Operational Semantics of μScheme

Bill Harrison

Computer Science Department, University of Missouri

CS4450: Principles of Programming Languages

Today

- Today we begin looking at operational semantics for μScheme
- An *operational semantics* for a language gives a precise account of what it means to execute programs in that language
- It is one common means of defining precisely what a language means
- μScheme contains ImpCore: every ImpCore program is a μScheme program

Abstract Syntax for μScheme

```
TopLevel
   = EXP (Exp)
   | DEFINE (Name, Lambda)
   | VAL (Name, Exp)
   | USE (Name)

Value
   = NIL
   | BOOL
   | NUM
   | SYM
   | PAIR (Value,Value)
   | CLOSURE (Lambda,Env)
   | PRIMITIVE(Name)
```

```
Exp = LITERAL (Value)
    | VAR (Name)
    | SET (Name, Exp)
    | IF (Exp, Exp, Exp)
    | WHILE (Exp, Exp)
    | BEGIN (Explist)
    | LET (LetX,Namelist,Explist,Exp)
    | Lambda
    | APPLY (Exp, Explist)

LetX = LET | LETSTAR | LETREC

Lambda = (Namelist,Exp)
```

ImpCore vs. μScheme

The operational semantics for μScheme has a different structure

ImpCore vs. μScheme

The operational semantics for μScheme has a different structure

- Environments:
  - Impcore: three environments for globals, function defs & args
  - μScheme: all names bound in one environment

ImpCore vs. μScheme

The operational semantics for μScheme has a different structure

- Environments:
  - Impcore: three environments for globals, function defs & args
  - μScheme: all names bound in one environment
- Locations and Explicit Store
  - Impcore: names bound to *values*
  - μScheme: environments bind *names* to *locations*; store σ binds *locations* to *values*

ImpCore vs. μScheme

The operational semantics for μScheme has a different structure

- Environments:
  - Impcore: three environments for globals, function defs & args
  - μScheme: all names bound in one environment
- Locations and Explicit Store
  - Impcore: names bound to *values*
  - μScheme: environments bind *names* to *locations*; store σ binds *locations* to *values*
- Functions
  - Impcore: only first-order
  - μScheme: higher-order (functions are values). *Closures* are used to model functions.

States & Judgments

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store
- I.e., for any store σ, can always pick a **fresh** location, $l \notin dom(\sigma)$

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store
- I.e., for any store σ, can always pick a **fresh** location, $l \notin dom(\sigma)$
- Store σ is a map $Loc \rightarrow Value$ (*Value* as defined in R&K!)

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store
- I.e., for any store σ, can always pick a **fresh** location, $l \notin dom(\sigma)$
- Store σ is a map $Loc \rightarrow Value$ ($Value$ as defined in R&K!)
- Synonyms: state = store = heap

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store
- I.e., for any store σ, can always pick a **fresh** location, $l \notin dom(\sigma)$
- Store σ is a map $Loc \rightarrow Value$ ($Value$ as defined in R&K!)
- Synonyms: state = store = heap
- Environment ρ is a map $Name \rightarrow Loc$

States & Judgments

The state of an abstract machine evaluating e is: $\langle e, \rho, \sigma \rangle$

A judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

- evaluating e in ρ and σ produces the value v
- ...and results in a (possibly) changed store σ'
- ...but never in a changed environment ρ' (why?)

Remarks

- Assume ∞ number of locations in store
- I.e., for any store σ, can always pick a **fresh** location, $l \notin dom(\sigma)$
- Store σ is a map $Loc \rightarrow Value$ ($Value$ as defined in R&K!)
- Synonyms: state $=$ store $=$ heap
- Environment ρ is a map $Name \rightarrow Loc$
- Current value $\mathtt{x}$ is $\sigma(\rho(\mathtt{x}))$

Variables & Functions

- Variables refer to locations

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ
  - to look up/change x, read/write location l.

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ
  - to look up/change x, read/write location l.

- New bindings of source variables mean are stored in the heap

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ
  - to look up/change x, read/write location l.

- New bindings of source variables mean are stored in the heap
  - If a variable x is given a new binding to value v in the source program,

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ
  - to look up/change x, read/write location l.

- New bindings of source variables mean are stored in the heap
  - If a variable x is given a new binding to value v in the source program,
  - ...then a fresh heap location $l \notin \sigma$ is allocated, and

Variables & Functions

- Variables refer to locations
  - $\rho(x) = l$ means x is stored at location l
  - contents of x stored at l in σ
  - to look up/change x, read/write location l.

- New bindings of source variables mean are stored in the heap
  - If a variable x is given a new binding to value v in the source program,
  - ...then a fresh heap location $l \notin \sigma$ is allocated, and
  - ...the fresh location is initialized to v

Where do new bindings come from?

New bindings come from two sources in μScheme:

- Let expressions and function applications:

```
(let ((x 10)) (+ x x))
((lambda (x) (+ x x)) 10)
-- binding of x
-- reference to x
```

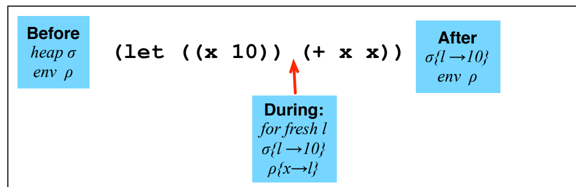
Where do new bindings come from?

New bindings come from two sources in μScheme:

- Let expressions and function applications:

```
(let ((x 10)) (+ x x))
((lambda (x) (+ x x)) 10)
-- binding of x
-- reference to x
```

- Pictorially:



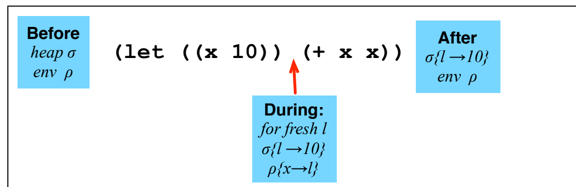
Where do new bindings come from?

New bindings come from two sources in μScheme:

- Let expressions and function applications:

```
(let ((x 10)) (+ x x))
((lambda (x) (+ x x)) 10)
-- binding of x
-- reference to x
```

- Pictorially:



- N.b., ρ only changes inside the let; after, heap contains garbage.

Simplified Let Rule

$$\frac{\begin{array}{c}\langle e,\rho,\sigma\rangle \Downarrow \langle u,\sigma'\rangle \\ l \notin dom(\sigma') \\ \langle e',\rho\{x\mapsto l\},\sigma'\{l\mapsto u\}\rangle \Downarrow \langle v,\sigma''\rangle\end{array}}{\langle (let\ ((x\ \ e))\ \ e'),\rho,\sigma\rangle \Downarrow \langle v,\sigma''\rangle}\ SimpleLet$$

Let Rule

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ l_1, \ldots, l_n \notin dom(\sigma_n) \\ \rho^* = \rho\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\} \\ \sigma^* = \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \\ \langle e, \rho^*, \sigma^* \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle LET(\langle x_1, e1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \; (LET)$$

LETSTAR Rule

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct}, \rho_1 = \rho, \sigma_1 = \sigma \\ \langle e_1, \rho_1, \sigma_1 \rangle \Downarrow \langle v_1, \sigma'_1 \rangle, l_1 \notin dom(\sigma'_1), \\ \rho_2 = \rho_1\{x_1 \mapsto l_1\}, \sigma_2 = \sigma'_1\{l_1 \mapsto v_1\} \\ \vdots \\ \rho_n = \rho_{n-1}\{x_{n-1} \mapsto l_{n-1}\}, \sigma_n = \sigma'_{n-1}\{l_{n-1} \mapsto v_{n-1}\} \\ \langle e_n, \rho_n, \sigma_n \rangle \Downarrow \langle v_n, \sigma'_n \rangle, l_n \notin dom(\sigma'_n), \\ \langle e, \rho_n, \sigma'_n\{l_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle LETSTAR(\langle x_1, e1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \;(LETSTAR)$$