Control Flow and the R2 language

CIS531 — Fall 2025, Syracuse

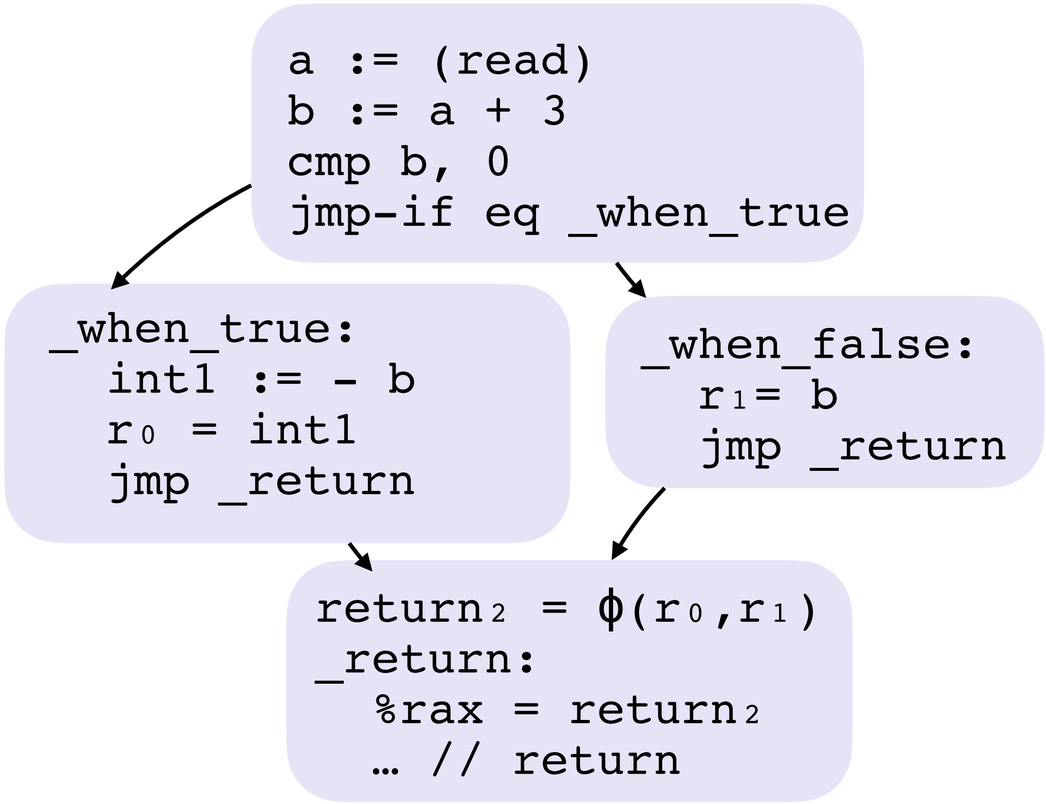
Kristopher Micinski

Syracuse University
College of Engineering
& Computer Science

Branching Control-Flow

- Branching control-flow necessary for *making decisions*
  - R2 is the language of *decision diagrams* over finite input streams
  - R2 also includes several more forms, along with type checking

```
(program
  (let ([a (read)])
    (let ([b (+ a 3)])
      (if (eq? b 0)
          b
          (- b)))))
```



R2: new syntactic forms

- *Boolean literals true/false*
- *cmp* is a syntactic set of *comparators*, which yield true/false
- And/or (short circuit) and not
- If allows branching
- Language power: ***decision trees***

cmp	::=	`eq?` | `<` | `<=` | `>` | `>=`
exp	::=	*int* | (`read`) | (`-` *exp*) | (`+` *exp exp*) | (`-` *exp exp*)
	|	*var* | (`let` ([*var exp*]) *exp*)
	|	`#t` | `#f` | (`and` *exp exp*) | (`or` *exp exp*) | (`not` *exp*)
	|	(*cmp exp exp*) | (`if` *exp exp exp*)
R_2	::=	(`program` *info exp*)

Figure 4.1: The syntax of R_2, extending R_1 (Figure 2.1) with Booleans and conditionals.

Type Checking Expressions

- In this setting, we can see *type checking* as very simple
  - We can write a recursive function, type-check-exp—which returns the expression's type

```
(define (type-check-exp env)
  (lambda (e)
    (define recur (type-check-exp env))
    (match e
      [(? fixnum?) 'Integer]
      [(? boolean?) 'Boolean]
      [(? symbol? x) (dict-ref env x)]
      [`(read) 'Integer]
      [`(let ([,x ,e]) ,body)
        (define T (recur e))
        (define new-env (cons (cons x T) env))
        (type-check-exp new-env body)]
      ...
      [`(not ,e)
        (match (recur e)
          ['Boolean 'Boolean]
          [else (error 'type-check-exp "'not' expects a Boolean" e)])]
      ...
      )))
```

- While I find type theory very interesting, I do not think this setting is a particularly rich one in which to explore type theory:
  - We can't do much interesting with type theory until we have arrows / implication, which (via Curry Howard) correspond to functions
- In this case, there is never *any* need for type *inference*, since every expression's type is immediately apparent via looking "down" the AST
- Type checking becomes much harder when we deal with parameters: in that case, we need to look at every function callsite to determine the type:
  - This is what leads to different notions of polymorphism, etc.

- Type checking is a small part of p3

The pass *shrink*

- Compilers often have relatively high-level passes whose goal is to eliminate forms
  - Why it's good: fewer forms means fewer cases to handle in lower passes
  - Why it's bad: might add complexity that makes optimization more difficult

- The pass *shrink* gets rid of several more complex features…
  - Removes subtraction (x - y = x + (-y))
  - and / or (implement using if)
  - <=, >, and >= (express in terms of and/or and <)—end up with only <

x86 language must expand...

$$
\begin{array}{lcl}
\mathit{arg} & ::= & (\texttt{int}\ \mathit{int}) \mid (\texttt{reg}\ \mathit{register}) \mid (\texttt{deref}\ \mathit{register}\ \mathit{int}) \\
& \mid & (\texttt{byte-reg}\ \mathit{register}) \\
\mathit{cc} & ::= & \texttt{e} \mid \texttt{l} \mid \texttt{le} \mid \texttt{g} \mid \texttt{ge} \\
\mathit{instr} & ::= & (\texttt{addq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{subq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{negq}\ \mathit{arg}) \mid (\texttt{movq}\ \mathit{arg}\ \mathit{arg}) \\
& \mid & (\texttt{callq}\ \mathit{label}) \mid (\texttt{pushq}\ \mathit{arg}) \mid (\texttt{popq}\ \mathit{arg}) \mid (\texttt{retq}) \\
& \mid & (\texttt{xorq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{cmpq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{set}\ \mathit{cc}\ \mathit{arg}) \\
& \mid & (\texttt{movzbq}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{jmp}\ \mathit{label}) \mid (\texttt{jmp-if}\ \mathit{cc}\ \mathit{label}) \\
& \mid & (\texttt{label}\ \mathit{label}) \\
x86_1 & ::= & (\texttt{program}\ \mathit{info}\ (\texttt{type}\ \mathit{type})\ \mathit{instr}^{+})
\end{array}
$$

Figure 4.4: The $x86_1$ language (extends $x86_0$ of Figure 2.7).

C1 language...

$$
\begin{array}{lcl}
\mathit{arg} & ::= & \mathit{int} \mid \mathit{var} \mid \texttt{\#t} \mid \texttt{\#f} \\
\mathit{cmp} & ::= & \texttt{eq?} \mid \texttt{<} \\
\mathit{exp} & ::= & \mathit{arg} \mid (\texttt{read}) \mid (\texttt{-}\ \mathit{arg}) \mid (\texttt{+}\ \mathit{arg}\ \mathit{arg}) \mid (\texttt{not}\ \mathit{arg}) \mid (\mathit{cmp}\ \mathit{arg}\ \mathit{arg}) \\
\mathit{stmt} & ::= & (\texttt{assign}\ \mathit{var}\ \mathit{exp}) \\
\mathit{tail} & ::= & (\texttt{return}\ \mathit{exp}) \mid (\texttt{seq}\ \mathit{stmt}\ \mathit{tail}) \\
 & \mid & (\texttt{goto}\ \mathit{label}) \mid (\texttt{if}\ (\mathit{cmp}\ \mathit{arg}\ \mathit{arg})\ (\texttt{goto}\ \mathit{label})\ (\texttt{goto}\ \mathit{label})) \\
C_1 & ::= & (\texttt{program}\ \mathit{info}\ ((\mathit{label}\ .\ \mathit{tail})^{+}))
\end{array}
$$

Figure 4.5: The C_1 language, extending C_0 with Booleans and conditionals.

explicate-control

```
(program ()
  (if (if (eq? (read) 1)
          (eq? (read) 0)
          (eq? (read) 2))
      (+ 10 32)
      (+ 700 77)))
```

⇓

```
(program ()                                    ⇒
  (if (if (let ([tmp52 (read)])
            (eq? tmp52 1))
          (let ([tmp53 (read)])
            (eq? tmp53 0))
          (let ([tmp54 (read)])
            (eq? tmp54 2)))
    (+ 10 32)
    (+ 700 77)))
```

```
(program ()
  ((block62 .
     (seq (assign tmp54 (read))
          (if (eq? tmp54 2)
              (goto block59)
              (goto block60))))
   (block61 .
     (seq (assign tmp53 (read))
          (if (eq? tmp53 0)
               (goto block57)
               (goto block58))))
   (block60 . (goto block56))
   (block59 . (goto block55))
   (block58 . (goto block56))
   (block57 . (goto block55))
   (block56 . (return (+ 700 77)))
   (block55 . (return (+ 10 32)))
   (start .
     (seq (assign tmp52 (read))
          (if (eq? tmp52 1)
               (goto block61)
               (goto block62))))))
```

End-to-end example...

```
(program ()
  (if (eq? (read) 1) 42 0))
```

⇓

```
(program ()
 ((block32 . (return 0))
  (block31 . (return 42))
  (start .
     (seq (assign tmp30 (read))
          (if (eq? tmp30 1)
              (goto block31)
              (goto block32))))))
```

⇓

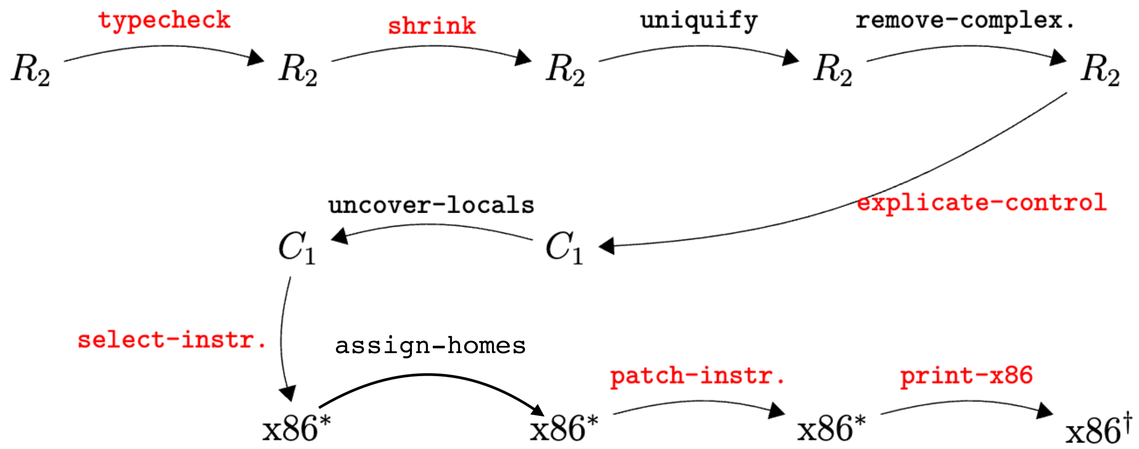
```
(program ((locals . (tmp30)))
 ((block32 .
    (block ()
      (movq (int 0) (reg rax))
      (jmp conclusion)))
  (block31 .
    (block ()
      (movq (int 42) (reg rax))
      (jmp conclusion)))
  (start .
    (block ()
      (callq read_int)
      (movq (reg rax) (var tmp30))
      (cmpq (int 1) (var tmp30))
      (jmp-if e block31)
      (jmp block32)))))
```

⇒

```
_block31:
        movq    $42, %rax
        jmp _conclusion
_block32:
        movq    $0, %rax
        jmp _conclusion
_start:
        callq   _read_int
        movq    %rax, %rcx
        cmpq    $1, %rcx
        je _block31
        jmp _block32

        .globl _main
_main:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %r12
        pushq   %rbx
        pushq   %r13
        pushq   %r14
        subq    $0, %rsp
        jmp _start
_conclusion:
        addq    $0, %rsp
        popq    %r14
        popq    %r13
        popq    %rbx
        popq    %r12
        popq    %rbp
        retq
```

S



We are **not** doing register allocation in this project…