A-Normal Form and Continuation Passing Style

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

- We need to discuss some concepts to understand the point of Project 3, which encompasses…
    - Assignment conversion (boxing), removing `set!`
    - ANF conversion (simplifying args to functions)
    - CPS conversion (removing `call/cc`)
- This will leave us with a tiny language consisting of **just** lambdas, prims, applications, and if
- Today: will talk about these passes at a high level, dig into details next 2-3 lectures.

Looking Forward…

Output language of P3 (post-CPS-conversion)

```
e ::= (let ([x (apply-prim op ae)]) e)
    | (let ([x (prim op ae ...)]) e)
    | (apply ae ae)
    | (ae ae ...)
    | (if ae e e)
ae ::= (lambda (x ...) e)
    | (lambda x e)
    | x
    | (quote dat)
```

At this point there are ***only*** tail calls (apply), let (with atomic args), and `if`

This is then simple to translate to LLVM/machine code.

SSA

- All variables are **assigned once**, or `const` (in C/C++ terms).
- No variable name is reused (at least in an overlapping scope).
- Instead of a variable X with multiple assignment points, SSA requires these points to be explicit syntactically as distinct variables X_0, X_1, … X_i.
- When control diverges and then joins back together, join points are made explicit using a special phi form, e.g.,

$$X_5 \leftarrow \phi(X_2, X_4)$$

Assignment conversion…

We will first remove **set!** by explicitly "boxing" all prims

```
(define (bar x)
  (define y (+ x 1))
  (define (h x)
    (if (= x 0)
        y
        (begin
          (set! y (+ y x))
          (h (- x 1)))))
  (h x))
```

```
(define (bar x)
  (define y (prim make-vector (+ x 1)))
  (define (h x)
    (if (= x 0)
        (prim vector-ref y 0)
        (begin
          (prim vector-set! y 0 (+ (vector-ref y 0) x))
          (h (- x 1)))))
  (h x))
```

C-like IR

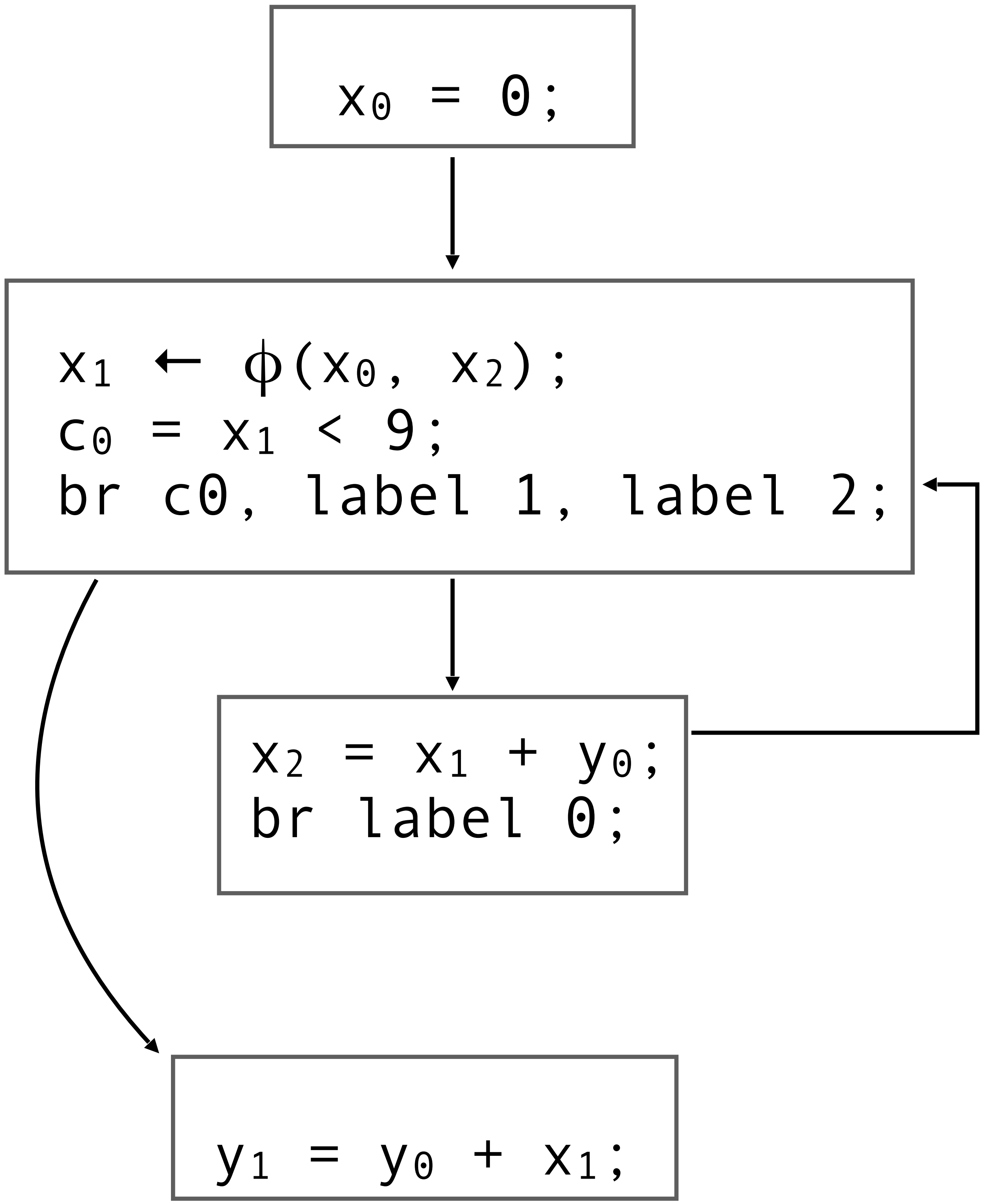
```
x = f(x);

if (x > y)
    x = 0;
else
{
    x += y;
    x *= x;
}

return x;
```

In SSA form

```
x_1 = f(x_0);

if (x_1 > y_0)
    x_2 = 0;
else
{
    x_3 = x_1 + y_0;
    x_4 = x_3 * x_3;
}
x_5 ← φ(x_2, x_4);

return x_5;
```

```
x = 0;

while (x < 9)
    x = x + y;

y += x;
```

```
x₀ = 0;

label 0:
 x₁ ← ϕ(x₀, x₂);
 c₀ = x₁ < 9;
 br c0, label 1, label 2;

label 1:
 x₂ = x₁ + y₀;
 br label 0;

label 2:
 y₁ = y₀ + x₁;
```



$x_0 = 0;$

label 0:

$x_1 \leftarrow \phi(x_0, x_2);$
$c_0 = x_1 < 9;$
br c0, label 1, label 2;

label 1:

$x_2 = x_1 + y_0;$
br label 0;

label 2:

$y_1 = y_0 + x_1;$

SSA in a Scheme IR?

- Assignment conversion
  - Eliminates `set!` by heap-allocating mutable values.
  - Replaces `(set! x y)` with `(prim vector-set! x 0 y)`.
- Alpha-renaming
  - Eliminates shadowing issues via alpha-conversion.
- Administrative normal form (ANF) conversion
  - Uses `let` to administratively bind all subexpressions.
  - Assigns subexpressions to a temporary intermediate variable.

Assignment conversion

- “Boxes” all mutable values, placing them on the heap.
- A box is a (heap-allocated) length-1 mutable vector.
- Mutable variables, when initialized, are placed in a box.
- When assigned, a mutable variable’s box is updated.
- When referenced, its value is retrieved from this box.

```
(lambda (x y)
  (set! x y)
  x)
```

⟶

```
(lambda (x y)
  (let ([x (make-vector 1 x)])
    (vector-set! x 0 y)
    (vector-ref x 0)))
```

α-renaming (“alphatization”)

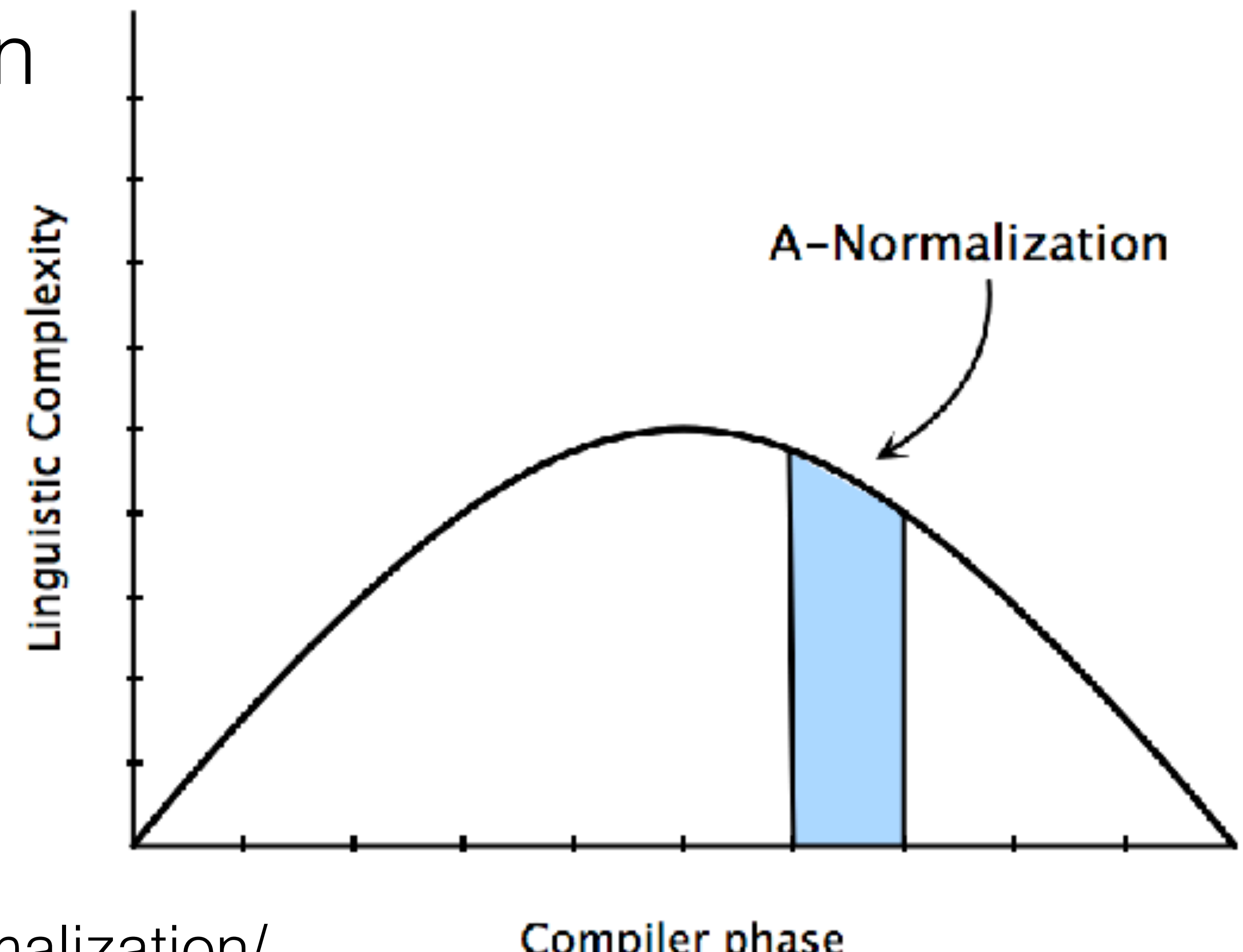
- Assign every binding point (e.g., at let- or lambda-forms) a unique variable name and rename all its references in a capture-avoiding manner.

- Can be done with a recursive AST walk and substitution env!

```
(define (alphatize e env)
  (match e
    [`(lambda (,x) ,e0)
     (define x+ (gensym x))
     `(lambda (,x+)
        ,(alphatize e0 (hash-set env x x+)))]
    [(? symbol? x)
     (hash-ref env x)]
    ...))
```

A-Normal Form

- Core IR for functional compilers
- Every argument to a function is **atomic**
- All non-tail calls must occur as a binding to a **let** or result from function



https://matt.might.net/articles/a-normalization/

Administrative normal form (ANF)

- Partitions the grammar into complex expressions (e) and atomic expressions (ae)—variables, datums, etc.

- Expressions cannot contain sub-expressions, except possibly in tail position, and therefore must be `let`-bound.

- ANF-conversion syntactically enforces an evaluation order as an explicit stack of let forms binding each expression in turn.

- Replaces a multitude of different continuations with a single type of continuation: the `let`-continuation.

```
(define (foo x y)
  (+ (+ x y) y))
```

Intermediate result is
administratively bound

```
(define (foo-anf x y)
  (let ([r0 (+ x y)])
    (+ r0 y)))
```

Still allow implicit return points

Why ANF convert?

ANF conversion can be thought of as *explicating subcomputations*

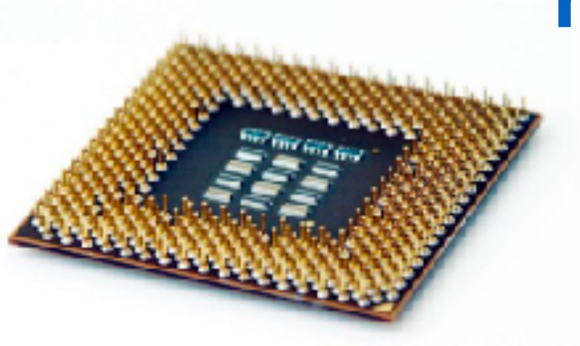
If you've ever "single stepped" in a debugger, executing each subcomputation one at a time…

```
x = 2 + 3 * (4 + 5);
```

True assembly languages **require** every operation be atomic

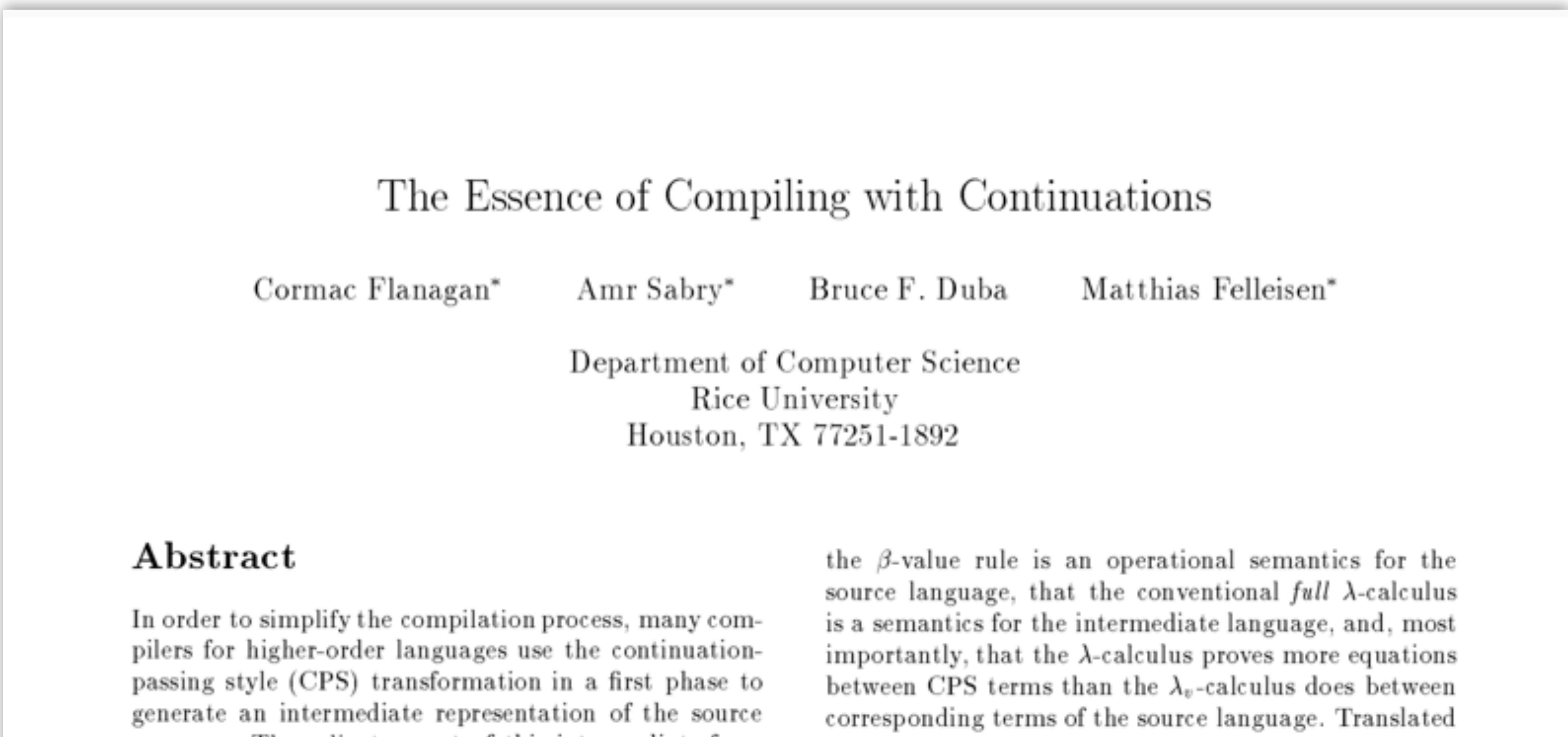
```
movq $r0, 4
movq $r1, 5
addq $r0, $r1
movq $r1, 3
mulq $r0, $r1
movq $r1, 2
addq $r0, $r1
```

Because atomic values **can fit into registers**



ANF Conversion Algorithm

- We will cover it in class, required for P3
- Today, will work some examples by hand…



The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

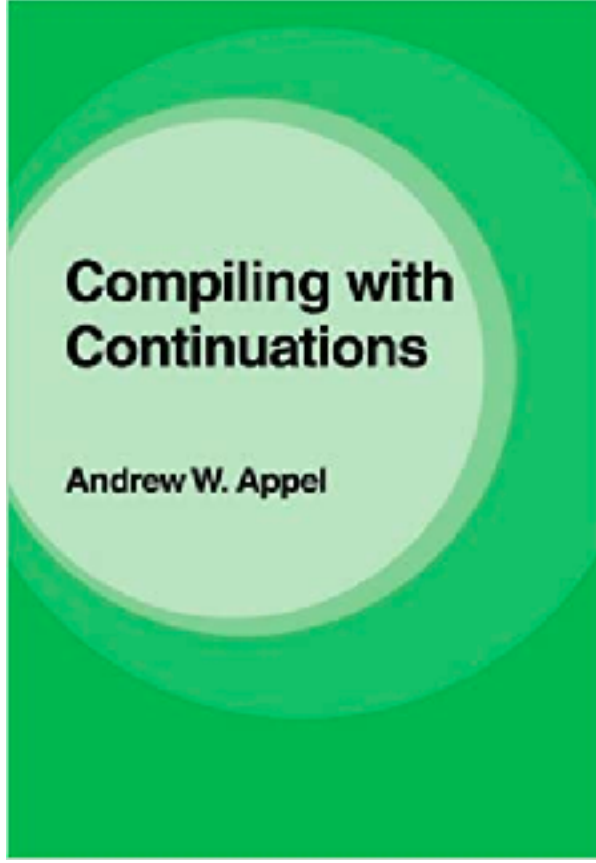
Abstract

In order to simplify the compilation process, many compilers for higher-order languages use the continuation-passing style (CPS) transformation in a first phase to generate an intermediate representation of the source

the β-value rule is an operational semantics for the source language, that the conventional *full* λ-calculus is a semantics for the intermediate language, and, most importantly, that the λ-calculus proves more equations between CPS terms than the λ_v-calculus does between corresponding terms of the source language. Translated

Eliminating call/cc requires conversion to
continuation-passing-style

Continuation Passing Style (CPS)



- Core IR for functional compilers.
- **Every** argument to **every** function must be an value
  - Thus, subcomputations do not incur stack space (beyond constant factors)
- Every function in the program ***also*** takes an explicit "current continuation" argument.
- No "implicit returns" allowed: all returns must tail-call-invoke the current continuation

Transforming to CPS

```
(define (foo-anf x y)
  (call/cc (lambda (k)
             (let ([r0 (+ x y)])
               (k (+ r0 y))))))

;; in a real compiler this would
;; be a special form
(define (+-k x y k) (k (+ x y)))

(define (foo-cps x y k)
  (+-k x y (lambda (r0) (+-k r0 y k))))
```

`call/cc`

- Compilation to CPS makes `call/cc` **dirt simple**

- Every function in the program has an explicit current continuation argument. So you can simply compile call/cc to apply the current continuation *guaranteed to be in scope via the transformation*

```
(define (foo x y)
  (call/cc (lambda (k) …)))
```

```
(define (foo x y k)
  (k (lambda (k) …)))
```

```
((f g) (+ a 1) (* b b))
```

ANF conversion

```
(let ([t0 (f g)])
  (let ([t1 (+ a 1)])
    (let ([t2 (* b b)])
      (t0 t1 t2))))
```

```
x = a+1;
y = b*2;
y = (3*x) + (y*y);
```

```
(let ([x (+ a 1)])
  (let ([y (* b 2)])
    (let ([y (+ (* 3 x) (* y y))])
       ...)))
```

ANF conversion & alpha-renaming

```
(let ([x0 (+ a0 1)])
  (let ([y0 (* b0 2)])
    (let ([t0 (* 3 x0)])
      (let ([t1 (* y0 y0)])
        (let ([y1 (+ t0 t1)])
          ...)))))
```

What about join points?

```
x₁ = f(x₀);

if (x₁ > y₀)
    x₂ = 0;
else
{
    x₃ = x₁ + y₀;
    x₄ = x₃ * x₃;
}
X₅ ← ϕ(x₂, x₄);

return x₅;
```

```
(let ([x1 (f x0)])
  (let ([x5
         (if (> x1 y0)
           (let ([x2 0]) x2)
           (let ([x3 (+ x1 y0)])
             (let ([x4 (* x3 x3)]))
               x4))])
    x5))
```

What about join points?

```
x₀ = 0;


label 0:
 x₁ ← ϕ(x₀, x₂);
 c₀ = x₁ < 9;
 br c0, label 1, label 2;

label 1:
 x₂ = x₁ + y₀;
 br label 0;

label 2:
 x₃ ← ϕ(x₁, x₂);
 y₁ = y₀ + x₃;
```

```
(let ([x0 0])
  (let ([x3
          (let loop0 ([x1 x0])
            (if (< x1 9)
              (let ([x2 (+ x1 y0)])
                (loop0 x2))
              x1))])
    (let ([y1 (+ y0 x3)])
      ...)))
```

They're just calls/returns!

```
(let ([x0 0])
  (let ([x3
         (letrec* ([loop0
                    (lambda (x1)
                      (if (< x1 9)
                        (let ([x2 (+ x1 y0)])
                          (loop0 x2))
                        x1))])
           (loop0 x0))])
    (let ([y1 (+ y0 x3)])
      ...)))
```

```
(let ([x0 0])
  (let ([x3
         (letrec* ([loop0
                    (lambda (x1)
                      (if (< x1 9)
                        (let ([x2 (+ x1 y0)])
                          (loop0 x2))
                        x1))])
           (loop0 x0))])
    (let ([y1 (+ y0 x3)])
      ...)))
```

```
(let ([x0 0])
  (let ([x3
         (let ([loop0 ‘()])
           (set! loop0
                 (lambda (x1)
                   (if (< x1 9)
                       (let ([x2 (+ x1 y0)])
                         (loop0 x2))
                       x1)))
           (loop0 x0))])
    (let ([y1 (+ y0 x3)])
      ...)))
```

```
(let ([x0 0])
  (let ([x3
         (let ([loop0 ‘()])
           (set! loop0
                 (lambda (x1)
                   (if (< x1 9)
                       (let ([x2 (+ x1 y0)])
                         (loop0 x2))
                       x1)))
           (loop0 x0))])
    (let ([y1 (+ y0 x3)])
      ...)))
```

```
(let ([x0 0])
  (let ([x3
         (let ([loop0 (make-vector 1 '())])
           (vector-set! loop0 0
             (lambda (x1)
               (if (< x1 9)
                   (let ([x2 (+ x1 y0)])
                     (let ([loop2
                            (vector-ref loop0 0)])
                       (loop2 x2)))
                   x1)))
           (let ([loop1 (vector-ref loop0 0)])
             (loop1 x0)))])
    (let ([y1 (+ y0 x3)])
      ...)))
```