Compiler Construction 2009/2010: Intermediate Representation

Annette Bieniusa

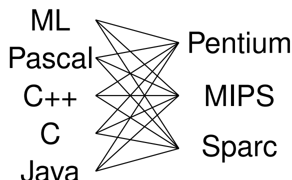
November 17, 2009

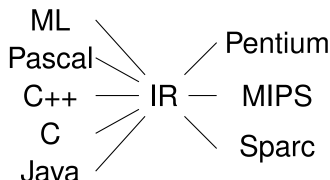
Outline

1. Intermediate representation
2. Registers, heap and stack frames
3. Memory layout

Motivation

We could go directly from the AST to machine code, but ...





Intermediate representation

- front end: lexical analysis, parsing, semantic analysis
- back end: machine specific optimization, translation to machine language
- intermediate code: machine and language independent optimization

Specifics of Intermediate Representation

A good IR is

- convenient to produce from AST;
- convenient to translate into machine language;
- small, with clear and simple semantics.

Main differences: AST vs. IR

- Conditionals: if-then-else vs. comparisons and conditional jumps
- Method calls: various number of arguments vs. simple call ($\rightarrow$ activation frames)
- Memory layout: array and field deferencing vs. load/store on heap or stack

IR: Expressions

CONST(i)	integer constant i
NAME(n)	symbolic constant n [code label]
TEMP(t)	temporary t, one of arbitrary many ”registers”
BINOP(o,e_1,e_2)	binary operator o with operands e_1 and e_2
MEM(e)	contents of a word of memory at address e
CALL(f,[$e_1, \ldots, e_n$])	procedure call
ESEQ(s,e)	expression sequence; evaluate statement s for side-effects, expression e for result

IR: Statements

MOVE(TEMP(t), e)	Evaluate e and move it into t.
MOVE(MEM (e_1),e_2)	Evaluate e_1 yielding address a; evaluate e_2 and move it into a.
EXP(e)	Evaluate e and discard result.
JUMP(e,[$l_1, \ldots, l_n$])	Transfer control (jump) to address e; $l_1, \ldots, l_n$ are all possible values for e. Often used: JUMP(l).
CJUMP(o,e_1,e_2,t,f)	Evaluate e_1, then e_2; compare their results using relational operator o. If true, jump to label t, else jump to label f.
SEQ(s_1,s_2)	Statement s_1 followed by statement s_2.
LABEL(n)	Define constant value of name n as current code address. NAME(n) can then be used as targets of jumps, calls, etc.

IR: Operators

Binary arithmetic and logical operators:

PLUS, MINUS, MUL, DIV	integer arithmetic operators
AND, OR, XOR	integer bitwise logical operators
LSHIFT, RSHIFT	integer logical shift operators
ARSHIFT	integer arithmetic right-shift

Relational operators:

EQ, NE	integer equality and non-equality (signed or unsigned)
LT, GT, LE, GE	integer inequalities (signed)
ULT, UGT, ULE, UGE	integer inequalities (unsigned)

Examples

Translate the following MiniJava statements to IR:

1. `if (x < y) x = y; else x = 0;`
2. `y = z[4];`

Examples

1. `if (x < y) x = y; else x = 0;`

    - Assume, *x* corresponds to TEMP 5, y corresponds to TEMP 27.
    - Define three (new) label names *L*1, *L*2, and *L*3.

```
     CJUMP (LT, TEMP 5, TEMP 27, L1, L2)
L1   MOVE  (TEMP 5, TEMP 27)
     JUMP L3
L2   MOVE  (TEMP 5, CONST 0)
L3   ...
```

Examples

2. `y = z[4];`

- Assume *y* corresponds to TEMP 27, and the array *z* is at memory location MEM *a*.
- Let *w* be the word size of MiniJava (e.g. 4 bytes).
- Calculate the offset for array index *i*.

```
MOVE (TEMP 27, +(MEM a, *(CONST 4, CONST w)))
```

Here, we use `o(e1,e2)` as abbreviation for `BINOP(o,e1,e2)`.

Outline

1. Intermediate representation
2. Registers, heap and stack frames
3. Memory layout

Concepts of Memory layout

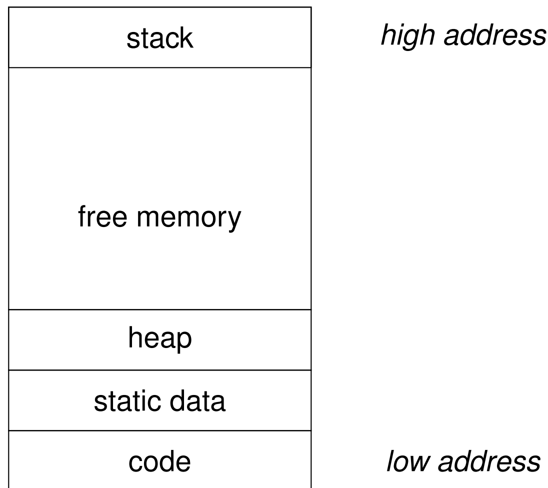
Registers store local variables and temporary results; pass parameters and return results (for function calls), depending on the architecture's calling conventions.

Heap area of memory used for dynamic memory allocation (e.g. arrays, objects)

Stack frames maintained in program's virtual address space

Non-local data can be either referenced via static links to stack locations (also as local data of other frames), or to heap locations.

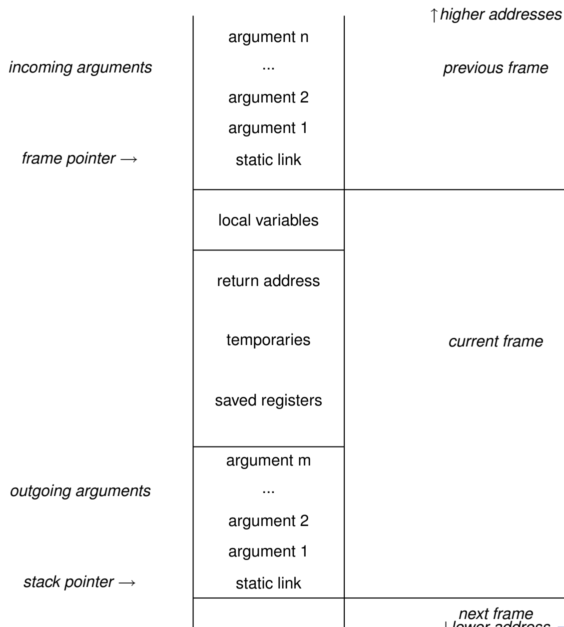
Traditional heap - stack arrangement



high address

low address

Stack frames



When calling a function...

The following actions are divided between the caller and the callee:

1. Evaluates actual arguments and puts values on the top of the caller's SF.
2. Stores return address in caller's SF (sometimes in the callee's SF).
3. Stores the caller's frame pointer register in callee's SF.
4. Modifies the frame pointer fp, making it point to callee's SF.
5. Modifies the stack pointer sp, making it point to the to top of the stack.
6. Go to callee's first instruction.
7. Callee begins execution.

When exciting a function...

1. Caller needs to retrieve the function return value.
2. Restores saved stack pointer for caller.
3. Restores saved register contents for caller.
4. Return to the caller.

Calling conventions

- Modern machines have a large set of registers (typically 32 registers).
- Register access is faster than memory loads and stores.
- Most functions have few parameters. Therefore, use small number of registers to pass parameters. The rest of the parameters, if any, can be passed in the stack.
- Returning function's results through registers.
- Caller-safe registers: caller is responsible to save and restore register contents.
- Callee-safe registers: callee is responsible to save and restore register contents.
- Convention is described in machine architecture manual.

When are variables written to memory?

- Variables passed by reference need to have a memory address ($\rightarrow$ escaping vars).
- Variables accessed by a procedure nested inside the current one.
- Values which are too big to fit into a single register.
- Variable is an array ($\rightarrow$ address arithmetic).
- Register holding the variable is needed for specific purpose.
- There are too many local variables and temporary values to fit all in registers ($\rightarrow$ spilling).

Outline

1. Intermediate representation
2. Registers, heap and stack frames
3. Memory layout

Memory layout

Pointers/References

- Size is given by the natural word size of the given machine architecture.

Basic data types

- Integers are scalar, i.e. they occupy one word each.
- Boolean false is represented as 0, true by every non-zero value (e.g. 1).
- Other data types may be padded.

Memory layout

Strings

- Typically implemented statically at constant address of a segment of memory.
- In Java byte code, strings are collectively put into the constant pool.
- In assembly language, referred to by a label.
- PASCAL: fixed-length arrays of characters
- C: zero-terminated array of characters, variable length

Memory layout

Arrays (one-dimensional)

1. Size: reserve one word for the size of the array.
2. Entries: reserve space for entry of the array.

E.g. `new int[4]`

4	*size*
0	*array base*
0	
0	
0	

Memory layout

Objects

1. Methods: pointer to the *vtable* (virtual method table) of the corresponding class.
2. Fields: reserve space for fields of the class and for fields of the super classes

Memory layout

For OO languages with single-inheritance, a *prefixing* technique is used.

```
1 class A           {int x = 0; int f() {...} }
2 class B extends A {int g() {...} }
3 class C extends B {int g() {...} }
4 class D extends C {int y = 0; int f() {...} }
```