

Implementing Type Checking with Classes

We used to have two records for each class:

- *Class declarations* = abstract syntax
- *Class* = run-time class information
 - flattened field and method lists

Now we'll have three:

- *Class declarations* = abstract syntax
- *Static class* = check-time class information
 - flattened lists with types
- *Class* = run-time class information
 - flattened lists

Checking Class Declarations

Check:

- Superclass exists, and no cyclic inheritance
- Methods bodies ok
 - Use host class for type of **self**
- Overriding method signatures are the same as in superclass
 - Except for **initialize**

```
class c2 extends c1
  method void m(int x, bool y)
    if y then +(2, x) else send self w()
```

Static Class Elaboration

```
;; type-of-program : program -> type
(define (type-of-program pgm)
  (cases program pgm
    (a-program (c-decls exp)
      (statically-elaborate-class-decls! c-decls)
      (type-of-expression exp (empty-tenv))))))
```

Checking Class Declarations

- Cyclic inheritance covered by requirement that classes are ordered

```
(define statically-elaborate-class-decls!
  (lambda (c-decls)
    (for-each statically-elaborate-class-decl!
      c-decls)
    (for-each check-class-method-bodies!
      c-decls)))
```

Checking Class Declarations: Methods

```
(define (check-class-method-bodies! c-decl)
  ...
  (for-each
    (lambda (m-decl)
      (typecheck-method-decl!
        m-decl
        class-name super-name
        field-ids field-tys))
    m-decls))
```

Checking Object Creation

Check:

- Class exists, and is not abstract
- Class has an **initialize** method
- **initialize**'s argument types match the operand types

```
class c1 extends object
  method void initialize(int x, bool y)
```

...

```
new c1(1, false)
```

Checking Class Declarations: Methods

```
(define (typecheck-method-decl! m-decl self-name
  super-name field-ids field-types)
  (cases method-decl m-decl
    (a-methd-decl (res-texp name id-texps ids body)
      (let* ((id-tys (expand-ty-exprs id-texps))
             (tenv
              (extend-tenv
                (cons '%super (cons 'self ids))
                (cons (class-type super-name)
                      (cons (class-type self-name)
                            id-tys))))
              (extend-tenv
                field-ids field-tys (empty-tenv))))
            (body-ty (type-of-expr body tenv)))
        (check-is-subtype!
          body-ty (expand-ty-expr res-texp) m-decl)))
    (an-abstract-method-decl (...) #t)))
```

Checking Object Creation

```
(define (type-of-new-obj-exp rand-types)
  (cases static-class (static-lookup class-name)
    (a-static-class (...)
      (cases abstraction-specifier specifier
        (abstract-specifier ()
          (eopl:error ...))
        (concrete-specifier ()
          (type-of-method-app-exp
            #t ;; means from 'new'
            (class-type class-name)
            'initialize
            rand-types)
          ;; Result:
            (class-type class-name))))))
```

Checking Method Calls

Check:

- Receiver expression is an object
- Method is in the object-type's class
 - Except **initialize**...
- Method's argument types match the operand types

```
class c1 extends object
  method void initialize() ...
  method void m(int x, bool y)
  ...
let o1 = new c1()
in send o1 m(1, false)
```

Checking Super Calls

Check:

Same as method calls, but simpler:

- No check for **initialize**
- No possibility of a non-object type

```
(define (type-of-super-call-exp super-name
                                msg rand-types)
  (type-of-method-app-or-super-call
   #t super-name msg rand-types)
```

Checking Method Calls

```
(define (type-of-method-app-exp for-new? obj-type
                                msg rand-types)
  (if (and (eq? msg 'initialize) (not for-new?))
      (eopl:error ...)
      (cases type obj-type
        (class-type (class-name)
                     (type-of-method-app-or-super-call
                      #f class-name msg rand-types))
        (else
         (eopl:error ...))))))
```

Checking Method Application

```
(define (type-of-method-app-or-super-call
        super-call? host-name msg rand-tys)
  (let ((method (statically-lookup-method msg
                                           (static-class->methods
                                            (static-lookup host-name)))))
    (if (static-method? method)
        (cases static-method method
          (a-static-method (method-name spec
                                method-ty super-name)
                           (let ((result-ty (type-of-app
                                              method-ty rand-tys)))
                             (if super-call?
                                 (cases abstraction-specifier spec
                                  (concrete-spec () result-ty)
                                  (abstract-spec () (error ...)))
                                 result-ty))))
        (eopl:error ...))))))
```

Checking Casts

Check:

- Operand has an object type (for any class)
- Target class exists
- Class for operand and target must be comparable
 - Otherwise, cast cannot possibly succeed

```
class c1 extends object ...
class c2 extends object ...
cast new c1() c2
```

Checking Casts

```
(define (type-of-cast-exp ty name2 exp)
  (cases type ty
    (class-type (name1)
      (if (or (statically-is-subclass? name1 name2)
              (statically-is-subclass? name2 name1))
          (class-type name2)
          (eopl:error ...)))
    (else
      (eopl:error ...))))
```

Checking Other Expressions

- Other expression forms checked as before
- `check-is-subtype!` often used instead of `check-equal-type!`

Compiling with Classes (Optionally)

- Recall that a **compiler** takes a program in language A and produces a program in language B
- To make compilation optional, a common trick is to set $B = A$, with the expectation that source programs use only a subset of A

Grammar with Compiler-target Cases

```
<expr> ::= <num>
        ::= <id>
        ::= <prim><expr>*(.)
        ...
        ::= send <expr> <id><expr>*(.)
        ...
        ::= <<num>,<num>>
        ::= send <expr> <<num>><expr>*(.)
```

Grammar with Compiler-target Cases

```
(define the-grammar
  '((program ((arbno class-decl) expression)
             a-program)

    (expression (number) lit-exp)
    (expression ("true") true-exp)
    ...
    (expression ("lexvar" number number)
                lexvar-exp)
    (expression
     ("imethod" expression number
      (separated-list expression ","))
     apply-method-indexed-exp)))
```

Interpreter with Compiler-target Cases

```
(define (eval-expression exp env)
  (cases expression exp
    (lit-exp (datum) datum)
    (var-exp (id) (apply-env env id))
    ...
    (lexvar-exp (depth pos)
                 (apply-env-lexvar env depth pos))
    (apply-method-indexed-exp (obj-exp pos rands)
                               (let ((obj (eval-expression obj-exp env))
                                     (args (eval-rands rands env))
                                     (c-name (object->class-name obj)))
                                 (apply-method
                                  (list-ref
                                   (class->methods (lookup-class c-name))
                                   pos)
                                  ...))))))
```