CS 403: Intermediate Representations and Code Generation

Stefan D. Bruda

Winter 2015

Intermediate Representations

- Code generation is typically not done directly
- The functioning of a compiler is typically split into a front-end and a back-end
  - The interface between the front- and the back-end is the intermediate representation (or IR) = an assembly-like language, only nicer
  - A compiler collection for s languages and running on t architectures would need $s \times t$ specific compilers, but only needs $s + t$ compilers when an IR is used (s front-ends and t back-ends)
  - Code optimization easier to do with an intermediate representation
  - The IR can also serve as a (portable) reference definition for the language being compiled
  - The IR simplifies the task of porting a compiler to a new platform
- There are many intermediate representations in use (almost as many as compilers)
  - They are actually more alike than they are different – once you become familiar with one it is not hard to learn others
  - IRs are categorized according to where they fall between a high-level language and machine code; we thus have high-level and low-level IRs

Notable IR Examples

- GCC IR = RTL (register-transfer language)
  - LISP-like textual syntax and also binary (internal) representation
  - Fairly low-level IR
  - Assumes a general purpose register machine and incorporates some notion of register allocation and instruction scheduling
  - The gcc compiler does most of its optimizations on the RTL representation, saving only machine-dependent tweaks to be done as part of final code generation

Notable IR Examples

- GCC IR = RTL (register-transfer language)
  - LISP-like textual syntax and also binary (internal) representation
  - Fairly low-level IR
  - Assumes a general purpose register machine and incorporates some notion of register allocation and instruction scheduling
  - The gcc compiler does most of its optimizations on the RTL representation, saving only machine-dependent tweaks to be done as part of final code generation
- Java bytecode
  - Fairly high-level IR – see textbook for details
  - Based on a stack-based machine architecture
  - Includes abstract notions such as `getstatic` and `invokevirtual` along with more low-level instructions such as `ldc` (load constant) and `add`
  - Java bytecode is usually not translated into assembly language, but executed by a Java virtual machine instead
  - Some compilers however do translate it into assembly (e.g. `gcj`)

RTL Example

```
;; Function main
   (note 3 2 4 "" NOTE_INSN_FUNCTION_BEG)
   (note 6 4 7 0 NOTE_INSN_BLOCK_BEG)
   (insn 7 6 8 (set (reg:SI 106)
           (high:SI (symbol_ref:SI ("*.LLC0")))) -1 (nil)
       (nil))
   (insn 8 7 10 (set (reg:SI 8 %o0)
           (lo_sum:SI (reg:SI 106)
               (symbol_ref:SI ("*.LLC0")))) -1 (nil)(nil))
   (call_insn 10 8 12 (parallel[
               (set (reg:SI 8 %o0)
           (call (mem:SI (symbol_ref:SI ("printf")) 0) (const_int 0 [0x0])))
               (clobber (reg:SI 15 %o7))
           ] ) -1 (nil)
       (nil)
       (expr_list (use (reg:SI 8 %o0))
           (nil)))
   (note 12 10 13 0 NOTE_INSN_BLOCK_END)
   (note 13 12 15 "" NOTE_INSN_FUNCTION_END)
```

Java Bytecode Example

```
Method Main()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello world">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 return
```

Code Generation

- Code generation translates an AST into some low-ish level language (IR or assembly)
- It is a typical recursive walk through the tree
- Examples:
  - To translate $E_1 + E_2$, generate code for E_1 and E_2 recursively, then generate code for addition
  - To translate `while` (E) B generate code for `E` (recursively), generate code for branching to the end of the loop when E is false, generate code for B (recursively), then generate a jump at the beginning of the code for E
  - Etc.
- The actual process of generating code depends heavily on the target language

Three-Address Code (TAC)

- Three-address code (TAC) is essentially a generic assembly language
- Falls in the lower-end of the mid-level IRs
- Variants commonly used as an IR, since it maps well to most assembly languages
  - A TAC instruction can have at most three operands
  - The operands could be two operands to a binary arithmetic operator with the third being the result location, or an operand to compare to zero and a second location to branch to, etc.

High-level	TAC

`a = b * c + b * d;`	`_t1 = b * c;`
	`_t2 = b * d;`
	`_t3 = _t1 + _t2;`
	`a = _t3;`

`if (a < b + c)`
`  a = a - c;`
`c = b * c;`
	`_t1 = b + c;`
	`_t2 = a < _t1;`
	`IfZ _t2 Goto _L0;`
	`_t3 = a - c;`
	`a = _t3;`
	`_L0: _t4 = b * c;`
	`c = _t4;`

TAC Examples (cont'd)

High-level	TAC

`n = ReadInteger();`
`Foo(vec[n]);`
	`_t0 = LCall _ReadInteger;`
	`n = _t0;`
	`_t1 = 4;`
	`_t2 = _t1 * n;`
	`_t3 = vec + _t2;`
	`_t4 = *(_t3);`
	`PushParam _t4;`
	`LCall _Foo;`
	`PopParams 4;`

DECAF TAC INSTRUCTIONS

- Assignment: `t2 = t1;  t1 = "abcdefg";  t3 = L0;`
  - The rvalue can be a variable, literal, or label
- Arithmetic: `t3 = t2 + t1;  t3 = t2 - t1;  t3 = t2 * t1;`
  `t3 = t2 / t1;  t3 = t2 % t1;`
  - Other operators must be synthesized using the available primitives
- Relational, equality, and logical: `t3 = t2 == t1;  t3 = t2 < t1;`
  `t3 = t2 && t1;  t3 = t2 || t1;`
  - Other operators must be synthesized using the available primitives
- Labels and branches: `L1:  Goto L1;  IfZ t1 Goto L1;` (branch if `t1` is zero)
- Parameters:
  - Before making a call parameters must be pushed from right to left: `PushParam t1;`
  - Upon returning from the call the parameters must be popped using `PopParams x;` where `x` is the number of bytes to be popped
- Function/method call: `LCall L1;  t1 = LCall L1;`
  `ACall t1;  t0 = ACall t1;`
  - `LCall` is for function labels known at compile time, while `ACall` is for computed function addresses (most often from vtables)
  - Note the different uses for void and non-void functions

DECAF TAC INSTRUCTIONS (CONT'D)

- Function/method definition: `BeginFunc 12;  ...  EndFunc;`
  - The argument for `BeginFunc` is the size in bytes of the space needed for all locals in the stack frame
  - Returning from a function: `Return t1;  Return;`
- Memory reference: `t1 = *(t2);  t1 = *(t2 + 8);`
  `*(t1) = t2;  *(t1 + -4) = t2;`
  - The (optional) offset must be an integer constant and can be positive or negative
  - Array indexing is done by adding to the base address the offset multiplied by the size of the element of the array
- Object fields and method dispatch:
  - To access member variables add offset to base and dereference
  - To access methods retrieve the address of the method from the vrable and then use `ACall`
- Method specification: `VTable ClassName = L1, L2, ...;`

DECAF LIBRARY FUNCTIONS

- `_Alloc` → one integer parameter, returns address of heap-allocated memory of that size in bytes
- `_ReadLine` → no parameters, returns string read from user input
- `_ReadInteger` → no parameters, returns integer read from user input
- `_StringEqual` → two string parameters, returns 1 if strings are equal and 0 otherwise
- `_PrintInt` → one integer parameter, prints that number to the console
- `_PrintString` → one string parameter, prints that string to the console
- `_PrintBool` → one boolean parameter, prints true/false to the console
- `_Halt` → no parameters, stops program execution

DECAF TRANSLATION EXAMPLES

- Translating DECAF into TAC means generating the TAC program but also figuring out temporary variables, creating labels, calling functions, etc.
- Simplifying assumptions: no `double` type, booleans are 4-bit integers, all the integers and pointers (for classes, arrays, strings) have a size of 4 bytes

```
void main() {
    Print("hello world");
}
```

```
main:
    BeginFunc 4;
    _t0 = "hello world";
    PushParam _t0;
    LCall _PrintString;
    PopParams 4;
    EndFunc;
```

  - Go down from the Program node into the declaration list that has one element (the function `main`)
  - Generate the function label and the `BeginFunc` (leaving a placeholder for the size)
  - Generate recursively the body of the function (one single call)
  - Come back and fill in the parameter of `BeginFunc`
  - Generate the function postamble

DECAF TRANSLATION EXAMPLES: ARITHMETIC

```
void main()
{
    int a;
    a = 2 + a;
    Print(a);
}
```

```
main:
    BeginFunc 12;
    _t0 = 2;
    _t1 = _t0 + a;
    a = _t1;
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

```
void main()
{

    int b;
    int a;

    b = 3;
    a = 12;
    a = (b + 2)-
        (a*3)/6;
}
```

```
main:
    BeginFunc 44;
    _t0 = 3;
    b = _t0;
    _t1 = 12;
    a = _t1;
    _t2 = 2;
    _t3 = b + _t2;
    _t4 = 3;
    _t5 = a * _t4;
    _t6 = 6;
    _t7 = _t5 / _t6;
    _t8 = _t3 - _t7;
    a = _t8;
    EndFunc;
```

DECAF TRANSLATION EXAMPLES: ARRAYS

```
void Foo(int[] arr)
{
     arr[1] = arr[0] * 2;
}
```

```
_Foo:
    BeginFunc 44;
    _t0 = 1;
    _t1 = 4;
    _t2 = _t1 * _t0;
    _t3 = arr + _t2;
    _t4 = 0;
    _t5 = 4;
    _t6 = _t5 * _t4;
    _t7 = arr + _t6;
    _t8 = *(_t7);
    _t9 = 2;
    _t10 = _t8 * _t9;
    *(_t3) = _t10;
    EndFunc;
```

DECAF TRANSLATION EXAMPLES: FUNCTIONS

```
int foo(int a, int b)
{
    return a + b;
}

void main()
{
    int c;
    int d;

    foo(c, d);
}
```

```
_foo:
    BeginFunc 4;
    _t0 = a + b;
    Return _t0;
    EndFunc;

main:
    BeginFunc 12;
    PushParam d;
    PushParam c;
    _t1 = LCall _foo;
    PopParams 8;
    EndFunc;
```

DECAF TRANSLATION EXAMPLES: CLASSES

```
class Animal {
    int height;
    void InitAnimal(int h) {
        this.height = h;
    }
}

class Cow extends Animal {
    void InitCow(int h) {
        InitAnimal(h);
    }
}

void Foo(Cow betsy) {
    betsy.InitCow(5);
}
```

```
_Animal.InitAnimal:
    BeginFunc 0;
    *(this + 4) = h;
    EndFunc;

VTable Animal =
    _Animal.InitAnimal,
;

_Cow.InitCow:
    BeginFunc 8;
    _t0 = *(this);
    _t1 = *(_t0);
    PushParam h;
    PushParam this;
    ACall _t1;
    PopParams 8;
    EndFunc;
 VTable Cow =
    _Animal.InitAnimal,
    _Cow.InitCow,
;
```

```
_Foo:
    BeginFunc 12;
    _t2 = 5;
    _t3 = *(betsy);
    _t4 = *(_t3 + 4);
    PushParam _t2;
    PushParam betsy;
    ACall _t4;
    PopParams 8;
    EndFunc;
```

- Note how `this` is passed as a "secret" first argument to a method calls!

DECAF TRANSLATION EXAMPLES: CONDITIONALS

```
void main()
{
    int a;

    a = 23;
    if (a == 23)
        a = 10;
    else
        a = 19;
}
```

```
main:
    BeginFunc 24;
    _t0 = 23;
    a = _t0;
    _t1 = 23;
    _t2 = a == _t1;
    IfZ _t2 Goto _L0;
    _t3 = 10;
    a = _t3;
    Goto _L1;
_L0:
    _t4 = 19;
    a = _t4;
_L1:
    EndFunc;
```

DECAF TRANSLATION EXAMPLES: LOOPS

```
void main()
{
    int a;
    a = 0;

    while (a < 10) {
        Print(a % 2 == 0);
        a = a + 1;
    }
}
```

```
main:
    BeginFunc 40;
    _t0 = 0;
    a = _t0;
_L0:
    _t1 = 10;
    _t2 = a < _t1;
    IfZ _t2 Goto _L1;
    _t3 = 2;
    _t4 = a % _t3;
    _t5 = 0;
    _t6 = _t4 == _t5;
    PushParam _t6;
    LCall _PrintBool;
    PopParams 4;
    _t7 = 1;
    _t8 = a + _t7;
    a = _t8;
    Goto _L0;
_L1:
    EndFunc;
```

Storage Allocation

- The code generator, you must assign a location to each local variable, parameter, and temporary variable
- These locations occur in a particular stack frame (relative to the frame pointer fp in MIPS) and are called fp-relative
  - Parameters begin at address fp + 4 and grow upward
  - Locals and temporaries begin at address fp - 8 and grow downward
- From your point of view:

```
Location* location =
    new Location(fpRelative, +4, locName);
```

Storage Allocation

- The code generator, you must assign a location to each local variable, parameter, and temporary variable
- These locations occur in a particular stack frame (relative to the frame pointer fp in MIPS) and are called fp-relative
  - Parameters begin at address fp + 4 and grow upward
  - Locals and temporaries begin at address fp - 8 and grow downward
- From your point of view:

```
Location* location =
    new Location(fpRelative, +4, locName);
```

- Global variables are stored starting from the MIPS global pointer (gp)
  - Memory pointed at by gp is treated as an array of values that grows upward (starting at gp+0)
  - Must choose an offset into this array for each global variable
- From your point of view:

```
Location* global =
    new Location(gpRelative, +8, locName);
```