Notes on specifying user defined types
Recall how we divide the universe of values into types.

Note similarities between PairV and ConsV.

Almost $\frac{1}{2}$ of the language is devoted to Pairs and Lists.
Data as Heap Pointers

data Value
  = IntV Int
  | PairV Addr
  | CharV Char
  | BoolV Bool
  | ConsV Addr
  | NilV

What distinguishes PairV, ConsV, and NilV?

• They have different names
• They point to consecutive blocks in the heap of different sizes.
Generic Constructors

```
data Value
    = IntV Int
    | PairV Addr
    | CharV Char
    | BoolV Bool
    | ConsV Addr
    | NilV

A constructor with N arguments, starting at Addr in Heap with name Cname
```
What operations?

• Construction
  – cons, nil, pair

• Selection
  – head, tail, fst, snd

• Predicate
  – null
Expressions

data Exp
    = While Exp Exp
    ...
    | Bool Bool
    | If Exp Exp Exp
    ...
    | Pair Exp Exp
    | Fst Exp
    | Snd Exp
    | Cons Exp Exp
    | Nil
    | Head Exp
    | Tail Exp
    | Null Exp

data Exp
    = While Exp Exp
    ...
    | Bool SourcePos Bool
    | If Exp Exp Exp
    ...
    | At Exp SourcePos [Exp]
    | Lambda SourcePos [Vname] Exp
    | Construction SourcePos Cname [Exp]
    | Selection SourcePos Cname Int Exp
    | Predicate SourcePos Cname Exp
data Typ
  = TyVar String
  | TyFun [Typ] Typ
  | TyPair Typ Typ
  | TyList Typ
  | TyCon String

data Typ
  = TyVar String
  | TyFun [Typ] Typ
  | TyCon String [Typ]

intT = TyCon "Int" []
charT = TyCon "Char" []
boolT = TyCon "Bool" []
stringT = tylist charT
typair x y = TyCon "Pair" [x,y]
tylist x = TyCon "List" [x]
What operations?

• Construction
  – (cons a b), nil, (pair a b)
  – (#cons a b), (# nil), (#pair a b)

• Selection
  – (head x), (tail x), (fst x), (snd x)
  – (!cons 0 x), (!cons 1 x)
  – (!pair 0 x), (!pair 1 x)

• Predicate
  – (null x), (@not (null x))
  – (?nil x), (?cons x)
run state (Construction _ c es) = do { (vals,state2) <- interpList vars state es 
    ; let count = length es 
        (addr,state3) = allocate count vals state2 
    ; return(ConV c count addr,state3) } 

(#node 3 'x' (#leaf) (#leaf)) 

   c ___________                     es
Semantics Selection

run state (term@(Selection p c n e)) =
  do { (v, state2) <- interpE vars state e
        ; case v of
            (ConV d m addr)
                | c==d && n<m
                ->  return(access (addr+n) state2, state2)
            (ConV d m addr) | not(c==d) -> error ...
            (ConV d m addr) | not(n<m)  -> error ...
            other -> error ("Non Construction in Selection")}

(!pair 0 (@ f 5))     -- this is “fst”
C   n    e
run state (term@(Predicate p c e)) =
  do { (v,state2) <- interpE vars state e
       ; case v of
         (ConV d m addr)
           | c==d -> return(BoolV True,state2)
           (ConV d m addr) -> return(BoolV False,state2)
         other -> error ("Non construction in Predicate")
   }

(Cons (@append x y))
Some samples

(global nil [a] (# nil))
(fun head h (x [h]) (!cons 0 x))
(fun tail [a] (x [a]) (!cons 1 x))
(fun fst a (x (a.b)) (!pair 0 x))
(fun isnil Bool (l [a]) (? nil l))

(fun list1 [a] (x a) (#cons x nil))
(fun list2 [a] (x a y a)
    (#cons x (#cons y nil)))
(fun list3 [a] (x a y a z a)
    (#cons x (#cons y (#cons z nil))))

(fun snd b (x (a.b)) (!pair 1 x))
(fun fst b (x (a.b)) (!pair 0 x))
Defining new types

(data (Tree a)
  (#tip a)
  (#fork (Tree a) (Tree a)))

{ A type with no arguments }
(data (Color) (#red) (#blue) (#green))

(data (Result a) (#found a) (#notFound))
Example

(fun length Int (l [a])
  (local (temp 0)
    (block
      (:= temp 0)
      (while (@not (?nil l))
        (block
          (:= temp (+ temp 1))
          (:= l (@ tail l)))
        temp)))
Abstract Data types

- Data definitions create types that have operations of
  - Construction
  - Selection
  - Predicate
- Other kinds of types are defined by their operations
  - (Env a)
  - lookup  ((Env a) -> Int -> (Result a))
  - extend  (Int -> a -> (Env a) -> (Env a))
  - empty   (Env a)
Example

(adt (Env a) [(Char . a)])
  (global empty (Env a) nil)

(fun extend (Env a) (key Char object a table (Env a))
  (#cons (#pair key object) table))

(fun lookup (Result a) (tab (Env a) key Char)
  (if (?nil tab) (#notFound)
    (if (= key (@fst (@head tab)))
      (#found (@snd (@head tab)))
      (@lookup (@tail tab) key)))) )
Another Example

(adt (Stack a) [a]
  (global emptySt (Stack a) (#nil))
  (fun push (Stack a) (x a xs (Stack a)) (#cons x xs))
  (fun pop ( a . (Stack a)) (xs (Stack a))
    (#pair (!cons 0 xs) (!cons 1 xs)))
)

Modules

• Modules allow breaking a program into separate files
• Track what a file needs from others to compile successfully
• Track what a file might provide to other files
• Control names
• Track types across files.
Sig-Item

- A Sig-Item specifies the type of an item. It says nothing about how it is implemented

- (type (T a b))
- (val x Int)
- (val f (Int -> Bool))
- (data (T x) (#make x Int) (#none Bool))
A signature is a set of Sig-Items

(sign Stack
  (type (Stack a))
  (val push (a -> (Stack a)-> (Stack a)))
  (val emptySt (Stack a))
  (val pop ((Stack a)-> (a . (Stack a)))))
Signatures

• Appear in programs

  (sig Stack
   (type (Stack a))
   (val push (a -> (Stack a)-> (Stack a)))
   (val emptySt (Stack a))
   (val pop ((Stack a)-> (a . (Stack a)))))

• And also in *.sig files

  (defsig Stack
   (sig
    (type (Stack a))
    (val push (a -> (Stack a)-> (Stack a)))
    (val emptySt (Stack a))
    (val pop ((Stack a)-> (a . (Stack a)))))
Signatures can be read from files

(defsig Stack
    (sig
        (type (Stack a))
        (val push (a -> (Stack a)-> (Stack a))))
    (val emptySt (Stack a))
    (val pop ((Stack a)-> (a . (Stack a))))))

(signature Stack "test.sig")
SigExp

- A sigExp is a way of creating a set of sig-Items
- There is a syntax for SigExp

sigExp :=
  Id
  | 'prelude'
  | 'everything'
  | '(' 'sig' { sigExp } ')
  | '(' hide' sigExp '(' {Id | id } ')' ')'
  | '(' 'union' { sigExp } ')'
Examples

- prelude
- everything
- (hide prelude (Int Bool nil))
- (sig (val x Int) (data (T) (#a Int) (#b)))
- (union prelude
  (sig (val x Int) (data (T) (#a Int) (#b))))
Use of SigExp

• A SigExp is used to compute a set of sigItems for three different reasons

1. Describe what external functions a file depends on.
   – (module T in sigExp out sigExp)

2. Describe what subset of the definitions in a file should be exported
   – (module T in sigExp out sigExp)

3. Describe what subset of the exported functions should be imported
   – (import “test.e7” implementing sigExp )
   – (import “test.e7” hiding sigExp)
What needs to be imported

(module Small in (sig (val tom Int)) out everything)
(global temp Int 5)
(adt (Stack a) [a]
  (global emptySt (Stack a) (#nil))
  (fun push (Stack a) (x a xs (Stack a)) (#cons x xs))
  (fun pop ( a . (Stack a)) (xs (Stack a))
    (#pair (!cons 0 xs) (!cons 1 xs)))
)
(global www Char 'c')
main
(:= temp (+ tom 1))
What should be exported

(signature E "envSig.sig")
(module Env2 in prelude out E )

(data (Tree a)
   (#leaf)
   (#node Int a (Tree a) (Tree a)))

... Main 0
What should be imported

(signature Stack “stack3.sig”)

(import "small.e7" implementing Stack)

(defsig Stack
  (sig (type (Stack a))
       (val push (a -> (Stack a) -> (Stack a)))
       (val emptySt (Stack a))
       (val pop ((Stack a) -> (a.(Stack a)))))
  )