$;
then $T = US'SU_0R$, and
$\bar{f} = U((\textit{if } S'SU_0\bar{d} \textit{ then } S'\bar{e} \textit{ else } \bar{e}')_\sigma)$.

(iv) If f is $(\lambda x \cdot d)$, then:

let $(R, \bar{d}) = \mathscr{W}(\bar{p} \cdot \lambda x_\beta, d)$, where β is new;
then $T = R$, and $\bar{f} = (\lambda x_{R\beta} \cdot \bar{d}_\rho)_{R\beta \to \rho}$.

(v) If f is (*fix* $x \cdot d$), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p} \cdot \textit{fix } x_\beta, d)$, β new;
let $U = \mathscr{U}(R\beta, \rho)$;
then $T = UR$, and $\bar{f} = (\textit{fix } x_{UR\beta} \cdot U\bar{d})_{UR\beta}$.

(vi) If f is (*let* $x = d$ *in* e), then:

let $(R, \bar{d}_\rho) = \mathscr{W}(\bar{p}, d)$;
let $(S, e_\sigma) = \mathscr{W}(R\bar{p} \cdot \textit{let } x_\rho, e)$;
then $T = SR$, and $\bar{f} = (\textit{let } x_{S\rho} = S\bar{d} \text{ in } \bar{e})_\sigma$. ∎

- First used in SML
- A Theory of Type Polymorphism in Programming
  - Robin Milner (1977)
- Algoritmn basically builds and solves equations over type expressions
- Now in use in:
  - Haskell, C#, F#, Visual Basic .Net 9.0

Subtyping

The interpretation of a type as a set of values, and the fact that one set may be a subset of another set, make it natural to think about when a value of one type may be considered to be a value of another type.

Example: the set of integers is a subset of the set of real numbers. Correspondingly, we might like to consider the type Integer to be a *subtype* of the type Float. This is often written Integer <: Float.

The subtype relation enjoys the following properties:
X <: X (indempotent)
X<:Y and Y<:Z then X<:Z (transitivity)

Different languages provide subtyping in different ways, including (in some cases) not at all. In object-oriented languages, subtyping arises from inheritance between classes.

Subtyping and Polymorphism

```
abstract class Shape {
  abstract float area( ); }
```

the idea is to define several classes of Shape, all of which define the area function

```
class Square extends Shape {
  float side;
  float area( ) {return (side * side); } }
```

Square <: Shape

```
class Circle extends Shape {
  float radius;
  float area( ) {return ( PI * radius * radius); } }
```

Circle <: Shape

Objects can be thought of as (extendible) records of fields and methods.
That is why Square <: Shape and Circle <: Shape

Subtyping and Polymorphism

```
float totalarea(Shape[] s)  {
   float t = 0.0;
   for (int i = 0; i < s.length; i++) {
      t = t + s[i].area( ); };
   return t;
}
```

totalarea can be applied to any array whose elements are subtypes of Shape. (This is why we want Square[] <: Shape[] etc.)

This is an example of a concept called *bounded polymorphism*.

Subtyping for Product Types

The rule is:

if $A <: T$ and $B <: U$ then $A \times B <: T \times U$

This rule, and corresponding rules for other structured types, can be worked out by following the principle:

$T <: U$ means that whenever a value of type U is expected, it is safe to use a value of type T instead.

What can we do with a value *v* of type $T \times U$?

- use *fst(v)* , which is a value of type T
- use *snd(v)* , which is a value of type U

If *w* is a value of type $A \times B$ then *fst(w)* has type A and can be used instead of *fst(v)*. Similarly *snd(w)* can be used instead of *snd(v)*. Therefore *w* can be used where *v* is expected.

Subtyping for Function Types

Suppose we have $f : A \to B$ and $g: T \to U$ and we want to use f in place of g.

It must be possible for the result of f to be used in place of the result of g, so we must have B <: U.

It must be possible for a value which could be a parameter of g to be given as a parameter to f, so we must have T <: A.

Therefore: if T <: A and B <: U then $A \to B$ <: $T \to U$

Compare this with the rule for product types, and notice the *contravariance*: the condition on subtyping between A and T is the other way around.

Correctness of Type Systems

How does a language designer (or a programmer) know that correctly-typed programs really have the desired run-time properties?

To answer this question we need to see how to specify type systems, and how to prove that a type system is *sound*.

To do this we can use techniques similar to those from SOS

To prove soundness we also need to specify the *semantics* (meaning) of programs - what happens when they are run.

So studying types will lead us to a deeper understanding of the meaning of programs.

Connection with Semantics

- Type system is part of the static semantics
  - Static semantics: the well-formed programs
  - Dynamic semantics: the execution model
- Safety theorem: types predict behaviour.
  - Types describe the states of an abstract machine model.
  - Execution behaviour must cohere with these descriptions.
  - **Theorem:** If Γ |- E:τ and E$\rightarrow$ E' then Γ |- E':τ
  - See Theorem 13.9 p. 196 in Transitions and Trees
- Thus a type is a specification and a type checker is a theorem prover.
- Type checking is the most successful formal method!
  - In principle there are limits.
  - In practice there is no end in sight.
- Examples:
  - Using types for low-level languages, say inside a compiler.
  - Extending the expressiveness of type systems for high-level languages.

Summary

- Static typing is important
- Type system has to be an integral part of the language design
- There are a lot of nitty-gritty decisions about primitive data types
- Composite types are best understood independently of language manifestation to ensure correctness of implementation
- Type systems can (and should) be formalised