

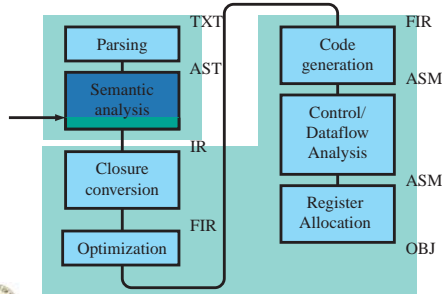
CS134b: IR

- So far:
 - Method body (expression) conversion
 - Binops, conditional
 - Implicit coercions
 - Int -> float, etc.
 - Method and function overloading
 - Application (5 cases)
 - Class and interface representation
- Finally:
 - Building interfaces and classes



Current position

- About half done



Objectives

- For interfaces, define interface types
- For classes
 - Build the global function table containing all methods and constructors
 - Define initialization functions to initialize the fields of a class
 - Build constructors
 - Build methods
 - Build class type



Interfaces

- An interface is a type specifying the types of each method
- What is the “this” type for a method in an interface?



“this” type in interfaces

- Formally, an interface has existential type:

```
interface X {  
  int f(float x);  
}  
  
∃α. {  
  vma : { f : (α, float) -> int; };  
  this : α;  
}
```



Polymorphism

- Parametric polymorphism makes the type system more complicated
 - Existential, universal types
 - Type applications
 - Value polymorphism (handling polymorphic side-effects)
 - Phase distinction
 - Very cool stuff, but best to put off
- Alternatives:
 - Use the “Object” type for “this”
 - Not safe, unless every method does a typecase on “this”
 - Fake it with “abstract” types
- Why use a type system?



Safety

- Safety spec, for some definition of "valid":
 - All data accesses are to valid locations
 - All jumps are to valid locations
- Formally
 - Subject reduction (types are preserved by evaluation)
 - Progress (evaluator never gets "stuck")
- Ideally, safety can be proved statically
- If static analysis is too hard (undecidable), have to insert runtime checks
 - Divide-by-zero (processor usually checks)
 - Array bounds (but we can usually catch 95% statically)



Abstract interface types

- IR program:

```
type prog =  
{ prog_types   : tydef SymbolTable.t; (* Classes and interfaces *)  
  prog_abstypes : SymbolSet.t;      (* Abstract interface types *)  
  prog_funs    : fun_info SymbolTable.t; (* Function table *)  
  prog_main    : var;                (* Name of "main" *)  
  prog_object  : var;                (* FjObject type *)  
}
```



Interface definition

- Every interface has
 - An abstract "this" type
 - A table of methods
 - Let's use a really big font

```
type interface_info =  
{ intf_this : var;  
  intf_methods : ty FieldMTable.t  
}
```



Converting interfaces

- Define a new abstract "this" type
- Add all the method types to a table



Interfaces, part #1

```
let build_interface_type info env name methods pos =  
  (* Make up a name for the "this" parameter *)  
  let this_sym = new_symbol_string "this" in  
  let ty_this = TyObject this_sym in  
  
  (* Collect all the methods *)  
  let table =  
    List.fold_left (fun table (v_ext, ty, pos) ->  
      let pos = string_pos "build_interface_type" (ast_pos pos) in  
      let ty = build_type env pos ty in  
      let ty_args, ty_res = dest_fun_type env pos ty in  
      let ty = TyMethod (ty_this, ty_args, ty_res) in  
      let v_int = new_symbol v_ext in  
      FieldMTable.add table v_ext v_int ty) (FieldMTable.create ()) methods  
  in
```



Interfaces, part #2

```
let intf =  
  { intf_this = this_sym;  
    intf_methods = table  
  }  
in  
let tydef = TyDefInterface (Some intf) in  
let info = info_add_tydef info name tydef in  
let info = info_add_abs_type info this_sym in  
let env = env_add_tydef env name tydef in  
info, env
```



What's info?

- A common model during conversion:
 - Define a record to hold all the new definitions
 - As definitions are encountered, add them to the info
 - At the end, use the parts of info to build the program

```
(*  
 * This is the info we collect from the program.  
 *)  
type prog_info =  
{ info_types : tydef SymbolTable.t;  
  info_abstypes : SymbolSet.t;  
  info_funs : (fun_class * ty * var list * exp) SymbolTable.t;  
}
```



Class conversion

- Lots of stuff to do
 - Collect methods, fields, parents, interfaces, old pizzas, constructors, initializers, and more...

```
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 conversion

- The class definition is flat
 - Includes
 - All the parents
 - All the interfaces, including those defined in parents,
 - All the constructors, including those defined in parents,
 - All the methods, including those defined in parents,
 - All the fields, including those defined in parents.
 - Fortunately, the parent is flat too.
- Two phases
 - Collect all the methods, fields, and constructors in AST form
 - Convert all constructors and methods, and define the initializer function (to initialize the fields)



Method definitions (phase #1)

- Add to the table

```
let f = new_symbol f in
let ty_fun = TyMethod (TyObject name, ty_vars, build_type env loc ty_res) in
{ fields with field_methods = (f, f, ty_fun, vars, body, pos) :: fields.field_methods }
```



Class conversion, part #2

- In phase #1
 - Collect a list of methods, fields, and constructors
- In phase #2
 - Expand method bodies
 - Collect field initializers into an initialization function
 - Expand constructors
 - Add default constructors



Method conversion

- Add all the vars to the env
- Convert the method body
- Add the method function to the function table



Methods, step #1

```
let build_method env info (f, f', ty, vars, body, pos) =  
  let pos = string_pos "build_method" (ast_pos pos) in  
  let ty_this, ty_vars, ty_res = dest_method_type env pos ty in  
  
  (* Add the return label with the new function name *)  
  let env = env_add_return env f' ty_res in  
  
  (* Add all the vars to the env *)  
  let env = List.fold_left2 env_add_var env vars (ty_this :: ty_vars) in
```



Methods, step #2

```
(* Build the body *)  
let e =  
  build_exp env body (fun env a ->  
    if is_unit_type env pos ty_res then  
      Return (f', AtomUnit)  
    else  
      let v = new_symbol f in  
      LetObject (v, ty_object, object_var, Raise (AtomVar v))  
  ) in
```



Methods, step #3

```
(* Add the function to the program *)  
let info = info_add_fun info f' (FunMethodClass, ty, vars, e) in  
info
```



Constructors

- Constructors are not methods
- Objects are *allocated* with *LetObject*
 - All fields have default values
- A constructor takes the initial value, and evaluates the constructor body
 - Returns *unit*