Relation Semantics of Local Variable Scoping

Kamal Aboul-Hosn & Dexter Kozen

PLDG

22 September 2005

Motivation

- Prove equivalence of programs w/local variables using Kleene algebra with tests (KAT)

```
swap(x,y)
{
  t := x
  x := y
  y := t
}
```

$$\stackrel{?}{=}$$

```
swap(x,y)
{
  x := x + y
  y := x - y
  x := x - y
}
```

Goals

- Define notion of local variable scoping with semantics based on binary relations that is:
  - Purely compositional
  - Fully abstract
  - Able to capture contextual considerations
- Provide axioms to prove equivalence under with local variables and no context

(Lots of) Related Work

- Meyer & Sieber: Using store model of Halpern-Meyer-Trakhtenbrot to prove equivalence of ALGOL programs
- Mason & Talcott: Contextual assertions for reasoning with context and semantic reasoning
- Pitts: Equivalence of ML programs with references using operational semantics
- Operational semantics, denotational semantics, game semantics,...

Relational Semantics

- *Domain of computation*: first-order structure $\mathfrak{A}$ over signature Σ
- *Partial Valuation f*: $\mathsf{Var} \rightarrow |\mathfrak{A}|$
- Domain of *f:* dom *f*
- *Environment*: Stack of partial valuations σ, τ
  - $f :: \sigma$: environment with head *f* and tail σ
  - *Shape* of $f_1 :: \cdots :: f_n$: $\bigcup_{i=1}^{n} \mathrm{dom}\ f_i$

Programs

- Binary relations on environments
- Built inductively from atomic programs and tests
  - Atomic programs $x := t$
  - Operators: $+$, $;$, $*$
- Simpler to work with than `while` and `if` programming constructs

Scoping Expression

$\textsf{let } x_1 = t_1, \ldots, x_n = t_n \textsf{ in } p \textsf{ end}$

- Pushes new partial valuation onto environment
- Domain: $\{x_1, \ldots, x_n\}$
- Initial values are values of $t_1, \ldots, t_n$ evaluated in old environment
- Shadow any occurrences of variables further down the stack
- Pop partial valuation when leaving scope

Evaluating a Variable

- Evaluating an undefined variable: no input/output pair in relational semantics
- *Rebinding operator* on partial valuations

$$f[x/a](y) = \begin{cases} f(y), & \text{if } y \in \text{ dom } f \text{ and } y \neq x, \\ a, & \text{if } y \in \text{ dom } f \text{ and } y = x, \\ \text{undefined}, & \text{if } y \notin \text{ dom } f. \end{cases}$$

$$\sigma(x) = \begin{cases} f(x), & \text{if } \sigma = f :: \tau \text{ and } x \in \text{dom } f, \\ \tau(x), & \text{if } \sigma = f :: \tau \text{ and } x \notin \text{dom } f, \\ \text{undefined}, & \text{if } \sigma = \epsilon. \end{cases}$$

$$\sigma[x/a] = \begin{cases} f[x/a] :: \tau, & \text{if } \sigma = f :: \tau \text{ and } x \in \text{dom } f, \\ f :: \tau[x/a], & \text{if } \sigma = f :: \tau \text{ and } x \notin \text{dom } f, \\ \epsilon, & \text{if } \sigma = \epsilon. \end{cases}$$

Evaluating a Program

$$[\![x := t]\!] = \{(\sigma, \sigma[x/\sigma(t)]) \mid \sigma(t) \text{ and } \sigma(x) \text{ are defined}\}$$

$$\begin{aligned} &[\![\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}]\!] \\ &\quad = \quad \{(\sigma, \mathsf{tail}(\tau)) \mid \sigma(t_i) \text{ is defined, } 1 \leq i \leq n, \text{ and } (f :: \sigma, \tau) \in [\![p]\!]\} \end{aligned}$$

- f is the environment such that $f(x_i) = \sigma(t_i)$, $1 \leq i \leq n$
- Semantics for +, ;, and * are union, relational composition, and reflexive transitive closure, respectively

Evaluating Tests

$$\begin{aligned}&[\![R(t_1,\ldots,t_n)]\!]\\ &\quad = \quad \{(\sigma,\sigma) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } \mathfrak{A},\sigma \vDash R(t_1,\ldots,t_n)\}\end{aligned}$$

$$\begin{aligned}&[\![!R(t_1,\ldots,t_n)]\!]\\ &\quad = \quad \{(\sigma,\sigma) \mid \sigma(t_i) \text{ is defined, } 1 \le i \le n, \text{ and } \mathfrak{A},\sigma \vDash \neg R(t_1,\ldots,t_n)\}\end{aligned}$$

- Not classical negation (does not contain states in which $\sigma(t_i)$ is undefined)
- Need to have test for undefined variables

$$[\![\mathsf{undefined}(x)]\!] \quad = \quad \{(\sigma,\sigma) \mid \sigma(x) \text{ is undefined}\}.$$

Short-Circuit Boolean Ops

$$
\begin{aligned}
[\![\varphi \ \&\& \ \psi]\!] &= [\![\varphi]\!] \cap [\![\psi]\!] \\
[\![\varphi \ || \ \psi]\!] &= [\![\varphi]\!] \cup ([\![!\varphi]\!] \cap [\![\psi]\!]) \\
[\![!(\varphi \ \&\& \ \psi)]\!] &= [\![!\varphi]\!] \cup ([\![\varphi]\!] \cap [\![!\psi]\!]) \quad = \quad [\![!\varphi \ || \ !\psi]\!] \\
[\![!(\varphi \ || \ \psi)]\!] &= [\![!\varphi]\!] \cap [\![!\psi]\!] \quad = \quad [\![!\varphi \ \&\& \ !\psi]\!] \\
[\![!!\varphi]\!] &= [\![\varphi]\!]
\end{aligned}
$$

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
                              (y = 5,z = 20)
```

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

(y = 5,z = 20)

(x = 1)::(y = 5,z = 20)

Example

```
let x=1 in x:=y+z;
             let y=x+2 in y:=y+z; z:=y+1 end;
             y:=x
end
```

```
                              (y = 5,z = 20)
                     (x = 1)::(y = 5,z = 20)
                    (x = 25)::(y = 5,z = 20)
```

Example

```
let x=1 in x:=y+z;
              let y=x+2 in y:=y+z; z:=y+1 end;
              y:=x
end
                                       (y = 5,z = 20)
                             (x = 1)::(y = 5,z = 20)
                            (x = 25)::(y = 5,z = 20)
                 (y = 27)::(x = 25)::(y = 5,z = 20)
```

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

```
                                                    (y = 5,z = 20)
                                          (x = 1)::(y = 5,z = 20)
                                         (x = 25)::(y = 5,z = 20)
                               (y = 27)::(x = 25)::(y = 5,z = 20)
                               (y = 47)::(x = 25)::(y = 5,z = 20)
```

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

```
                                   (y = 5,z = 20)
                          (x = 1)::(y = 5,z = 20)
                         (x = 25)::(y = 5,z = 20)
               (y = 27)::(x = 25)::(y = 5,z = 20)
               (y = 47)::(x = 25)::(y = 5,z = 20)
               (y = 47)::(x = 25)::(y = 5,z = 48)
```

Example

```
let x=1 in x:=y+z;
              let y=x+2 in y:=y+z; z:=y+1 end;
              y:=x
end
```

```
                                       (y = 5,z = 20)
                            (x = 1)::(y = 5,z = 20)
                           (x = 25)::(y = 5,z = 20)
                (y = 27)::(x = 25)::(y = 5,z = 20)
                (y = 47)::(x = 25)::(y = 5,z = 20)
                (y = 47)::(x = 25)::(y = 5,z = 48)
                           (x = 25)::(y = 5,z = 48)
```

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

```
                                       (y = 5,z = 20)
                              (x = 1)::(y = 5,z = 20)
                             (x = 25)::(y = 5,z = 20)
                   (y = 27)::(x = 25)::(y = 5,z = 20)
                   (y = 47)::(x = 25)::(y = 5,z = 20)
                   (y = 47)::(x = 25)::(y = 5,z = 48)
                             (x = 25)::(y = 5,z = 48)
                            (x = 25)::(y = 25,z = 48)
```

Example

```
let x=1 in x:=y+z;
           let y=x+2 in y:=y+z; z:=y+1 end;
           y:=x
end
```

```
                       (x = 1)::(y = 5,z = 20)
                      (x = 25)::(y = 5,z = 20)
            (y = 27)::(x = 25)::(y = 5,z = 20)
            (y = 47)::(x = 25)::(y = 5,z = 20)
            (y = 47)::(x = 25)::(y = 5,z = 48)
                      (x = 25)::(y = 5,z = 48)
                     (x = 25)::(y = 25,z = 48)
                               (y = 25,z = 48)
```

Axioms & Properties

- If the y_i are distinct and do not occur in p, $1 \leq i \leq n$, then the following two programs are equivalent:

  $$\text{let } x_1 = t_1, \ldots, x_n = t_n \text{ in } p \text{ end}$$
  $$\text{let } y_1 = t_1, \ldots, y_n = t_n \text{ in } p[x_i/y_i \mid 1 \leq i \leq n] \text{ end}$$

- If y does not occur in s, ttftpae:

  $$\text{let } x = s \text{ in let } y = t \text{ in } p \text{ end end}$$
  $$\text{let } y = t[x/s] \text{ in let } x = s \text{ in } p \text{ end end}$$

- If x does not occur in s, ttftpae:

  $$\text{let } x = s \text{ in let } y = t \text{ in } p \text{ end end}$$
  $$\text{let } x = s \text{ in let } y = t[x/s] \text{ in } p \text{ end end}$$

Axioms & Properties

- If x_1 does not occur in $t_2,\ldots,t_n$, ttftpae:

  $\mathsf{let}\ x_1 = t_1, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}$

  $\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ \mathsf{let}\ x_2 = t_2, \ldots, x_n = t_n\ \mathsf{in}\ p\ \mathsf{end}\ \mathsf{end}$

- If t is a closed term, ttftpae:

  $\mathbf{skip} \qquad \mathsf{let}\ x = t\ \mathsf{in}\ \mathbf{skip}\ \mathsf{end}$

- If x does not occur in $p;r$, ttftpae:

  $p; \mathsf{let}\ x = t\ \mathsf{in}\ q\ \mathsf{end}; r \qquad \mathsf{let}\ x = t\ \mathsf{in}\ p; q; r\ \mathsf{end}$

- If x does not occur in p and t is closed, ttftpae:

  $p + \mathsf{let}\ x = t\ \mathsf{in}\ q\ \mathsf{end} \qquad \mathsf{let}\ x = t\ \mathsf{in}\ p + q\ \mathsf{end}$

Axioms & Properties

- If x does not occur in t, ttftpae for any closed a:

$$(\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end})^* \qquad \mathsf{let}\ x = a\ \mathsf{in}\ (x := t; p)^*\ \mathsf{end}$$

- If x does not occur in t and a is closed, ttftpae:

$$\mathsf{let}\ x = t\ \mathsf{in}\ p\ \mathsf{end} \qquad \mathsf{let}\ x = a\ \mathsf{in}\ x := t; p\ \mathsf{end}$$

- If x does not occur in t, ttftpae:

$$\mathsf{let}\ x = s\ \mathsf{in}\ p\ \mathsf{end}; x := t \qquad x := s; p; x := t$$

- The axioms are sound with respect to the binary relation semantics

Axioms & Properties

- For any permutation π : *{1,...,n}*➞*{1,...,n}*, tftpae:

$$\text{let } x_1 = t_1, \ldots, x_n = t_n \text{ in } p \text{ end}$$
$$\text{let } x_{\pi(1)} = t_{\pi(1)}, \ldots, x_{\pi(n)} = t_{\pi(n)} \text{ in } p \text{ end}$$

- If x does not occur in p, and if t is a closed term, ttftpae:

$$p \qquad\qquad \text{let } x = t \text{ in } p \text{ end}$$

Flattening (Globalization)

1. Apply α-conversion to both programs to make bound variables unique
2. Let $x_1,\ldots,x_n$ be all bound variables in either program. Use axioms to get both programs to the form

$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ p\ \mathsf{end}$$
$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ q\ \mathsf{end}$$

where a is a closed term; p and q have no scoping expressions

Equivalence

For p,q with no scoping and a a closed term, the two programs

$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ p\ \mathsf{end}$$
$$\mathsf{let}\ x_1 = a, \ldots, x_n = a\ \mathsf{in}\ q\ \mathsf{end}$$

if and only if the two programs

$$x_1 := a; \cdots ; x_n := a; p; x_1 := a; \cdots ; x_n := a$$
$$x_1 := a; \cdots ; x_n := a; q; x_1 := a; \cdots ; x_n := a$$

are equivalent w.r.t. the “flat” binary relation semantics.

Swap Example

```
let t = x
in  x := y;
    y := t
end
```

=

```
x := x + y;
y := x - y;
x := x - y
```

Swap Example

```
let t = x
in  x := y;
    y := t
end
```

=

```
let t = a
in x := x + y;
   y := x - y;
   x := x - y
end
```

Swap Example

```
let t = a
in  t := x;
    x := y;
    y := t
end
```

=

```
let t = a
in x := x + y;
   y := x - y;
   x := x - y
end
```

Swap Example

Suffices to show

```
t := a;                 t := a;
t := x;                 x := x + y;
x := y;        =        y := x - y;
y := t;                 x := x - y;
t := a                  t := a
```

using

$$
\begin{array}{lcll}
x := s; y := t & = & y := t[x/s]; x := s & (y \notin FV(s)) \\
x := s; y := t & = & x := s; y := t[x/s] & (x \notin FV(s)) \\
x := s; x := t & = & x := t[x/s] &
\end{array}
$$

Swap Example

Suffices to show

```
t := a;                 t := a;
t := x;                 x := x + y;
x := y;        =        y := x - y;
y := t;                 x := x - y;
t := a                  t := y;
                        t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$
$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$
$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;
t := x;
x := y;
y := t;
t := a
```

=

```
t := a;
x := x + y;
y := x - y;
t := y;
x := x - t;
t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$

$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$

$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;
t := x;
x := y;
y := t;
t := a
```

=

```
t := a;
x := x + y;
t := x - y;
y := x - y;
x := x - t;
t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$

$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$

$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;          t := a;
t := x;          t := x;
x := y;     =    x := x + y;
y := t;          y := x - y;
t := a           x := x - t;
                 t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$
$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$
$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;
t := x;
x := y;
y := t;
t := a
```

=

```
t := a;
t := x;
x := t + y;
y := x - y;
x := x - t;
t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$

$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$

$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;
t := x;
x := y;
y := t;
t := a
```

=

```
t := a;
t := x;
x := t + y;
y := t;
x := x - t;
t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$

$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$

$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;          t := a;
t := x;          t := x;
x := y;    =     x := t + y;
y := t;          x := x - t;
t := a           y := t;
                 t := a
```

using

$$x := s; y := t \quad = \quad y := t[x/s]; x := s \quad (y \notin FV(s))$$

$$x := s; y := t \quad = \quad x := s; y := t[x/s] \quad (x \notin FV(s))$$

$$x := s; x := t \quad = \quad x := t[x/s]$$

Swap Example

Suffices to show

```
t := a;                    t := a;
t := x;                    t := x;
x := y;          =         x := y;
y := t;                    y := t;
t := a                     t := a
```

using

$$
\begin{array}{lcll}
x := s; y := t & = & y := t[x/s]; x := s & (y \notin FV(s)) \\
x := s; y := t & = & x := s; y := t[x/s] & (x \notin FV(s)) \\
x := s; x := t & = & x := t[x/s] &
\end{array}
$$

Higher-Order Functions

- Provide relational semantics for language with first-class functions
- Functions & data are not conflated
- Two kinds of expressions:
  - *Value expressions*: binary relations between execution states and values
  - *Program expressions*: binary relations between execution states and execution states

Value Expression (V.E.)

- A variable
- A constant or function symbol in $\mathfrak{A}$
- A λ-term $\lambda x.p$ with variable x, program expression p
- A λ-term $\lambda x.p; e$ with variable x, program expression p, v.e. e
- Application $P(d)$ with procedural expression with non-void return type P and v.e. d

Program Expression (P.E.)

- Assignment $x := d$ with variable x and v.e. d
- Test $R(d)$ relation symbol of $\mathfrak{A}$ R and v.e. d
- Nondeterministic choice $p + q$ with p.e.s p and q
- Sequential composition $p;q$ with p.e.s p and q
- Iteration p^* with p.e. p
- Application $P(d)$ with procedural expression P with void or non-void return type, v.e. d

Closure Structures

- Generalize execution state to pointed tree structure
- Motivated by operational semantics of ML, Scheme
- *Closure structure*: pair $\sigma = (T,\alpha)$ with tree of bindings T of the form $x = c$ and α, a pointer into T
- α is the *active pointer*, $\mathsf{active}(\sigma)$, points to *active environment*
- cs are elements and functions of $\mathfrak{A}$ or pair of λ-term and pointer into T

Value Expression Semantics

$$\begin{aligned}
[x] &= \{(\sigma, \sigma(x)) \mid \sigma \in \mathsf{CS},\ \sigma(x) \text{ is defined}\} \\
[f] &= \{(\sigma, f^{\mathfrak{A}}) \mid \sigma \in \mathsf{CS}\} \\
[\lambda x.p] &= \{(\sigma, (\lambda x.p, \mathsf{active}(\sigma))) \mid \sigma \in \mathsf{CS}\} \\
[\lambda x.p; e] &= \{(\sigma, (\lambda x.p; e, \mathsf{active}(\sigma))) \mid \sigma \in \mathsf{CS}\} \\
[f(d)] &= \{(\sigma, f^{\mathfrak{A}}(c)) \mid (\sigma, c) \in [d]\} \\
[P(d)] &= \{((T,\alpha), b) \ \mid \ ((T,\alpha), c) \in [d], \\
&\qquad ((T,\alpha), (\lambda x.p; e,\ \beta)) \in [P], \\
&\qquad ((x = c) :: (T,\beta),\ b) \in [\![p]\!] \circ [e]\} \\
&\quad \cup\ \{((T,\alpha), f(c)) \ \mid \ ((T,\alpha), c) \in [d], \\
&\qquad ((T,\alpha), f) \in [P]\}
\end{aligned}$$

Program Expression Semantics

$$
\begin{aligned}
\llbracket x := d \rrbracket &= \{(\sigma, \sigma[x/a]) \mid (\sigma, a) \in [d],\ \sigma(x) \text{ is defined}\} \\
\llbracket R(d) \rrbracket &= \{(\sigma, \sigma) \mid (\sigma, a) \in [d] \text{ and } R^{\mathfrak{A}}(a)\} \\
\llbracket p + q \rrbracket &= \llbracket p \rrbracket \cup \llbracket q \rrbracket \\
\llbracket p\ ;\ q \rrbracket &= \llbracket p \rrbracket \circ \llbracket q \rrbracket \\
\llbracket p^* \rrbracket &= \textstyle\bigcup_n \llbracket p \rrbracket^n \\
\llbracket P(d) \rrbracket &= \{((T, \alpha), (S, \alpha)) \mid \begin{array}[t]{l} ((T, \alpha), c) \in [d], \\ ((T, \alpha), (\lambda x.p, \beta)) \in [P], \\ ((x = c) :: (T, \beta), (x = d) :: (S, \beta)) \in \llbracket p \rrbracket\}. \end{array}
\end{aligned}
$$

Contexts

- A *context C[-]*: program expression with distinguished free program variable
- Relational semantics is fully abstract
- Context is not necessary for equivalence arguments
- Theorem: For program expressions *p* and *q* and all contexts *C[-]*,

$$[\![C[p]]\!] = [\![C[q]]\!] \Leftrightarrow [\![p]\!] = [\![q]\!]$$

Conclusions

- Relational semantics capture contextual information
- No need to reason at contextual level
- Ability to prove equivalence of programs with local variables using KAT
- Relational semantics for first-class functions