Alternate implementation strategy: compilation

Divide interpreter work into two parts:

- compile-time
- run-time

Compile-time does preprocessing

- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs

Compile-time processing

Decide layout of run-time data values

- use direct reference at precomputed offsets, not e.g. hash table lookups

Decide where variable contents will be stored

- registers
- stack frame slots at precomputed offsets
- global memory

Generate machine code to do basic operations

- just like interpreting expression, except generate code that will evaluate it later

Do optimizations across instructions if desired

Compilation plan

First, translate typechecked ASTs into linear sequence of simple statements called **intermediate code**

- a program in an **intermediate language** (IL)
- source-language, target-language independent

Then, translate intermediate code into target code

Two-step process helps separate concerns

- intermediate code generation from ASTs focuses on breaking down source-language contructs into simple and explicit pieces
- target code generation from intermediate code focuses on constraints of particular target machines

Can write many target code generators (back-ends), many language-specific front-ends sharing same IL

Can implement optimizer for IL, shared by front- and back-ends

MiniJava's intermediate language

Want intermediate language to have only simple, explicit operations, without "helpful" features

- humans won't write IL programs!
- C-like is good

Use simple declaration primitives

- global functions, global variables
- no classes, no implicit method lookup, no nesting

Use simple data types

- ints, doubles, explicit pointers, records, arrays
- no booleans
- no class types, no implicit class fields
- arrays are naked sequences; no implicit length or bounds checks

Use explicit gotos instead of control structures

Make all implicit checks explicit (e.g. array bounds checks)

Implement method lookup via explicit data structures and code

MiniJava's IL (part 1)

```
Program ::= {GlobalVarDecl} {FunDecl}

GlobalVarDecl ::= Type ID [= Value] ;

Type    ::= int | double | * Type
          | Type [ ] | { {Type ID}/, } | fun

Value   ::= Int | Double | & ID
          | [ {Value}/, ] | { {ID = Value}/, }

FunDecl ::= Type ID ( {Type ID}/, )
            { {VarDecl} {Stmt} }

VarDecl ::= Type ID ;

Stmt    ::= Expr ;
          | LHSExpr = Expr ;
          | iffalse Expr goto Label ;
          | iftrue Expr goto Label ;
          | goto Label ;
          | label Label ;
          | throw new Exception( String ) ;
          | return Expr ;
```


MiniJava's IL (part 2)

```
Expr    ::= LHSExpr
          | Unop Expr
          | Expr Binop Expr
          | Callee ( {Expr}/, )
          | new Type [[ Expr ]]
          | Int
          | Double
          | & ID

LHSExpr ::= ID
          | * Expr
          | Expr -> ID [[ Expr ]]

Unop    ::= -.int | -.double | not | int2double

Binop   ::= (+|-|*|/).(int|double)
          | (<|<=|>=|>|==|!=).(int|double)
          | <.unsigned

Callee  ::= ID
          | ( * Expr )
          | String
```


Intermediate code generation in MiniJava

Choose representations for source-level data types

- translate each `ResolvedType` into `ILType`(s)

Recursively traverse ASTs, creating corresponding IL program

- `Expr` ASTs create `ILExpr` ASTs
- `Stmt` ASTs create `ILStmt` ASTs
- `MethodDecl` ASTs create `ILFunDecl` ASTs
- `ClassDecl` ASTs create `ILGlobalVarDecl` ASTs
- `Program` ASTs create `ILProgram` ASTs

Traversal parallels typechecking and evaluation traversals

ICG operations on (source) ASTs named `lower`

IL AST classes in `IL` subdirectory

Data type representation (part 1)

What IL type to use for each source type?

- (what operations are we going to need on them?)

`int`:

`boolean`:

`double`:

Data type representation (part 2)

What IL type to use for each source type?

- (what operations are we going to need on them?)

Example:

```
class B {
   int i;
   D j;
}
```

instance of class `B`:

Inheritance

How to lay out subclasses?

- subclass inherits features of superclass
- subclass can be assigned to variable of superclass's type
  ⇒ subclass layout must "match" superclass's layout

Example:

```
class B {
   int i;
   D j;
}
class C extends B {
   int x;
   F y;
}
```

instance of class `C`:

Methods

How to translate a method?

Use a function

- name is "mangled": name of class + name of method

Make `this` an explicit argument

Example:

```
class B {
   ...
   int m(int i, double d) { ... body ... }
}
```

`B`'s method `m` translates to

```
int B_m(*{...B...} this, int i, double d) {
   ... translation of body ... }
```


Methods in instances

To support run-time method lookup, need to make method function pointers accessible from each instance

Build a record of pointers to functions for each class, with members for each of a class's methods (a.k.a. virtual function table, or vtbl)

Example:

```
class B {
   ...
   int m(...) { ... }
   E n(...) { ... }
}
```

`B`'s method record value:

```
{ *fun m = &B_m, *fun n = &B_n }
```

Method inheritance

A subclass inherits all the methods of its superclasses

- its method record includes all fields of its superclass

Overriding methods in subclass share same member of superclass, but change its value

Example:

```
class B {
   ...
   int m(...) { ... }
   E n(...) { ... }
}
class C extends B {
   ...
   int m(...) { ... }  // override
   F p(...) { ... }
}
```

`B`'s method record value:

```
{ *fun m = &B_m, *fun n = &B_n }
```

`C`'s method record value:

```
{ *fun m = &C_m, *fun n = &B_n, *fun p = &C_p }
```


Shared method records

Every instance of a class shares same method record value
⇒ each instance stores a pointer to class's method record

`B`'s instance layout (type):

```
*{ *{ *fun m, *fun n } vtbl,
   int i,
   *{...D...} j }
```

`C`'s instance layout (type):

```
*{ *{ *fun m, *fun n, *fun p } vtbl,
   int i,
   *{...D...} j,
   int x,
   *{...F...} y }
```

`C`'s vtbl layout extends `B`'s

`C`'s instance layout extends `B`'s

`B` instances' `vtbl` field initialized to `B`'s vtbl record

`C` instances' `vtbl` field initialized to `C`'s vtbl record

Method calls

Translate a method invocation on an instance into
a lookup in the instance's vtbl
then an indirect function call

Example:

```
B b;
...
b.m(3, 4.5)
```

Translates to

```
*{ *{ *fun m, *fun n } vtbl,
   int i,
   *{...D...} j } b;
...
*{ *fun m, *fun n } b_vtbl = b->vtbl;
*fun b_m = b_vtbl->m;
(*b_m)(b, 3, 4.5)
```


Data type representation (part 3)

What IL type to use for each source type?

- (what operations are we going to need on them?)

array of *T*: