Lecture 19: Modules

Specification

- Definitions should not depend on implementation details.
- Constants, types, variables, and operations
  - Behavior must be specified abstractly
    - pre- and postconditions
    - Axioms and rules: \( \text{pop}(\text{push}(S,x)) = S \)
      \[ \text{if not empty}(S) \text{ then } \text{push}(\text{pop}(S), \text{top}(S)) = S \]
- Details of implementation provided elsewhere
- Data + Operations (+ equations) = Algebra

Implementation

- Details of representation and implementation of operations.
- Details not accessible outside unit.

Modules

- Reusable modules:
  - Separate, but not independent compilation
  - Maintain type checking
  - Control over export and import of names
Representation Independence & Information Hiding

• Choice of representation doesn't affect computation. E.g., rationals.

• If represent new type in terms of old:
  - Rep may have values not corresponding to new type. E.g., (3,0)
  - Rep may have several values corresponding to same abstract value. E.g., (1,2) and (2,4).
  - Values of new type can be confused w/values of rep type.

Modula 2

• Similar to Ada except
  - no generics
  - no “private” section

• Require all private types to take same amount of space – a pointer

Last Time

• Simula 67
• Clu
• Ada

DEFINITION MODULE stackMod;
IMPORT element FROM elementMod;
  TYPE stack;
  PROCEDURE make_empty (VAR S : stack);
  PROCEDURE push (VAR S : stack; X : element);
  PROCEDURE pop (VAR S : stack; X : element);
  PROCEDURE empty (S : stack): BOOLEAN;
END stackMod.

IMPLEMENTATION MODULE stackMod;
  TYPE stack = POINTER TO RECORD
    space   : array[1..length] of element;
    top     : INTEGER;
  END;
  PROCEDURE make_empty (VAR S : stack);
    BEGIN
      S^.top := 0;
    END make_empty ;

    ... (* can be start-up code too to initialize *)
END stackMod;
Ada vs. Modula 2

- Representations fairly similar
  - can import from other units (modules or packages) and export items to other units.
- For external representations not much difference.
- Private types in Ada vs opaque types in Modula.

Ada vs. Modula 2

- Use of opaque types require Pointer types
- Representation changes
  - in Ada forces recompilation of user programs
  - Not in Modula 2
- Internal reps of ADT's almost identical.
- Ada more flexible via generic routines -
  - can parameterize on types and sizes
  - Create new instances of packages

Easier if Uniform Reps

- LISP, Scheme, ML, Haskell, Clu, Eiffel, and Java have uniform reps for values so can share same code.
- Non-uniform representations:
  - Ada requires different implementation, but still type-checks statically.
  - C++ type-checks only when instantiated
- Automatic boxing and unboxing now helps with primitive types in Java and C#.

Modules in Haskell

- Module:
  - Control namespace
  - define abstract data types.
  - No nesting of modules
- Examples: PcfLexer.hs, ParsePCF.hs
  - Name starts with cap
  - list functions and types (including constructors) to be exported.
  - Make more abstract by leaving off constructors
  - if no export list, then all exported
Importing

• Module must be explicitly imported
  - Can narrow more by listing items to be imported
    • import PcfLexer(getTokens, Token(ID,NUM))
  - If name conflicts, can “import qualified” and access:
    • PcfLexer.getTokens
  - “hiding” clause can hide imported items.
  - “as” clause can rename imported features

ML

• datatype like Haskell data declaration.
• No information hiding

ML Modules

• SML has two sub-languages:
  - Core — programming in the small
    • details of types and expressions
  - Modules — programming in the large — architecture.
    • group defs of types & expressions into units w/interfaces
  • Separate interfaces (signatures) from implementations (structures)
    - Explicitly typed!
    - Can reveal implementations if want.

Signature

signature INTSTACKSIG =
  sig
    type intstack;
    exception stackUnderflow;
    val emptyStk: intstack;
    val push: int -> intstack -> intstack;
    val pop: intstack -> intstack;
    val top: intstack -> int;
    val IsEmpty: intstack -> bool;
  end;
Structure

structure IntStack: INTSTACKSIG =
struct
    type intstack = int list;
    exception stackUnderflow;

    val emptyStk = [];
    fun push (e:int) (s:intstack) = (e::s);
    fun pop [] = raise stackUnderflow
        | pop (e::s) = s;
    ...
    fun extra ... (* not visible outside *)
end;

Accessing Structure

• IntStack.push 12 IntStack.emptyStk
• open IntStack;
push 12 emptyStk;
• Considered bad style to open outside structure.
• Rather than open, rename:
  - val push = IntStack.push;
  - val emptyStk = IntStack.emptyStk;

Ascription

• structure IntStack: INTSTACKSIG = ...
  - lets type definitions escape - transparent
  - hides extra features
• structure IntStack:> INTSTACKSIG = ...
  - also hides type definitions - opaque
• Can further restrict structure to create “views” by giving new name and signature
  - structure ResStack:> RESSTACKSIG = IntStack

More ML Modules

• Modules may be nested -- helping modules
• Functors: Modules parameterized by other modules.
• Supports code reuse -- apply to many different structures
Functor Examples

signature EQ =
  sig
    type t
    val eq : t * t -> bool
  end;

functor PairEQ(P : EQ) : EQ = struct
  type t = P.t * P.t
  fun eq((x,y),(u,v)) = P.eq(x,u) andalso P.eq(y,v)
end;

structure IntEq : EQ = struct
  type t = int
  val eq : t*t->bool = (op =)
end;

structure IntPairEQ : EQ = PairEQ(IntEq);

Module Languages

- Signatures like types
  - Describe families of structures
  - Make functor argument specification possible
  - But ... components can themselves be types
- Structures like records
  - But ... can contain types
- Complications:
  - Opacity sometimes requires “sharing constraints”
  - force types in parameters to match up -- omit details

Evaluating ML Modules

- Limitations:
  - Functors are first-order only
    * can't be applied to or return functor
  - Structures & functors are second-class values
    * Compile-time structures, can't be constructed or stored at run-time.

Key Features of ADT’s

- Encapsulation of all features in one place
- Information hiding -- explicit control over what imported and exported.
- Generally separate specification and implementation (except in Clu)
  - Separately compiled (except in ML)
- Ada, Clu, and ML provide parameterized modules.
Subtyping: Making it easier to reuse code!

- Can be added to non-DD languages
- Matching structures and signatures similar but more restricted.
- Provides support for using values from new types in old (unexpected) contexts

Subtype Polymorphism

S is a subtype of T, written $S <: T$, iff a value of type S can be used in any context expecting a value of type T, i.e., $S$ can masquerade as a $T$.

Subsumption: $e: S \& S <: T \Rightarrow e: T$. 

Immutable Records

Records without field update (like Haskell/ML):

Sandwich = { bread: BreadType;
              filling: FoodType }

s: Sandwich = { bread = rye;
                filling = pastrami }

Only operation is extracting field:

... s.filling ...

Specializing Record Types

CheeseSandwich = { bread: BreadType;
                   filling: CheeseType;
                   sauce: SauceType }

c_s: CheeseSandwich = { bread = white;
                       filling = cheddar;
                       sauce = mustard }

Subtyping Immutable Records

If r: \{ l_i : T_i \}_{i=k} then expect r. l_i : T_i

When is \{ l_i : T_i' \}_{i=n} <: \{ l_i : T_i \}_{i=k} ?

Suppose r': \{ l_i : T_i' \}_{i=n}

When can r' masquerade as elt of \{ l_i : T_i \}_{i=k} ?

Need r'. l_i : T_i.

Masquerading

\{ l_i : T_i' \}_{i=n} <: \{ l_i : T_i \}_{i=k}

iff

k \leq n and for all 1 \leq i \leq k, \ T'_i <: T_i.