Intermediate Formats

for object oriented languages
Program Representation Goals

• Enable Program Analysis and Transformation
  – Semantic Checks, Correctness Checks, Optimizations
• Structure Translation to Machine Code
  – Sequence of Steps
High Level IR

• Preserves Object Structure
• Preserves Structured Flow of Control
• Primary Goal: Analyze Program

Low Level IR

• Moves Data Model to Flat Address Space
• Eliminates Structured Control Flow
• Suitable for Low Level Compilation Tasks
  – Register Allocation
  – Instruction Selection
Examples of Object Representation and Program Execution
(This happens when program runs)
Example Vector Class

class vector {
    int v[ ];

    ...}

void add(int x) {
    int i;
    i = 0;

    while (i < v.length) {
        v[i] = v[i]+x; i = i+1;
    }

}
Representing Arrays

• Items Stored Contiguously In Memory
• Length Stored In First Word

\[ \begin{array}{cccc}
3 & 7 & 4 & 8 \\
\end{array} \]

• Color Code
  – Red - generated by compiler automatically
  – Blue, Yellow, Lavender - program data or code
  – Magenta - executing code or data
Representing Vector Objects

- First Word Points to Class Information
  - Method Table, Garbage Collector Data

- Next Words Have Object Fields
  - For vectors, Next Word is Reference to Array
Invoking Vector Add Method

vect.add(1);

- Create Activation Record
Invoking Vector Add Method

```java
vect.add(1);
```

- Create Activation Record
  - `this` onto stack
Invoking Vector Add Method

```java
vect.add(1);
```

- Create Activation Record
  - this onto stack
  - parameters onto stack
Invoking Vector Add Method

```java
vect.add(1);
```

- Create Activation Record
  - `this` onto stack
  - parameters onto stack
  - space for locals on stack
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Executing Vector Add Method

Class Info

1
x
this
i

3 7 4 8
Executing Vector Add Method

```java
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}
```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Class Info

1
0
this
x
i

3 7 4 8
void add(int x) {
    int i;
    i = 0;
    while (i < v.length) {
        v[i] = v[i]+x;
        i = i+1;
    }
}
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

Executing Vector Add Method
void add(int x) {
    int i;
    i = 0;
    while (i < v.length) {
        v[i] = v[i] + x;
        i = i + 1;
    }
}
Executing Vector Add Method

```java
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}
```
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

Class Info

this
1
0
x
i

3 7 4 8
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

Class Info

1
0
this
x
i
Executing Vector Add Method

void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

Class Info

this
1
0
x
i
3 7 4 8
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

1 0

this
x
i

Class Info

3 8 4 8
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i] + x;
    i = i + 1;
}

Class Info
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

Class Info

this

x

i

v

1

1

3 8 4 8
void add(int x) {
    int i;
    i = 0;
    while (i < v.length)
        v[i] = v[i]+x;
    i = i+1;
}

1
3
x
i

this

3 8 5 9

Class Info
What does the compiler have to do to make all of this work?
Compilation Tasks

- Determine Format of Objects and Arrays
- Determine Format of Call Stack
- Generate Code to Read Values
  - this, parameters, locals, array elements, object fields
- Generate Code to Evaluate Expressions
- Generate Code to Write Values
- Generate Code for Control Constructs
Compilation Tasks

• Determine Object Format in Memory
  – Fields from Parent Classes
  – Fields from Current Class

• Generate Code for Methods
  – Field, Local Variable and Parameter Accesses
  – Method Invocations
Symbol Tables - Key Concept in Compilation

- Compiler Uses Symbol Tables to Produce
  - Object Layout in Memory
  - Code to
    - Access Object Fields
    - Access Local Variables
    - Access Parameters
    - Invoke Methods
Symbol Tables During Translation From Parse Tree to IR

• Symbol Tables Map Identifiers (strings) to Descriptors (information about identifiers)

• Basic Operation: Lookup
  – Given A String, find Descriptor
  – Typical Implementation: Hash Table

• Examples
  – Given a class name, find class descriptor
  – Given variable name, find descriptor
    • local descriptor, parameter descriptor, field descriptor
Hierarchy In Symbol Tables

• Hierarchy Comes From
  – Nested Scopes - Local Scope Inside Field Scope
  – Inheritance - Child Class Inside Parent Class

• Symbol Table Hierarchy Reflects These Hierarchies

• Lookup Proceeds Up Hierarchy Until Descriptor is Found
Hierarchy in vector add Method

Symbol Table for Fields of vector Class

Symbol Table for Parameters of add

Symbol Table for Locals of add

v → descriptor for field v

x → descriptor for parameter x

this → descriptor for this

i → descriptor for local i
Lookup In vector Example

- $v[i] = v[i] + x$;

$v$ descriptor for field $v$

$x$ descriptor for parameter $x$

$this$ descriptor for this

$i$ descriptor for local $i$
• $v[i] = v[i]+x$;
• $v[i] = v[i] + x$;
Lookup x In vector Example

- $v[i] = v[i] + x$;
Lookup x In vector Example

- $v[i] = v[i] + x$;
Lookup x In vector Example

- \( v[i] = v[i] + x; \)

\( v \) descriptor for field \( v \)

\( x \) descriptor for parameter \( x \)

\( \text{this} \) descriptor for \( \text{this} \)

\( i \) descriptor for local \( i \)
Descriptors

• What do descriptors contain?
• Information used for code generation and semantic analysis
  – local descriptors - name, type, stack offset
  – field descriptors - name, type, object offset
  – method descriptors
    • signature (type of return value, receiver, and parameters)
    • reference to local symbol table
    • reference to code for method
Program Symbol Table

- Maps class names to class descriptors
- Typical Implementation: Hash Table

![Diagram showing class names mapped to class descriptors]

- vector
- point
- cartesianPoint
- polarPoint

class descriptor for vector
class descriptor for point
class descriptor for cartesianPoint
class descriptor for polarPoint
Class Descriptor

• Has Two Symbol Tables
  – Symbol Table for Methods
    • Parent Symbol Table is Symbol Table for Methods of Parent Class
  – Symbol Table for Fields
    • Parent Symbol Table is Symbol Table for Fields of Parent Class

• Reference to Descriptor of Parent Class
Field, Parameter and Local and Type Descriptors

- Field, Parameter and Local Descriptors Refer to Type Descriptors
  - Base type descriptor: int, boolean
  - Array type descriptor, which contains reference to type descriptor for array elements
  - Class descriptor

- Relatively Simple Type Descriptors
- Base Type Descriptors and Array Descriptors Stored in Type Symbol Table
Example Type Symbol Table

- int
- int []
- boolean
- boolean []
- vector
- vector []

- int descriptor
- array descriptor
- boolean descriptor
- array descriptor
- array descriptor
- class descriptor for vector
Method Descriptors

- Contain Reference to Code for Method
- Contain Reference to Local Symbol Table for Local Variables of Method
- Parent Symbol Table of Local Symbol Table is Parameter Symbol Table for Parameters of Method
Method Descriptor for add Method

- field symbol table for vector class
- parameter symbol table
- this descriptor
- parameter descriptor
- local variable symbol table
- local descriptor
- code for add method
- Method descriptor for add
Symbol Table Summary

• Program Symbol Table (Class Descriptors)

• Class Descriptors
  – Field Symbol Table (Field Descriptors)
    • Field Symbol Table for SuperClass
  – Method Symbol Table (Method Descriptors)
    • Method Symbol Table for Superclass

• Method Descriptors
  – Local Variable Symbol Table (Local Variable Descriptors)
    • Parameter Symbol Table (Parameter Descriptors)
      – Field Symbol Table of Receiver Class

• Local, Parameter and Field Descriptors
  – Type Descriptors in Type Symbol Table or Class Descriptors
class descriptor
for vector

class_decl

vector

field_decl

int v []
class descriptor for vector

field descriptor

field symbol table

parameter descriptor

parameter symbol table

this descriptor

class _decl

vector

field _decl

int v []
class descriptor for vector

add

field descriptor

method descriptor for add

method symbol table

local symbol table

code for add method

int

int []

boolean

boolean []

vector []

field symbol table

parameter symbol table

parameter descriptor

this descriptor

local descriptor

int descriptor

array descriptor

boolean descriptor

array descriptor

array descriptor

vector []
Representing Code in High-Level Intermediate Representation
Basic Idea

• Move towards assembly language
• Preserve high-level structure
  – object format
  – structured control flow
  – distinction between parameters, locals and fields
• High-level abstractions of assembly language
  – load and store nodes
  – access abstract locals, parameters and fields, not memory locations directly
What is a Parse Tree?

• Parse Tree Records Results of Parse
• External nodes are terminals/tokens
• Internal nodes are non-terminals

```plaintext
class_decl ::= 'class' name '{' field_decl method_decl '}'
field_decl ::= 'int' name '[]';
method_decl ::= 'void' name '(' param_decl ')' '{' var_decl stats '}'
```
Abstract Versus Concrete Trees

- Remember grammar hacks
  - left factoring, ambiguity elimination, precedence of binary operators
- Hacks lead to a tree that may not reflect cleanest interpretation of program
- May be more convenient to work with abstract syntax tree (roughly, parse tree from grammar before hacks)
Building IR Alternatives

• Build concrete parse tree in parser, translate to abstract syntax tree, translate to IR
• Build abstract syntax tree in parser, translate to IR
• Roll IR construction into parsing
From Abstract Syntax Trees to Symbol Tables

- Recursively Traverse Tree
- Build Up Symbol Tables As Traversal Visits Nodes
Representing Expressions

• Expression Trees Represent Expressions
  – Internal Nodes - Operations like +, -, etc.
  – Leaves - Load Nodes Represent Variable Accesses

• Load Nodes
  – ldf node for field accesses - field descriptor
    • (implicitly accesses this - could add a reference to accessed object)
  – ldl node for local variable accesses - local descriptor
  – ldp node for parameter accesses - parameter descriptor
  – lda node for array accesses
    • expression tree for array
    • expression tree for index
Example

\[ x^2 + y^2 \]

Field descriptor for \( x \) in field symbol table for cartesianPoint class

Field descriptor for \( y \) in field symbol table for cartesianPoint class
Example

\[ v[i] + x \]

- **lda**: local descriptor for \( i \) in local symbol table of vector add
- **ldl**: local descriptor for \( i \) in local symbol table of vector add
- **ldf**: field descriptor for \( v \) in field symbol table for vector class
- **ldp**: parameter descriptor for \( x \) in parameter symbol table of vector add
Special Case: Array Length

- len node represents length of array
  - expression tree for array

- Example: v.length

\[
\text{len} \downarrow \\
\text{ldf} \downarrow \\
\text{field descriptor for } v \\
\text{in field symbol table} \\
\text{for vector class}
\]
Representing Assignment Statements

• Store Nodes
  – stf for stores to fields
    • field descriptor
    • expression tree for stored value
  – stl for stores to local variables
    • local descriptor
    • expression tree for stored value
  – sta for stores to array elements
    • expression tree for array
    • expression tree for index
    • expression tree for stored value
Representing Procedure Calls

- Call statement
- Refers to method descriptor for invoked method
- Has list of parameters (this is first parameter)

vect.add(1)

- method descriptor for add in method symbol table for vector class
- local descriptor for vect in local symbol table of method containing the call statement vect.add(1)
- constant 1
Example

\[ v[i] = v[i] + x \]

- **sta**: local descriptor for \( i \) in local symbol table of vector add
- **ldf**: field descriptor for \( v \) in field symbol table for vector class
- **ldl**: local descriptor for \( v \) in local symbol table of vector add
- **lda**: parameter descriptor for \( x \) in parameter symbol table of vector add
- **ldp**:
Representing Flow of Control

- **Statement Nodes**
  - sequence node - first statement, next statement
  - if node
    - expression tree for condition
    - then statement node and else statement node
  - while node
    - expression tree for condition
    - statement node for loop body
  - return node
    - expression tree for return value
Example

while (i < v.length)

\[ v[i] = v[i] + x; \]
Translating from Abstract Syntax Trees to Symbol Tables
Example Abstract Syntax Tree

class vector {
    int v[];
    void add(int x) {
        int i; i = 0;
        while (i < v.length) { v[i] = v[i]+x; i = i+1; }
    }
}

```
```
class_decl
  └── vector
      └── field_decl
          └── int v
  └── method_decl
      └── statements
          ├── add
              └── param_decl
                  └── int x
          └── var_decl
                  └── int i
```plaintext
class Decl
  vector Decl
    int v
  field Decl
    method Decl
      add Decl
        param Decl
          int x
        var Decl
          int i
  statements

vector

class symbol table

class descriptor for vector

Method descriptor for add

field descriptor

this descriptor

this
```
vector field descriptor

class descriptor for vector

add method descriptor for add

vector field decl

int v

method decl

add param decl

int x

var decl

int i

field descriptor

x this

this descriptor

local descriptor

i
From Abstract Syntax Trees to Intermediate Representation
while (i < v.length)
  v[i] = v[i]+x;
while (i < v.length)
    v[i] = v[i] + x;
while (i < v.length)
  v[i] = v[i]+x;
while (i < v.length)
    v[i] = v[i] + x;

field descriptor for v  local descriptor for i  parameter descriptor for x
while (i < v.length)
    v[i] = v[i] + x;

field descriptor for v  local descriptor for i  parameter descriptor for x
while (i < v.length)
  v[i] = v[i]+x;

field descriptor for v  local descriptor for i  parameter descriptor for x
while (i < v.length)
    v[i] = v[i] + x;
while (i < v.length)

v[i] = v[i] + x;
while (i < v.length) 

\[ v[i] = v[i] + x; \]
while (i < v.length)
    v[i] = v[i] + x;

<

ldl len

ldf

lda ldp

ldf ldl

field descriptor for v
local descriptor for i
parameter descriptor for x
while (i < v.length)
\[v[i] = v[i] + x;\]
while (i < v.length)  

    v[i] = v[i] + x;

field descriptor for v  
local descriptor for i  
parameter descriptor for x
while (i < v.length)  
  v[i] = v[i]+x;

field descriptor for v  
local descriptor for i  
parameter descriptor for x
while (i < v.length)

v[i] = v[i] + x;

while

<

ldl
len
ldf
ldl
sta
+

ldf
ldl
lda
ldp

field descriptor for v
local descriptor for i
parameter descriptor for x
while (i < v.length)

v[i] = v[i] + x;

while

<

ldl

len

ldf

ldl

sta

+

ldf

ldf

ldl

ldf

ldl

lda

ldp

field descriptor for v

local descriptor for i

parameter descriptor for x
while (i < v.length)
    v[i] = v[i]+x;

while 
    sta 
        < 
            ldf v 
            ldl i 
            lda 
            ldp x 
            ldf v 
            ldl i 
            len
From Abstract Syntax Trees to IR

• Recursively Traverse Abstract Syntax Tree
• Build Up Representation Bottom-Up Manner
  – Look Up Variable Identifiers in Symbol Tables
  – Build Load Nodes to Access Variables
  – Build Expressions Out of Load Nodes and Operator Nodes
  – Build Store Nodes for Assignment Statements
  – Combine Store Nodes with Flow of Control Nodes
High-Level Intermediate Representation

• Goal: represent program in an intuitive way that supports future compilation tasks

• Representing program data
  – Symbol tables
  – Hierarchical organization

• Representing computation
  – Expression trees
  – Various types of load and store nodes
  – Structured flow of control

• Traverse abstract syntax tree to build IR
Eliminating Parse Tree Construction

• Parser actions build symbol tables
  – Reduce actions build tables in bottom-up fashion
  – Actions correspond to activities that take place in top-down fashion in parse tree traversal
• Eliminates intermediate construction of parse tree - improves performance
• Also less code to write (but code may be harder to write than if just traverse parse tree)
class vector {
    int v[];
    void add(int x) {
        int i;
    }
}

class symbol

table
class symbol table

field decl

int v

field descriptor

class vector { int v[]; void add(int x) { int i; ... }}
class vector { int v[]; void add(int x) { int i; ... }}

class symbol table

field_decl

int v

param_decl

int x

field descriptor

parameter descriptor
class vector { int v[]; void add(int x) { int i; ... }}
class vector { int v[]; void add(int x) { int i; ... } }
class vector { int v[]; void add(int x) { int i; ... }}

```
#include <vector>

class Vector {
public:
    Vector();
    void add(int x);
private:
    std::vector<int> v;
};
```

**Method descriptor for add**

```
    method_decl
        add
        param_decl
            int x
        var_decl
            int i
```

**Field descriptor**

```
    field_decl
        int v
```

**Parameter descriptor**

```
    param_decl
```

**Local descriptor**

```
    var_decl
```

**Code for add method**

```
    void add(int x) {
        int i = x;
        // ... code...
    }
```
class vector { int v[]; void add(int x) { int i; ... }}
Nested Scopes

• So far, have seen several kinds of nesting
  – Method symbol tables nested inside class symbol tables
  – Local symbol tables nesting inside method symbol tables

• Nesting disambiguates potential name clashes
  – Same name used for class field and local variable
  – Name refers to local variable inside method
Nested Code Scopes

• Symbol tables can be nested arbitrarily deeply with code nesting:

```java
class bar {
    baz x;
    int foo(int x) {
        double x = 5.0;
        { float x = 10.0;
            { int x = 1; ... x ... }
            ... x ...
        }
        ... x ...
    }
    ... x ...
}
```

Note: Name clashes with nesting can reflect programming error. Compilers often generate warning messages if it occurs.
Traversing Class Declarations

- Extract Class Name and Superclass Name
- Create Class Descriptor (field and method symbol tables), Put Descriptor Into Class Symbol Table
- Put Array Descriptor Into Type Symbol Table
- Lookup Superclass Name in Class Symbol Table, Make Superclass Link in Class Descriptor Point to Retrieved Class Descriptor
- Traverse Field Declarations to Fill Up Field Symbol Table
- Traverse Method Declarations to Fill Up Method Symbol Table
Further Complication - Inheritance

Object Extension
Inheritance Example - Point Class

class point {
    int c;
    int getColor() { return(c); }
    int distance() { return(0); }
}

class cartesianPoint extends point{
    int x, y;
    int distance() { return(x*x + y*y); }
}

class polarPoint extends point {
    int r, t;
    int distance() { return(r*r); }
    int angle() { return(t); }
}
Implementing Object Fields

- Each object is a contiguous piece of memory
- Fields from inheritance hierarchy allocated sequentially in piece of memory
- Example: polarPoint object
Point Objects

- **c**: 2
- **c**: 1
- **x**: 4
- **y**: 6
- **c**: 4
- **r**: 1
- **t**: 3

Class Info:
- **point**
- **cartesianPoint**
- **polarPoint**
Class Descriptors for point and cartesianPoint

class descriptor for point

- c
- field descriptor for c
- method descriptor for getColor
- method descriptor for distance

x
- field descriptor for x

y
- field descriptor for y

distance
- method descriptor for distance

class descriptor for cartesianPoint
if (x == 0) {
    p = new point();
} else if (x < 0) {
    p = new cartesianPoint();
} else if (x > 0) {
    p = new polarPoint();
}
y = p.distance();

Which distance method is invoked?

- if p is a point
  return(0)
- if p is a cartesianPoint
  return(x*x + y*y)
- if p is a polarPoint
  return(r*r)
- Invoked Method Depends on Type of Receiver!
Implementing Dynamic Dispatch

• Basic Mechanism: Method Table

  method table for point objects
  getColor method for point
  distance method for point

  method table for cartesianPoint objects
  getColor method for point
  distance method for cartesianPoint

  method table for polarPoint objects
  getColor method for point
  distance method for polarPoint
  angle method for polarPoint
Invoking Methods

• Compiler Numbers Methods In Each Inheritance Hierarchy
  – getColor is Method 0, distance is Method 1, angle is Method 2

• Method Invocation Sites Access Corresponding Entry in Method Table

• Works For Single Inheritance Only
  – not for multiple inheritance, multiple dispatch, or interfaces
Hierarchy in Method Symbol Tables for Points

- getColor
  - method descriptor for getColor
  - method descriptor for distance

- distance
  - method descriptor for distance

- angle
  - method descriptor for angle
Lookup In Method Symbol Tables

• Starts with method table of declared class of receiver object
• Goes up class hierarchy until method found
  – point p; p = new point(); p.distance();
    • finds distance in point method symbol table
  – point p; p = new cartesianPoint(); p.distance();
    • finds distance in point method symbol table
  – cartesianPoint p; p = new cartesianPoint();
    p.getColor();
    • finds getColor in point method symbol table
Static Versus Dynamic Lookup

• Static lookup done at compile time for type checking and code generation
• Dynamic lookup done when program runs to dispatch method call
• Static and dynamic lookup results may differ!
  – point p; p = new cartesianPoint(); p.distance();
    • Static lookup finds distance in point method table
    • Dynamic lookup invokes distance in cartesianPoint class
    • Dynamic dispatch mechanism used to make this happen
Static and Dynamic Tables

• Static Method Symbol Table
  – Used to look up method definitions at compile time
  – Index is method name
  – Lookup starts at method symbol table determined by declared type of receiver object
  – Lookup may traverse multiple symbol tables

• Dynamic Method Table
  – Used to look up method to invoke at run time
  – Index is method number
  – Lookup simply accesses a single table element