Lecture 16

CIS 341: COMPILERS

Announcements

- Midterm Exam:
  - Graded and entered
  - Pick up exams from Levine 308 Laura Fox's office

- Project 4 is on the course web pages
  - Due on Thursday, March 21st.
  - As usual, start early and ask questions if you get stuck
  - Note: revised version of LL intermediate representation to be more compliant with "real" LLVM IR

Beyond describing "structure"… describing "properties"

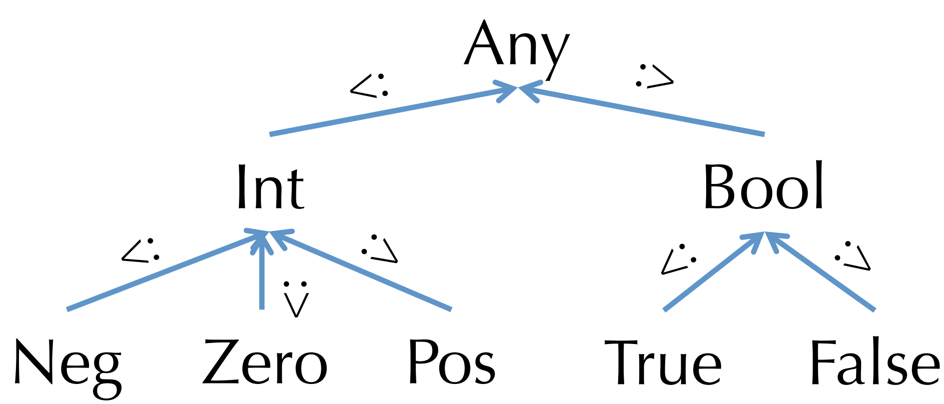
Types as sets

Subsumption

TYPES, MORE GENERALLY

Subtyping and Upper Bounds

- If we think of types as sets of values, we have a natural inclusion relation: Pos $\subseteq$ Int
- This subset relation gives rise to a *subtype* relation: Pos <: Int
- Such inclusions give rise to a *subtyping hierarchy*:



- Given any two types T_1 and T_2, we can calculate their *least upper bound* (LUB) according to the hierarchy.
  - Example: LUB(True, False) = Bool, LUB(Int, Bool) = Any
  - Note: might want to add types for "NonZero", "NonNegative", and "NonPositive" so that set union on values corresponds to taking LUBs on types.

"If" Typing Rule Revisited

- For statically unknown conditionals, we want the return value to be the LUB of the types of the branches:

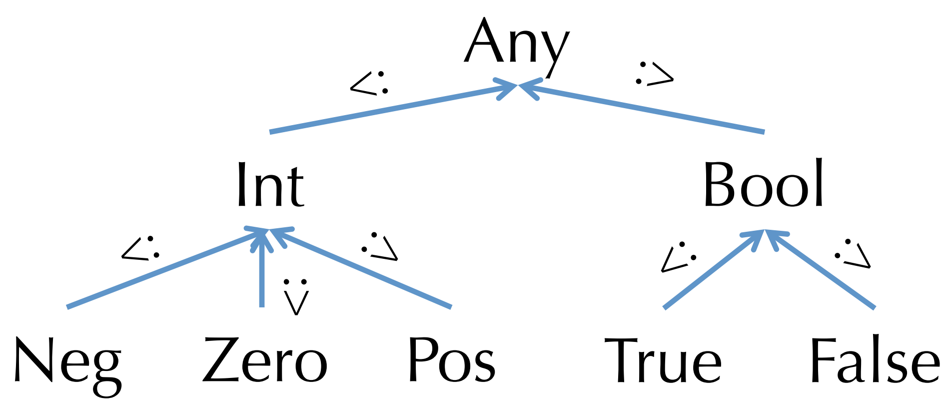
IF-BOOL

$$\frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \quad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \text{LUB}(T_1, T_2)}$$

- Note that LUB(T_1, T_2) is the most precise type (according to the hierarchy) that is able to describe any value that has either type T_1 or type T_2.
- In math notation, LUB(T1, T2) is sometimes written $T_1 \vee T_2$
- LUB is also called the *join* operation.

Subtyping Hierarchy

- A *subtyping hierarchy*:



- The subtyping relation is a *partial order*:
  - Reflexive: T <: T for any type T
  - Transitive: T_1 <: T_2 and T_2 <: T_3 then T_1 <: T_3
  - Antisymmetric: It T_1 <: T_2 and T_2 <: T_1 then $T_1 = T_2$

Soundness of Subtyping Relations

- We don't have to treat *every* subset of the integers as a type.
  - e.g., we left out the type NonNeg

- A subtyping relation $T_1 <: T_2$ is *sound* if it approximates the underlying semantic subset relation.
- Formally: write $[\![T]\!]$ for the subset of (closed) values of type T
  - i.e. $[\![T]\!] = \{v \mid \vdash v : T\}$
  - e.g. $[\![Zero]\!] = \{0\}$, $[\![Pos]\!] = \{1, 2, 3, \ldots\}$

- If $T_1 <: T_2$ implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then $T_1 <: T_2$ is sound.
  - e.g. Pos <: Int is sound, since $\{1,2,3,\ldots\} \subseteq \{\ldots,-3,-2,-1,0,1,2,3,\ldots\}$
  - e.g. Int <: Pos is not sound, since it is *not* the case that $\{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \subseteq \{1,2,3,\ldots\}$

Soundness of LUBs

- Whenever you have a sound subtyping relation, it follows that:

$$[\![\mathrm{LUB}(T_1, T_2)]\!] \supseteq [\![T_1]\!] \cup [\![T_2]\!]$$

  - Note that the LUB is an over approximation of the "semantic union"
  - Example: $[\![\mathrm{LUB}(\mathrm{Zero}, \mathrm{Pos})]\!] = [\![\mathrm{Int}]\!] = \{\ldots,-3,-2,-1,0,1,2,3,\ldots\} \supseteq$
    $\{0,1,2,3,\ldots\} = \{0\} \cup \{1,2,3,\ldots\} = [\![\mathrm{Zero}]\!] \cup [\![\mathrm{Pos}]\!]$

- Using LUBs in the typing rules yields sound approximations of the program behavior (as if the IF-B rule).
- It just so happens that LUBs on types <: Int correspond to +

ADD

$$\frac{E \vdash e_1 : T_1 \qquad E \vdash e_2 : T_2 \qquad T_1 <: \mathrm{Int} \qquad T_2 <: \mathrm{Int}}{E \vdash e_1 + e_2 : T_1 \vee T_2}$$

Subsumption Rule

- When we add subtyping judgments of the form T <: S we can uniformly integrate it into the type system generically:

SUBSUMTION

$$\frac{E \vdash e : T \qquad T <: S}{E \vdash e : S}$$

- Subsumption allows any value of type T to be treated as an S whenever T <: S.
- Adding this rule makes the search for typing derivations more difficult – this rule can be applied anywhere, since T <: T.
  - But careful engineering of the typing system can incorporate the subsumption rule into a deterministic algorithm.

Downcasting

- What happens if we have an Int but need something of type Pos?
  - At compile time, we don't know whether the Int is greater than zero.
  - At run time, we do.

- Add a "checked downcast"

$$\frac{E \vdash e_1 : \text{Int} \qquad E, x : \text{Pos} \vdash e_2 : T_2 \qquad E \vdash e_3 : T_3}{E \vdash \text{ifPos } (x = e_1)\ e_2 \text{ else } e_3 : T_2 \vee T_3}$$

- At runtime, ifPos checks whether e_1 is > 0. If so, branches to e_2 and otherwise branches to e_3.
- Inside the expression e_2, x is the name for e_1's value, which is known to be strictly positive because of the dynamic check.
- Note that such rules force the programmer to add the appropriate checks
  - We could give integer division the type: Int -> NonZero -> Int

SUBTYPING OTHER TYPES

Extending Subtyping to Other Types

- What about subtyping for tuples?
  - Intuition: whenever a program expects something of type $S_1 * S_2$, it is sound to give it a $T_1 * T_2$.
  - Example: (Pos * Neg) <: (Int * Int)

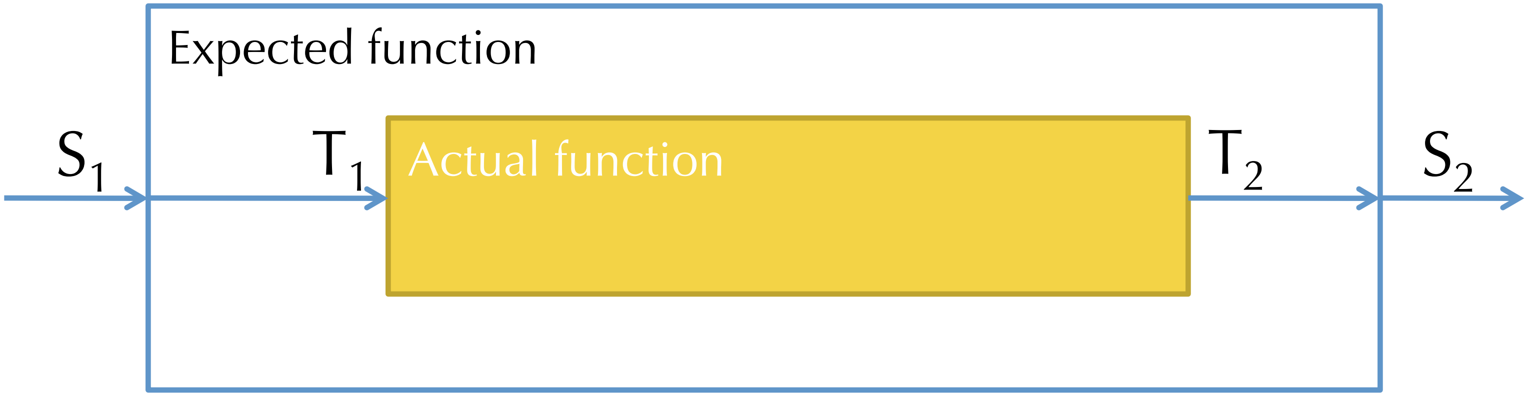
$$\frac{T_1 <: S_1 \quad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

- What about functions?

- When is $T_1 \rightarrow T_2 <: S_1 \rightarrow S_2$?

Subtyping for Function Types

- One way to see it:



- Need to convert an S1 to a T1 and T2 to S2, so the argument type is *contravariant* and the output type is *covariant*.

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \to T_2) <: (S_1 \to S_2)}$$

Immutable Records

- Record type: $\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}$
  - Each lab_i is a label drawn from a set of identifiers.

RECORD

$$\frac{E \vdash e_1 : T_1 \qquad E \vdash e_2 : T_2 \qquad \ldots \qquad E \vdash e_n : T_n}{E \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

PROJECTION

$$\frac{E \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{E \vdash e.lab_i : T_i}$$

Immutable Record Subtyping

- Depth subtyping:
  - Corresponding fields may be subtypes

DEPTH

$$\frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \ldots\ ;\ lab_n{:}U_n\}}$$

- Width subtyping:
  - Subtype record may have *more* fields:

WIDTH

$$\frac{m \leq n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_m{:}T_m\}}$$

Immutable Record Subtyping (cont'd)

- Width subtyping assumes an implementation in which order of fields in a record matters:

  {x:int; y:int} ≠ {y:int; x:int}

- But: {x:int; y:int; z:int} <: {x:int; y:int}
  - Implementation: a record is a struct, subtypes just add fields at the *end* of the struct.

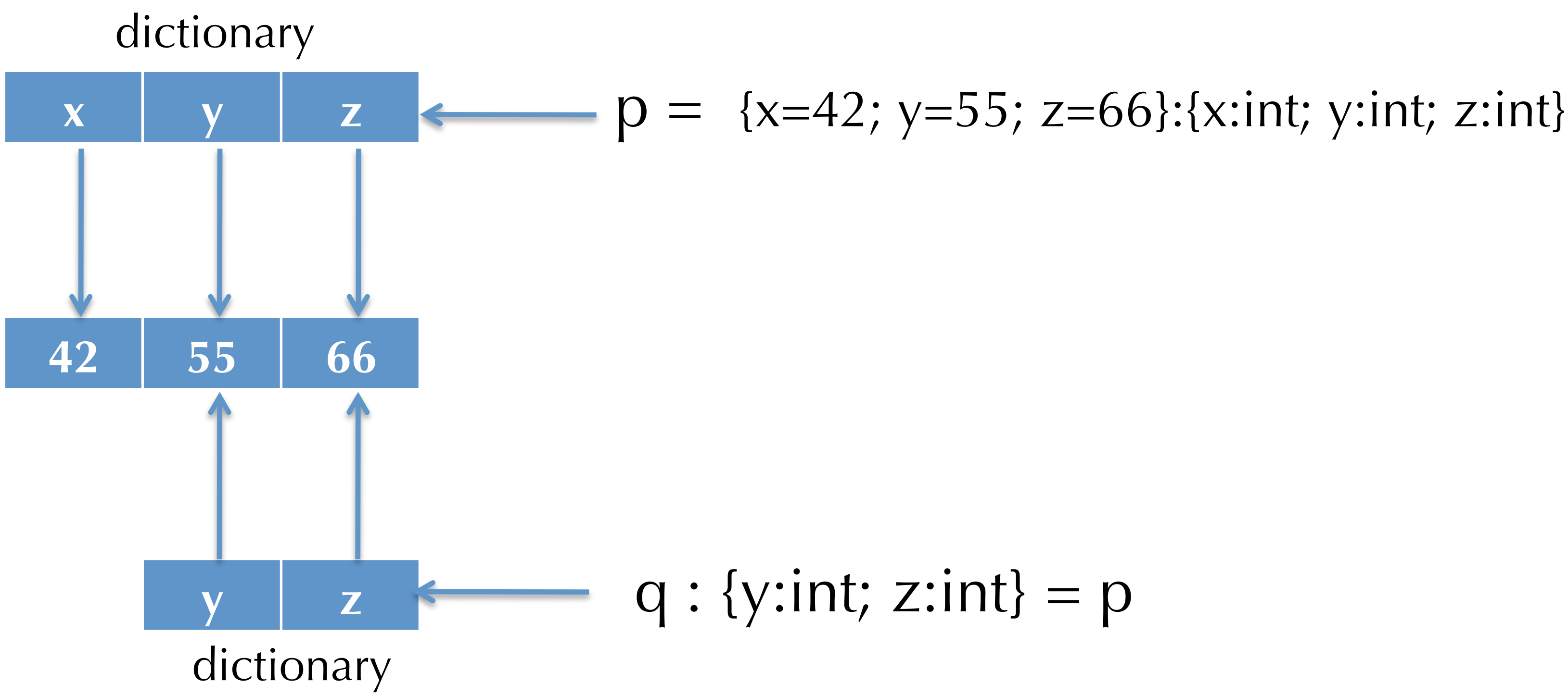
- Alternative: allow permutation of record fields:

  {x:int; y:int} = {y:int; x:int}
  - Implementation: compiler sorts the fields before code generation.
  - Need to know *all* of the fields to generate the code
- Permutation is not directly compatible with width subtyping:

  {x:int; z:int; y:int} = {x:int; y:int; z:int} </: {y:int; z:int}

If you want both:

- If you want permutability & dropping, you need to either copy (to rearrange the fields) or use a dictionary like this:



Subtyping and References

- What is the proper subtyping relationship for references and arrays?

- Suppose we have NonZero as a type and the division operation has type: Int -> NonZero -> Int
  - Recall that NonZero <: Int
- Should (NonZero ref) <: (Int ref) ?
- Consider this program:

```
Int bad(NonZero ref r) {
  Int ref a = r;   (* OK because (NonZero ref <: Int ref*)
  a := 0;          (* OK because 0 : Zero <: Int *)
  return (42 / !r) (* OK because !r has type NonZero *)
}
```

Mutable Structures are Invariant

- Covariant reference types are unsound
  - As demonstrated in the previous example
- Contravariant reference types are also unsound
  - i.e. If T_1 <: T_2 then ref T_2 <: ref T_1 is also unsound
  - Exercise: construct a program that breaks contravariant references.

- Moral: Mutable structures are invariant:

  T_1 ref <: T_2 ref implies $T_1 = T_2$

- Same holds for arrays, OCaml-style mutable records, object fields, etc.
  - Note: Java and C# get this wrong. They allows covariant array subtyping, but then compensate by adding a dynamic check on *every* array update!

Another Way to See It

- We can think of a reference cell as an immutable record (object) with two functions (methods) and some hidden state:

  T ref ≃ {get: unit -> T; set: T -> unit}

  – get returns the value hidden in the state.
  – set updates the value hidden in the state.

- When is T ref <: S ref?
- Records are like tuples: subtyping extends pointwise over each component.
- {get: unit -> T; set: T -> unit} <: {get: unit -> S; set: S -> unit}
  – get components are subtypes: unit -> T <: unit -> S
    set components are subtypes: T -> unit <: S -> unit
- From get, we must have T <: S (covariant return)
- From set, we must have S <: T (contravariant arg.)
- From T <: S and S <: T we conclude T = S.

STRUCTURAL VS. NOMINAL TYPES

Structural vs. Nominal Typing

- Is type equality / subsumption defined by the *structure* of the data or the *name* of the data?
- Example 1: type abbreviations (OCaml) vs. "newtypes" (a la Haskell)

```
(* OCaml: *)
type cents = int     (* cents = int in this scope *)
type age = int

let foo (x:cents) (y:age) = x + y
```

```
(* Haskell: *)
newtype Cents = Cents Integer  (* Integer and Cents arr
                                isomorphic, not identical. *)
newtype Age = Age Integer

foo :: Cents -> Age -> Int
foo x y = x + y                (* Ill typed! *)
```

- Type abbreviations are treated "structurally"
  Newtypes are treated "by name"

Nominal Subtyping in Java

- In Java, Classes and Interfaces must be named and their relationships *explicitly* declared:

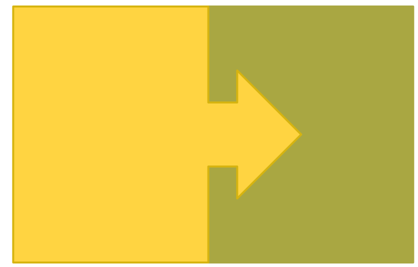
```
(* Java: *)
interface Foo {
  int foo();
}

class C {       /* Does not implement the Foo interface */
  int foo() {return 2;}
}

class D implements Foo {
  int foo() {return 341;}
}
```

- Similarly for inheritance: programmers must declare the subclass relation via the "**`extends`**" keyword.
  - Typechecker still checks that the classes are structurally compatible

MODULARITY & ABSTRACTION

Modular Programming

- Programs are typically composed of many modules.
  - Separate compilation – scalable to millions of lines
  - Code reuse – libraries, sharing
  - Namespace management
  - Encapsulation – hiding complexity
  - Abstraction & abstract data-types
  - Security



- What is a module?
  - A collection of named, related values and types
  - Definitions (partially) hidden from the outside

- Examples: Java classes & packages, C++ classes, Modula-3 modules, SML/Ocaml structures & functors, CLU clusters, C source files, …

Separate Compilation

- Program is made of several *compilation units*
  - Independent inputs to the compiler

- Avoids needing to recompile the whole program for every change
- Code is more reusable (libraries)
- Examples:
  - C: .c files / Java: .java files / OCaml: .ml files

- For building a whole program out of compilation units:
- Need to know how to reference values in other units
  - Solution: namespaces + linking
- Need to know datatype sizes (for code generation) or types (for type safety)
  - Solution: interfaces (C: .h files / Java: .class files / OCaml: .mli files)

Namespaces

- In C and FORTRAN: all global identifiers are visible everywhere
- Problem:
  - Can't have two global variables or functions with the same name
  - (Also, linker doesn't type check)
- Solutions:
  - C++, Java qualified identifiers: C.x or $P_1.P_2.P_3.C.x$ (where C is a class name)
  - Modula-3, OCaml: qualified identifiers + renaming
  - Java, Modula-3, OCaml: link-time type checking

- Wrinkle: object code formats typically have a flat name space
  - Need to *mangle* qualified identifiers
  - e.g. C++: `int C::f(int x)` becomes `f__1Ci`

Linking

Input: File f1.c

```
extern int x;

void main() {
  printf("%d", x);
}
```

File f2.c

```
int x = 341;
```

compiles to asm:	f1.s	f2.s
assembles to obj:	f1.o	f2.o
linker	a.out	

- *Problem:* compiler can't generate code to access variable x because its address is unknown.
- *Solution:* Generate placeholder reference to x in f1.s, generate definition of x in f2.s, linker patches the files together, replacing placeholders in f1.s with actual value from f2.s
  - Exact mechanism depends on linker/OS object file format

Encapsulation

- It's often useful to hide some information contained in a module.
- Example:

```
String[] names;        // should be hidden
String[] passwords;    // should be hidden
bool check_password(String n, String p) {
  int j = 0;
  while (j < names.length) {
    if (names[j] == n & passwords[j] == p)
      return true;
    j = j + 1;
  }
  return false;
}
```

- Encapsulation can protect a module's data from tampering
  - Good software engineering practices rely on encapsulation.

Encapsulation Mechanisms

- Fundamentally, need a way to indicate which identifiers should be exported from a module.
- C++/Java: "public" vs. "private" qualifiers:

```
class PWChecker {
    private String[] names;        // should be hidden
    private String[] passwords;    // should be hidden
    public bool check_password(String n, String p) {…} }
```

- ML / Modula-3: separate interfaces (omit hidden identifiers):

```
module type PWChecker = sig
  val check_password : String * String -> bool
  (* Note: no declaration for names or password *)
end
```

- C: "static" qualifier

```
static int check_password(char *n, char *p)
```

Modules as Records

- Records (or structs) bundle values together, mapping names to values.
- Modules *also* bundle values together…
  - Except that modules are computed a *load* time
  - They are (usually) 2nd class (e.g. modules cannot be passed arguments to functions). (OCaml v. 3.12 has support for first-class modules.)
- But… module interfaces look like record types:

```
module PWC = struct
  let names : string array = …
  let passwords : string array = …
  let check_password (n:string, p:string):bool = …
  let is_name (n:string):bool = …
end :
sig
  val check_password : string * string -> bool
  val is_name : string -> bool
end
```

More on Encapsulation

- Example: sets of integers
  - operations: empty, insert, has

In OCaml:

```
type intset = int list
let empty = []
let insert i s = i::s
let rec has i s =
  match s with
  | [] -> false
  | (j::rest) -> if i == j then true else has i rest
```

- Problem: can't write down the interface unless
  - We expose the implementation of intset as equal to int list
  - Or, alternatively, we expose intset as an *abstract* type

Alternate Implementation of Integer Sets

- Consider this alternate implementation of integer sets as binary search trees:

```
type intset = Leaf | Node of intset * int * intset
let empty = Leaf
let rec insert i s =
  match s with
  | Leaf -> Node(Leaf, i, Leaf)
  | Node(left, j, right) ->
     if i = j then s else
     if i < j then Node(insert i left, j, right)
       else Node(left, j, insert i right)
let rec has i s =
  match s with
  | Leaf -> false
  | Node(left, j, right) ->
     if i = j then true else
       if i < j then has i left
       else has i right
```

Problem of Exposed Representations

- If we expose the representation type:
  `intset = Leaf | Node of intset * int * intset`
- Client code can break the representation invariant that `intset` is a search tree.
  - Concretely, a client could construct a value of type `intset` such as:
    `let bad = Node(Leaf, 10, Node(Leaf, 5, Leaf))`
  - Note that "`has 5 bad`" will return `false`, even though 5 appears as a node in the tree.

- We need encapsulation of values *exported* from the module, not just components inside the module.
  - Only way to create `insets` is via the operations in the interface.

Abstract Data Types

- Key idea: *abstract type*
  - An identifier representing an *unknown* type

- Abstract Data Type is
  - A type identifier (possibly parameterized) +
  - Declared operations on that type +
  - Concrete type definition (a representation) +
  - Concrete implementation of the operations

  (first two: Interface; last two: Implementation)

- IntSet interface in OCaml:

```
module type IntSet = sig
  type intset          (* Note: no type definition *)
  val empty : intset
  val insert : int -> intset -> intset
  val has : int -> intset -> bool
end
```

IntSet example in OCaml

```
module IntSet1 : IntSet = struct
  type intset = int list
  let empty = []
  let insert i s = i::s
  let rec has = …
end
```

This signature ascription *seals* the modules with an abstract type, hiding the representation of intset.

```
module IntSet2 : IntSet = struct
  type intset = Leaf | Node of intset * int * intset
  let empty = Leaf
  let rec insert i s = …
  let rec has = …
end
```

Implementing Abstract Types

- Representation of the abstract type is hidden from code other than the implementation itself
  - CLU, Ada, Modula-3, ML
- Because external code doesn't know representation, it can't violate the abstraction boundary
  - e.g. break representation invariants
- Positive: The same interface can be reimplemented multiple ways.
- Negative: Compiler doesn't know representation either
  - When compiling external code it must use level of indirection
  - No stack allocation of abstract types

IntSet Example in Java

```
public interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
}

class IntSet1 implements IntSet {
    private List<Integer> rep;          // note hidden state

    public IntSet1() {
      this.rep = new LinkedList<Integer>();
    }

    public IntSet1 insert(int i) {
      rep.add(new Integer(i));
      return this;
    }

    public boolean has(int i) {
      return rep.contains(new Integer(i));
    }

}
```

Classes in C++/Java

- Classes have private/public visibility qualifiers that hide part of the object.
- A class is a *partially* abstract type
  - (Note: do not confuse with Java's 'abstract' keyword)

- Interface file declares the representation
  - Method code is (mostly) hidden from the outside

- Positive: This mechanism allows external code to know how much space each object takes while still providing encapsulation
  - Objects can be stack allocated (good for cache coherence/performance)
- Negative: Change to representation can require complete recompilation, even of external code
  - C++ is notoriously slow to compile

IntSet example in C

- intset.h:

```
struct intset;
extern struct intset *empty;
struct intset *insert(int i, struct intset *s);
int has(int i, struct intset *s);
```

- intset.c:

```
#include "intset.h"

struct intset {struct intset *left;  int val; struct
   intset *right; };

struct intset *empty = NULL;

struct intset *insert(int i, struct intset *s) {…}
int has(int I, struct intset *s) {…}
```

No Abstraction in C

- C provides hiding/encapsulation but no abstraction.
- (Unchecked) Casts allow any client code to violate the representation invariants of the module.