CS 7400 December 7–9, 2009
Dr. Wand Readings: see references

Lecture 11: Flow Analysis

Key Concepts:

Control-Flow Analysis
Constraint-based analysis
0CFA

In this lecture we will consider a different way of making predictions about the behavior of a program: *flow analysis*. We want to predict in more detail what are the possible values of any expression in the program, or the possible values of a variable. We'll see how it is possible to predict not just the type, but the possible values of each subexpression.

Along the way we can deduce how data flows through the program; this will be helpful for doing compiler optimizations. We'll see a little bit of this.

Now, in order to track the data through the program you need to know its *control-flow analysis* (which procedures are called from which applications). But we have a higher-order programming languages, in which we can write `proc(x) ... (x y) ...`, so you need to know the data flow before you can figure out the control flow. The moral of the story is that in a higher-order language like ours, you have to build both the control-flow analysis and the data-flow analysis at the same time, by mutual recursion.

So the key to our analysis will be tracking closures and how they flow through the program. So the analysis we will study is cometimes called *closure analysis*.

In order to generate a prediction, we will proceed much as we did for type inference: First, we will walk through the program, generating constraints that any sound prediction must satisfy. Then we will solve the constraints. This style of prediction is called *constraint-based analysis*. The particular analysis that we will do is called *0CFA*.

We will do this for our core language. As usual, we will first develop the theory, and then look at a simple implementation. Last, we will consider some extensions to more complex language features.

11.1 The Language

We will consider our core language, with an environment semantics. In order to uniquely identify subexpressions, we will label both expressions and values. This gives us the following grammar:

Core Language: Syntax

n	Integers
x	Variables
$e ::= n^l \mid -(e_1, e_2)^l$	Expressions
$\quad \mid x^l \mid (e_1\ e_2)^l \mid (\lambda x.e)^l$	
$v ::= n^l \mid ((\lambda x.e)^l, \rho)$	Values
$\rho ::= [] \mid [x = v]\rho$	Environments

Every expression has a label. We write lab(e) for the label of expression e. We define $\text{lab}(((\lambda x.e)^l, \rho)) = l$.

The environment semantics becomes:

Core Language: Environment-Passing Evaluation Rules

$$\textsc{Eval Const}\quad (n^l, \rho) \Downarrow n^l$$

$$\textsc{Eval Diff}\quad \frac{(e_1, \rho) \Downarrow n^{l_1} \qquad (e_2, \rho) \Downarrow m^{l_2} \qquad p = n - m}{(-(e_1, e_2)^l, \rho) \Downarrow p^l}$$

$$\textsc{Eval Var}\quad \frac{\rho(x) = v}{(x^l, \rho) \Downarrow v}$$

$$\textsc{Eval Abs}\quad ((\lambda x.e)^l, \rho) \Downarrow ((\lambda x.e)^l, \rho)$$

$$\textsc{Eval App}\quad \frac{(e_1, \rho) \Downarrow ((\lambda x.e')^{l_1}, \rho') \qquad (e_2, \rho) \Downarrow v \qquad (e', \rho'[v/x]) \Downarrow w}{((e_1\ e_2)^l, \rho) \Downarrow w}$$

Notice that every value is labelled with the label of the expression that created it.

11.2 Abstract Values and Predictions

We can't predict all the values, of course. We need something to predict that's finer than types. We will call these *abstract values*.

We choose our abstract values $\widehat{Val}$ to be the labels of value-creating expressions (constants, difference expressions, and abstractions).

An *analysis* (a prediction) will be a relation $pval \subseteq (Lab \cup Var) \times \widehat{Val}$. (Think: "possible value") Our intention is that for some set of computations (yet to be spelled out), $pval(l, l')$ means that one possible value of the expression labelled l is a value labelled l', and that $pval(x, l')$ means that x could be bound to a value labelled l'. In this case we say the *pval* is a correct prediction (for that set of computations).

Our goal will be to get from an expression e to a relation *pval* that is a correct prediction. We will do this by generating a logical formula involving *pval*, and using that formula to generate the relation.

11.3 Generating a Prediction

How can we write an analyzer, that is, a program to generate correct predictions? We'll do this in two steps: first we'll walk through the program and generate a set of constraints on *pval*. Then we'll solve the constraints.

We do this in the language of *Datalog*. Datalog formulas are described by the following grammar:

Datalog Formulas

p	Predicate Symbols
x	Variables
c	Constants
$t ::= x \mid c$	Terms
$at ::= p(t, \ldots, t)$	Atomic Formulas
$F ::= (at_1 \wedge \ldots \wedge at_n) \Rightarrow (at'_1 \wedge \ldots \wedge at'_n)$	Formulas
$\quad \mid F \wedge F$	

Formulas are subject to the condition that every variable that appears in the conclusion must have occurred in at least one of the hypotheses. We call a formula with no hypotheses and a single conclusion (which therefore must have no variables) an *assertion*.

So Datalog formulas are like Prolog clauses, except that the terms contain no function symbols.

Datalog formulas have the nice property that that they have *finite closures*: that is, given any finite set of formulas, their deductive closure is finite. So we can take a set of Datalog clauses and find all its consequences safely.

Given a Datalog formula F over predicate symbols $p_1, \ldots, p_n$, a tuple σ of actual predicates $P_1, \ldots P_n$ is said to be a *solution* of F if the predicates P_i make F true (in the obvious way). We write or $\sigma \models F$ or $(P_1, \ldots P_n) \models F$

Given a Datalog formula F, we can find a solution by taking its deductive closure, and then defining each predicate P_i to be true at $(c_1, \ldots, c_n)$ iff $p_i(c_1, \ldots, c_n)$ is deducible from F. Furthermore, this is the *smallest* solution of F.

For example, given the formulas

$$P(a, b)$$
$$P(b, c)$$
$$P(c, d)$$
$$P(x, y) \wedge P(y, z) \Rightarrow P(x, z)$$

the deductive closure is the set of assertions

$$P(a, b)$$
$$P(a, c)$$
$$P(a, d)$$
$$P(b, c)$$
$$P(b, d)$$
$$P(c, d)$$

We'll look at this process more closely later.

To do an analysis, we start with a labelled expression, and we represent it in Datalog by a set of assertions, one for each node in its syntax tree.

Representing Syntax in Datalog

Node	Assertion
n^l	$const(l)$
x^l	$var(l, x)$
$-(e_1, e_2)^l$	$diff(\text{lab}(e_1), \text{lab}(e_2))$
$(\lambda x.e)^l$	$abs(l, x, \text{lab}(e))$
$(e_1\ e_2)^l$	$app(l, \text{lab}(e_1), \text{lab}(e_2))$

We can define the set $\langle\langle e \rangle\rangle$ of assertions for any expression inductively:

$$\begin{aligned}
\langle\langle n^l \rangle\rangle &= const(l) \\
\langle\langle x^l \rangle\rangle &= var(l, x) \\
\langle\langle -(e_1, e_2)^l \rangle\rangle &= diff(\text{lab}(e_1), \text{lab}(e_2)) \wedge \langle\langle e_1 \rangle\rangle \wedge \langle\langle e_2 \rangle\rangle \\
\langle\langle (\lambda x.e)^l \rangle\rangle &= abs(l, x, \text{lab}(e)) \wedge \langle\langle e \rangle\rangle \\
\langle\langle (e_1\ e_2)^l \rangle\rangle &= app(l, \text{lab}(e_1), \text{lab}(e_2)) \wedge \langle\langle e_1 \rangle\rangle \wedge \langle\langle e_2 \rangle\rangle
\end{aligned}$$

While we're at it, we might as well extend this to values and environments:

$$\begin{aligned}
\langle\langle ((\lambda x.e)^l, \rho) \rangle\rangle &= \langle\langle (\lambda x.e)^l \rangle\rangle \wedge \langle\langle \rho \rangle\rangle \\
\langle\langle \rho \rangle\rangle &= \textstyle\bigwedge_{x \in dom(\rho)} (pval(x, \text{lab}(\rho(x))) \wedge \langle\langle \rho(x) \rangle\rangle)
\end{aligned}$$

This says that we are going to put in assertions for every subexpression appearing in the expression, and for every expression and subexpression appearing in the environment (recursively!) and also add the assertion that every binding in the environment is consistent with *pval*.

Here's an example of an expression, its labelling (in a slightly different concrete syntax than we have here), and the corresponding assertions:

```
> (assertions "(proc x -(x,22)  33)")
((0 : (app-exp
        (2 : (proc-exp x
               (3 : (diff-exp
                      (5 : (var-exp x))
                      (4 : (const-exp 22))))))
        (1 : (const-exp 33))))

 (const 1)
 (const 4)
 (var 5 x)
 (diff 3 5 4)
 (abs 2 x 3)
 (app 0 2 1))
```

Next we will write some Datalog formulas that describe flow of values in the program. In these formulas, we will use two more predicate symbols:

- $pval(l_1, l_2)$ means that l_1 is the label of a possible value of the expression at l_2. Similarly, $pval(x, l_1, x)$ means that l_1 is the label of a possible binding of the variable x.
- $flow(l_1, l_2)$ means that all values from label l_1 might flow to label l_2. We similarly write $flow(x, l)$ or $flow(l, x)$.

Now we can write down the formulas. We call this set of formulas *CFA*. Here l, etc., may range over labels and variable names.

1. $flow(l_1, l_2) \wedge pval(l_1, l) \Rightarrow pval(l_2, l)$.
2. $const(l) \Rightarrow pval(l, l)$
3. $diff(l, l_1, l_2) \Rightarrow pval(l, l)$
4. $abs(l, x, l') \Rightarrow pval(l, l)$
5. $var(l, x) \Rightarrow flow(x, l)$
6. $app(l, l_1, l_2) \wedge pval(l_1, l') \wedge abs(l', x, l'') \Rightarrow flow(l_2, x) \wedge flow(l'', l)$

- Formula (1) says that if there is a flow from l_1 to l_2, then any possible value of l_1 is also a possible value of l_2.
- Formulas (2)-(4) say that constants, difference expressions, and abstractions are all sources of values: each value is labelled by its source.
- Formula (5) says that if l is the label of a use of variable x, then there is a flow from x to l.
- Formula (6) describes the flows around a procedure call. *If* l is an application *and* l' is a possible value of the rator, *and* l' labels an abstraction whose bound variable is x and whose body is labelled l'', *then*: any possible value of the operand is a possible value of x, and any possible value of the procedure body is a possible value of the application.

  This is the key: keep it under your pillow at night.

11.4 The Soundness Theorem

Our soundness theorem states that if σ is a solution to the constraints $\langle\langle e \rangle\rangle \wedge \mathit{CFA}$, then σ makes correct predictions about e.

But to do this, we need a stronger induction hypothesis.

> **Theorem:** If $\sigma \models \langle\langle e \rangle\rangle \wedge \langle\langle \rho \rangle\rangle \wedge \mathit{CFA}$, and $(e, \rho) \Downarrow v$, then $\sigma \models \mathit{pval}(\mathrm{lab}(e), \mathrm{lab}(v))$ and $\sigma \models \langle\langle v \rangle\rangle$.

Proof: We proceed by induction on the height of the derivation of $(e, \rho) \Downarrow v$.

The interesting case is application.

The other rules are all straightforward.

11.5 `interps/cfa`: Implementing a Flow Analysis

To implement this system, we need to first label the parsed expression and generate the constraints, and then solve the generated constraints.

```
(module labels (lib "eopl.ss" "eopl")
  ;; produce syntactic assertions from a program

  (require "lang.scm")
  (provide assertions-of-program assertions)

  ;; state of the labeller:
  (define assertion-list '())
  (define next-free-label 0)

  ;; main entry points:
  (define (assertions-of-program pgm)
    (cases program pgm
      (a-program (exp)
        (set! assertion-list '())
        (set! next-free-label 0)
        (assertions-of-exp exp))))

  (define (assertions str)
    (assertions-of-program (scan&parse str)))

  ;; manipulating the state:
  (define next-label
    (lambda ()
      (let ((x next-free-label))
        (set! next-free-label (+ next-free-label 1))
        x)))

  (define assert!
    (lambda (x)
      (set! assertion-list (cons x assertion-list))))
```

```
  ;; assertions:

  ;; assertion ::= (CONST l ) | (DIFF l l1 l2) | (VAR l x)
  ;;             |  (ABS l x l') | (APP l l1 l2)

  ;; first argument is always the label of the current expression.

  ;; assertions-of : exp -> !(pair-of labelled-exp (list-of assertion))

  ;; main loop (called label) takes exp and its label, and returns the
  ;; labelled expression, adding its constraints to the
  ;; list by local side-effect.

(define (assertions-of-exp exp0)
  (let ((labelled-exp
         (let label ;; exp * label -> !labelled-exp
           ((exp exp0) (lab (next-label)))
           (cases expression exp
             (const-exp (number)
               (assert! (list 'const lab))
               (list lab ': (list 'const number)))
             (var-exp (id)
               (assert! (list 'var lab id))
               (list lab ': (list 'var-exp id)))
             (diff-exp (exp1 exp2)
               (let ((lab1 (next-label))
                     (lab2 (next-label)))
                 (begin
                   (assert! (list 'diff lab lab1 lab2))
                   (list lab ':
                     (list 'diff-exp (label exp1 lab1) (label exp2 lab2))))))
             (proc-exp (id body)
               (let ((lab1 (next-label)))
                 (begin
                   (assert! (list 'abs lab id lab1))
                   (list lab ': (list 'proc-exp id (label body lab1))))))
```

```
            (app-exp (rator rand)
              (let ((lab1 (next-label))
                    (lab2 (next-label)))
                (begin
                  (assert! (list 'app lab lab1 lab2))
                  (list lab ':
                    (list 'app-exp
                      (label rator lab1)
                      (label rand lab2)))))))))
  ;; now return the pair consisting of the labelled expression and
  ;; the list of generated assertions
  (cons labelled-exp assertion-list)))

)
```

To solve the constraints, we build a little Datalog solver.

```
(module datalog (lib "eopl.ss" "eopl")

  (require "utils.scm")

  ;; extremely simple-minded datalog solver
  ;; using the worklist algorithm

  (provide (all-defined))

  ;; assertion ::= (symbol literal ...)
  ;; literal ::= number | symbol

  (define assertion?
    (pair-of
      symbol?
      (list-of (either number? symbol?))))

  (define-datatype formula formula?
    (a-formula
      (patvars (list-of symbol?))       ; pattern variables
      (hypotheses (both
                   non-empty?
                   (list-of assertion?)))
      (conclusions (list-of assertion?))))

  ;; every pattern variable in the conclusions should be bound by in the
  ;; hypotheses, but we don't check this.  Indeed define-datatype
  ;; doesn't provide an easy way to do so :-{
```

```
;;;;;;;;;;;;;;;; the solver ;;;;;;;;;;;;;;;;

;; state of the solver is the worklist and the conclusions, which
;; are disjoint sets of formulas or assertions, without repetitions.

;; furthermore, a formula can be in conclusions only if it's already
;; been resolved against all the other formulas in the conclusions

(define conclusions '())
(define worklist '())

;; manipulating the solver state.  A formulas always starts in the
;; worklist and then gets moved to the conclusions once its been
;; rubbed against all the conclusions.

(define (add-to-worklist! formula)
  (if (and
       (not (member formula worklist))
       (not (member formula conclusions)))
    (begin
      (set! worklist (cons formula worklist))
      )
    ))

;; formula will always come from worklist, which is disjoint from
;; conclusions.

(define (add-to-conclusions! formula)
  (set! conclusions (cons formula conclusions)))

(define (take-from-worklist)
  (let ((this (car worklist)))
    (set! worklist (cdr worklist))
    this))
```

```
;; main entry points for the solver

;; (list-of formula) -> (list-of formula)

(define (datalog-closure formulas)
  (set! worklist formulas)
  (set! conclusions '())              ; global
  (let outer-loop! () ; yes, it's a while loop
    (if (null? worklist)
      conclusions
      (begin
        (let ((this-formula (take-from-worklist)))
          (rub-formula-against-list! this-formula conclusions)
          (add-to-conclusions! this-formula)
          (outer-loop!))))))

;; take a list of formulas and remove everything but the assertions.
(define (filter-assertions formulas)
  (cond
    ((null? formulas) '())
    ((assertion? (car formulas))
     (cons (car formulas)
       (filter-assertions (cdr formulas))))
    (else (filter-assertions (cdr formulas)))))
```

```
;; rub this-formula against each of the other-formulas.  This could
;; be inlined in datalog-closure.

(define rub-formula-against-list!
  (lambda (this-formula other-formulas)
    (for-each
      (lambda (other-formula)
        (rub-two-formulas! this-formula other-formula))
      other-formulas)))

;; rub two formulas, and put any conclusions on the worklist

(define rub-two-formulas!
  (lambda (fmla1 fmla2)
    (cond
      ((and (assertion? fmla1) (not (assertion? fmla2)))
       (rub-assertion-against-implication! fmla1 fmla2))
      ((and (assertion? fmla2) (not (assertion? fmla1)))
       (rub-assertion-against-implication! fmla2 fmla1))
      (else #t))))
```

```
;; this is kinda ugly because it has to deal with the different data
;; structures for formulas and atomic formulas.

(define (rub-assertion-against-implication! atom fmla)
  (cases formula fmla
    (a-formula (patvars hypotheses conclusions)
      (cond
        ((match-assertion (car hypotheses) atom patvars)
         =>
         (lambda (env)
           (if (null? (cdr hypotheses))
             (for-each
               (lambda (conclusion)
                 (add-to-worklist!
                   (subst-in-assertion conclusion env)))
               conclusions)
             (add-to-worklist!
               (subst-in-formula
                 (a-formula patvars (cdr hypotheses) conclusions)
                 env)))))))))
```

```
;;;;;;;;;;;;;;;; pattern-matching and substitution ;;;;;;;;;;;;;;;;

  (define (match-assertion pattern0 subject0 patvars)
    (let loop ((pattern pattern0) (subject subject0) (env '()))
      (cond
        ;; nothing to match, return accumulated environment
        ((null? pattern) env)
        (else
          (let ((pat-item (subst-in-literal (car pattern) env)))
            (cond
              ;; pat-item is an unbound pattern variable.  So bind it.
              ((memv pat-item patvars)
               (loop (cdr pattern) (cdr subject)
                 (cons
                   (cons pat-item (car subject))
                   env)))
              ;; pat-item is a literal
              ((same-literal? pat-item (car subject))
               (loop (cdr pattern) (cdr subject) env))
              (else #f)))))))

  (define same-literal? eqv?)
```

```
;; substitution functions

(define subst-in-literal
  (lambda (literal env)
    (cond
      ((assv literal env) => cdr)
      (else literal))))

(define subst-in-assertion
  (lambda (assertion env)
    (map (lambda (x) (subst-in-literal x env)) assertion)))

(define subst-in-assertions
  (lambda (assertions env)
    (map (lambda (x) (subst-in-assertion x env)) assertions)))

(define subst-in-formula
  (lambda (fmla env)
    (cases formula fmla
      (a-formula (patvars hypotheses conclusions)
        (a-formula
          patvars
          (subst-in-assertions hypotheses env)
          (subst-in-assertions conclusions env))))))


)
```

Next, we write down the theory of 0CFA:

```
(module 0cfa (lib "eopl.ss" "eopl")

  ;; the theory of 0cfa

  (require "datalog.scm")               ; for a-formula
  (provide 0cfa)

  (define 0cfa
    (list
      (a-formula '(l1 l2 l3)
        '((flow l1 l2) (pval l1 l3))
        '((pval l2 l3)))
      (a-formula '(l)
        '((const l))
        '((pval l l)))
      (a-formula '(l l1 l2)
        '((diff l l1 l2))
        '((pval l l)))
      (a-formula '(l x)
        '((var l x))
        '((flow x l)))
      (a-formula '(l x l1)
        '((abs l x l1))
        '((pval l l)))
      (a-formula '(lapp lrator lrand labs bv lbody)
        '((app lapp lrator lrand)
          (pval lrator labs)
          (abs labs bv lbody))
        '((flow lrand bv)
          (flow lbody lapp)))))

)
```

Last, we tie it all together:

```
(module top (lib "eopl.ss" "eopl")

  (require "labels.scm")
  (require "datalog.scm")
  (require "0cfa.scm")
  (require "tabulate.scm")
  (require "tests.scm")

  (define (test-all)
    (map analyze test-list))

  (define (analyze str)
    (let* ((res (assertions str))
           (soln
             (filter-assertions
               (datalog-closure
                 (append 0cfa (cdr res))))))
      (list
        (car res)
        (tabulate soln)
        )))

  )
```

Let’s do a few examples:

```
> (analyze "((proc x proc y -(x, y) 33) 44)")
((0
  :
  (app-exp
    (1
     :
     (app-exp
       (3
        :
        (proc-exp
          x
          (5 : (proc-exp y
                 (6 : (diff-exp (7 : (var-exp x)) (8 : (var-exp y))))))))
       (4 : (const 33))))
    (2 : (const 44))))
 (pval:
  ((0 (6)) (1 (5)) (2 (2)) (3 (3)) (4 (4)) (5 (5))
   (6 (6)) (7 (4)) (8 (2)) (x (4)) (y (2)))
  flow:
  ((2 y) (6 0) (4 x) (5 1) (x 7) (y 8))))
```

```
>  (analyze "(proc f ((f 33) 44)   proc x proc y -(x,y))")
((0
  :
  (app-exp
    (1
     :
     (proc-exp
       f
       (3 : (app-exp
              (4 : (app-exp (6 : (var-exp f)) (7 : (const 33))))
              (5 : (const 44))))))
    (2
     :
     (proc-exp
       x
       (8 : (proc-exp y
              (9 : (diff-exp (10 : (var-exp x)) (11 : (var-exp y)))))))))
 (pval:
  ((0 (9)) (1 (1)) (2 (2)) (3 (9)) (4 (8)) (5 (5)) (6 (2)) (7 (7))
   (8 (8)) (9 (9)) (10 (7)) (11 (5)) (f (2)) (x (7)) (y (5)))
  flow:
  ((7 x) (5 y) (9 3) (8 4) (2 f) (3 0) (f 6) (x 10) (y 11))))
```

The next example shows multiple flows into x.

```
>  (analyze "(proc f -((f 11), (f 22))  proc x -(x,10))")
((0 : (app-exp
        (1 : (proc-exp f
               (3 : (diff-exp
                      (4 : (app-exp (6 : (var-exp f)) (7 : (const 11))))
                      (5 : (app-exp (8 : (var-exp f)) (9 : (const 22))))))))
        (2 : (proc-exp x
               (10 : (diff-exp (11 : (var-exp x)) (12 : (const 10)))))))))
 (pval:
   ((0 (3))
    (1 (1))
    (2 (2))
    (3 (3))
    (4 (10))
    (5 (10))
    (6 (2))
    (7 (7))
    (8 (2))
    (9 (9))
    (10 (10))
    (11 (9 7))
    (12 (12))
    (f (2))
    (x (9 7)))
   flow:
   ((7 x) (10 4) (9 x) (10 5) (2 f) (3 0) (f 6) (f 8) (x 11))))
```

We don't *have* to do anything special for letrec. Just putting in the regular combinator will do the right thing.

```
(analyze "
     (proc fix
      ((fix proc g proc x -(x, (g -(x, 1))))
       66)
      proc f                    % definition of fix
       (proc y (f (y y))
        proc z (f (z z))))")
((0 :
   (app-exp
     (1 :
       (proc-exp fix
         (3 :
           (app-exp
             (4 :
               (app-exp
                 (6 : (var-exp fix))
                 (7 : (proc-exp g
                        (8 :
                          (proc-exp x
                            (9 :
                              (diff-exp
                                (10 : (var-exp x))
                                (11 : (app-exp
                                        (12 : (var-exp g))
                                        (13 : (diff-exp
                                                (14 : (var-exp x))
                                                (15 : (const 1))))))))))))))
             (5 : (const 66))))))
     (2 :
       (proc-exp f
         (16 :
           (app-exp
             (17 :
               (proc-exp y
                 (19 :
```

```
      (app-exp
        (20 : (var-exp f))
        (21 : (app-exp
                (22 : (var-exp y))
                (23 : (var-exp y))))))))
(18 :
  (proc-exp z
    (24 :
      (app-exp
        (25 : (var-exp f))
        (26 : (app-exp
                (27 : (var-exp z))
                (28 : (var-exp z)))))))))))))
```

```
(pval:
 ((0 (9))  ;; the value of the whole expression comes from the diff-exp
  (1 (1))
  (2 (2))
  (3 (9))
  (4 (8))  ;; the value of the call to fix comes from its argumet
  (5 (5))
  (6 (2))
  (7 (7))
  (8 (8))
  (9 (9))
  (10 (13 5))  ;; the value of x may come from the starting value or from
               ;; the -(x,1)
  (11 (9))     ;; what goes around, comes around!
  (12 (8))
  (13 (13))
  (14 (13 5))  ;; another use of x
  (15 (15))
  (16 (8))
  (17 (17))
  (18 (18))
  (19 (8))
  (20 (7))
  (21 (8))
  (22 (18))
  (23 (18))
  (24 (8))
  (25 (7))
  (26 (8))
  (27 (18))
  (28 (18))
  (f (7))
  (fix (2))
  (g (8))
  (x (13 5))  ;; here's the value for x.
  (y (18))
  (z (18)))
```

```
  flow:
  ((21 g)
   (5 x)
   (9 3)
   (8 19)
   (13 x)
   (9 11)
   (26 g)
   (8 24)
   (7 f)
   (16 4)
   (2 fix)
   (3 0)
   (fix 6)
   (x 10)
   (g 12)
   (x 14)
   (28 z)
   (24 26)
   (23 z)
   (24 21)
   (18 y)
   (19 16)
   (f 20)
   (y 22)
   (y 23)
   (f 25)
   (z 27)
   (z 28))))
>
```

11.6 Enhancements

- `let`, `letrec`, etc. We didn't do `if`, either. What should the constraints be?
- Tracking scalars more finely. We could have evens, odds, as abstract values.
- Tracking errors. If a scalar or a procedure of the wrong number of arguments shows up as a possible value of an operator, then an error should be reported as a possible value of the application. And errors need to be propagated, both in the evaluation semantics and in the analysis. Do something similar if a procedure shows up in an operand of a primapp.
- Tracking booleans. Now that we're tracking scalars, we can check the test part of a conditional. You can check not only for a procedure showing up as the value of the test, but since you know the points of origin, you can check to see that every scalar arriving at the test is a boolean. Better yet, if you're lucky you can predict that the value will always be `true` (or `false`)– then the compiler can eliminate dead code for the branch that's never taken.
- Side-effects. Extend this idea to store operations like those in `explicit-store`, and to mutable data structures like those in `mutable-pairs`.

  `newref`'s allocate new locations in the store, so we can take the labels of `newref`'s to be abstract locations. Then we can add a new predicate *sto*, with the interpretation that:

  > $sto(l_1, l_2)$ iff l_1 is the label of a possible contents of a location allocated at l_2.

Then the assertions and flows associated with each of the store operations will be:

Analyses for store operations

$\texttt{newref}(e)^l$	$\mathit{newref}(l, \mathrm{lab}(e))$	$\mathit{newref}(l,l') \Rightarrow \sigma(l,l)$ $\wedge\ (\mathit{newref}(l,l') \wedge \sigma(l'',l')) \Rightarrow \mathit{sto}(l'',l)$
$\texttt{deref}(e)^l$	$\mathit{deref}(l, \mathrm{lab}(e))$	$\mathit{deref}(l,l_1) \wedge \mathit{sto}(l_2,l_1) \Rightarrow \sigma(l_2,l)$
$\texttt{setref}(e_1,e_2)^l$	$\mathit{setref}(l, \mathrm{lab}(e_1), \mathrm{lab}(e_2))$	$\mathit{setref}(l,l_1,l_2) \wedge \sigma(l_1',l_1) \wedge \sigma(l_2',l_2) \Rightarrow \mathit{sto}(l_2',l_1')$

This can be extended similarly for mutable pairs.