SSA in Scheme

Static single assignment (SSA) :
assignment conversion ("boxing"),
alpha-renaming/alphatization,
and administrative normal form (ANF)
Roadmap

• Our compiler projects will target the LLVM backend.

• This will take us two more assignments:
  
  • Assignment 3: Fundamental simplifications, and implementation of continuations (& call/cc).
  
  • Assignment 4: Implementation of closures (closure conversion) and final code emission (LLVM IR).

• Assignment 5 focuses on top-level, matching, defines, etc
**LLVM**

- Major compiler framework: Clang (C/C++ on OSX), GHC, …

- LLVM IR: Assembly for an idealized virtual machine.

- IR allows an unbounded number of virtual registers:
  - Performs register allocation for various target platforms.
  - **But**, no register may shadow another or be mutated!

- Supports a variety of calling conventions (e.g., C, GHC, Swift, …).

- Uses low-level types (with flexible bit widths, e.g. i1, i32, i64, …).
x = a+1;
y = b*2;
y = (3*x) + (y*y);

Clang (C -> LLVM IR)

%x0 = add i64 %a0, 1
%y0 = mul i64 %b0, 2
%t0 = mul i64 3, %x0
%t1 = mul i64 %y0, %y0
%y1 = add i64 %t0, %t1
Static single assignment (SSA)?

- Significant added complexity in program analysis, optimization, and final code emission, arises from the fact that a single variable can be assigned in many places.

- This occurs both due to shadowing and direct mutation (\texttt{set!}).

- Thus each use of a variable \( X \) may hold a value assigned at one of several distinct points in the code.

- E.g., Constant propagation, common sub-expression elimination, type-recovery, control-flow analysis, etc.
SSA

- All variables are static, 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 $X_0, X_1, \ldots, X_i$.

- When control diverges and then joins back together, join points are made explicit using a special phi form, e.g.,

  $$X_5 \leftarrow \phi(X_2, X_4)$$
C-like IR

\[
x = f(x);
\]
\[
\text{if } (x > y) \text{ }
x = 0; \\
\text{else}
\{
\text{ } x += y; \\
\text{ } x *= x;
\}
\]
\[
\text{return } x;
\]

In SSA form

\[
x_1 = f(x_0);
\]
\[
\text{if } (x_1 > y_0) \text{ }
x_2 = 0; \\
\text{else}
\{
\text{ } x_3 = x_1 + y_0; \\
\text{ } x_4 = x_3 * x_3;
\}
\]
\[
x_5 \leftarrow \phi(x_2, x_4);
\]
\[
\text{return } x_5;
\]
\( x = \theta \);

while \((x < 9)\)
\[
\begin{align*}
    x &= x + y; \\
    y &= y + x;
\end{align*}
\]

\( x_0 = \theta; \)

label 0:
\[
\begin{align*}
    x_1 &\leftarrow \phi(x_0, x_2); \\
    c_0 &= x_1 < 9; \\
    \text{br} \ c_0, \text{label 1, label 2};
\end{align*}
\]

label 1:
\[
\begin{align*}
    x_2 &= x_1 + y_0; \\
    \text{br} \ \text{label 0};
\end{align*}
\]

label 2:
\[
\begin{align*}
    y_1 &= y_0 + x_1;
\end{align*}
\]
\[ x_0 = 0; \]

\[ x_1 \leftarrow \phi(x_0, x_2); \]
\[ c_0 = x_1 < 9; \]
\[ \text{br } c_0, \text{ label 1, label 2}; \]

\[ x_2 = x_1 + y_0; \]
\[ \text{br label 0}; \]

\[ y_1 = y_0 + x_1; \]
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)]
    [(...)])
```

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.
ANF conversion

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

(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);
\[(\text{let } ([x (+ a 1)])\]
\[(\text{let } ([y (* b 2)])\]
\[(\text{let } ([y (+ (* 3 x) (* y y))])\]
\[
\]
\[\text{...})])\]

\[
\]
\[
\]
\[\text{ANF conversion & alpha-renaming}\]

\[(\text{let } ([x0 (+ a0 1)])\]
\[(\text{let } ([y0 (* b0 2)])\]
\[(\text{let } ([t0 (* 3 x0)])\]
\[(\text{let } ([t1 (* y0 y0)])\]
\[(\text{let } ([y1 (+ t0 t1)])\]
\[
\]
\[\text{...})])])\]
What about join points?

\[ x_1 = f(x_0); \]

\[
\text{if } (x_1 > y_0) \text{ then } \\
\quad x_2 = 0; \\
\text{else } \\
\quad \{ \\
\qquad x_3 = x_1 + y_0; \\
\qquad x_4 = x_3 \times x_3; \\
\quad \} \\
\]

\[ x_5 \leftarrow \phi(x_2, x_4); \]

\[ \text{return } x_5; \]
What about join points?

\[ x_0 = 0; \]

```
(\text{let } ([x_0 \ 0])
 (\text{let } ([x_3
 (\text{let } \text{loop}_0 ([x_1 \ x_0])
 (\text{if } (< x_1 9)
 (\text{let } ([x_2 (+ x_1 y_0)])
 (\text{loop}_0 x_2))
 x_1))))
 (\text{let } ([y_1 (+ y_0 x_3)])
 ...)))

\text{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 ...
  (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)])
    ...))))