Simplification and Basic Blocks

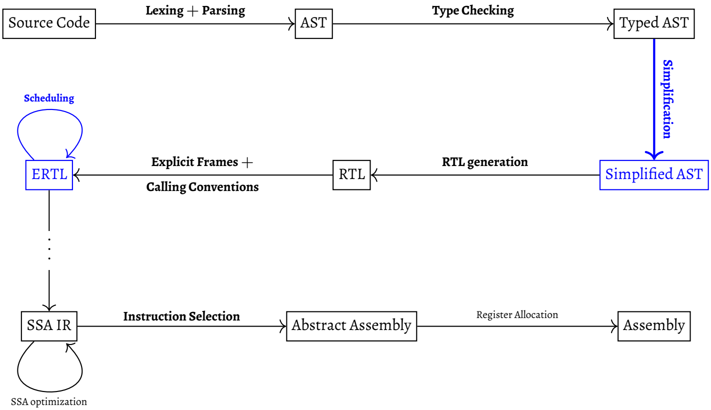
CSE 302 – Compilers – Week 6

Kaustuv Chaudhuri

Inria & École polytechnique

2019-10-17

Where in the compiler are we?



Reminder: BX2

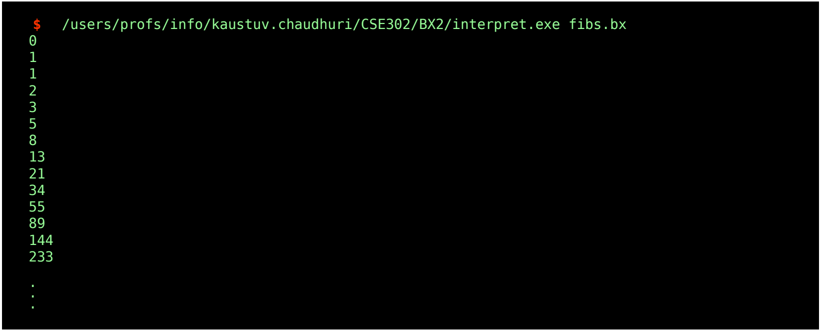
```
var num_fibs = 30 : int64;               // Global variable

proc fib(count : int64) {
  print fib_aux(0, 1, count);            // Recursive call
}

fun fib_aux(j, k, count : int64) : int64 {
  if (count < 1) {
    return k;
  }
  print j;
  var l = j + k : int64;                // Declarations anywhere
  return fib_aux(k, l, count - 1);      // Tail call
}

proc main() {                           // Required entry point
  fib(num_fibs);
}
```

Reminder: BX2 Interpreter



```
$  /users/profs/info/kaustuv.chaudhuri/CSE302/BX2/interpret.exe fibs.bx
0
1
1
2
3
5
8
13
21
34
55
89
144
233

.
.
.
```

Lab 4 Errata and Updates

(These are all fairly minor)

- While writing the interpreter I found some corner cases of the BX2 language that need to be fixed
- Global variables must be initialized:

⟨globalvar⟩ ::= "var" ⟨globalvar-init⟩ ("," ⟨globalvar-init⟩)* ":" ⟨type⟩ ";"
⟨globalvar-init⟩ ::= ⟨variable⟩ "=" (⟨number⟩ | ⟨bool⟩)

- Numerical literals can now be negative:

⟨number⟩ ::= `-?[0-9]+` (must be in range $[-2^{63}, 2^{63})$)

- The order of evaluation of procedure/function call arguments is unspecified, not left-to-right. (will revisit this later)

Quick Aside: Global Variables

- BX2 allows for global initialized variables
- These variables are stored in the `.data` section
- Translating: `var thing = 42 : int64;`

```
      .globl thing
      .section .data
      .align 8
thing:
      .quad 42        # can also write in hex with 0x notation
```

- Refer to variable `thing` as: `thing(%rip)` *(PC-relative addressing)*

```
main:
      movq thing(%rip), %rdi
      call bx_print_int64
```

Otherwise you will have to compile with `-no-pie` and the executables will be bigger and less portable.

Today's Agenda

1. Simplifying the AST
2. Basic Blocks
3. The Control Flow Graph (CFG)

Complications in the AST

- BX1 had a simple AST
  - All variables known up front, so stack space easy to compute
  - A flat namespace
  - Every variable name unique
- BX2 adds several complications
  - Declarations can be interspersed with other statements
  - Variables can be locally scoped
  - Variables can shadow names in outer scope
- BX3 will add even more complications
  - Arrays and records
  - References
  - `for` and ranged-`for` loops

Simplifying the AST: BX2 → BX1

Simplification steps:

1. Uniquely number every occurrence of a name
2. Separate declaration from initialization
3. Hoist all declarations to front of function/procedure
4. Flatten scopes

Simplifying the AST: BX2 → BX1

Step 1: uniquely number every occurrence of a variable name

```
proc foo(x, y : int64) {
  var u = x + y : int64;
  {
    var u = 2 * u + 1 : int64;
    print u;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 1: uniquely number every occurrence of a variable name

```
proc foo(x0, y0 : int64) {
  var u0 = x0 + y0 : int64;
  {
    var u1 = 2 * u0 + 1 : int64;
    print u1;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 2: separate declaration from initialization

```
proc foo(x0, y0 : int64) {
  var u0 = x0 + y0 : int64;

  {
    var u1 = 2 * u0 + 1 : int64;

    print u1;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 2: separate declaration from initialization

```
proc foo(x0, y0 : int64) {
  var u0 : int64;
  u0 = x0 + y0;
  {
    var u1 : int64;
    u1 = 2 * u0 + 1;
    print u1;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 3: hoist all declarations to front of function/procedure

```
proc foo(x0, y0 : int64) {
 var u0 : int64;

 u0 = x0 + y0;
 {
  var u1 : int64;
  u1 = 2 * u0 + 1;
  print u1;
 }
}
```

Simplifying the AST: BX2 → BX1

Step 3: hoist all declarations to front of function/procedure

```
proc foo(x0, y0 : int64) {
  var u0 : int64;
    var u1 : int64;
  u0 = x0 + y0;
  {

    u1 = 2 * u0 + 1;
    print u1;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 4: flatten scopes

```
proc foo(x0, y0 : int64) {
  var u0 : int64;
  var u1 : int64;
  u0 = x0 + y0;
  {
    u1 = 2 * u0 + 1;
    print u1;
  }
}
```

Simplifying the AST: BX2 → BX1

Step 4: flatten scopes

```
proc foo(x0, y0 : int64) {
  var u0 : int64;
  var u1 : int64;
  u0 = x0 + y0;

    u1 = 2 * u0 + 1;
    print u1;

}
```

Today's Agenda

1. Simplifying the AST
2. Basic Blocks
3. The Control Flow Graph (CFG)

Basic Blocks from the Instruction Graph

- In (E)RTL every instruction (except `return`) has an out-label
- We can merge linear sequences of instructions with single out-labels to form basic blocks

```
L1: move 10, #0 ⟶ L2
L2: move 20, #1 ⟶ L3
L3: binop add, #0, #1 ⟶ L4
L4: ubranch jnz, #1 ⟶ L5,L6
```

```
L1: ┌──────────────────────┐
    │ move 10, #0          │ ⟶ L5,L6
    │ move 20, #1          │
    │ binop add, #0, #1    │
    │ ubranch jnz, #1      │
    └──────────────────────┘
```

- Each such basic block:
  1. Has an in-label
  2. Ends with a jump (`ubranch`, `bbranch`, or `goto`) or `return`
  3. Has no other jump or `return`
  4. Has zero or more out-labels

Building Basic Blocks

A most obvious algorithm

- Maintain a working set I of in-labels. Initially this contains just the entry label of the procedure.
- While I is not empty:
    - Extract L from I and start a new basic block
    - While the instruction at L has only one out-label and that out-label is not the out-label of any other instruction:
        - Add the instruction to the end of the basic block
        - Set L to the out-label of the instruction
    - Add the last instruction L to the basic block and set the out-labels of the block to those of the instruction. Then ship the block.
- Complexity: all instructions visited only once, so linear in the number of instructions

Why Basic Blocks?

- When analyzing the control flow, they are the natural unit
  - Fewer basic blocks than instructions
  - Many optimizations can abstract away details of arithmetic
- Like (E)RTL instructions, basic blocks can be freely reordered
- Some peephole optimizations can be applied to instruction patterns inside a basic block

Today's Agenda

1. Simplifying the AST
2. Basic Blocks
3. The Control Flow Graph (CFG)

Control Flow Graph (CFG)

A directed graph with

- Nodes: basic blocks, labeled with in-labels
- Edges: out-label links
- A distinguished entry edge (can also be entry node)

Control Flow Optimization:

- Modification of the CFG that:
  - preserves all nodes reachable from the entry edge/node
  - preserves paths
- Think: no observable change in behavior

Unreachable Code Elimination (UCE)

AKA dead code elimination (DCE)

- Any CFG node that is unreachable from the entry node can simply be removed
- Such nodes can be created when compiling booleans
- The programmer may also intentionally write such code
  E.g., putting code after a `return`.

Traces and Schedules

- A trace is a linear sequence of basic blocks
- Scheduling: CFG → disjoint union of traces
- Many schedules for a given CFG
- Some schedules may be better than others for performance
  - `if` can put more likely outcome next in trace
  - `while` loops can schedule body right after test
  - Profile Guided Optimization:
    - Run the program on representative input
    - Gather statistics on which branches are taken more often
    - Reschedule the traces based on this data
    - Note: your processor does this already with sophisticated branch prediction units in hardware, but this has recently been shown to be the source of major security bugs (Spectre)

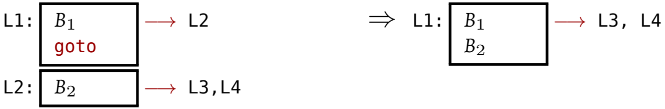
Scheduling Algorithm

(Basic depth first search)

- Put all the blocks into a worklist Q
- While Q is non-empty
  - Start a new trace T
  - Remove the front element $b \in Q$
  - While b is not marked visited
    - Mark b as visited
    - Add b to the end of T
    - Set b to one of its unmarked successors (if any)
  - End the current trace T
- Note: every block belongs to exactly one trace

Scheduling Optimizations

- Block Merging/Coalescing

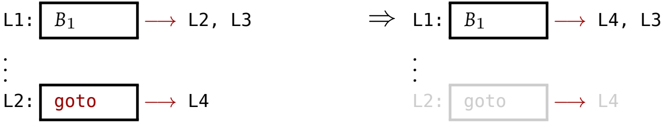


L1: B_1 goto ⟶ L2

L2: B_2 ⟶ L3,L4

⇒ L1: B_1 B_2 ⟶ L3, L4

- Shrink goto sequences



L1: B_1 ⟶ L2, L3

⋮

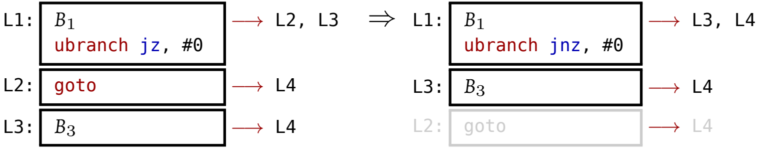
L2: goto ⟶ L4

⇒ L1: B_1 ⟶ L4, L3

⋮

L2: goto ⟶ L4

- Condition inversion (if with an empty then- or else-block)



L1: B_1 ubranch jz, #0 ⟶ L2, L3

L2: goto ⟶ L4

L3: B_3 ⟶ L4

⇒ L1: B_1 ubranch jnz, #0 ⟶ L3, L4

L3: B_3 ⟶ L4

L2: goto ⟶ L4

Lab 4 Discussion