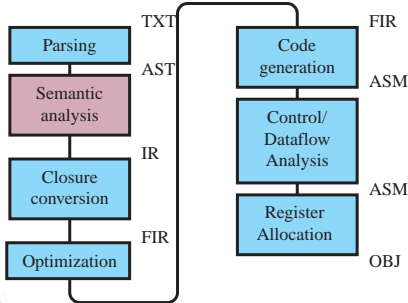


CS134b: Semantic Analysis

- Type checking
- IR representation



Compiler Stages



Semantic Analysis

- Build intermediate representation (IR)
- Why generate IR (many analyses can be performed on the AST directly)
 - IR is simpler
 - IR is more general (the same IR can be used with multiple front-ends)
 - We build a more generic compiler
- We'll cover
 - IR definition
 - Conversion of simple expressions to IR
 - Conversion of classes and interfaces



IR Syntax

- Three main parts:
 - *types*
 - *atoms (values like numbers, variables)*
 - *expressions*



IR type definition

```
type ty =  
  (* Basic types *)  
  | TyUnit | TyNil | TyBool | TyChar | TyString | TyInt | TyFloat  
  
  (* Arrays of values, all elements have the same type *)  
  | TyArray of ty  
  
  (* Functions have a list of types for the arguments, and a result type *)  
  | TyFun of ty list * ty  
  
  (* TObject is an instance of a class *)  
  | TObject of ty_var  
  
  (* TyVObject is an instance of an interface *)  
  | TyVObject of ty_var
```



IR type definitions

- Some basic types
 - *TyInt is the type of integers*
 - *TyArray(ty) is the type of arrays of elements with type ty*
 - *TyString is the type of strings*
 - Note, we are not implementing the *String*, *Array*, *System* classes
 - This is just for simplicity—we can do it by adding default definitions
 - *TyString* is not equivalent to *(TyArray TyChar)*
 - *TyNil is the type of null*
 - *Null* is polymorphic: it can be used with many different types
 - *TyFun (ty_vars, ty_res) is the type of functions with arguments of type ty_vars, and result ty_res*



Object types

- TyObject(v), TyVObject(v)
 - We'll discuss these in (significant) detail later this week
- For now:
 - We have to deal with classes and interfaces
 - "Objects" are values with a class type
 - "VObjects" are values with an interface type
 - Sun's original compiler folded these together
 - This meant that interfaces were very inefficient
 - We'll need to discuss object layout
- Note:
 - Classes are not types—they contain code



IR expressions

- Goals
 - Scoping should be explicit
 - The order of evaluation should be clearly defined
 - We'll decide if $((x = 1) + (x = 2))$ leaves x as 1 or 2
 - The IR should be typed
 - So we can help determine if our compiler is bogus
 - So we can give a proof that the code is safe
- Code
 - Separate expressions into two levels:
 - Atoms are simple values, like numbers and variables
 - Expressions specify evaluation order



Atoms

type atom =
 AtomUnit
 | AtomNil
 | AtomBool of bool
 | AtomChar of char
 | AtomInt of int
 | AtomFloat of float
 | AtomVar of var



Atoms

- Simple constants
 - *AtomUnit*
 - This is a value of "void" type
 - The Java/C void definition is incorrect—it corresponds to the ML unit type (a true void type would be empty)
 - *AtomChar, AtomBool, AtomInt, ...*
 - Constants
- Variable expressions
 - *AtomVar(v)*: the value in variable *v*



Expressions

- A program is an expression
- Nearly all expressions have the form
 - *let v : ty = <op> in e*
 - *The v makes the intermediate value explicit*
 - We have to define a place to put intermediate values
 - *The semantics is what you would expect*
 - Evaluate <op>, put the value in v, and then evaluate e



Simple expressions

- $e ::= \text{let } v : \text{ty} = \text{atom in } e$
 - | $\text{let } v : \text{ty} = \text{unop } a \text{ in } e$
 - | $\text{let } v : \text{ty} = a1 \text{ binop } a2 \text{ in } e$
 - | $\text{return } a$
- Example, translate $(x + y * 3)$
 - *let v1 = y * 3 in*
 - let v2 = x + v1 in*
 - return v2*
- v1 and v2 are the intermediate computations



Unary operations

- Normal arithmetic
 - *let v : ty = - a in e* (negation)
 - *let v : int = ! a in e* (logical negation)
- Coercions are explicit
 - *let v : int = (int) a in e* (coercion: a is float)
 - *let v : float = (float) a in e* (coercion: a is int)

```
type exp = ... | LetUnop of var * ty * unop * atom * exp
type unop =
  (* Arithmetic *)
  | UMinusIntOp
  | UMinusFloatOp
  | UNotBoolOp
```

(* Numeric coercions *)

```
| UIntOfFloat
| UFloatOfInt
```



Binary operations

- Normal binary arithmetic
 - *+*: arguments are integers, floats
 - *-*: arguments are integers, floats
 - ***, */*: integer/float
 - *&*, *|*, *^*, *<<*, *>>*, *%*: integers



Binary operations

```
type exp = ... | LetBinop of var * ty * binop * atom * atom * exp
type binop =
  | AddIntOp
  | SubIntOp
  | ...
  | EqIntOp
  | LeIntOp
  | ...
```

```
| AddFloatOp
| SubFloatOp
| ...
| EqFloatOp
| LeFloatOp
| ...
```



Functions

- Simultaneous function definition
 - $e ::= \text{let } f_1 : t_1 \text{ (vars1)} = e_1$
and $f_2 : t_2 \text{ (vars2)} = e_2$
...
and $f_n : t_n \text{ (varsn)} = e_n$
in e
- Functions may be arbitrarily nested
- All function names f_1, f_2, \dots, f_n are *bound* in function bodies e_1, e_2, \dots, e_n and in e



Function calls

- Tail-call:
 - $e ::= f(a_1, \dots, a_n)$
- Normal function call
 - $e ::= \text{let } v : ty = f(a_1, \dots, a_n) \text{ in } e$
- Returning a value from a function
 - $e ::= \text{return } a$
- Call a built-in function
 - $e ::= \text{let } v : ty = ("op" : ty)(a_1, \dots, a_n) \text{ in } e$
 - *op* includes "atoi", "atof", "itoa", "ftoa", "println", and "strcat"
 - *We need these to implement the builtin string ops*



Type definitions

```
type exp =  
  LetFuns of fundef list * exp  
| LetApply of var * ty * var * atom list * exp  
| LetExt of var * ty * string * ty * atom list * exp  
| TailCall of var * atom list  
| Return of var * atom  
| ...
```

```
and fundef = var * fun_info  
and fun_info = fun_class * ty * var list * exp
```

```
and fun_class =  
  FunGlobalClass  
| FunLocalClass
```



Conditional

- A conditional performs a comparison
- Else-branch can't be omitted
 - $e ::= \text{if } a \text{ then } e1 \text{ else } e2$
- The conditional does not "return" a value
 - *This code is not possible:*
 - $\text{let } v : \text{ty} = \text{if } \dots \text{ then } \dots \text{ else } \dots \text{ in } e$

type exp = ... | IfThenElse of atom * exp * exp



Aggregate operations

- Array subscripting
 - $e ::= \text{let } v : \text{ty} = a[a2] \text{ in } e$
 - | $a[a2] \leftarrow e1; e2$
- Record projection
 - $e ::= \text{let } v1 : \text{ty} = a.v2 \text{ in } e$
 - | $a1.v \leftarrow a2; e$

type exp =

...

| SetSubscript of atom * atom * ty * atom * exp
| LetSubscript of var * ty * atom * atom * exp
| SetProject of atom * var * ty * atom * exp
| LetProject of var * ty * atom * var * e



Side-effects

- $e ::= v \leftarrow a; e$

type exp = ... | SetVar of var * ty * atom * exp



Allocation

- String allocation
 - Strings are not atoms
 - let $v = "s"$ in e
- Multi-dimensional array allocation
 - let $v : ty = new\ array(dimens, a)$ in e
 - $dimens$ are a list of integers, a is the initializer

type exp =

...

| LetString of var * string * exp

| LetArray of var * ty * atom list * atom * exp



Simple translations

- int f(int i) { if(i != 0) 1; else 2; }
- Becomes:
 - let $f : ((int) \rightarrow int) (i) =$
 - if $i \neq 0$ then
 - return 1
 - else
 - return 2



Loops

- int fact(int i) {
 int v = 1, j = 1;
 while(j <= i)
 v *= j++;
 return v;
}
- let $f : ((int) \rightarrow int) (i) =$
 - let $v : int = 1$ in
 - let $j : int = 1$ in
 - let $g : ((int) \rightarrow int) (j') =$
 - if $j \leq i$ then
 - let $v' : int = v * j$ in
 - $v <- v'$;
 - let $v'' : int = j + 1$ in
 - $j <- v''$;
 - $g(j)$
 - else
 - return v
 - in
 - $g(j)$



Local and global functions

- A “global” function allows nested calls and **return** statements
- A “local” function is allowed to be called only with a TailCall
 - *return statements return to the nearest global function*



Fib

- ```
int fib(int i) {
 if(i == 0 || i == 1)
 1;
 else
 fib(i - 1) + fib(i - 2);
}
```
- ```
let global fib : ((int) -> int) (i) =
  let local g : (0 -> any) 0 = return 1
  in
  if i = 0 then
    g0;
  else if i = 1 then
    g0;
  else
    let v1 : int = i - 1 in
    let v2 : int = fib(v1) in
    let v3 : int = i - 2 in
    let v4 : int = fib(v3) in
    let v5 : int = v2 + v4 in
    return v5
```



Utilities

- Three essential tools
- Symbol (variable or type identifiers)
 - *Generation of new symbols (symbols that are different from any other symbol in the program)*
 - *Try to make symbol names meaningful (not just v1, v2, ...)*
 - (*) *Make a new symbol.*
 - *)
`val new_symbol : symbol -> symbol`
`val new_symbol_string : string -> symbol`



Scoping and environments

- A *table* is like an association list
- `SymbolTable.t`
 - `add : 'a t -> symbol -> 'a -> 'a t`
 - `find : 'a t -> symbol -> 'a`
 - raises `Not_found` on error
- Type environment:
 - `ty SymbolTable.t`
 - Gives type definition for a type identifier
- Variable environment
 - `ty SymbolTable.t`
 - Gives type of a variable



Fj_ir_env

- type env
- val env_empty : env
- (* Types: used for class and interface definitions *)
- val env_add_tydef : env -> var -> tydef -> env
- val env_lookup_tydef : env -> ir_pos -> var -> tydef
- val env_lookup_type : env -> ir_pos -> var -> ty
- (* Variables *)
- val env_add_var : env -> var -> ty -> env
- val env_lookup_var : env -> ir_pos -> var -> ty
- (* Functions can be overloaded, so there may be multiple entries *)
- val env_add_fun : env -> var -> var -> ty -> env
- val env_lookup_fun : env -> ir_pos -> var -> var * ty
- val env_lookup_funs : env -> ir_pos -> var -> (var * ty) list



Semant: type-checking/IR generation

- Want this function in `Fj_ir_exp`
 - `val build_exp : env -> Fj_ast.exp -> Fj_ir.exp`
- Not as straightforward as that
 - `int l = 0;`
`return l + 1;`
 - `build_expr tenv venv "int l = 0;" =`
`LetAtom (l, TyInt, AtomInt 0, ???)`



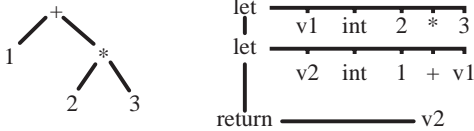
Continuation-passing style

- `val build_expr : env -> Fj_ast.exp -> (env -> atom -> exp) -> exp`
- `build_expr env "int l = 0; return l + 1;" cont =
 build_expr env "int l = 0" (fun env v ->
 build_expr env "return l + 1;" cont)`
- `build_expr env "int l = 0" cont =
 let env = env_add_var env l Tylnt in
 let e = cont env l in
 LetAtom (l, Tylnt, AtomInt 0, e)`



IR generation

- The AST is an expression tree
- The IR is a series of **let** statements
- IR inverts the tree structure



IR generation

- AST is a tree
- Read it from the top-down
- Build IR from bottom-up
- How do we do this?



IR generation: make_expr

- We're doing type checking at the same time
- build_expr returns the new expr

```
let build_exp env e =  
  match e with  
  | Fj_ast.NilExpr _ -> AtomNil  
  | Fj_ast.BoolExpr (b, _) -> AtomBool b  
  | Fj_ast.CharExpr (c, _) -> AtomChar c  
  | Fj_ast.IntExpr (i, _) -> AtomInt i  
  | Fj_ast.FloatExpr (x, _) -> AtomFloat x  
  | Fj_ast.StringExpr (s, _) ->  
    let v = new_symbol_string "str" in  
    let env = env_add_var env v TyString in  
    LetString (v, s, ???)
```



Second try: introduce "rest of program"

- As we build the IR:
 - Add new variables to envv
- The "cont" argument represents the "rest of the program"

```
let build_exp env e cont =  
  match e with  
  | Fj_ast.NilExpr _ -> cont env AtomNil  
  | Fj_ast.BoolExpr (b, _) -> cont env (AtomBool b)  
  | Fj_ast.CharExpr (c, _) -> cont env (AtomChar c)  
  | Fj_ast.IntExpr (i, _) -> cont env (AtomInt i)  
  | Fj_ast.FloatExpr (x, _) -> cont env (AtomFloat x)  
  | Fj_ast.StringExpr (s, _) ->  
    let v = new_symbol_string "str" in  
    let env = env_add_var env v TyString in  
    LetString (v, s, cont env (AtomVar v))
```



Binary expressions: first step

- Build the subexpressions

```
let rec build_exp env e cont =  
  let pos = Fj_ast_util.pos_of_expr e in  
  let pos = string_pos "build_exp" (ast_pos pos) in  
  match e with  
  ...  
  | Fj_ast.BinOpExpr (op, e1, e2, _) ->  
    build_binop_exp env pos op e1 e2 cont  
  
and build_binop_exp env pos op e1 e2 cont =  
  let pos = string_pos "build_binop_exp" pos in  
  build_exp env e1 (fun env a1 ->  
    build_exp env e2 (fun env a2 ->  
      build_binop_atom env pos op a1 a2 cont))
```



Binary expressions: second step

- Coerce the arguments to have equal types, and check the op

```
and build_binop_atom env pos op a1 a2 cont =  
  let pos = string_pos "build_binop_atom" pos in  
  coerce_equal env pos a1 a2 (fun env ty a1 a2 ->  
    match op, ty with  
    | Fj_ast.PlusOp, TyInt ->  
      build_binop env pos TyInt AddIntOp a1 a2 cont  
    | Fj_ast.MinusOp, TyInt ->  
      build_binop env pos TyInt SubIntOp a1 a2 cont  
    | Fj_ast.PlusOp, TyFloat ->  
      build_binop env pos TyFloat AddFloatOp a1 a2 cont
```



Binary expressions: third step

- Build the expression, and call the cont

```
and build_binop env pos ty op a1 a2 cont =  
  let pos = string_pos "build_binop" pos in  
  let v = new_symbol_string "binop" in  
  let env = env_add_var env v ty in  
  LetBinop (v, ty, op, a1, a2, cont env (AtomVar v))
```



Coercions:

- Java has implicit coercions
 - *Int -> float*
 - *Int -> string*
 - *Float -> string*
 - *Lots of object coercions*
- Plan: define a function `coerce_type` that takes an atom and a type, and coerces the atom to the type



Getting the type of an atom

- The type is in the environment

```
let type_of_atom env pos a =
  let pos = string_pos "type_of_atom" pos in
  match a with
  | AtomUnit -> TyUnit
  | AtomBool _ -> TyBool
  | AtomChar _ -> TyChar
  | AtomInt _ -> TyInt
  | AtomFloat _ -> TyFloat
  | AtomNil -> TyNil
  | AtomVar v -> env_lookup_var env pos v
```



Numeric coercions

```
let coerce_type strict env pos a ty1 cont =
  let pos = string_pos "coerce_type" pos in
  let ty2 = type_of_atom env pos a in
  (* If the types are already equal, don't need to do anything *)
  if equal_types env pos ty1 ty2 then
    cont env a
  else
    match ty1, ty2 with
    | TyInt, TyFloat ->
      let v = new_symbol_string "int_of_float" in
      let env = env_add_var env v TyInt in
      LetUnop (v, TyInt, UIntOfFloat, a, cont env (AtomVar v))
    | TyFloat, TyInt ->
      let v = new_symbol_string "float_of_int" in
      let env = env_add_var env v TyFloat in
      LetUnop (v, TyFloat, UFloatOfInt, a, cont env (AtomVar v))
```



String coercions

- There are several builtin functions: "atoi", "println", "itoa", "ftoa", "strcat"

```
| TyString, TyInt ->
  let v = new_symbol_string "string" in
  let env = env_add_var env v TyString in
  LetExt (v, TyString, "itoa", TyFun ([TyInt], TyString), [a], cont env (AtomVar v))
| TyString, TyFloat ->
  let v = new_symbol_string "string" in
  let env = env_add_var env v TyString in
  LetExt (v, TyString, "ftoa", TyFun ([TyFloat], TyString), [a], cont env (AtomVar v))
```



A more mathematical description

- $E[\dots]$: takes a list of bindings and translates them
- A binding is $e \mapsto v$
- Initial translation:

$$E[e \mapsto v :: \text{return } v]$$



Value translation

$$E[i \mapsto v :: e] \rightarrow \text{let } v = i \text{ in } E[e]$$

$$E[v' \mapsto v :: e] \rightarrow \text{let } v' = v \text{ in } E[e]$$



Arithmetic expression

$$E[(v_1 + v_2) \mapsto v :: e] \rightarrow \text{let } v = v_1 + v_2 \text{ in } E[e]$$

$$E[(e_1 + e_2) \mapsto v :: e] \rightarrow E[e_1 \mapsto v_1 :: e_2 \mapsto v_2 :: (v_1 + v_2) \mapsto v; e]$$



Tail-call equivalence

let $v = z$ in e

let $f(v) = e$ in

let $v = z$ in
 $f(v)$



Conditional expression

$E[(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e] \mapsto$
 $E[e_1 \mapsto v_1 :: (\text{if } v_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e]$

$E[(\text{if } v_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e] \mapsto$
let $f(v) = E[e]$ in
if v_1 then
 $E[e_2 \mapsto v :: f(v)]$
else
 $E[e_3 \mapsto v :: f(v)]$



Function calls

$E[f(v)] \mapsto f(v)$

$E[f(e_1) \mapsto v :: e] \mapsto$
 $E[e_1 \mapsto v_1 :: f(v_1) \mapsto v :: e]$

$E[f(v_1) \mapsto v :: e] \mapsto$
let $v = f(v_1)$ in $E[e]$



Array subscript

$$E[e_1[e_2] \mapsto v :: e] \rightarrow E[e_1 \mapsto v_1 :: e_2 \mapsto v_2 :: v_1[v_2] \mapsto v :: e]$$

$$E[v_1[v_2] \mapsto v :: e] \rightarrow \text{let } v = v_1[v_2] \text{ in } E[e]$$



Assignment

$$E[(v' \leftarrow e_1) \mapsto v :: e] \rightarrow E[e_1 \mapsto v_1 :: (v' \leftarrow v_1) \mapsto v :: e]$$

$$E[(v_1 \leftarrow v_2) \mapsto v :: e] \rightarrow v_1 \leftarrow v_2; \text{let } v = v_1 \text{ in } E[e]$$



Array assignment

$$E[(e_1[e_2] \leftarrow e_3) \mapsto v :: e] \rightarrow E[e_1 \mapsto v_1 :: e_2 \mapsto v_2 :: e_3 \mapsto v_3 :: (v_1[v_2] \leftarrow v_3) \mapsto v :: e]$$

$$E[(v_1[v_2] \leftarrow v_3) \mapsto v :: e] \rightarrow v_1[v_2] \leftarrow v_3; \text{let } v = v_3 \text{ in } E[e]$$



Translating a for loop

$$E[(\text{for}(e_1; e_2; e_3) e_4) \mapsto v :: e] \mapsto$$
$$E[e_1 \mapsto v_1 :: (\text{while}(e_2)\{e_4; e_3;\}) \mapsto v :: e]$$


While loop

$$E[(\text{while}(e_1) e_2) \mapsto v :: e] \mapsto$$
$$\text{let } brk() = E[e] \text{ in}$$
$$\text{let } f() = E[e_1 \mapsto v_1 :: \text{if } v_1 \text{ then } e_2; f() \text{ else } brk()] \text{ in}$$
$$f()$$
$$E[\text{break} \mapsto v :: e] \mapsto$$
$$brk()$$


CS134b: Semantic analysis

- Expression conversion
- Classes and interfaces



Arithmetic expression

$$E[(v_1 + v_2) \mapsto v :: e] \mapsto \\ \text{let } v = v_1 + v_2 \text{ in } E[e]$$
$$E[(e_1 + e_2) \mapsto v :: e] \mapsto \\ E[e_1 \mapsto v_1 :: e_2 \mapsto v_2 :: (v_1 + v_2) \mapsto v; e]$$


Binary expressions: first step

- Build the subexpressions

```
let rec build_exp env e cont =
  let pos = Fj_ast_util.pos_of_expr e in
  let pos = string_pos "build_exp" (ast_pos pos) in
  match e with
  ...
  | Fj_ast.BinOpExpr (op, e1, e2, _) ->
    build_binop_exp env pos op e1 e2 cont

and build_binop_exp env pos op e1 e2 cont =
  let pos = string_pos "build_binop_exp" pos in
  build_exp env e1 (fun env a1 ->
    build_exp env e2 (fun env a2 ->
      build_binop_atom env pos op a1 a2 cont))
```



Binary expressions: second step

- Coerce the arguments to have equal types, and check the op

```
and build_binop_atom env pos op a1 a2 cont =
  let pos = string_pos "build_binop_atom" pos in
  coerce_equal env pos a1 a2 (fun env ty a1 a2 ->
    match op, ty with
    | Fj_ast.PlusOp, TyInt ->
      build_binop env pos TyInt AddIntOp a1 a2 cont
    | Fj_ast.MinusOp, TyInt ->
      build_binop env pos TyInt SubIntOp a1 a2 cont
    | Fj_ast.PlusOp, TyFloat ->
      build_binop env pos TyFloat AddFloatOp a1 a2 cont
```



Binary expressions: third step

- Build the expression, and call the cont

```
and build_binop env pos ty op a1 a2 cont =  
  let pos = string_pos "build_binop" pos in  
  let v = new_symbol_string "binop" in  
  let env = env_add_var env v ty in  
    LetBinop (v, ty, op, a1, a2, cont env (AtomVar v))
```



Conditional expression

```
 $E[(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e] \mapsto$   
   $E[e_1 \mapsto v_1 :: (\text{if } v_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e]$   
  
 $E[(\text{if } v_1 \text{ then } e_2 \text{ else } e_3) \mapsto v :: e] \mapsto$   
  let  $f(v) = E[e]$  in  
  if  $v_1$  then  
     $E[e_2 \mapsto v :: f(v)]$   
  else  
     $E[e_3 \mapsto v :: f(v)]$ 
```



Conditional: step 1

- Build the code after the conditional

```
and build_if_exp env pos e1 e2 e3 cont =  
  let pos = string_pos "build_if_exp" pos in  
  
  (* Wrap the rest in a function *)  
  let ty_fun = TyFun ([], TyUnit) in  
  let f_cont = new_symbol_string "if_cont" in  
  let f_body = cont env AtomUnit in  
  let f_call = TailCall (f_cont, []) in
```



Conditional: step 2

- Build the code for the branches

```
(* Build the branches *)
let body2 =
  build_exp env e2 (fun env _ -> f_call)
in
let body3 =
  match e3 with
  | Some e3 ->
    build_exp env e3 (fun env _ -> f_call)
  | None ->
    f_call
in
```



Conditional: step 3

- Build the conditional and continuation fun

```
(* Build the test *)
let body =
  build_exp env e1 (fun env a1 ->
    coerce_bool env pos a1 (fun env a1 ->
      IfThenElse (a1, body2, body3)))
in
(* Add the function *)
LetFuns ([f_cont, (FunLocalClass, ty_fun, [], f_body)], body)
```



Classes, interfaces, objects, vobjects

- A class has four parts:
 - A parent
 - Some interfaces
 - Some methods (with code)
 - Some fields (with optional initializers)
- An interface is just a collection of methods

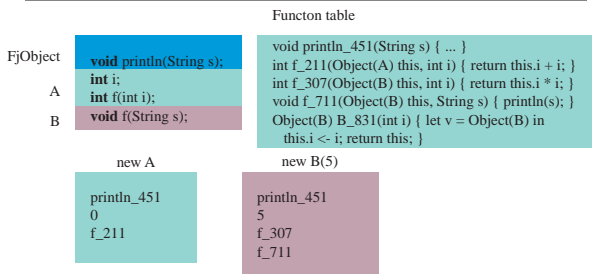


Inheritance

- class FjObject { void println(String s); }
- class A { int i = 1; int f(int j) { return i + j; } }
- interface S { int f(int k); }
- class B extends A implements S {
 - B(int i) { this.i = i; }
 - Int f(int i) { return this.i * i; }
 - Int f(String s) { println(s); }
- }
- An object of class B can be used anywhere where an object of type A, S, FjObject is expected
- Class C {
 - void g(A a) { println(a.f(5)); }
 - Void main(String[] argv) { g(new B(7)); }



Flat class representation, no interfaces



Classes

- Methods defined by children override methods with the same type in their parents
- Flat representation:
 - Ordering of fields is preserved so that a parent object is a prefix
 - Object(A) is a prefix of Object(B)
 - Coercion Object(B)->Object(A) is the identity
- Note: an *object* is not the same as a *class*
- Methods take object as first argument
 - (new B(5)).f(10) becomes:
 - let b = B_831(5) in
 - let f = b.f in
 - f(b, 10)



Table representation

- Methods are immutable
- An object may have a lot of methods
- Move the methods into a table to save space



Virtual method area

Class

```
VMA vma;  
int i;
```

Function table

```
void println_451(String s) { ... }  
int f_211(Object(A) this, int i) { return this.i + i; }  
int f_307(Object(B) this, int i) { return this.i * i; }  
void f_711(Object(B) this, String s) { println(s); }  
Object(B) B_831(int i) { let v = Object(B) in  
  this.i <- i; return this; }
```

new A:

```
a_vma  
0
```

a_vma:

```
println_451  
f_211
```

new B(5):

```
b_vma  
5
```

b_vma:

```
println_451  
f_307  
f_711
```



Interfaces

- Interfaces are like classes, but there is no order on the methods
 - Interface *S* { int f(int foo); int g(int bar); }
 - Class *A* implements *S* { int g(int i); int f(int j); }
 - Class *B* { int z(int k); int g(int m); int h(int k); }
 - Class *C* extends *B* implements *S* { int f(int i); }
- A global analysis could potentially solve this
 - Add an empty slot for *f* in *B*
- Another option: hash tables for method lookup (Sun did something like this in Java 1.1)



IR representation

- Exact representation is postponed
- Collect names, interfaces, methods, and fields

```
type class_info =  
{ class_parents : var list;  
  class_interfaces : (var * var * ty) list SymbolTable.t;  
  class_consts : (var * ty) list;  
  class_methods : ty FieldMTable.t;  
  class_fields : ty FieldMTable.t;  
}
```



Class interface list

- (var * var * ty) list SymbolTable.t
 - Each interface in the SymbolTable has a list of
 - External name for the method
 - Function implementing the method
 - Type of the method
 - Overloading introduces multiple methods with the same name (but different types)



FieldMTable

- Used for methods and fields
- Like a SymbolTable, but:
 - Each entry has external and internal names
 - Ordering is preserved

```
val add : 'a t -> symbol -> symbol -> 'a -> 'a t  
val find_ext : 'a t -> symbol -> (symbol * 'a) list  
val find_int : 'a t -> symbol -> symbol * 'a  
val to_list : 'a t -> (symbol * symbol * 'a) list
```