Procedures and Functions

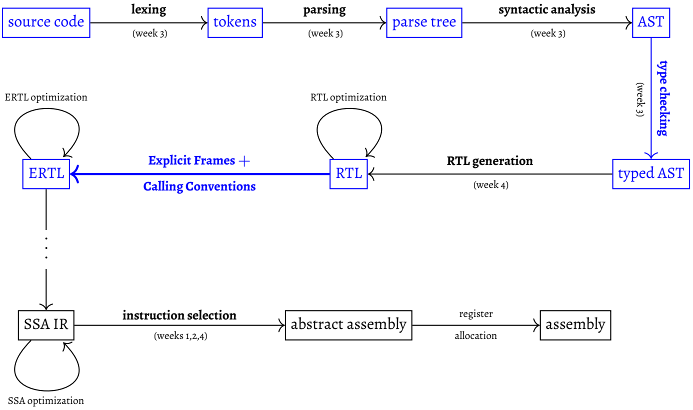
CSE 302 – Compilers – Week 5

Kaustuv Chaudhuri

Inria & École polytechnique

2019-10-07

Where in the compiler are we?



BX1 $\rightarrow$ BX2

Features of BX1:

- Integers (`int64`) and booleans (`bool`)
- Arithmetic, comparisons, boolean operators
- Conditionals (`if ... else`) and loops (`while`)

BX1 → BX2

Features of BX1:

- Integers (`int64`) and booleans (`bool`)
- Arithmetic, comparisons, boolean operators
- Conditionals (`if ... else`) and loops (`while`)

What BX2 adds:

- Defining and calling procedures and functions
- Recursion (including mutual recursion)
- *Tail Call Elimination* (TCE)

Example of BX2

```
// procedures
proc main() {
  var i = 0 : int64;
  while (i < 10) {
    print fibo(i);          // can call functions defined later
    i = i + 1;
  }
  // implicit return at end of proc
}

// self-recursion
fun fibo(n : int64) : int64 {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibo(n - 1) + fibo(n - 2);
  // all code paths must return a value
}
```

More Examples of BX2

```
// tail recursion

proc collatz(n : int64) {
  print n;
  if (n % 2 == 0) return collatz(n / 2);     // tail call
  return collatz(3 * n + 1);                 // tail call
}
```

```
// mutual recursion (not necessarily tail recursive)

fun is_even_nat(n : int64) : bool {
  if (n == 0) return true;
  return is_odd_nat(n - 1);                  // tail call
}

fun is_odd_nat(n : int64) : bool {
  return is_even_nat(n - 1);                // tail call
}
```

Lecture Plan

1. Parsing and Type-Checking Callables
   (note: callable = function or procedure)

2. The Stack and Calling Conventions

3. RTL with Explicit Frames (ERTL)

4. Tail Call Elimination

Lecture Plan

1. Parsing and Type-Checking Callables
   (note: callable = function or procedure)

2. The Stack and Calling Conventions

3. RTL with Explicit Frames (ERTL)

4. Tail Call Elimination

Grammar: Procedures

A procedure definition has:

- Zero or more formal arguments (with their types)
- Declarations of local variables
- A sequence of statements

Think: `main()`

⟨procedure⟩ ::= "proc" ⟨variable⟩ "(" ⟨params⟩? ")" ⟨body⟩

⟨params⟩ ::= ⟨param⟩ ("," ⟨param⟩) *
⟨param⟩ ::= ⟨variable⟩ ("," ⟨variable⟩) * ":" ⟨type⟩

⟨body⟩ ::= "{" ⟨vardecl⟩* ⟨stmt⟩* "}"

⟨stmt⟩ ::= ··· | "return"? ⟨variable⟩ "(" (⟨expr⟩ ("," ⟨expr⟩)*)? ")" ";"

Grammar: Functions

A function definition has:

- Zero or more formal arguments (with their types)
- The type of return values
- Declarations of local variables
- A sequence of statements

$\langle\text{function}\rangle$ ::= "`fun`" $\langle\text{variable}\rangle$ "`(`" $\langle\text{params}\rangle$? "`)`" "`:`" $\langle\text{type}\rangle$ $\langle\text{body}\rangle$

$\langle\text{expr}\rangle$ ::= $\cdots$ | $\langle\text{variable}\rangle$ "`(`" ($\langle\text{expr}\rangle$ ("`,`" $\langle\text{expr}\rangle$)*)? "`)`"

$\langle\text{stmt}\rangle$ ::= $\cdots$ | $\langle\text{expr}\rangle$ "`;`" | "`return`" $\langle\text{expr}\rangle$ "`;`"

Type-Checking Callables

Step 1/2: Accumulating Type Signatures

- Legal:
  - Calling function or procedure defined later
  - Making recursive call to self
- Illegal:
  - Calling something never defined
  - Supplying type-incorrect arguments
  - Using function result with incorrect type
  - Using procedure call in expressions
- Type-Signature: a “package” of the list of parameter types together with the return type (if relevant)
  - Order of parameter types important!
- Step 1: obtain the type-signatures of all the callables
  - Also check each callable defined *exactly once*

Type-Checking Callables

Step 2/2: Checking Expressions and Statements

- Store the map of callable names to type-signatures
- Step 2(a): For any procedure or function call:
  - Arguments must match parameter types in the type-signature
  - Procedure calls must not be in expressions
- Step 2(b): In every function or procedure body:
  - The input parameters must be used type-correctly.
  - Procedure bodies must have argument-less `return` statements
  - Function bodies must `return` values of correct type
  - Every code path in a function body must `return`

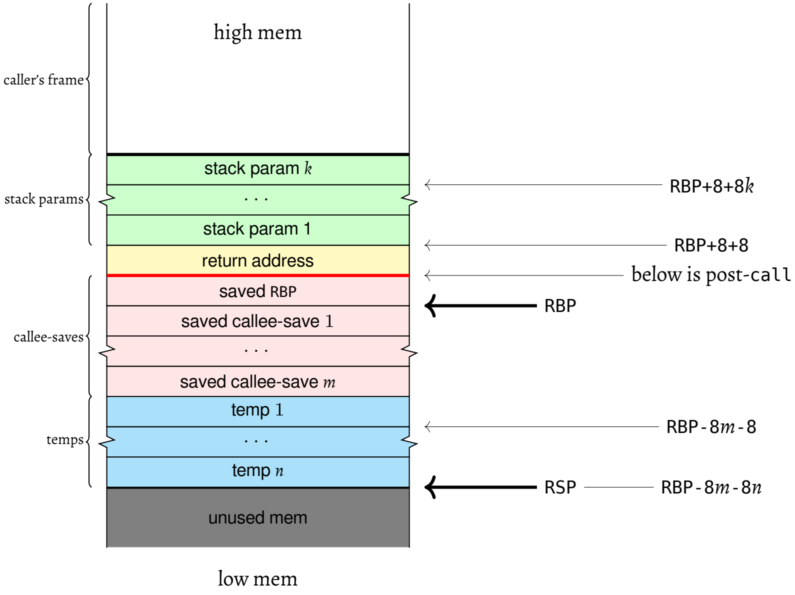
Lecture Plan

1. Parsing and Type-Checking Callables
2. The Stack and Calling Conventions
3. RTL with Explicit Frames (ERTL)
4. Tail Call Elimination

The Stack (Recap)

- Region of memory that grows downwards (high to low mem)
- Divided into stack frames
  - One frame per call
  - Only the last (i.e., bottommost) frame is "active"
  - When the call finishes, its stack frame is deleted
    A subsequent call can reuse that memory for its own frame
- Two key pointers into the last stack frame:
  - Stack Pointer: end (aka *top*) of the stack
  - Frame Pointer: start of the callee-modifiable region
    (aka *Base Pointer*)
- The frame also has other regions
  - Incoming arguments
  - Outgoing return value (not relevant for BX2)
  - Saved callee-save registers
  - Spilled temporaries
  - Dynamic stack-allocation (not relevant for BX2)

Regions of the Stack Frame



Calling Convention

- A calling convention is a contract between:
  - The compiler (*local functions*)
  - The runtime system (*intrinsics*)
  - External libraries (*foreign functions*)
  - The operating system (*system calls*)
  - The hardware (only for the stack pointer)
- Things that are fixed in the calling convention:
  - Which registers hold input arguments
  - Order of stack arguments
  - Which registers can be freely modified by the callee (*caller-save*)
  - Which registers must be preserved by the callee (*callee-save*)
  - Which register will hold the output value (if applicable)
- Things that are not fixed in the calling convention:
  - Whether the frame pointer register is used (as a frame pointer)
  - Order of callee-saves

The “Unix/C” Calling Convention

- Precisely: the *System V Release 4* (SVR4) *Application Binary Interface* (ABI) and *Executable and Linkable Format* (ELF)
- GNU libraries and Linux/FreeBSD/OpenBSD
- Also in Windows 10+ with *Windows Subsystem for Linux* (WSL)
- Note: not used by some Unix variants such as MacOS X

The “Unix/C” Calling Convention

- 6 callee-saves: `RBX`, `RBP`, `R12`–`R15`
- First 6 int-like arguments: `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`
  - Int-like: fits in 1 register, so `int64` or pointers (to come)
  - The order in which they are listed above matters – `RDI` is first and `R9` is last
- First 8 float arguments: `XMM0`–`XMM7` (not relevant for BX2)
- All other arguments pushed on stack in 16-bit alignment
  - Args pushed in reverse order
  - 16-bit alignment: `RSP` is always a multiple of 16
- Return values:
  - `RAX` (64-bit) or `RDX:RAX` (128-bit) for int-like returns
  - `XMM0`–`XMM7` for first 7 float return values
  - Other return values treated as stack parameters, i.e., caller allocates stack space at end of parameter list for the return value and callee writes into that space

Example: Fibonacci

```
fun fibo(n : int64) : int64 { ... }

proc main() {
  fibo(10);
}
```

⇓

```
fibo:
    # ...
    cmpq $0, %rdi          # RDI has argument ‘n‘
    je .Lfibo.0
    # ...
.Lfibo.0:                  # return fibo(0)
    movq $0, %rax
    ret

main:
    # ...
    movq $10, %rdi         # first parameter ‘n‘ of fibo()
    call fibo
    # ...
```

Example: Summing 7 Things

```
fun sum7(x1, x2, x3, x4, x5, x6, x7 : int64) : int64 {
  return x1 + x2 + x3 + x4 + x5 + x6 + x7;
}
```

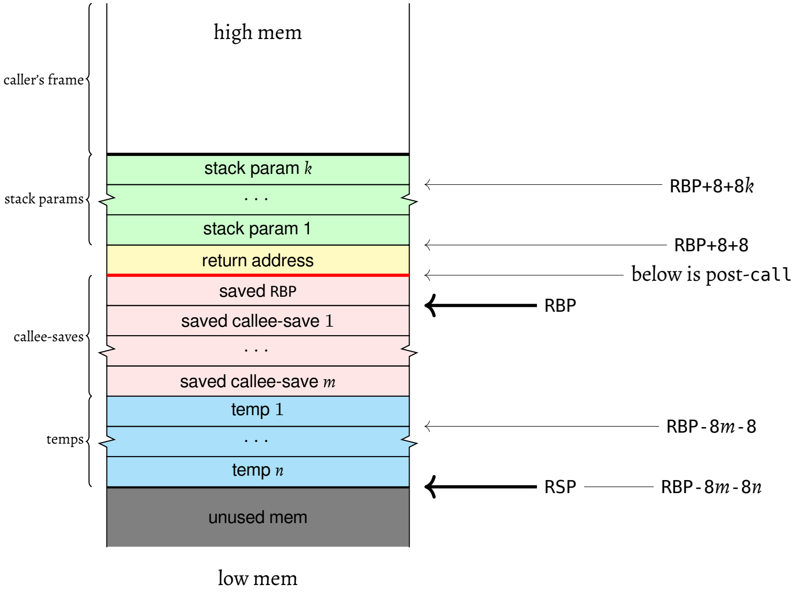
⇓

```
sum7:
    pushq %rbp                 # use RBP to index (optional)
    movq %rsp, %rbp

    addq %rsi, %rdi            # x1 in RDI, x2 in RSI
                               # RDI accumulates the sum
    addq %rdx, %rdi            # x2 in RDX
    # ... and so on until:
    addq %r9, %rdi             # x6 in R9
    movq 16(%rbp), %rdi        # x7 in stack param region

    movq %rdi, %rax
    popq %rbp
    ret
```

Regions of the Stack Frame



Example: Calling sum7

```
proc main() {
  sum7(1, 2, 3, 4, 5, 6, 7);
}
```

⇓

```
main:
    # ...
    movq $1, %rdi
    movq $2, %rsi
    # ... and so on until:
    movq $6, %r9
    pushq $7       # up to 32 bit immediates can be pushed
    call sum7
    addq $8, %rsp  # ”unpush” by just moving RSP up 1 slot
    # now %rax holds the result of the sum
```

Lecture Plan

1. Parsing and Type-Checking Callables
2. The Stack and Calling Conventions
3. RTL with Explicit Frames (ERTL)
4. Tail Call Elimination

RTL with Explicit Frames (ERTL)

aka *Explicit RTL*

- The calling convention mentions hardware registers
- RTL only contains pseudos
- In particular, RTL `call`/`return` uses only pseudo-registers
- ERTL = RTL
  + explicit stack frame allocation/deallocation
  + explicit in/out registers transferred to/from pseudos
  + callee-saves transferred to pseudos
  − free use of `call`

ERTL Instructions (Summary)

Instructions `move`, `unop`, `binop`, `ubranch`, `bbranch`, `goto` the same as RTL.

ERTL instruction	Description
`Li: newframe ⟶ Lo`	allocate new frame (size TBD)
`Li: delframe ⟶ Lo`	deallocate current frame
`Li: load_param n, r ⟶ Lo`	load stack param into pseudo
`Li: copy r1, r2 ⟶ Lo`	copy between pseudos
`Li: copy mr, r ⟶ Lo`	copy machine reg into pseudo
`Li: copy r, mr ⟶ Lo`	copy pseudo into machine reg
`Li: push r ⟶ Lo`	push pseudo onto stack
`Li: pop r ⟶ Lo`	pop pseudo from stack
`Li: call f(n) ⟶ Lo`	call with `n` non-stack args
`Li: return`	terminate the call

`n`	number/immediate (no $)	`Li`, `Lo`, ...	labels
`mr`	machine register	`r`, `r1`, `r2`, ...	pseudo-registers

ERTL Instructions: `newframe`

- Purpose: allocate a new stack frame in the callee
- Machine registers used before `newframe` invalid
- Compilation to AMD64 (full spillage):
  1. Compute list of pseudos in the function
  2. Allocate space in the stack for these pseudos

```
pushq %rbp
movq %rsp, %rbp
subq $8n, %rsp        # allocate n 64-bit slots
```

  3. Remember the mapping from pseudos to stack slots

ERTL Instructions: `delframe`

- Purpose: deallocate a stack frame and restore `RSP`
- Machine registers used after `newframe` invalid
- Compilation to AMD64:

```
movq %rbp, %rsp
popq %rbp
```

ERTL Instructions: `load_param` and `copy`

`load_param`:

- Purpose: load a parameter at a given offset (n) from the input parameter region of the stack into a pseudo `r`.
- Compilation to AMD64 (full spillage):

```
movq (n+8)(%rbp), %rcx
movq %rcx, r's stack slot
```

`copy` machine registers:

- Purpose: callee-save/restore and machine register args
- Most of the ERTL instructions still use pseudos
- Later passes will remove redundant `movq`s.

ERTL Instructions: push, pop, call, and return

push and pop:

- Similar to their AMD64 equivalents

call:

- Takes as argument just the number of register arguments (a number $\in \{0, 1, \ldots, 6\}$)
- All the other arguments must be pushed before the call and then popped after the call

call:

- The return value needs to be moved into %rax explicitly

RTL → ERTL Example

Illustrative example with only RBX callee-saved

```
fun fibo(#1):
  enter: L0
  ---------
  L0:  move 0, #2          ⟶ L1
  L1:  bbranch je, #1, #2 ⟶ L2, L3
  L2:  move 0, #5          ⟶ L15
  L3:  move 1, #2          ⟶ L1
  L4:  bbranch je, #1, #2 ⟶ L5, L6
  L5:  move 1, #5          ⟶ L15
  L6:  copy #1, #3         ⟶ L7
  L7:  move 1, #4          ⟶ L8
  L8:  binop sub, #4, #3   ⟶ L9
  L9:  call fibo(#3), #3   ⟶ L10
  L10: copy #1, #5         ⟶ L11
  L11: move 2, #6          ⟶ L12
  L12: binop sub #6, #5    ⟶ L13
  L13: call fibo(#5), #5   ⟶ L14
  L14: binop addq #3, #5   ⟶ L15
  L15: return #5
```

⇒

```
fun fibo:
  enter: L0
  ---------
  L0:  newframe            ⟶ L1
  L1:  copy %rbx, #100     ⟶ L2
  L2:  copy %rdi, #1       ⟶ L3
  L3:  move 0, #2          ⟶ L4
  L4:  bbranch je, #1, #2 ⟶ L5, L6
  L5:  move 0, #5          ⟶ L22
  L6:  move 1, #2          ⟶ L7
  L7:  bbranch je, #1, #2 ⟶ L8, L9
  L8:  move 1, #5          ⟶ L22
  L9:  copy #1, #3         ⟶ L10
  L10: move 1, #4          ⟶ L11
  L11: binop sub, #4, #3   ⟶ L12
  L12: copy #3, %rdi       ⟶ L13
  L13: call fibo(1)        ⟶ L14
  L14: copy %rax, #3       ⟶ L15
  L15: copy #1, #5         ⟶ L16
  L16: move 2, #6          ⟶ L17
  L17: binop sub #6, #5    ⟶ L18
  L18: copy #5, %rdi       ⟶ L19
  L19: call fibo(1)        ⟶ L20
  L20: copy %rax, #5       ⟶ L21
  L21: binop addq #3, #5   ⟶ L22
  L22: copy #5, %rax       ⟶ L23
  L23: copy #100, %rbx     ⟶ L24
  L24: delframe            ⟶ L25
  L25: return
```

RTL → ERTL Compilation

- One RTL instruction may become multiple ERTL instructions
- The number of temporaries may also grow
- Every exit from the function must be preceeded by restoring the callee-saved registers and a `delframe`
- What about redundant saves?
  - Register allocation will allocate the temporary to the same register if it can
  - E.g., if `#100` is allocated to `%rbx`, then we can just delete lines `L1` and `L23` that now would have a `copy` between the same registers.

```
L1:  copy %rbx, #100     --> L2
...
L23: copy #100, %rbx     --> L24
```

  - However, if `%rbx` is actually used for some purpose, then `#100` won't be allocated to it, so the save/restore of `%rbx` will persist.

Lecture Plan

1. Parsing and Type-Checking Callables
2. The Stack and Calling Conventions
3. RTL with Explicit Frames (ERTL)
4. Tail Call Elimination

Tail Calls

- A tail call is a `return` whose argument is a call to another callable.

```
fun is_even(n : int64) : bool {
  if (n == 0) return true;
  return is_odd(n - 1);
}

fun is_odd(n : int64) : bool {
  return is_even(n - 1);
}

fun is_nat(n : int64) : bool {
  return is_even(n) || is_odd(n);
}
```

- ERTL for these functions:

```
fun is_even:
  enter: L0
  ----
  L0:  newframe              --> L1
  L1:  copy %rdi, #1         --> L2
  ...
  L9:  call is_odd(1)        --> L10
  L10: delframe              --> L11
  L11: return
```

```
fun is_odd:
  enter: L0
  ----
  L0: newframe               --> L1
  L1: copy %rdi, #2          --> L2
  L2: move 1, #3             --> L3
  L3: binop subq, #3, #2     --> L4
  L4: copy #2, %rdi          --> L5
  L5: call is_even(1)        --> L6
  L6: delframe               --> L7
  L7: return
```

Tail Calls: A Sneaky Trick

```
fun is_even:
  enter: L0
  ----
  L0:  newframe            ⟶ L1
  L1:  copy %rdi, #2       ⟶ L2
  ...
  L9:  call is_odd(1)      ⟶ L10
  L10: delframe            ⟶ L11
  L11: return
```

```
fun is_odd:
  enter: L0
  ----
  L0: newframe             ⟶ L1
  L1: copy %rdi, #2        ⟶ L2
  L2: move 1, #3           ⟶ L3
  L3: binop subq, #3, #2   ⟶ L4
  L4: copy #2, %rdi        ⟶ L5
  L5: delframe             ⟶ L6
  L6: goto             ⟶ is_even
```

- Now both `is_even()` and `is_odd()` run in constant stack space! (vs. $O(n)$ without this hack)
- In essence, reduces a complex mutual recursion graph of function calls into an iteration over a finite state machine (the call graph)
- Important case: self tail recursion becomes just a loop!

Tail Call Elimination

- Requirements: two callables `f()` and `g()` (could be the same function) that have compatible stack frames
  - Same number of arguments
  - Same arguments passed in registers vs. stacks
- When the caller `f()` tail calls the callee `g()`:
  - Set up the inputs for `g()` (input registers, stack parameters)
  - Do everything up to and including the `delframe` for `f()`
  - Unconditionally jump (`goto`) to `g()`
- From the perspective of the program, it is as if `f()` wasn't called at all; indeed `g()` returns directly to `f()`'s caller
- Improvements:
  - Self tail calls can even potentially avoid even the `delframe`/`newframe`!
  - Tail calls with incompatible stack frames: hard!

Lab 4

Lab 4/BX2

- You may now work in pairs
- Lab 3 complete solutions will be given in both C++ and Java. You can start with these for lab 4 if you want.
- Once again a two week timeline
  - Week 1: extend grammar, parse, type-check, submit 5 tests
  - Week 2: implement ERTL, RTL→ERTL, and ERTL→AMD64
  - Extra credit: implement TCE
- Date/time switch idea: still waiting to hear from the scola