Smalltalk is highly dynamic

- Semantics reflect this
- Almost everything can change at runtime
- (In the full language, even more so!)

Major new features

- Values are objects
- Object carries its class with it
- Even classes like SmallInteger can be redefined
- ... so the behavior of a literal could change during program execution
- Method dispatch - many rules!
- Environments - global and parameter
- Closures capture only parameter environment

Agenda

- The semantics of uSmalltalk
Abstract syntax

Expressions

datatype exp = VAR of name
  | SET of name * exp
  | SEND of srcloc * name * exp * exp list
  | BEGIN of exp list
  | BLOCK of name list * exp list
  | LITERAL of rep
  | VALUE of value
  | SUPER

For technical reasons values can be treated as expressions

Top-level items

The main new thing is the class definition

toplevel = DEFINE of name * name list * exp
  | CLASSD of
  { name : string
  , super : string
  , ivars : string list (* instance variables *)
  , methods : (method_kind * name * method_impl) list
  }
  | EXP of exp | VAL of name * exp | USE of name
and method_kind = IMETHOD | CMETHOD
and method_impl = USER_IMPL of name list * name list * exp
  | PRIM_IMPL of name

Values

Values (objects) are pairs containing the class and the representation:

withtype value = class * rep

Representations (closures need the static superclass)

rep = USER of value ref env (* instance vars *)
  | ARRAY of value Array.array
  | NUM of int
  | SYM of name
  | CLOSURE of
    name list * exp list * value ref env * class
  | CLASSREP of class

Class representation

Classes are constructed from ML records

class = CLASS of
  { name : name
  , super : class option (* superclass, if any *)
  , ivars : string list (* instance variables *)
  , methods : method env
  , id : int (* unique identifier *)
  }

The option datatype is used to represent things that might not be there - class Object has no superclass
Side trip: the option datatype

- A standard ML datatype, used throughout the interpreter to represent optional things
  
  ```plaintext
datatype 'a option =  NONE |  SOME of 'a
  
  Creating optional values:
  
  fun stringReader []     = NONE
  | stringReader (h::t) = SOME (h, t)
  ```

Using option

- Distinguishing between SOME and NONE
  
  ```plaintext
  case super
  of SOME c => fm c
  | NONE   => ...
  ```

- Raising an exception if NONE
  
  ```plaintext
  fun mkInteger n = (valOf (!intClass), NUM n)
  handle Option => badlit "...
  ```

Methods in class reps

- Methods are either primitive or user-defined:
  
  ```plaintext
  method
  = PRIM_METHOD of name * (value * value list -> value)
  | USER_METHOD of
  { name : name
  , formals : name list
  , temps : name list
  , body : exp
  , superclass : class (* static superclass *)
  }
  ```

Expression evaluation

- Context is a message send:
  
  - global environment $\xi$
  - local (parameter) environment $\rho$
  - static superclass (superclass of the class where the message send occurs) $C_{\text{super}}$
  - Environments map identifiers to locations in the store
Judgments

- Expression evaluation
  \[\langle e, \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma' \rangle\]

- Top-level evaluation
  \[\langle t, \xi, \sigma \rangle \rightarrow \langle \xi', \sigma' \rangle\]

Variables

- Just like Impcore (we ignore the superclass)
- \textit{self} is an instance variable, and \textit{super} behaves like \textit{self} except as the receiver of a message:
  \[\langle \text{VAR}(\text{self}), \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle\]
  \[\langle \text{SUPER}, \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle\]

Literals

- Array literals are parsed as VALUES, but numbers and symbols as LITERALS
  \[\langle \text{VALUE}(v), \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle\]

Blocks

- We make an object of class \textit{Block}, with a closure as representation, which captures the parameter environment (not the globals) as well as the static superclass:
  \[\langle \text{BLOCK}((x_1 \ldots x_n), es), \rho, cs, \xi, \sigma \rangle \Downarrow \langle (\sigma(\xi(\text{Block})), \text{CLO}(x_1 \ldots x_n), es, cs, \rho)), \sigma \rangle\]
Message send

- Five cases:
  - user-defined method, receiver is not super
  - user-defined method, receiver is super
  - primitive method, receiver is not super
  - primitive method, receiver is super
  - value method

Message send cont’d

- Allocate space for the method's parameters and locals
  \( l_1, \ldots, l_n \notin \text{dom } \sigma_n \quad l'_1, \ldots, l'_k \notin \text{dom } \sigma_n \)

\[
\hat{\sigma} = \sigma_n \{ l_1 \mapsto v_1, \ldots, l_n \mapsto v_n, l'_1 \mapsto \text{nil}, \ldots, l'_k \mapsto \text{nil} \}
\]

- Create the environment and eval the body

\[
\rho' = \text{instanceVars}(r)
\]

\[
\langle e_m, \rho' \{ x_1 \mapsto l_1, \ldots, x_n \mapsto l_n, y_1 \mapsto l'_1, \ldots, y_k \mapsto l'_k \}, s, \xi, \hat{\sigma} \rangle \Downarrow \langle v, \sigma' \rangle
\]

- Notice which static superclass is used!

Ordinary user message send

- To evaluate

\[
\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_s, \xi, \sigma \rangle \Downarrow \langle v, \sigma' \rangle
\]

- eval receiver and parameters, threading the store:

\[
\langle e, \rho, c_s, \xi, \sigma \rangle \Downarrow \langle \langle c, r \rangle, \sigma_0 \rangle
\]

\[
\langle e_i, \rho, c_s, \xi, \sigma_{i-1} \rangle \Downarrow \langle \langle c_i, \sigma_i \rangle, \sigma_i \rangle
\]

- look up method using receiver's class

\[
\text{findMethod}(m, c) = \text{user method}(\ , \langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_k \rangle, e_m, s)
\]

Message send to super

- The only difference is that we use the static superclass to start the method lookup.

\[
\text{findMethod}(m, c_s) = \text{USER METHOD}(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_k \rangle, e_m, s)
\]
Primitive methods

- The rules are simpler, because primitive methods are just functions.
- The *value* method sent to a block acts like a user method, but without local variables. The body of the block is evaluated in a context where the static superclass is the one that was stored when the block was created.

Top level variables

- Defining a new global:

\[
\begin{align*}
& x \not\in \text{dom} \quad l \not\in \text{dom} \\
& \langle e, \emptyset, \xi_0(\text{Object}), \xi, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \\
& \langle \text{VAL}(x, e), \xi, \sigma \rangle \rightarrow \langle \xi\{x \mapsto l\}, \sigma'\{l \mapsto v\} \rangle
\end{align*}
\]

- Recall that a closure doesn't capture the global environment. That's why we can define recursive blocks at the top level.

The top-level environment

- ...not captured in closures:

```
-> (define foo (n) (if (<= n 0)
    [(value bar n)] [(value foo (- n 1))]))
<Block>
-> (define bar (n) (+ n 1))
<Block>
-> (value foo 5)
1
-> (val bar 0)
0
-> (value foo 5)
run-time error: SmallInteger does not understand message value
```
Flow-of-control view

- Entry point from command line `main`, calls `runInterpreter` which calls `readEvalPrint`
- Lexing and parsing "hidden" in the reader created by `readEvalPrint`
- `readEvalPrint`: loop "forever", calling top level evaluator and handling errors
- `topEval`: evaluates one top-level item; but "use" recursively calls `readEvalPrint`

Where does abstract syntax come from?

- How do we get
  \[
  \text{SET}("x", \text{VAR}("y"))
  \]
  from
  \[
  (\text{set } x \ y)
  \]
- Answer: lexer turns characters into lists of lexical items (datatype `par`) and parser turns that into abstract syntax
Lexing and parsing

- **read** is the entry point to lexing
- **toplevel** is the entry point to parsing (this identifier is both a datatype and a function name)
- The guts of parsing are in function **parse**

Circularities: Booleans

- A chicken-and-egg problem:
  - We need class **Object** because everything inherits from it
  - Class **Object** defines method **notNil**, which returns a **Boolean**
  - Class **Boolean** inherits from class **Object**

Circularities: literals

- When the evaluator sees a literal, it must create a value of one of the classes **Integer**, **Symbol**, or **Array**
- ... but we need the evaluator to read these classes from the initial basis

Circularities: the solution

- As we did for self-reference in recursion, use reference cells
- for classes **Integer**, **Symbol**, **Array**, **Block**
- for **Booleans** true, false
- During bootstrapping (reading the basis) these cells contain nonsense - so the initial basis must avoid evaluating certain kinds of expressions
- After bootstrapping update the cells
Side trip: building in classes

- ML and a clean design made it easy for me to build in new primitive classes for the homework

Building CacheControl

- Just a collection of functions turned into methods:

```haskell
fun getHits _ = mkInteger (!cacheHits)
fun getMisses _ = mkInteger (!cacheMisses)
fun resetCacheCounts _ = (cacheHits := 0 ; cacheMisses := 0 ; nilValue)
fun turnCachingOn _ = (cachingOn := true ; nilValue)
fun turnCachingOff _ = (cachingOn := false ; nilValue)
val statsClass = mkClass "CacheControl" objectClass []
  [ primMethod "hits" (unaryPrim getHits),
    primMethod "misses" (unaryPrim getMisses),
    primMethod "reset" (unaryPrim resetCacheCounts),
    primMethod "cachingOn" (unaryPrim turnCachingOn),
    primMethod "cachingOff" (unaryPrim turnCachingOff)]
```

Building Timer

- Some primitive methods and a user method:

```haskell
local val timer = ref NONE in
  fun startTimer _ = ...
  and stopTimer _ = ...
end

val timerClass = mkClass "Timer" objectClass []
  [ primMethod "start" (unaryPrim startTimer),
    primMethod "stop" (unaryPrim stopTimer),
    userMethod "timeBlock:" ["aBlock"] [] 
      "(begin (start self) (value aBlock) (stop self))"
  ]
```

Binding the class names

- Class names have to be bound in the global environment:

```haskell
val initialXi = foldl addClass initialXi [ objectClass, nilClass, classClass, statsClass, timerClass ]
```
Evaluation

- Straightforward translation of semantics, so `eval` has four parameters: expression, local environment, static superclass, global environment.
- Functions `findMethod` and `instanceVars` were used in the semantics without formally being specified.

Recall class representation

- Classes are constructed from ML records:
  ```ml
  class = CLASS of
    { name : name
    , super : class option (* superclass, if any *)
    , ivars   : string list (* instance variables *)
    , methods : method env
    , id      : int         (* unique identifier *)
    }
  ```

Method lookup

- Finding a method:
  ```ml
  fun findMethod (name, class) =
    let fun fm (CLASS { methods, super, ...}) =
      find (name, methods)
      handle NotFound m =>
        case super
          of SOME c => fm c
          | NONE   => raise RuntimeError
            (className class^" does not understand message "^m)
      in  fm class
      end
  ```

Instance variables of an object

- Remember from the semantics: to send a message we need to create an environment from the receiver's instance variables:
  ```ml
  ρ′ = instanceVars(r)
  <e_m, ρ′{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n, y_1 \mapsto l'_1, \ldots, y_k \mapsto l'_k}, s, ξ, ̂σ> \Downarrow <v, σ'>
  ```
  and `instanceVars (_, USER rep) = rep`.
  ```ml
  instanceVars self =
    bind("self", ref self, emptyEnv)
  ```
Creating closures

- Capture local environment and static superclass

```haskell
fun mkBlock c = (valOf (!blockClass), CLOSURE c)
    handle Option =>
    raise InternalError
    "Bad blockClass; evaluated block
    expression in initial basis?"
```

Read-eval-print

- Besides tracing, the loop updates the definitions of
  the classes that have literals:

```haskell
fun closeLiteralsCycle xi =
    (intClass := SOME (findInitialClass ("SmallInteger", xi))
    ; symbolClass := SOME (findInitialClass ("Symbol", xi))
    ; arrayClass := SOME (findInitialClass ("Array", xi))
    )
```

Full Smalltalk

- Originally intimately associated with early GUI design research
- Thus, no official concrete syntax for classes
- Innovative concrete syntax for message send. To send a two-argument message named $m1:m2$: we write
  
  ```smalltalk
  receiver m1: arg1 m2: arg2 ...
  ```
- Assignment is left-arrow. Hurray!

Syntax examples

- With judicious choice of names the syntax can look rather natural:

  ```smalltalk
  myAccount spend: 10 for: #dinner
  myAccount totalSpentFor: #dinner
  ```
Semantics

- Many more literals
- Class variables (like Java statics)
- Nonlocal return
- Huge predefined class hierarchy
- Reflection; everything is an object, including methods; supports Smalltalk processing itself:
  - compilers, debuggers, browsers, ...