Sequence 4.1 – Intermediate Representation

P. de Oliveira Castro S. Tardieu

Going beyond the AST

The abstract syntactic tree (AST) is a tree representation of the source, strongly tied to the source language.

In order to advance the compilation process, we need to go beyond the AST and approach the machine representation. For this, we use a language independent **intermediate representation** (IR).

Intermediate Representation

- An Intermediate Representation (IR) should be,
  - Simple and terse (remove high-level *syntactic sugar*)
  - Optimizable (must preserve enough *information* to enable optimizations)
- **Tradeoffs** in the level of abstraction are necessary
- Eg. Should the IR have an array type?
  - **Yes**: eases dependencies analysis
    - easy to know that two references map to the same array
  - **No**: complicates scalar optimizations such as constant propagation or force reduction
    - require a special case for array elements

Design Choices

- **Register vs. Stack**: keep local variables in named registers or a stack?
  - Stack: simple code generation and interpretation
  - Register: eases dependency analysis and optimizations
- **Flat vs. Hierarchical**: is the IR a list of instructions or a tree?
  - Hierarchical: preserves scopes and structure of the code
  - Flat: closer to target assembly; eases moving around instructions

Lowering and Three Address Code

Lowering: transforming a high-level representation (AST) into IR

- Classical lowering of expressions into **three-address code**, where each operation has at most three operands,

```
y := 4*x*x - 2*x + 1
```

becomes

```
x1 := x * x;
x2 := 4 * x1;
x3 := 2 * x;
x4 := x2 - x3;
y := x4 + 1;
```

Multiple IR?

In some compilers multiple IR levels are used.

GCC

- Generic (common AST format for all frontends)
- Gimple (register + hybrid: hierarchical / flat)
  - lowers all control structures to a canonical form
  - lowers expressions to 3-address code
- Gimple SSA (a Single Static Assignment form of GIMPLE)
  - we'll see later what SSA is
- RTL, Register Transfer Language (hybrid: register / stack + hierarchical)
  - lisp like
  - very close to machine assembly

Original Program

```
int f(int x) {
  int y = 0;
  if (x>0)
    y = 4*x*x + 1;
  return y;
}
```

GIMPLE

```
[...]
  if (x > 0) goto <D.4198>; else goto <D.4199>;
  <D.4198>:
  _1 = x * x;
  _2 = _1 * 4;
  y = _2 + 1;
  <D.4199>:
  D.4200 = y;
  goto <D.4201>;
  <D.4201>:
  return D.4200;
}
```

RTL

```
[...]
(note 9 8 10 4 [bb 4] NOTE_INSN_BASIC_BLOCK)
(insn 10 9 11 4 (set (reg:SI 114)
  (mult:SI (reg/v:SI 113 [ x ])
   (reg/v:SI 113 [ x ]))) "ex1.c":4 -1
  (nil))
(insn 11 10 12 4 (set (reg:SI 115)
  (ashift:SI (reg:SI 114)
   (const_int 2 [0x2]))) "ex1.c":4 -1
  (nil))
(insn 12 11 21 4 (set (reg/v:SI 112 [ <retval> ])
  (plus:SI (reg:SI 115)
   (const_int 1 [0x1]))) "ex1.c":4 -1
  (nil))
(jump_insn 21 12 22 4 (set (pc)
  (label_ref:SI 17)) 211 {*arm_jump}
  (nil)
 -> 17)
(barrier 22 21 24)
(code_label 24 22 23 5 3 (nil) [1 uses])
```