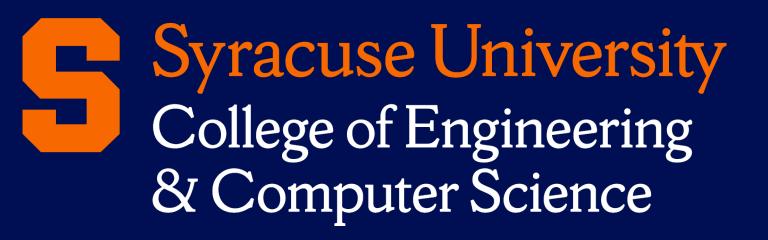
Static Single Assignment, A-Normal Form, and CPS

CIS531 — Fall 2025, Syracuse

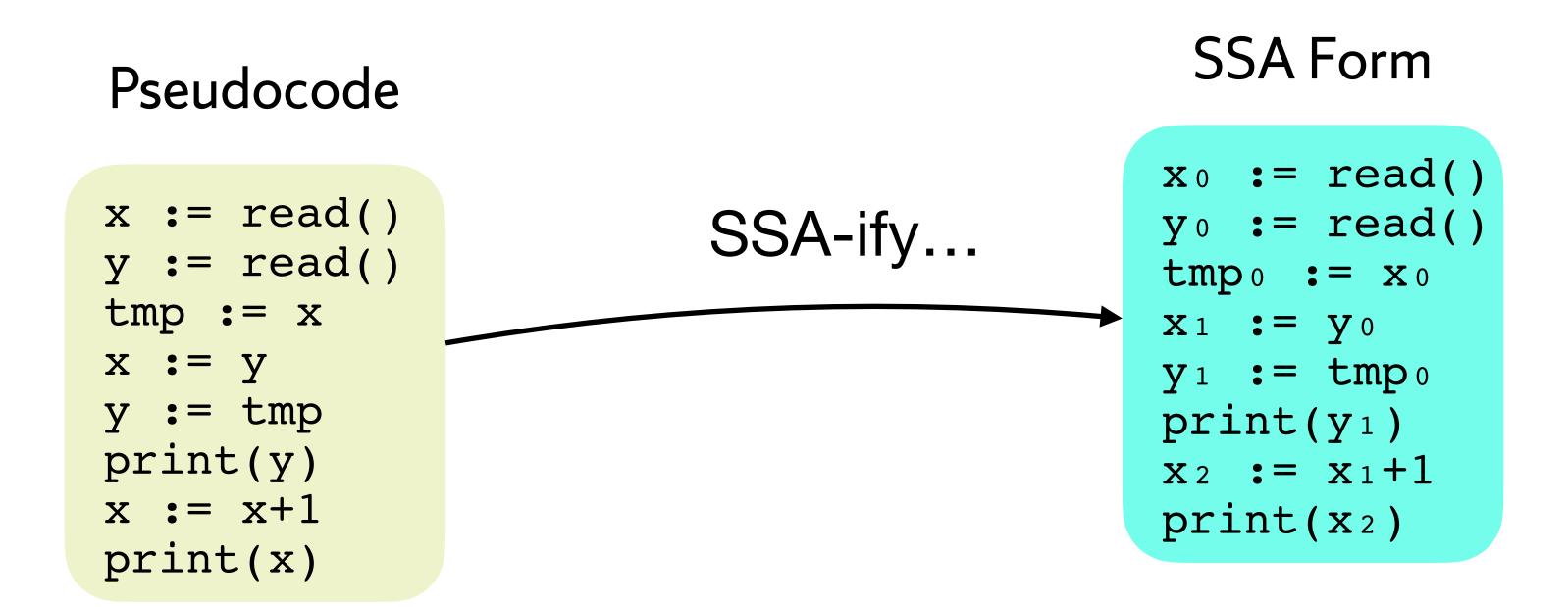
Kristopher Micinski



1

Static Single Assignment (SSA) Form

- Pervasive methodology of IR design: simplifies dataflow reasoning, optimization, analysis, register allocation, etc.
- Main rule: every variable defined once (single assignment)
 - If a variable x is written multiple times, replace it by several unique xs: x1, x2, etc...



SSA Form: who cares?

- When ultimately compiling to registers:
 - Many machine instructions use **source** registers **as destinations**
 - Thus, old values of registers will be clobbered, e.g., add %rbx, %r8
 - ♦ If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another variable, it is now replaced by result of add

 If %r8 was previously holding another was previously hol
- SSA makes obvious what version of a variable is <u>currently live</u> at every point
- Benefits:
 - One unique definition implies simpler program analysis
 - Easy to map SSA variables back to stack locations / registers

A-Normal Form

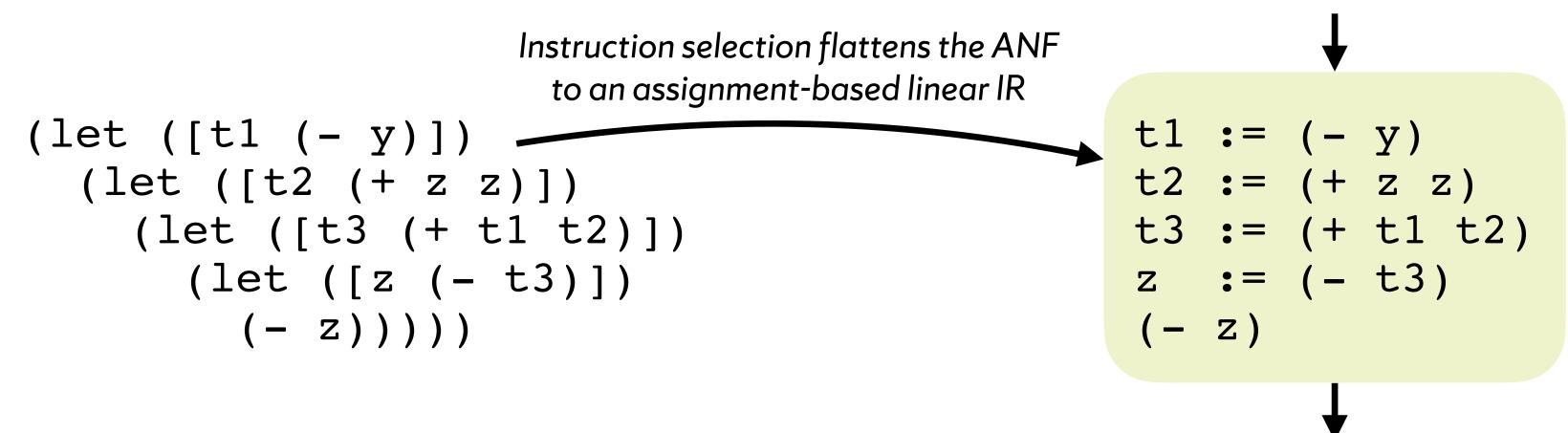
- A-Normal Form (ANF) is a functional-style equivalent of SSA
 - Not really different, just different ways of seeing SSA
- In ANF, we have "simple" and "complex" expressions
 - Atomic (simple) expressions are those which can be evaluated in a bounded number of steps
 - Complex expressions perform computation, or combine results
 - Function applications, conditional branching (if), etc.
- **ANF Rule**: every complex expression involves only simple arguments

Exercise: Convert to ANF

Exercise: Convert to ANF

ANF is SSA

- After uniqueify pass, every variable is distinct
 - * ANF introduces administrative bindings with new (gensym'd) names
 - Thus, in ANF, every variable assigned exactly once
- In LVar/R1, all expressions (post-A-normalization) are linear chains of lets
- We can view these as analogous to a flat SSA-style basic block



- We are seeing only some aspects of SSA right now—SSA also involves branching control-flow. We will talk about that soon, in this lecture I discuss SSA as viewed through the lens of LVar / R1
- In particular, SSA allows us to **branch** and **join**
 - Phi nodes allow combining variables from different branches / loops
- Branching control-flow is coming up next...

Continuation Passing Style (CPS)

- Continuation Passing Style (CPS) is an idea from functional programming
- A continuation is a function which represents the "rest of the computation"
- Many different styles of continuations:
 - Baked into the language itself (call/cc)
 - Internal to the compiler (compiler emits continuations) but not exposed
 - Delimited continuations
 - Exceptions are not continuations, but related
- Important to differentiate them!

Manual CPS

- We can **manually** write code in CPS by following a calling convention:
 - Every function accepts a "continuation" as its last argument
 - Every computation returns its result to its continuation
- No longer possible to "evaluate to" a value—must invoke the continuation

Manual CPS is a pain...

Literally **everything** needs a continuation if we're pedantic, even builtins like +, add1, etc.

Question: what about let?

Can see let as "left-left" lambda

First compute e, pass it a continuation that binds x and executes e-b

Strings of lets are roughly in CPS...

If every argument is **simple** then long strings of lets are essentially in CPS:

- Instead of explicit continuation passed to +, -, etc..., the continuation is the body of the let
- In real assembly, primitive functions don't accept continuations as arguments
- Instead, control implicitly continues on to the next instruction (in absence of branch instructions)

```
x0 := (- 1)
x1 := (+ x0 x0)
x2 := ...
x3 := ...
```

Manual CPS: Worked Example

- I do not think writing in manual CPS is good style:
 - CPS is programming with goto, and goto is to be discouraged
- But I think it is reasonable to show a worked, motivating example...
- Example: tail-recursive tree-traversal / iteration

```
;; How can we do a similar accumulator walk over a tree?
(define (tree? t)
   (match t
      ['empty #t]
      [`(node ,v ,(? tree? t0) ,(? tree? t1)) #t]
      [_ #f]))
```

Easy to write binary search in a tail-recursive fashion

- Always know "which way to go," never need to do backtracking
- By contrast, summing all the nodes requires us to backtrack
 - Because we must explore every node, not just single path down the tree

Direct-style sum-tree

- Sum-even-tree sums all even nodes
 - Motice it is in direct style: neither call is in tail position

```
(define (sum-tree t)
  (match t
    ['empty 0]
    [`(node ,v ,t0 ,t1)
          (+ v (sum-even-tree t0) (sum-even-tree t1))]))
```

Tail-recursive sum-even-tree, using manual CPS

- Notice how sum-tree takes a continuation k
- We still call functions like + in direct style—in practice rarely want full CPS

https://gist.github.com/kmicinski/42abb2faaef4b7b1fb632514f564725b

Converting to ANF/CPS

- Basic idea: write a function c-e which takes a continuation as an argument
- Continuation accepts an **atom** (variable or integer) as an argument
- Algorithm generates administrative bindings
- Uses continuations to flatten the control flow of complex expressions
- What I show is really a CPS conversion algorithm: I will discuss an optimization later in class which generates fewer administrative bindings

```
;; anf-convert : r1-e? -> r1-anf?
(define (anf-convert expr)
  ;; c-e converts a complex expression e
  ;; whenever it's done, we need to pass
  ;; its result to k.
  (define (c-e e k)
    (match e
      [(? integer? i) (k i)]
      [(? symbol? x) (k x)]
      [ (- ,e)
       (c-e e (\lambda (a0))
                (define x (gensym 'int))
                `(let ([,x (- ,a0)])
                   ,(k x))))]
      [ (+ ,e0 ,e1)
       (define e0-res (gensym 'int))
       (c-e e0 (lambda (a0)
                 (c-e el (lambda (al)
                            `(let ([,e0-res (+ ,a0 ,a1)])
                               ,(k e0-res))))))))
  (c-e expr (lambda (x) x)))
```