

Mixin Modules and Computational Effects

Davide Ancona and **Sonia Fagorzi** and Eugenio Moggi and Elena Zucca
{davide, fagorzi, moggi, zucca}@disi.unige.it

Dipartimento di Informatica e Scienze dell'Informazione (DISI)
University of Genova

To appear in ICALP 2003

Aim

Aim

To study the interaction between the notions of mixin (modules) and computational effects

Aim

To study the interaction between the notions of mixin (modules) and computational effects

- All formalizations of mixin systems defined so far ([\[Bra92\]](#), [\[DS96@ICFP\]](#), [\[AZ99@PPDP\]](#) [\[AZ02@JFP\]](#) [\[MT00@ESOP\]](#) [\[WV00@ESOP\]](#)) only model mixins in purely functional settings

Aim

To study the interaction between the notions of mixin (modules) and computational effects

- All formalizations of mixin systems defined so far ([\[Bra92\]](#), [\[DS96@ICFP\]](#), [\[AZ99@PPDP\]](#) [\[AZ02@JFP\]](#) [\[MT00@ESOP\]](#) [\[WV00@ESOP\]](#)) only model mixins in purely functional settings
- Our proposal is a (foundational) **monadic mixin calculus**, called CMS_{do} , obtained by combining:
 - the purely functional mixin calculus CMS [\[AZ99@PPDP,AZ02@JFP\]](#)
 - a monadic metalanguage [\[MF03@FOSSACS\]](#) equipped with a Haskell-like recursive monadic binding [\[EL00@ICFP,EL02@HASKELL\]](#)

Aim

To study the interaction between the notions of mixin (modules) and computational effects

- All formalizations of mixin systems defined so far ([\[Bra92\]](#), [\[DS96@ICFP\]](#), [\[AZ99@PPDP\]](#) [\[AZ02@JFP\]](#) [\[MT00@ESOP\]](#) [\[WV00@ESOP\]](#)) only model mixins in purely functional settings
- Our proposal is a (foundational) **monadic mixin calculus**, called CMS_{do} , obtained by combining:
 - **the purely functional mixin calculus CMS** [\[AZ99@PPDP,AZ02@JFP\]](#)
 - **a monadic metalanguage** [\[MF03@FOSSACS\]](#) equipped with a Haskell-like recursive monadic binding [\[EL00@ICFP,EL02@HASKELL\]](#)

The semantics of the monadic language consists of two parts:

- **local** (semantics preserving) **simplification rules**
- **computation steps** able to modify the store

This has simplified the integration since **CMS rules** are **all local**

Summary

Summary

- An informal overview of the *CMS* calculus

Summary

- An informal overview of the *CMS* calculus
- Mixin modules and imperative features: semantic issues

Summary

- An informal overview of the *CMS* calculus
- Mixin modules and imperative features: semantic issues
- CMS_{do} : an informal overview of the novelties with respect to *CMS*
 - Example 1: basic modules and doall operation
 - Examples 2 and 3: mutual recursion

Summary

- An informal overview of the *CMS* calculus
- Mixin modules and imperative features: semantic issues
- CMS_{do} : an informal overview of the novelties with respect to *CMS*
 - Example 1: basic modules and doall operation
 - Examples 2 and 3: mutual recursion
- Future work and conclusion

An informal overview of the CMS calculus

An informal overview of the CMS calculus

A CMS basic mixin module

Example

```
M1 = mix      import N2 as x      (* deferred *)
              export N1 = e1[x,y] (* defined *)
              local y = e2[x,y]   (* local *)
              end
```

- It consists of **defined** and **local** components, bound to an expression, and **deferred** components, declared but not yet defined
- Separation between **component names** and **variables**

An informal overview of the CMS calculus

A CMS basic mixin module

Example

```
M1 = mix      import N2 as x      (* deferred *)
              export N1 = e1[x,y] (* defined *)
              local y = e2[x,y]   (* local *)
              end
```

- It consists of **defined** and **local** components, bound to an expression, and **deferred** components, declared but not yet defined
- Separation between **component names** and **variables**

CMS provides three operations on mixins: **sum**, **freeze** and **delete**

[Back to the Summary](#)

Operations: sum

Operations: sum

The **sum** operator: performs the union of the deferred components, and the disjoint union of the defined and local components of the two mixins

Example:

```
M1 = mix  import N2 as x
         export N1 = e1[x,y]
         local y = e2[x,y]
         end
```

```
M2 = mix  import N1 as x
         export N2 = e3[x,y]
         local y = e4[x,y]
         end
```

$M3 = M1 + M2$

Operations: sum

The **sum** operator: performs the union of the deferred components, and the disjoint union of the defined and local components of the two mixins

Example:

```
M1 = mix   import N2 as x
          export N1 = e1[x,y]
          local y = e2[x,y]
          end
```

```
M2 = mix   import N1 as x
          export N2 = e3[x,y]
          local y = e4[x,y]
          end
```

$M3 = M1 + M2$ simplifies to

```
mix   import N2 as x1, N1 as x2
      export N1 = e1[x1,y1]
      local y1 = e2[x1,y1]
      export N2 = e3[x2,y2]
      local y2 = e4[x2,y2]
      end
```

The sum operation supports **mutual dependencies** and **cross-module recursion**

Operations: freeze and delete

Operations: freeze and delete

The **freeze** operator: connects deferred and defined components having the same name inside a mixin

Example:

(mix	import N as x		mix	local x = e1[x,y]
	export N = e1[x,y]	simplifies to		export N = x
	local y = e2[x,y]			local y = e2[x,y]
end) ! N			end	

Operations: freeze and delete

The **freeze** operator: connects deferred and defined components having the same name inside a mixin

Example:

(mix	import N as x		mix	local x = e1[x,y]
	export N = e1[x,y]	simplifies to		export N = x
	local y = e2[x,y]			local y = e2[x,y]
end) ! N			end	

The **delete** operator: is used for hiding defined components

Example:

(mix	import N as x		mix	import N as x
	export N = e1[x,y]	simplifies to		local y = e2[x,y]
	local y = e2[x,y]			
end) \ N			end	

[Back to the Summary](#)

Mixin modules and imperative features: semantic issues

Mixin modules and imperative features: semantic issues

In the presence of primitives for store manipulation, expressions inside mixins can have [side-effects](#).

This raises the following [issues](#):

- because of side-effects, the [evaluation order](#) of components inside a mixin must be [deterministic](#), while still retaining cross-module-recursion
- [when](#) computations inside a mixin must be evaluated and [how many times](#)?

[Back to the Summary](#)

Example 1: basic modules and doall operation

Example 1: basic modules and doall operation

CM1 =

```
mix    local    l ← new(x-1),                (* computational *)
        x = 1
        export  Inc = mdo v ← get(l) in set(l,v+1),
        Val ← get(l)                        (* computational *)
end
```

Example 1: basic modules and doall operation

CM1 =

```
mix    local    l ← new(x-1),                (* computational *)
        x = 1
        export  Inc = mdo v ← get(l) in set(l,v+1),
        Val ← get(l)                          (* computational *)
end
```

Additions to *CMS*:

- $\tau \in \mathbf{T} ::= \dots \mid M\tau \mid \text{ref}\tau \mid \{\Pi\} \mid [\Pi; \Pi']$
- primitives on the store
- the monadic constructs `mdo` (recursive do) and `ret`
- a new kind of mixin component called **computational**, of the form $x \leftarrow e$ (e with monadic type)
- `doall`: $[\emptyset; \Pi] \rightarrow M\{\Pi\}$

Example 1: basic modules and doall operation

CM1 =

```
mix    local    l ← new(x-1),                (* computational *)
        x = 1
        export  Inc = mdo v ← get(l) in set(l,v+1),
        Val ← get(l)                          (* computational *)
end
```

Additions to *CMS*:

- $\tau \in T ::= \dots \mid M\tau \mid \text{ref}\tau \mid \{\Pi\} \mid [\Pi; \Pi']$
- primitives on the store
- the monadic constructs `mdo` (recursive do) and `ret`
- a new kind of mixin component called **computational**, of the form $x \leftarrow e$ (e with monadic type)
- `doall`: $[\emptyset; \Pi] \rightarrow M\{\Pi\}$

Mutual recursion has the following informal semantics: if $i \leq j$, then e_i can depend on the variable x_j , provided that the computation e_i can be successfully performed without knowing the value of e_j

Example 1: basic modules and doall operation

```
CM1 =  
  mix    local    l ← new(x-1), (* computational *)  
          x = 1  
  export Inc = mdo v ← get(l) in set(l,v+1),  
          Val ← get(l) (* computational *)  
end
```

A new mixin operation **doall** to transform a mixin (without deferred components) into a record (selection is allowed only over record):

- it **evaluates** all computational definitions $x_i \leftarrow e_i$ **in the order they are declared** (order matters)
- it **binds the value** returned by e_i to x_i **immediately**, to make it available to the subsequent computations e_j with $j > i$

```
doall(CM1) evaluates to  $r \triangleq \{$  Inc = mdo v ← get( $l$ ) in set( $l$ ,v+1),  
Val = 0  $\}$ 
```

where l is the location generated by new(x-1)

Example 1: basic modules and doall operation

CM1 =

```
mix    local    l ← new(x-1),                (* computational *)
        x = 1
        export  Inc = mdo v ← get(l) in set(l,v+1),
        Val ← get(l)                          (* computational *)
end
```

$$r \triangleq \{ \text{Inc} = \text{mdo } v \leftarrow \text{get}(l) \text{ in } \text{set}(l, v+1), \\ \text{Val} = 0 \}$$

Inc can be reevaluated several times:

Example

`mdo l ← r.Inc in get(l)` evaluates to 1

[Back to the Summary](#)

Examples 2 and 3: mutual recursion

Examples 2 and 3: mutual recursion

Example 2: computational components can be mutually recursive

```
doall (mix export Loc1=l1, Loc2=l2
      local l1 ← new(l2), l2 ← new(l1)
      end
```

evaluates to

$$\{ \text{Loc1} = l_1, \text{Loc2} = l_2 \}$$

Examples 2 and 3: mutual recursion

Example 2: computational components can be mutually recursive

```
doall (mix export Loc1=l1, Loc2=l2
      local l1 ← new(l2), l2 ← new(l1)
      end)
```

evaluates to

{ Loc1 = l_1 , Loc2 = l_2 }

Example 3: bad recursive declarations

```
doall (mix local x ← set(y, 1)
      local y ← new(0)
      end) evaluates to error
```

[Back to the Summary](#)

Conclusion and future work

Conclusion and future work

Emphasis of the paper is on the operational aspects, we adopt a simple type system

More refined type systems:

Conclusion and future work

Emphasis of the paper is on the operational aspects, we adopt a simple type system

More refined type systems:

- dynamic errors due to bad recursive declarations mentioned above could be detected by introducing a type system similar to that in [HL02@ESOP,Bou02@JFP]

Conclusion and future work

Emphasis of the paper is on the operational aspects, we adopt a simple type system

More refined type systems:

- dynamic errors due to bad recursive declarations mentioned above could be detected by introducing a type system similar to that in [HL02@ESOP,Bou02@JFP]
- a type system distinguishing between modules possibly containing some computational definitions and those with no computational definitions, would allow selection on CMS_{do} mixins without any need to apply `doall(_)`

[Back to the Summary](#)

References

- [AZ99@PPDP] D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, 1999*, number 1702 in Lecture Notes in Computer Science, pages 62–79. Springer Verlag, 1999
- [AZ02@JFP] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, March 2002
- [Bou02@JFP] G. Boudol.. The recursive record semantics of objects revisited. To appear in *Journal of Functional Programming*, 2002
- [Bra92] G. Bracha. The programming language JIGSAW: Mixin, modularity and multiple inheritance. PhD thesis, Department of Computer Science, University of Utah, 1992
- [DS96@ICFP] D. Duggan and C. Sourelis. Mixin modules. Intl Conf. on Functional Programming, pp. 262-273, ACM Press, 1996
- [EL00@ICFP] L. Erkök and J. Launchbury. Recursive monadic bindings. In *Intl. Conf. on Functional Programming 2000*, pages 174–185, 2000

[Back to the Summary](#)

References (cont)

- [EL02@HASKELL] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell Workshop'02*, pages 29–37, 2002
- [HL02@ESOP] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *ESOP 2002 - Programming Languages and Systems*, number 2305 in Lecture Notes in Computer Science, pages 6–20. Springer Verlag, 2002
- [MF03@FOSSACS] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *Proceedings Workshop on Foundations of Software Science and Computation Structures, Warsaw 2003*, LNCS. Springer Verlag, 2003. To appear
- [MT00@ESOP] E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In G. Smolka, editor, *ESOP 2000 - Programming Languages and Systems*, number 1782 in Lecture Notes in Computer Science, pages 260–274, Berlin. Springer Verlag, 2000

References (cont)

- [WV00@ESOP] J.B. Wells and R. Vestergaard. Equational reasoning for linking with first-class primitive modules. In G. Smolka, editor, *ESOP 2000 - Programming Languages and Systems*, number 1782 in Lecture Notes in Computer Science, pages 412–428, Berlin, 2000. Springer Verlag, 2000

[Back to the Summary](#)

Summary of the technical contents

- [Syntax](#)
- [Type system](#)
- [Semantics](#)
 - [Simplification](#)
 - [Computation](#)
- [Technical results](#)

[Back to the First Part Summary](#)

Syntax

Syntax

Syntax

$$\begin{aligned} e \in \text{Exp} ::= & x \mid \{o\} \mid e.X \mid \text{let}(\rho; e) \mid \text{ret}(e) \mid \text{mdo}(\Theta; e) \mid \text{doall}(e) \\ & \mid l \mid \text{new}(e) \mid \text{get}(e) \mid \text{set}(e_1, e_2) \mid e_1 + e_2 \mid \text{freeze}_X(e) \mid e \setminus X \\ & \mid [\iota; \Delta] \quad \text{with } \iota \text{ injective and } \text{dom}(\iota) \cap \text{DV}(\Delta) = \emptyset \\ \Theta ::= & \emptyset \mid \Theta, x \leftarrow e \quad \text{with } x \notin \text{DV}(\Theta) \\ \Delta ::= & \emptyset \mid \Delta, D \quad \text{with } \text{DV}(\Delta) \cap \text{DV}(D) = \text{DN}(\Delta) \cap \text{DN}(D) = \emptyset \\ D ::= & X \triangleleft e \mid x \triangleleft e \quad \text{with } \triangleleft \text{ either } = \text{ or } \leftarrow \end{aligned}$$

where: $X \in \text{Name}$ (component) names

$x \in \text{Var}$ variables

$l \in \text{L}$ locations

$\iota: \text{Var} \xrightarrow{\text{fin}} \text{Name}$ is an input assignment

$o: \text{Name} \xrightarrow{\text{fin}} \text{Exp}$ is an output assignment

$\rho: \text{Var} \xrightarrow{\text{fin}} \text{Exp}$ is a local assignment

[$\text{DV}(_)$, $\text{DN}(_)$, $\text{FV}(_)$ definition for D , Δ and Θ]

[Back to the Summary](#)

Type system

Type system

■ **Types:** $\tau \in \mathsf{T} ::= \dots \mid M\tau \mid \text{ref}\tau \mid \{\Pi\} \mid [\Pi; \Pi']$ where $\Pi: \text{Name} \xrightarrow{\text{fin}} \mathsf{T}$

■ **Judgment:**

$\Gamma \vdash_{\Sigma} e: \tau$ “ e is a well-typed term of type τ in Γ and Σ ”

with $\Gamma: \text{Var} \xrightarrow{\text{fin}} \mathsf{T}$ type assignment

$\Sigma: \text{L} \xrightarrow{\text{fin}} \mathsf{T}$ signature for locations

Type system

■ **Types:** $\tau \in \mathbb{T} ::= \dots \mid M\tau \mid \text{ref}\tau \mid \{\Pi\} \mid [\Pi; \Pi']$ where $\Pi: \text{Name} \xrightarrow{\text{fin}} \mathbb{T}$

■ **Judgment:**

$\Gamma \vdash_{\Sigma} e: \tau$ “ e is a well-typed term of type τ in Γ and Σ ”

with $\Gamma: \text{Var} \xrightarrow{\text{fin}} \mathbb{T}$ type assignment

$\Sigma: \mathbb{L} \xrightarrow{\text{fin}} \mathbb{T}$ signature for locations

■ **Rules:**

$$\text{(var)} \frac{}{\Gamma \vdash_{\Sigma} x: \tau} \quad \Gamma(x) = \tau \quad \text{(ret)} \frac{\Gamma \vdash_{\Sigma} e: \tau}{\Gamma \vdash_{\Sigma} \text{ret}(e): M\tau}$$

$$\text{(mdo)} \frac{\{\Gamma, \Gamma_{\Theta} \vdash_{\Sigma} e: M\tau \mid (x \Leftarrow e) \in \Theta \wedge \tau = \Gamma_{\Theta}(x)\} \quad \Gamma, \Gamma_{\Theta} \vdash_{\Sigma} e': M\tau'}{\Gamma \vdash_{\Sigma} \text{mdo}(\Theta; e'): M\tau'} \quad \text{dom}(\Gamma_{\Theta}) = \text{DV}(\Theta)$$

$$\text{(let)} \frac{\{\Gamma, \Gamma_{\rho} \vdash_{\Sigma} e: \tau \mid e = \rho(x) \wedge \tau = \Gamma_{\rho}(x)\} \quad \Gamma, \Gamma_{\rho} \vdash_{\Sigma} e': \tau}{\Gamma \vdash_{\Sigma} \text{let}(\rho; e'): \tau} \quad \text{dom}(\Gamma_{\rho}) = \text{dom}(\rho)$$

Type system

Type system

$$\begin{array}{l} \text{(l)} \frac{}{\Gamma \vdash_{\Sigma} l: \text{ref}\tau} \quad \Sigma(l) = \tau \quad \text{(new)} \frac{\Gamma \vdash_{\Sigma} e: \tau}{\Gamma \vdash_{\Sigma} \text{new}(e): M(\text{ref}\tau)} \\ \text{(get)} \frac{\Gamma \vdash_{\Sigma} e: \text{ref}\tau}{\Gamma \vdash_{\Sigma} \text{get}(e): M\tau} \quad \text{(set)} \frac{\Gamma \vdash_{\Sigma} e_2: \tau \quad \Gamma \vdash_{\Sigma} e_1: \text{ref}\tau}{\Gamma \vdash_{\Sigma} \text{set}(e_1, e_2): M(\text{ref}\tau)} \end{array}$$

[Back to the Summary](#)

Type system

Type system

$$\begin{array}{c}
 \{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Sigma} e: \tau \mid (X = e) \in \Delta \wedge \tau = \Pi'(X)\} \\
 \{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Sigma} e: M\tau \mid (X \Leftarrow e) \in \Delta \wedge \tau = \Pi'(X)\} \\
 \{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Sigma} e: \tau \mid (x = e) \in \Delta \wedge \tau = \Gamma_2(x)\} \\
 \{\Gamma, \Gamma_1, \Gamma_2 \vdash_{\Sigma} e: M\tau \mid (x \Leftarrow e) \in \Delta \wedge \tau = \Gamma_2(x)\} \\
 \text{(mixin)} \frac{}{\Gamma \vdash_{\Sigma} [\iota; \Delta]: [\Pi; \Pi']}
 \end{array}$$

$$\begin{array}{l}
 \text{img}(\iota) = \text{dom}(\Pi) \\
 \Gamma_1 \stackrel{\Delta}{=} \Pi \circ \iota \\
 \text{DN}(\Delta) = \text{dom}(\Pi') \\
 \text{DV}(\Delta) = \text{dom}(\Gamma_2)
 \end{array}$$

$$\begin{array}{c}
 \Gamma \vdash_{\Sigma} e: [\emptyset; \Pi] \\
 \text{(doall)} \frac{}{\Gamma \vdash_{\Sigma} \text{doall}(e): M\{\Pi\}}
 \end{array}$$

$$\begin{array}{c}
 \{\Gamma \vdash_{\Sigma} e: \tau \mid e = o(X) \wedge \tau = \Pi(X)\} \\
 \text{(record)} \frac{}{\Gamma \vdash_{\Sigma} \{o\}: \{\Pi\}} \quad \text{dom}(\Pi) = \text{dom}(o)
 \end{array}$$

$$\begin{array}{c}
 \Gamma \vdash_{\Sigma} e: \{\Pi\} \\
 \text{(select)} \frac{}{\Gamma \vdash_{\Sigma} e.X: \tau} \quad \tau = \Pi(X)
 \end{array}$$

[Back to the Summary](#)

Type system

Type system

$$\text{(sum)} \frac{\Gamma \vdash_{\Sigma} e_1 : [\Pi_1; \Pi'_1] \quad \Gamma \vdash_{\Sigma} e_2 : [\Pi_2; \Pi'_2] \quad \Pi_1 \text{ compatible with } \Pi_2 \quad \text{dom}(\Pi'_1) \cap \text{dom}(\Pi'_2) = \emptyset}{\Gamma \vdash_{\Sigma} e_1 + e_2 : [\Pi_1, \Pi_2; \Pi'_1, \Pi'_2]}$$

$$\text{(freeze)} \frac{\Gamma \vdash_{\Sigma} e : [\Pi; \Pi']}{\Gamma \vdash_{\Sigma} \text{freeze}_X(e) : [\Pi \setminus X; \Pi']} \quad \Pi(X) = \Pi'(X)$$

$$\text{(delete)} \frac{\Gamma \vdash_{\Sigma} e : [\Pi; \Pi']}{\Gamma \vdash_{\Sigma} e \setminus X : [\Pi; \Pi' \setminus X]} \quad X \in \text{dom}(\Pi')$$

[Back to the Summary](#)

Simplification

Simplification

(R) $\{o\}.X \longrightarrow e$ provided $e = o(X)$

(L) $\text{let}(\rho; e) \longrightarrow e\{x: \text{let}(\rho; \rho(x)) \mid x \in \text{dom}(\rho)\}$

(S) $[\iota_1; \Delta_1] + [\iota_2; \Delta_2] \longrightarrow [\iota_1, \iota_2; \Delta_1, \Delta_2]$ provided $[\iota_1, \iota_2; \Delta_1, \Delta_2]$ is well-formed, i.e.

■ $\text{DN}(\Delta_1) \cap \text{DN}(\Delta_2) = \text{DV}(\Delta_1) \cap \text{DV}(\Delta_2) = \text{dom}(\iota_1, \iota_2) \cap \text{DV}(\Delta_1, \Delta_2) = \emptyset$

■ ι_1, ι_2 is an injection (therefore ι_1 is compatible with ι_2)

■ $\text{FV}(\Delta_1) \cap (\text{dom}(\iota_2) \cup \text{DV}(\Delta_2)) = \text{FV}(\Delta_2) \cap (\text{dom}(\iota_1) \cup \text{DV}(\Delta_1)) = \emptyset$

(F) $\text{freeze}_X([\iota, x: X; \Delta, X \triangleleft e, \Delta']) \longrightarrow [\iota; \Delta, x \triangleleft e, X = x, \Delta']$

(D) $[\iota; \Delta, X \triangleleft e, \Delta'] \setminus X \longrightarrow [\iota; \Delta, \Delta']$

$[\text{DV}(_), \text{DN}(_), \text{FV}(_)]$ definition for D, Δ and Θ

[Back to the Summary](#)

Simplification

Simplification

(A) $\text{doall}([\emptyset; \Delta]) \longrightarrow \text{mdo}(|\Delta|; \text{ret}\{o_1, o_2\})\{x: \text{let}(\rho; x) | x \in \text{dom}(\rho)\}$ where

- $\rho = \{x: e | (x = e) \in \Delta\}$
- $o_1 = \{X: e | (X = e) \in \Delta\}$ and
 $o_2 = \{X: x_X | X \leftarrow e \in \Delta\}$, with x_X freshly chosen
- $|\Delta|$ is defined by induction on Δ as follows:
 - $|\emptyset| = \emptyset$
 - $|(\Delta, X = e)| = |(\Delta, x = e)| = |\Delta|$
 - $|(\Delta, X \leftarrow e)| = |\Delta|, x_X \leftarrow e$
 - $|(\Delta, x \leftarrow e)| = |\Delta|, x \leftarrow e$

[Back to the Summary](#)

Computation

Computation

- Stores: $\mu \in S \stackrel{\Delta}{=} L \xrightarrow{fin} \text{Exp}$
- Evaluation Contexts: $E \in \text{EC} ::= \square \mid E[\text{mdo}(x \leftarrow \square, \Theta; e)]$
- configuration: $(\mu, e, E) \in \text{Conf} \stackrel{\Delta}{=} S \times \text{Exp} \times \text{EC}$
- Bad terms: $b \in \text{BE} ::= x \mid b + e \mid e + b \mid \text{freeze}_X(b) \mid b \setminus X \mid \text{doall}(b) \mid b.X \mid \text{get}(b) \mid \text{set}(b, e)$
- Computational Redexes: $r \in R ::= \text{mdo}(\Theta; e) \mid \text{ret}(e) \mid \text{new}(e) \mid \text{get}(l) \mid \text{set}(l, e) \mid b$

Computation

Computation

- **Completion step:** (done) $(\mu, \text{ret}(e), \square) \mapsto \text{done}$

Computation

■ **Completion step:** (done) $(\mu, \text{ret}(e), \square) \longmapsto \text{done}$

■ **Recursive monadic binding steps:**

(M.0) $(\mu, \text{mdo}(\emptyset; e), E) \longmapsto (\mu, e, E)$

(M.1) $(\mu, \text{mdo}(x_1 \leftarrow e_1, \Theta; e), E) \longmapsto (\mu, e_1, E[\text{mdo}(x_1 \leftarrow \square, \Theta; e)])$, with variables in $DV(x_1 \leftarrow e_1, \Theta)$ renamed to avoid clashes with $CV(E)$

(M.2) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square; e)]) \longmapsto (\mu\{\rho\}, e\{\rho\}, E)$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

(M.3) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square, x_2 \leftarrow e_2, \Theta; e)]) \longmapsto$
 $(\mu\{\rho\}, e_2\{\rho\}, E[\text{mdo}(x_2 \leftarrow \square, \Theta; e)\{\rho\}])$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

[CV() definition]

Computation

■ **Completion step:** (done) $(\mu, \text{ret}(e), \square) \longmapsto \text{done}$

■ **Recursive monadic binding steps:**

(M.0) $(\mu, \text{mdo}(\emptyset; e), E) \longmapsto (\mu, e, E)$

(M.1) $(\mu, \text{mdo}(x_1 \leftarrow e_1, \Theta; e), E) \longmapsto (\mu, e_1, E[\text{mdo}(x_1 \leftarrow \square, \Theta; e)])$, with variables in $DV(x_1 \leftarrow e_1, \Theta)$ renamed to avoid clashes with $CV(E)$

(M.2) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square; e)]) \longmapsto (\mu\{\rho\}, e\{\rho\}, E)$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

(M.3) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square, x_2 \leftarrow e_2, \Theta; e)]) \longmapsto$
 $(\mu\{\rho\}, e_2\{\rho\}, E[\text{mdo}(x_2 \leftarrow \square, \Theta; e)\{\rho\}])$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

[CV () definition]

■ **Imperative steps:**

(I.1) $(\mu, \text{new}(e), E) \longmapsto (\mu\{l: e\}, \text{ret}(l), E)$, $l \notin \text{dom}(\mu)$

(I.2) $(\mu, \text{get}(l), E) \longmapsto (\mu, \text{ret}(e), E)$, $e = \mu(l)$

(I.3) $(\mu, \text{set}(l, e), E) \longmapsto (\mu\{l: e\}, \text{ret}(l), E)$, $l \in \text{dom}(\mu)$

Computation

■ **Completion step:** (done) $(\mu, \text{ret}(e), \square) \longmapsto \text{done}$

■ **Recursive monadic binding steps:**

(M.0) $(\mu, \text{mdo}(\emptyset; e), E) \longmapsto (\mu, e, E)$

(M.1) $(\mu, \text{mdo}(x_1 \leftarrow e_1, \Theta; e), E) \longmapsto (\mu, e_1, E[\text{mdo}(x_1 \leftarrow \square, \Theta; e)])$, with variables in $DV(x_1 \leftarrow e_1, \Theta)$ renamed to avoid clashes with $CV(E)$

(M.2) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square; e)]) \longmapsto (\mu\{\rho\}, e\{\rho\}, E)$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

(M.3) $(\mu, \text{ret}(e_1), E[\text{mdo}(x_1 \leftarrow \square, x_2 \leftarrow e_2, \Theta; e)]) \longmapsto$
 $(\mu\{\rho\}, e_2\{\rho\}, E[\text{mdo}(x_2 \leftarrow \square, \Theta; e)\{\rho\}])$,
with $\rho \triangleq \{x_1: \text{let}(x_1: e_1; x_1)\}$

[CV () definition]

■ **Imperative steps:**

(I.1) $(\mu, \text{new}(e), E) \longmapsto (\mu\{l: e\}, \text{ret}(l), E)$, $l \notin \text{dom}(\mu)$

(I.2) $(\mu, \text{get}(l), E) \longmapsto (\mu, \text{ret}(e), E)$, $e = \mu(l)$

(I.3) $(\mu, \text{set}(l, e), E) \longmapsto (\mu\{l: e\}, \text{ret}(l), E)$, $l \in \text{dom}(\mu)$

■ **Error step** caused by a bad term:

(err) $(\mu, b, E) \longmapsto \text{err}$

Technical results

Technical results

- **CR for \longrightarrow** : The relation \longrightarrow is confluent
- **SR for \longrightarrow** : If $\Gamma \vdash_{\Sigma} e:\tau$ and $e \longrightarrow e'$, then $\Gamma \vdash_{\Sigma} e':\tau$
- **Bisimulation:** If $(\mu_1, r_1, E_1) \xrightarrow{*} (\mu_2, r_2, E_2)$, then
 1. $(\mu_1, r_1, E_1) \longmapsto Id_1$ implies $\exists Id_2$ s.t. $(\mu_2, r_2, E_2) \longmapsto Id_2$ and $Id_1 \xrightarrow{*} Id_2$
 2. $(\mu_2, r_2, E_2) \longmapsto Id_2$ implies $\exists Id_1$ s.t. $(\mu_1, r_1, E_1) \longmapsto Id_1$ and $Id_1 \xrightarrow{*} Id_2$

where Id_1 and Id_2 range over $\text{Conf} \cup \{\text{done}, \text{err}\}$.

- **SR:**
 1. If $\Gamma \vdash_{\Sigma} (\mu, e, E)$ and $(\mu, e, E) \longrightarrow (\mu', e', E')$, then $\Gamma \vdash_{\Sigma} (\mu', e', E')$.
 2. If $\Gamma \vdash_{\Sigma} (\mu, e, E)$ and $(\mu, e, E) \longmapsto (\mu', e', E')$, then there exist $\Sigma' \supseteq \Sigma$ and Γ' compatible with Γ such that $\Gamma' \vdash_{\Sigma'} (\mu', e', E')$.
- **Progress:** If $\Gamma \vdash_{\Sigma} (\mu, e, E)$, then
 1. $e \in R$ and $(\mu, e, E) \longmapsto$, or
 2. $e \notin R$ and $e \longrightarrow$

[\[\$\Gamma \vdash_{\Sigma} \(\mu, e, E\)\$: well-formed configuration definition\]](#)

[Back to the Summary](#)

Definitions: functions $FV(_)$, $DV(_)$ and $DN(_)$

Definitions: functions function $FV(_)$, $DV(_)$ and $DN(_)$

$e \in \text{Exp}$	$FV(_) \subseteq_{fin} \text{Var}$
x	$\{x\}$
l	\emptyset
$\text{ret}(e) \mid \text{new}(e) \mid \text{get}(e)$ $\text{doall}(e) \mid e.X \mid \text{freeze}_X(e) \mid e \setminus X$	$FV(e)$
$\text{set}(e_1, e_2) \mid e_1 + e_2$	$FV(e_1) \cup FV(e_2)$
$\text{mdo}(\Theta; e)$	$(FV(\Theta) \cup FV(e)) \setminus DV(\Theta)$
$\text{let}(\rho; e)$	$(FV(\rho) \cup FV(e)) \setminus \text{dom}(\rho)$
$\{o\}$	$FV(o)$
$[\iota; \Delta]$	$FV(\Delta) \setminus (DV(\Delta) \cup \text{dom}(\iota))$

$FV(f) \triangleq$
 $\cup \{FV(f(a)) \mid$
 $a \in \text{dom}(f)\}$
 with
 $f: A \xrightarrow{fin} \text{Exp}$

$D/\Delta/\Theta$	$FV(_) \subseteq_{fin} \text{Var}$	$DV(_) \subseteq_{fin} \text{Var}$	$DN(_) \subseteq_{fin} \text{Name}$
$X \triangleleft e$	$FV(e)$	\emptyset	$\{X\}$
$x \triangleleft e$	$FV(e)$	$\{x\}$	\emptyset
\emptyset	\emptyset	\emptyset	\emptyset
Δ, D	$FV(\Delta) \cup FV(D)$	$DV(\Delta) \cup DV(D)$	$DN(\Delta) \cup DN(D)$
$\Theta, x \leftarrow e$	$FV(\Theta) \cup FV(e)$	$DV(\Theta) \cup \{x\}$	\emptyset

Definitions: functions $FV(_)$ and $CV(_)$ for E

Definitions: functions $FV(_)$ and $CV(_)$ for E

$E \in EC$	$FV(_) \subseteq_{fin} Var$	$CV(_) \subseteq_{fin} Var$
\square	\emptyset	\emptyset
$E[\text{mdo}(x \leftarrow \square, \Theta; e)]$	$FV(E) \cup (FV(\Theta, e) \setminus CV(E[\text{mdo}(x \leftarrow \square, \Theta; e)]))$	$CV(E) \cup \{x\} \cup DV(\Theta)$

[Back to the Summary](#)

Definitions: well-formed configurations

Definitions: well-formed configurations

$$\Gamma \vdash_{\Sigma} (\mu, e, E) \stackrel{\Delta}{\iff}$$

- $\text{dom}(\Sigma) = \text{dom}(\mu)$ and $\text{dom}(\Gamma) = \text{CV}(E)$
- $\mu(l) = e_l$ and $\Sigma(l) = \tau_l$ imply $\Gamma \vdash_{\Sigma} e_l : \tau_l$
- exists τ such that $\Gamma \vdash_{\Sigma} e : M_{\tau}$ derivable
- exists τ' such that $\Gamma, \square : M_{\tau} \vdash_{\Sigma} E : M_{\tau'}$ (i.e., it is well-formed)

where:

Well-formed evaluation context:

$$(\square) \frac{}{\emptyset, \square : M_{\tau} \vdash_{\Sigma} \square : M_{\tau}}$$

$$\{\Gamma, x_1 : \tau_1, \Gamma_{\Theta} \vdash_{\Sigma} e' : M_{\tau'} \mid (x' \Leftarrow e') \in \Theta \wedge \tau' = \Gamma_{\Theta}(x')\}$$

$$\Gamma, x_1 : \tau_1, \Gamma_{\Theta} \vdash_{\Sigma} e : M_{\tau_2}$$

$$\Gamma, \square : M_{\tau_2} \vdash_{\Sigma} E : M_{\tau}$$

$$(\text{mdo}) \frac{}{\Gamma, x_1 : \tau_1, \Gamma_{\Theta}, \square : M_{\tau_1} \vdash_{\Sigma} E[\text{mdo}(x_1 \Leftarrow \square, \Theta; e)] : M_{\tau}} \quad \text{dom}(\Gamma_{\Theta}) = \text{DV}(\Theta)$$

[Back to the Summary](#)