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
}

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 $\mathcal{A}$ over signature $\Sigma$

- **Partial Valuation** $f : \text{Var} \to |\mathcal{A}|$

- **Domain of $f$**: $\text{dom } f$

- **Environment**: Stack of partial valuations $\sigma$, $\tau$
  
  - $f :: \sigma$: environment with head $f$ and tail $\sigma$
  
  - **Shape of $f_1 :: \cdots :: f_n$**: $\bigcup_{i=1}^{n} \text{dom } f_i$
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

```
let x_1 = t_1, \ldots, x_n = t_n in p 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

\[
\begin{align*}
f[x/a](y) = & \begin{cases} 
  f(y), & \text{if } y \in \text{dom } f \text{ and } y \neq x, \\
  a, & \text{if } y \in \text{dom } f \text{ and } y = x, \\
  \text{undefined}, & \text{if } y \notin \text{dom } f.
\end{cases} \\
\sigma(x) = & \begin{cases} 
  f(x), & \text{if } \sigma = f :: \tau \text{ and } x \in \text{dom } f, \\
  \tau(x), & \text{if } \sigma = f :: \tau \text{ and } x \notin \text{dom } f, \\
  \text{undefined}, & \text{if } \sigma = \epsilon.
\end{cases} \\
\sigma[x/a] = & \begin{cases} 
  f[x/a] :: \tau, & \text{if } \sigma = f :: \tau \text{ and } x \in \text{dom } f, \\
  f :: \tau[x/a], & \text{if } \sigma = f :: \tau \text{ and } x \notin \text{dom } f, \\
  \epsilon, & \text{if } \sigma = \epsilon.
\end{cases}
\end{align*}
\]
Evaluating a Program

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

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

- \( f \) 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

\[
\begin{align*}
\llbracket R(t_1, \ldots, t_n) \rrbracket &= \{ (\sigma, \sigma) \mid \sigma(t_i) \text{ is defined, } 1 \leq i \leq n, \text{ and } \mathcal{A}, \sigma \models R(t_1, \ldots, t_n) \} \\
\llbracket \neg R(t_1, \ldots, t_n) \rrbracket &= \{ (\sigma, \sigma) \mid \sigma(t_i) \text{ is defined, } 1 \leq i \leq n, \text{ and } \mathcal{A}, \sigma \models \neg R(t_1, \ldots, t_n) \}
\end{align*}
\]

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

\[
\llbracket \text{undefined}(x) \rrbracket = \{ (\sigma, \sigma) \mid \sigma(x) \text{ is undefined} \}.
\]
Short-Circuit Boolean Ops

\[
\begin{align*}
[\varphi \&\& \psi] &= [\varphi] \cap [\psi] \\
[\varphi \| \psi] &= [\varphi] \cup ([\neg \varphi] \cap [\psi]) \\
[\neg(\varphi \&\& \psi)] &= [\neg \varphi] \cup ([\varphi] \cap [\neg \psi]) = [\neg \varphi \| \neg \psi] \\
[\neg(\varphi \| \psi)] &= [\neg \varphi] \cap [\neg \psi] = [\neg \varphi \&\& \neg \psi] \\
[\neg\neg \varphi] &= [\varphi]
\end{align*}
\]
Example

\[
\text{let } x=1 \text{ in } x:=y+z; \\
\text{let } y=x+2 \text{ in } y:=y+z; \ z:=y+1 \text{ end;} \\
y:=x \\
\text{end}
\]

(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

\[
\text{let } x=1 \text{ in } x:=y+z; \\
\text{let } y=x+2 \text{ in } y:=y+z; \ z:=y+1 \text{ end;} \\
y:=x \text{ end}
\]

\[
(y = 5, z = 20) \\
(x = 1) :: (y = 5, z = 20) \\
(x = 25) :: (y = 5, z = 20)
\]
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

```plaintext
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:

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

- If $y$ does not occur in $s$, $\text{ttftpae}$:

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

- If $x$ does not occur in $s$, $\text{ttftpae}$:

  \[
  \begin{align*}
  \text{let } x = s \text{ in let } y = t \text{ in } p \text{ end end} \\
  \text{let } x = s \text{ in let } y = t[x/s] \text{ in } p \text{ end end}
  \end{align*}
  \]
Axioms & Properties

- If $x_1$ does not occur in $t_2, \ldots, t_n$, ttftpae:
  \[
  \text{let } x_1 = t_1, \ldots, x_n = t_n \text{ in } p \text{ end}
  \]
  \[
  \text{let } x_1 = t_1 \text{ in let } x_2 = t_2, \ldots, x_n = t_n \text{ in } p \text{ end end}
  \]

- If $t$ is a closed term, ttftpae:
  \[
  \text{skip let } x = t \text{ in skip end}
  \]

- If $x$ does not occur in $p; r$, ttftpae:
  \[
  p; \text{let } x = t \text{ in } q \text{ end}; r \quad \text{let } x = t \text{ in } p; q; r \text{ end}
  \]

- If $x$ does not occur in $p$ and $t$ is closed, ttftpae:
  \[
  p + \text{let } x = t \text{ in } q \text{ end} \quad \text{let } x = t \text{ in } p + q \text{ end}
  \]
Axioms & Properties

- If $x$ does not occur in $t$, $t t f t p a e$ for any closed $a$:
  
  $(\text{let } x = t \text{ in } p \text{ end})^* \quad \text{let } x = a \text{ in } (x := t; p)^* \text{ end}$

- If $x$ does not occur in $t$ and $a$ is closed, $t t f t p a e$:
  
  $\text{let } x = t \text{ in } p \text{ end} \quad \text{let } x = a \text{ in } x := t; p \text{ end}$

- If $x$ does not occur in $t$, $t t f t p a e$:
  
  $\text{let } x = s \text{ in } p \text{ end}; x := t \quad x := s; p; x := t$

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

- For any permutation $\pi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$, tftpae:
  
  \[
  \begin{align*}
  &\text{let } x_1 = t_1, \ldots, x_n = t_n \text{ in } p \text{ end} \\
  &\text{let } x_{\pi(1)} = t_{\pi(1)}, \ldots, x_{\pi(n)} = t_{\pi(n)} \text{ in } p \text{ end}
  \end{align*}
  \]

- If $x$ does not occur in $p$, and if $t$ is a closed term, ttftpae:
  
  \[
  p \quad \text{let } x = t \text{ in } p \text{ end}
  \]
Flattening (Globalization)

1. Apply $\alpha$-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

   \[
   \begin{align*}
   &\text{let } x_1 = a, \ldots, x_n = a \text{ in } p \text{ end} \\
   &\text{let } x_1 = a, \ldots, x_n = a \text{ in } q \text{ end}
   \end{align*}
   \]

   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

\[
\text{let } x_1 = a, \ldots, x_n = a \text{ in } p \text{ end} \\
\text{let } x_1 = a, \ldots, x_n = a \text{ in } q \text{ end}
\]

if and only if the two programs

\[
\begin{align*}
x_1 &:= a; \cdots; x_n := a; p; x_1 := a; \cdots; x_n := a \\
x_1 &:= a; \cdots; x_n := a; q; x_1 := a; \cdots; x_n := a
\end{align*}
\]

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

\[
\text{let } t = a \\
\text{in } t := x; \\
\quad x := y; \\
\quad y := t \\
\text{end}
\]

\[
\text{let } t = a \\
\text{in } x := x + y; \\
\quad y := x - y; \\
\quad x := x - y \\
\text{end}
\]
Swap Example

Suffices to show

\[
\begin{align*}
t & := a; \\
t & := x; \\
x & := y; \\
y & := t; \\
t & := a \\
\end{align*}
\]

\[
\begin{align*}
t & := a; \\
x & := x + y; \\
y & := x - y; \\
x & := x - y; \\
t & := a
\end{align*}
\]

using

\[
\begin{align*}
x := s; y := t & = y := t[x/s]; x := s \quad (y \notin FV(s)) \\
x := s; y := t & = x := s; y := t[x/s] \quad (x \notin FV(s)) \\
x := s; x := t & = x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
t &:= a; \\
t &:= x; \\
x &:= y; \\
y &:= t; \\
t &:= a
\end{align*}
\]

using

\[
\begin{align*}
x := s; y := t & = y := t[x/s]; x := s & \quad (y \notin FV(s)) \\
x := s; y := t & = x := s; y := t[x/s] & \quad (x \notin FV(s)) \\
x := s; x := t & = x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
t &:= a; \\
t &:= x; \\
x &:= y; \\
y &:= t; \\
t &:= a
\end{align*}
\]

using

\[
\begin{align*}
x := s; y := t &= y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
x := s; y := t &= x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
x := s; x := t &= x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
t & := a; \\
t & := x; \\
x & := y; \\
y & := t; \\
t & := a
\end{align*}
\]

using

\[
\begin{align*}
x := s; y := t & \quad = \quad y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
x := s; y := t & \quad = \quad x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
x := s; x := t & \quad = \quad x := t[x/s]
\end{align*}
\]

\[
\begin{align*}
t & := a; \\
x & := x + y; \\
t & := x - y; \\
y & := x - y; \\
x & := x - t; \\
t & := a
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
    t & := a; \\
    t & := x; \\
    x & := y; \\
    y & := t; \\
    t & := a
\end{align*}
\]

\[
\begin{align*}
    t & := a; \\
    t & := x; \\
    x & := x + y; \\
    y & := x - y; \\
    x & := x - t; \\
    t & := a
\end{align*}
\]

using

\[
\begin{align*}
    x := s; y := t & = y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
    x := s; y := t & = x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
    x := s; x := t & = x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
t & := a; \\
t & := x; \\
x & := y; \\
y & := t; \\
t & := a
\end{align*}
\]

\[
\begin{align*}
t & := a; \\
t & := x; \\
x & := t + y; \\
y & := x - y; \\
x & := x - t; \\
t & := a
\end{align*}
\]

using

\[
\begin{align*}
x := s; y := t & = y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
x := s; y := t & = x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
x := s; x := t & = x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show
\[
\begin{align*}
  t & := a; \\
  t & := x; \\
  x & := y; \\
  y & := t; \\
  t & := a
\end{align*}
\]

using
\[
\begin{align*}
  x := s; y := t & = y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
  x := s; y := t & = x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
  x := s; x := t & = x := t[x/s]
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
  t &:= a; \\
t &:= x; \\
x &:= y; \\
y &:= t; \\
t &:= a \\
\end{align*}
\]

\[
\begin{align*}
  t &:= a; \\
t &:= x; \\
x &:= t + y; \\
y &:= x - t; \\
t &:= a \\
\end{align*}
\]

using

\[
\begin{align*}
  x := s; y := t & = y := t[x/s]; x := s \quad (y \not\in FV(s)) \\
  x := s; y := t & = x := s; y := t[x/s] \quad (x \not\in FV(s)) \\
  x := s; x := t & = x := t[x/s] \\
\end{align*}
\]
Swap Example

Suffices to show

\[
\begin{align*}
\text{t} & := \text{a}; \\
\text{t} & := \text{x}; \\
\text{x} & := \text{y}; \\
\text{y} & := \text{t}; \\
\text{t} & := \text{a}
\end{align*}
\]

using

\[
\begin{align*}
\text{x} & := \text{s}; \text{y} := \text{t} & = & \text{y} := \text{t}[\text{x}/\text{s}]; \text{x} := \text{s} & (\text{y} \not\in \text{FV}(\text{s})) \\
\text{x} & := \text{s}; \text{y} := \text{t} & = & \text{x} := \text{s}; \text{y} := \text{t}[\text{x}/\text{s}] & (\text{x} \not\in \text{FV}(\text{s})) \\
\text{x} & := \text{s}; \text{x} := \text{t} & = & \text{x} := \text{t}[\text{x}/\text{s}]
\end{align*}
\]
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 $\mathcal{A}$
- A $\lambda$-term $\lambda x. p$ with variable $x$, program expression $p$
- A $\lambda$-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 $\mathcal{A} R$ 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 $\alpha$, a pointer into $T$
- $\alpha$ is the *active pointer*, $\text{active}(\sigma)$, points to *active environment*
- $cs$ are elements and functions of $\mathfrak{A}$ or pair of $\lambda$-term and pointer into $T$
Value Expression Semantics

\[ [x] = \{ (\sigma, \sigma(x)) \mid \sigma \in CS, \sigma(x) \text{ is defined} \} \]

\[ [f] = \{ (\sigma, f^\mathcal{A}) \mid \sigma \in CS \} \]

\[ [\lambda x.p] = \{ (\sigma, (\lambda x.p, \text{active}(\sigma))) \mid \sigma \in CS \} \]

\[ [\lambda x.p; e] = \{ (\sigma, (\lambda x.p; e, \text{active}(\sigma))) \mid \sigma \in CS \} \]

\[ [f(d)] = \{ (\sigma, f^\mathcal{A}(c)) \mid (\sigma, c) \in [d] \} \]

\[ [P(d)] = \{ (((T, \alpha), b) \mid (((T, \alpha), c) \in [d],
\quad ((T, \alpha), (\lambda x.p; e, \beta)) \in [P],
\quad ((x = c) :: (T, \beta), b) \in [p] \circ [e])
\quad \cup \{ (((T, \alpha), f(c)) \mid (((T, \alpha), c) \in [d],
\quad ((T, \alpha), f) \in [P]) \}
\} \]
Program Expression Semantics

\[[x := d]\] = \{ (\sigma, \sigma[x/a]) \mid (\sigma, a) \in \lbrack d \rbrack, \sigma(x) \text{ is defined} \}

\[[R(d)]\] = \{ (\sigma, \sigma) \mid (\sigma, a) \in \lbrack d \rbrack \text{ and } R^a(a) \}

\[[p + q]\] = \[[p]\] \cup \[[q]\]

\[[p ; q]\] = \[[p]\] \circ \[[q]\]

\[[p^*]\] = \bigcup_n \[[p]\]^n

\[[P(d)]\] = \{ (((T, \alpha), (S, \alpha)) \mid ((T, \alpha), c) \in \lbrack d \rbrack, (T, \alpha), (\lambda x.p, \beta)) \in \lbrack P \rbrack, ((x = c) :: (T, \beta), (x = d) :: (S, \beta)) \in \lbrack p \rbrack \}.
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 $C[-]$,

  $$\sem{C[p]} = \sem{C[q]} \iff \sem{p} = \sem{q}$$
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