

# **Modular operational semantics for fundamental programming constructs**

Peter Mosses

Swansea University, UK

**DiverSE Seminar on SLE**

28 May 2015 • Rennes, France

# Overview

## Modular semantics

modularity

- ▶ SOS (Structural Operational Semantics)
- ▶ MSOS (Modular SOS)
- ▶ I-MSOS (Implicitly-MSOS)
- ▶ Bisimulation

## Component-based semantics

reuse !

- ▶ Funcons (fundamental programming constructs)
- ▶ Language specifications

## Appendix

# Conventional SOS

# SOS: sequencing

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$e \rightarrow e'$

## Rules

$$\frac{e_1 \rightarrow e'_1}{(e_1;e_2) \rightarrow (e'_1;e_2)}$$

$$(v_1;e_2) \rightarrow e_2$$

## Auxiliary

$v \in Val$

# SOS: binding

## Syntax

$e ::=$   
 $v$   
 $e;e$

$x$

## Semantics

$\rho \vdash e \rightarrow e'$

## Rules

$\rho \vdash x \rightarrow v$  if  $\rho(x) = v$

## Auxiliary

$v \in Val$

$\rho \in Env$

# SOS: binding

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let  $x = e$  in  $e$**

## Semantics

$\rho \vdash e \rightarrow e'$

## Rules

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash (\text{let } x = e_1 \text{ in } e_2) \rightarrow (\text{let } x = e'_1 \text{ in } e_2)}$$

$$\frac{\rho[x \mapsto v_1] \vdash e_2 \rightarrow e'_2}{\rho \vdash (\text{let } x = v_1 \text{ in } e_2) \rightarrow (\text{let } x = v_1 \text{ in } e'_2)}$$

$$\rho \vdash (\text{let } x = v_1 \text{ in } v_2) \rightarrow v_2$$

## Auxiliary

$v \in Val$

$\rho \in Env$

# SOS: sequencing + binding

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

## Semantics

$\rho \vdash e \rightarrow e'$

## Rules

$$\frac{\rho \vdash e_1 \rightarrow e'_1}{\rho \vdash (e_1;e_2) \rightarrow (e'_1;e_2)}$$
$$\rho \vdash (v_1;e_2) \rightarrow e_2$$

## Auxiliary

$v \in Val$

$\rho \in Env$

# SOS: storing

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

## Semantics

$\rho \vdash (e, \sigma) \rightarrow (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e, \sigma) \rightarrow (e', \sigma')}{\rho \vdash (!e, \sigma) \rightarrow (!e', \sigma')}$$

$\rho \vdash (!l, \sigma) \rightarrow (v, \sigma) \text{ if } \sigma(l) = v$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$



# SOS: storing

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

## Semantics

$\rho \vdash (e, \sigma) \rightarrow (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \rightarrow (e'_1, \sigma')}{\rho \vdash (e_1 := e_2, \sigma) \rightarrow (e'_1 := e_2, \sigma')}$$

$$\frac{\rho \vdash (e_2, \sigma) \rightarrow (e'_2, \sigma')}{\rho \vdash (e_1 := e_2, \sigma) \rightarrow (e_1 := e'_2, \sigma')}$$

$$\rho \vdash (l_1 := v_2, \sigma) \rightarrow (v_2, \sigma[l_1 \mapsto v_2]) \text{ if } l_1 \in \text{dom } \sigma$$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

# SOS: sequencing + binding + storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$

## Semantics

$\rho \vdash (e, \sigma) \rightarrow (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \rightarrow (e'_1, \sigma')}{\rho \vdash (e_1; e_2, \sigma) \rightarrow (e'_1; e_2, \sigma')}$$
$$\rho \vdash (v_1; e_2, \sigma) \rightarrow (e_2, \sigma)$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

# SOS: binding + storing

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

## Semantics

$\rho \vdash (e, \sigma) \rightarrow (e', \sigma')$

## Rules

$\rho \vdash (x, \sigma) \rightarrow (v, \sigma)$  if  $\rho(x) = v$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

# SOS: binding + storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$

## Semantics

$\rho \vdash (e, \sigma) \rightarrow (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \rightarrow (e'_1, \sigma')}{\rho \vdash (\text{let } x = e_1 \text{ in } e_2, \sigma) \rightarrow (\text{let } x = e'_1 \text{ in } e_2, \sigma')}$$
$$\frac{\rho[x \mapsto v_1] \vdash (e_2, \sigma) \rightarrow (e'_2, \sigma')}{\rho \vdash (\text{let } x = v_1 \text{ in } e_2, \sigma) \rightarrow (\text{let } x = v_1 \text{ in } e'_2, \sigma')}$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

$$\rho \vdash (\text{let } x = v_1 \text{ in } v_2, \sigma) \rightarrow (v_2, \sigma)$$

# SOS: emitting

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

**print**  $e$

## Semantics

$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')}{\rho \vdash (\mathbf{print} \ e, \sigma) \xrightarrow{\alpha} (\mathbf{print} \ e', \sigma')}$$
$$\rho \vdash (\mathbf{print} \ v, \sigma) \xrightarrow{v} (v, \sigma)$$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

$\alpha \in Val \cup \{\tau\}$

# SOS: sequencing + ... + emitting

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

$$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \xrightarrow{\alpha} (e'_1, \sigma')}{\rho \vdash (e_1; e_2, \sigma) \xrightarrow{\alpha} (e'_1; e_2, \sigma')}$$

$$\rho \vdash (v_1; e_2, \sigma) \xrightarrow{\tau} (e_2, \sigma)$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val \cup \{\tau\}$

# SOS: binding + ... + emitting

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

**print**  $e$

## Semantics

$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$

## Rules

$\rho \vdash (x, \sigma) \xrightarrow{\tau} (v, \sigma)$  if  $\rho(x) = v$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

$\alpha \in Val \cup \{\tau\}$

# SOS: binding + ... + emitting

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \xrightarrow{\alpha} (e'_1, \sigma')}{\rho \vdash (\text{let } x = e_1 \text{ in } e_2, \sigma) \xrightarrow{\alpha} (\text{let } x = e'_1 \text{ in } e_2, \sigma')}$$
$$\frac{\rho[x \mapsto v_1] \vdash (e_2, \sigma) \xrightarrow{\alpha} (e'_2, \sigma')}{\rho \vdash (\text{let } x = v_1 \text{ in } e_2, \sigma) \xrightarrow{\alpha} (\text{let } x = v_1 \text{ in } e'_2, \sigma')}$$
$$\rho \vdash (\text{let } x = v_1 \text{ in } v_2, \sigma) \xrightarrow{\tau} (v_2, \sigma)$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val \cup \{\tau\}$



# SOS: storing + ... + emitting

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')}{\rho \vdash (!e, \sigma) \xrightarrow{\alpha} (!e', \sigma')}$$

$\rho \vdash (!l, \sigma) \xrightarrow{\tau} (v, \sigma) \text{ if } \sigma(l) = v$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val \cup \{\tau\}$

# SOS: storing + ... + emitting

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \xrightarrow{\alpha} (e'_1, \sigma')}{\rho \vdash (e_1 := e_2, \sigma) \xrightarrow{\alpha} (e'_1 := e_2, \sigma')}$$
$$\frac{\rho \vdash (e_2, \sigma) \xrightarrow{\alpha} (e'_2, \sigma')}{\rho \vdash (e_1 := e_2, \sigma) \xrightarrow{\alpha} (e_1 := e'_2, \sigma')}$$
$$\rho \vdash (l_1 := v_2, \sigma) \xrightarrow{\tau} (v_2, \sigma[l_1 \mapsto v_2]) \text{ if } l_1 \in \text{dom } \sigma$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val \cup \{\tau\}$

# SOS: abrupt termination handling ?

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

**print**  $e$

**$e$  otherwise  $e$**

## Semantics

$$\rho \vdash (e, \sigma) \xrightarrow{?} (?, \sigma')$$

## Rules

$$\rho \vdash (x, \sigma) \xrightarrow{?} (?, \sigma) \text{ if } x \notin \text{dom } \rho$$

$$\rho \vdash (!l, \sigma) \xrightarrow{?} (?, \sigma) \text{ if } l \notin \text{dom } \sigma$$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

$\alpha \in Val \cup \{\tau\}$

# SOS: summary

***Non-modular !***

# **MSOS: Modular SOS**



ELSEVIER

Available online at [www.sciencedirect.com](http://www.sciencedirect.com)

SCIENCE *@* DIRECT®

The Journal of Logic and  
Algebraic Programming 60–61 (2004) 195–228

---

THE JOURNAL OF  
LOGIC AND  
ALGEBRAIC  
PROGRAMMING

---

[www.elsevier.com/locate/jlap](http://www.elsevier.com/locate/jlap)

# Modular structural operational semantics<sup>☆</sup>

Peter D. Mosses

# SOS: sequencing + ...

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$$\rho \vdash (e, \sigma) \xrightarrow{\alpha} (e', \sigma')$$

## Rules

$$\frac{\rho \vdash (e_1, \sigma) \xrightarrow{\alpha} (e'_1, \sigma')}{\rho \vdash (e_1; e_2, \sigma) \xrightarrow{\alpha} (e'_1; e_2, \sigma')}$$

$$\rho \vdash (v_1; e_2, \sigma) \xrightarrow{\tau} (e_2, \sigma)$$

## Auxiliary

$v \in Val$

# SOS: sequencing + ...

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$e \xrightarrow{X} e'$

## Rules

$$\frac{e_1 \xrightarrow{X} e'_1}{(e_1;e_2) \xrightarrow{X} (e'_1;e_2)} \quad X = \{\rho, \sigma, \sigma', \alpha', \dots\}$$

## Auxiliary

$v \in Val$

$$(v_1;e_2) \xrightarrow{U} e_2 \quad U = \{\rho, \sigma, \sigma' = \sigma, \alpha' = \tau, \dots\}$$



# MSOS: label composition

## Semantics

$$e \xrightarrow{X} e'$$

## Computations

$$e_1 \xrightarrow{X_1} e_2 \xrightarrow{X_2} e_3 \cdots$$

$$X_1 = \{\rho_1, \sigma_1, \sigma'_1, \alpha'_1, \cdots\}$$

$$\rho_1 = \rho_2 \quad \downarrow \quad \sigma'_1 = \sigma_2$$

$$X_2 = \{\rho_2, \sigma_2, \sigma'_2, \alpha'_2, \cdots\}$$

labels in MSOS are  
morphisms of a  
category

unobservable steps  
are labelled by  
identity morphisms

independent label  
components :  
indexed product

# MSOS: sequencing

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$e \xrightarrow{\{\rho, \sigma, \sigma', \alpha', \dots\}} e'$

SML-style  
record  
pattern

## Rules

$$\frac{e_1 \xrightarrow{\{\rho, \sigma, \sigma', \alpha', \dots\}} e'_1}{(e_1; e_2) \xrightarrow{\{\rho, \sigma, \sigma', \alpha', \dots\}} (e'_1; e_2)}$$

$$(v_1; e_2) \xrightarrow{\{\rho, \sigma, \sigma' = \sigma, \alpha' = \tau, \text{---}\}} e_2$$

## Auxiliary

$v \in Val$

# MSOS: sequencing

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$e \xrightarrow{\{\dots\}} e'$

## Rules

$$\frac{e_1 \xrightarrow{\{\dots\}} e'_1}{(e_1;e_2) \xrightarrow{\{\dots\}} (e'_1;e_2)}$$

## Auxiliary

$v \in Val$

$$(v_1;e_2) \xrightarrow{\{\text{—}\}} e_2$$

# MSOS: binding

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$

## Semantics

$e \xrightarrow{\{\rho, \dots\}} e'$

## Rules

$x \xrightarrow{\{\rho, \_ \}} v \text{ if } \rho(x) = v$

## Auxiliary

$v \in Val$   
 $\rho \in Env$

# MSOS: binding

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let  $x = e$  in  $e$**

## Semantics

$$e \xrightarrow{\{\rho, \dots\}} e'$$

## Rules

$$\frac{e_1 \xrightarrow{\{\dots\}} e'_1}{(\text{let } x = e_1 \text{ in } e_2) \xrightarrow{\{\dots\}} (\text{let } x = e'_1 \text{ in } e_2)}$$

$$\frac{e_2 \xrightarrow{\{\rho[x \mapsto v_1], \dots\}} e'_2}{(\text{let } x = v_1 \text{ in } e_2) \xrightarrow{\{\rho, \dots\}} (\text{let } x = v_1 \text{ in } e'_2)}$$

$$(\text{let } x = v_1 \text{ in } v_2) \xrightarrow{\{\text{---}\}} v_2$$

## Auxiliary

$v \in Val$

$\rho \in Env$

# MSOS: storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$

## Semantics

$$e \xrightarrow{\{\sigma, \sigma', \dots\}} e'$$

## Rules

$$\frac{e \xrightarrow{\{\dots\}} e'}{!e \xrightarrow{\{\dots\}} !e'}$$

$$!l \xrightarrow{\{\sigma, \sigma' = \sigma, -\}} v \quad \text{if } \sigma(l) = v$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

# MSOS: storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
 $\text{let } x = e \text{ in } e$   
 $!e$   
 $e := e$

## Semantics

$$e \xrightarrow{\{\sigma, \sigma', \dots\}} e'$$

## Rules

$$\frac{e_1 \xrightarrow{\{\dots\}} e'_1}{(e_1 := e_2) \xrightarrow{\{\dots\}} (e'_1 := e_2)}$$

$$\frac{e_2 \xrightarrow{\{\dots\}} e'_2}{(e_1 := e_2) \xrightarrow{\{\dots\}} (e_1 := e'_2)}$$

$$(l_1 := v_2) \xrightarrow{\{\sigma, \sigma' = \sigma[l_1 \mapsto v_2], -\}} v_2 \text{ if } l_1 \in \text{dom } \sigma$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

# MSOS: emitting

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

**print**  $e$

## Semantics

$$e \xrightarrow{\{\alpha', \dots\}} e'$$

## Rules

$$\frac{e \xrightarrow{\{\dots\}} e'}{\text{print } e \xrightarrow{\{\dots\}} \text{print } e'}$$
$$\text{print } v \xrightarrow{\{\alpha' = v, \text{---}\}} v$$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

$\alpha \in Val^*$



# MSOS: abrupt termination

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

$$e \xrightarrow{\{\phi', \dots\}} e'$$

## Rules

$$x \xrightarrow{\{\rho, \text{---}\}} v \quad \text{if } \rho(x) = v$$

$$x \xrightarrow{\{\rho, \phi' = \text{err}, \text{---}\}} x \quad \text{if } x \notin \text{dom } \rho$$

## Auxiliary

$v \in \text{Val}$   
 $\rho \in \text{Env}$   
 $\sigma \in \text{Store}$   
 $l \in \text{Loc}$   
 $\alpha \in \text{Val}^*$

$\phi \in \{\text{ok}, \text{err}\}$

$$l \xrightarrow{\{\sigma, \sigma' = \sigma, \text{---}\}} v \quad \text{if } \sigma(l) = v$$

$$l \xrightarrow{\{\sigma, \sigma' = \sigma, \phi' = \text{err}, \text{---}\}} l \quad \text{if } l \notin \text{dom } \sigma$$

# MSOS: abrupt termination handling !

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$   
 $e$  **otherwise**  $e$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val^*$   
 $\phi \in \{ok, err\}$

## Semantics

$$e \xrightarrow{\{\phi', \dots\}} e'$$

## Rules

$$\frac{e_1 \xrightarrow{\{\phi' = ok, \dots\}} e'_1}{(e_1 \text{ otherwise } e_2) \xrightarrow{\{\phi' = ok, \dots\}} (e'_1 \text{ otherwise } e_2)}$$

$$\frac{e_1 \xrightarrow{\{\phi' = err, \dots\}} e'_1}{(e_1 \text{ otherwise } e_2) \xrightarrow{\{\phi' = ok, \dots\}} e_2}$$

$$(v_1 \text{ otherwise } e_2) \xrightarrow{\{\text{—}\}} v_1$$

# MSOS: summary

*Modular*

– *but labels unattractive*

# **I-MSOS: Implicitly-MSOS**

# SOS 2008



ELSEVIER

Available online at [www.sciencedirect.com](http://www.sciencedirect.com)



---

---

**Electronic Notes in  
Theoretical Computer  
Science**

---

---

Electronic Notes in Theoretical Computer Science 229 (2009) 49–66

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

## Implicit Propagation in Structural Operational Semantics

Peter D. Mosses<sup>1</sup> Mark J. New<sup>2</sup>

# MODULARITY '14

## Reusable Components of Semantic Specifications

Martin Churchill<sup>1</sup>, Peter D. Mosses<sup>2</sup>, Neil Sculthorpe<sup>2</sup>, and Paolo Torrini<sup>2</sup>

Extended version: *Trans.AOSD XII*, 2015.

- ▶ a component-based semantics of CAML LIGHT
- ▶ validated (by empirical testing)
- ▶ detailed introduction to the approach
- ▶ overview of preliminary tool support

# I-MSOS: a notation for MSOS

	I-MSOS	MSOS
transitions	$e_1 \rightarrow e_2$	$e_1 - \{\cdots\} \rightarrow e_2$
	$\rho \vdash e_1 \rightarrow e_2$	$e_1 - \{\rho, \cdots\} \rightarrow e_2$
	$(e_1, \sigma) \rightarrow (e_2, \sigma')$	$e_1 - \{\sigma, \sigma', \cdots\} \rightarrow e_2$
	$\rho \vdash (e_1, \sigma) \rightarrow (e_2, \sigma')$	$e_1 - \{\rho, \sigma, \sigma', \cdots\} \rightarrow e_2$
	$e_1 - \alpha \rightarrow e_2$	$e_1 - \{\alpha', \cdots\} \rightarrow e_2$
	$\rho \vdash e_1 - \alpha \rightarrow e_2$	$e_1 - \{\rho, \alpha', \cdots\} \rightarrow e_2$
	$(e_1, \sigma) - \alpha \rightarrow (e_2, \sigma')$	$e_1 - \{\sigma, \sigma', \alpha', \cdots\} \rightarrow e_2$
	$\rho \vdash (e_1, \sigma) - \alpha \rightarrow (e_2, \sigma')$	$e_1 - \{\rho, \sigma, \sigma', \alpha', \cdots\} \rightarrow e_2$

# I-MSOS: a notation for MSOS

	I-MSOS	MSOS
transitions	$e_1 \rightarrow e_2$	$e_1 - \{\text{---}\} \rightarrow e_2$
	$\rho \vdash e_1 \rightarrow e_2$	$e_1 - \{\rho, \text{---}\} \rightarrow e_2$
	$(e_1, \sigma) \rightarrow (e_2, \sigma')$	$e_1 - \{\sigma, \sigma', \text{---}\} \rightarrow e_2$
	$\rho \vdash (e_1, \sigma) \rightarrow (e_2, \sigma')$	$e_1 - \{\rho, \sigma, \sigma', \text{---}\} \rightarrow e_2$
	$e_1 - \alpha \rightarrow e_2$	$e_1 - \{\alpha', \text{---}\} \rightarrow e_2$
	$\rho \vdash e_1 - \alpha \rightarrow e_2$	$e_1 - \{\rho, \alpha', \text{---}\} \rightarrow e_2$
	$(e_1, \sigma) - \alpha \rightarrow (e_2, \sigma')$	$e_1 - \{\sigma, \sigma', \alpha', \text{---}\} \rightarrow e_2$
	$\rho \vdash (e_1, \sigma) - \alpha \rightarrow (e_2, \sigma')$	$e_1 - \{\rho, \sigma, \sigma', \alpha', \text{---}\} \rightarrow e_2$



# I-MSOS: sequencing

## Syntax

$e ::=$   
 $v$   
 $e;e$

## Semantics

$e \rightarrow e'$

## Rules

$$\frac{e_1 \rightarrow e'_1}{(e_1;e_2) \rightarrow (e'_1;e_2)}$$

$$(v_1;e_2) \rightarrow e_2$$

## Auxiliary

$v \in Val$

## MSOS:

### Conditional rules

$$\frac{e_1 \xrightarrow{\{\dots\}} e'_1}{(e_1;e_2) \xrightarrow{\{\dots\}} (e'_1;e_2)}$$

### Unconditional rules

$$(v_1;e_2) \xrightarrow{\{-\}} e_2$$

# I-MSOS: binding

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$

## Semantics

$\text{env}(\rho) \vdash e \rightarrow e'$

## Rules

$\text{env}(\rho) \vdash x \rightarrow v$  if  $\rho(x) = v$

## Auxiliary

$v \in Val$   
 $\rho \in Env$

# I-MSOS: binding

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let  $x = e$  in  $e$**

## Semantics

**env( $\rho$ )**  $\vdash e \rightarrow e'$

## Rules

$$\frac{e_1 \rightarrow e'_1}{(\text{let } x = e_1 \text{ in } e_2) \rightarrow (\text{let } x = e'_1 \text{ in } e_2)}$$

$$\frac{\text{env}(\rho[x \mapsto v_1]) \vdash e_2 \rightarrow e'_2}{\text{env}(\rho) \vdash (\text{let } x = v_1 \text{ in } e_2) \rightarrow (\text{let } x = v_1 \text{ in } e'_2)}$$

## Auxiliary

$v \in Val$

$\rho \in Env$

$$(\text{let } x = v_1 \text{ in } v_2) \rightarrow v_2$$

# I-MSOS: storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$

## Semantics

$(e, \text{store}(\sigma)) \rightarrow (e', \text{store}(\sigma'))$

## Rules

$$\frac{e \rightarrow e'}{!e \rightarrow !e'}$$

$(!l, \text{store}(\sigma)) \rightarrow (v, \text{store}(\sigma))$  if  $\sigma(l) = v$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

# I-MSOS: storing

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$

## Semantics

$(e, \text{store}(\sigma)) \rightarrow (e', \text{store}(\sigma'))$

## Rules

$$\frac{e_1 \rightarrow e'_1}{(e_1 := e_2) \rightarrow (e'_1 := e_2)}$$

$$\frac{e_2 \rightarrow e'_2}{(e_1 := e_2) \rightarrow (e_1 := e'_2)}$$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$

$(l_1 := v_2, \text{store}(\sigma)) \rightarrow (v_2, \text{store}(\sigma[l_1 \mapsto v_2]))$   
if  $l_1 \in \text{dom } \sigma$

# I-MSOS: emitting

## Syntax

$e ::=$

$v$

$e;e$

$x$

**let**  $x = e$  **in**  $e$

**!** $e$

$e := e$

**print**  $e$

## Auxiliary

$v \in Val$

$\rho \in Env$

$\sigma \in Store$

$l \in Loc$

$\alpha \in Val \cup \{\tau\}$

## Semantics

$$e \xrightarrow{\text{output}(\alpha)} e'$$

## Rules

$$\frac{e \longrightarrow e'}{\text{print } e \longrightarrow \text{print } e'}$$

$$\text{print } v \xrightarrow{\text{output}(v)} v$$

# I-MSOS: abrupt termination

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$

## Semantics

### Rules

$$\text{env}(\rho) \vdash (e, \text{store}(\sigma)) \xrightarrow{\text{abrupt}(\phi)} (e', \text{store}(\sigma'))$$

$$\text{env}(\rho) \vdash x \longrightarrow v \quad \text{if } \rho(x) = v$$

$$\text{env}(\rho) \vdash x \xrightarrow{\text{abrupt}(\text{err})} x \quad \text{if } x \notin \text{dom } \rho$$

## Auxiliary

$v \in \text{Val}$   
 $\rho \in \text{Env}$   
 $\sigma \in \text{Store}$   
 $l \in \text{Loc}$   
 $\alpha \in \text{Val}^*$   
 $\phi \in \{\text{ok}, \text{err}\}$

$$(l, \text{store}(\sigma)) \longrightarrow (v, \text{store}(\sigma)) \quad \text{if } \sigma(l) = v$$

$$(l, \text{store}(\sigma)) \xrightarrow{\text{abrupt}(\text{err})} (l, \text{store}(\sigma)) \quad \text{if } l \notin \text{dom } \sigma$$

# I-MSOS: abrupt termination handling!

## Syntax

$e ::=$   
 $v$   
 $e;e$   
 $x$   
**let**  $x = e$  **in**  $e$   
 $!e$   
 $e := e$   
**print**  $e$   
 $e$  **otherwise**  $e$

## Auxiliary

$v \in Val$   
 $\rho \in Env$   
 $\sigma \in Store$   
 $l \in Loc$   
 $\alpha \in Val^*$   
 $\phi \in \{ok, err\}$

## Semantics

$e \xrightarrow{\text{abrupt}(\phi)} e'$

## Rules

$$\frac{e_1 \xrightarrow{\text{abrupt}(ok)} e'_1}{(e_1 \text{ otherwise } e_2) \xrightarrow{\text{abrupt}(ok)} (e'_1 \text{ otherwise } e_2)}$$

$$\frac{e_1 \xrightarrow{\text{abrupt}(err)} e'_1}{(e_1 \text{ otherwise } e_2) \xrightarrow{\text{abrupt}(ok)} e_2}$$

$$(v_1 \text{ otherwise } e_2) \longrightarrow v_1$$



# I-MSOS: summary

*Modular*

– *OK?*

# Overview

## Modular semantics

- ▶ SOS (Structural Operational Semantics)
- ▶ MSOS (Modular SOS)
- ▶ I-MSOS (Implicitly-MSOS)
- ▶ Bisimulation

# Bisimulation

# Modular proofs

FOSSACS'13:

- ▶ bisimilarity  
congruence  
format
- ▶ preservation by  
disjoint extension

## Modular Bisimulation Theory for Computations and Values

Martin Churchill and Peter D. Mosses  
`{m.d.churchill,p.d.mosses}@swansea.ac.uk`

Department of Computer Science, Swansea University, Swansea, UK

**Abstract.** For structural operational semantics (SOS) of process algebras, various notions of bisimulation have been studied, together with rule formats ensuring that bisimilarity is a congruence. For programming languages, however, SOS generally involves auxiliary entities (e.g. stores) and computed values, and the standard bisimulation and rule formats are not directly applicable.

Here, we first introduce a notion of bisimulation based on the distinction between computations and values, with a corresponding liberal congruence format. We then provide metatheory for a modular variant of SOS (MSOS) which provides a systematic treatment of auxiliary entities. This is based on a higher order form of bisimulation, and we formulate an appropriate congruence format. Finally, we show how algebraic laws can be proved sound for bisimulation with reference only to the (M)SOS rules defining the programming constructs involved in them. Such laws remain sound for languages that involve further constructs.

# Overview

## Modular semantics

- ▶ SOS (Structural Operational Semantics)
- ▶ MSOS (Modular SOS)
- ▶ I-MSOS (Implicitly-MSOS)
- ▶ Bisimulation

## Component-based semantics

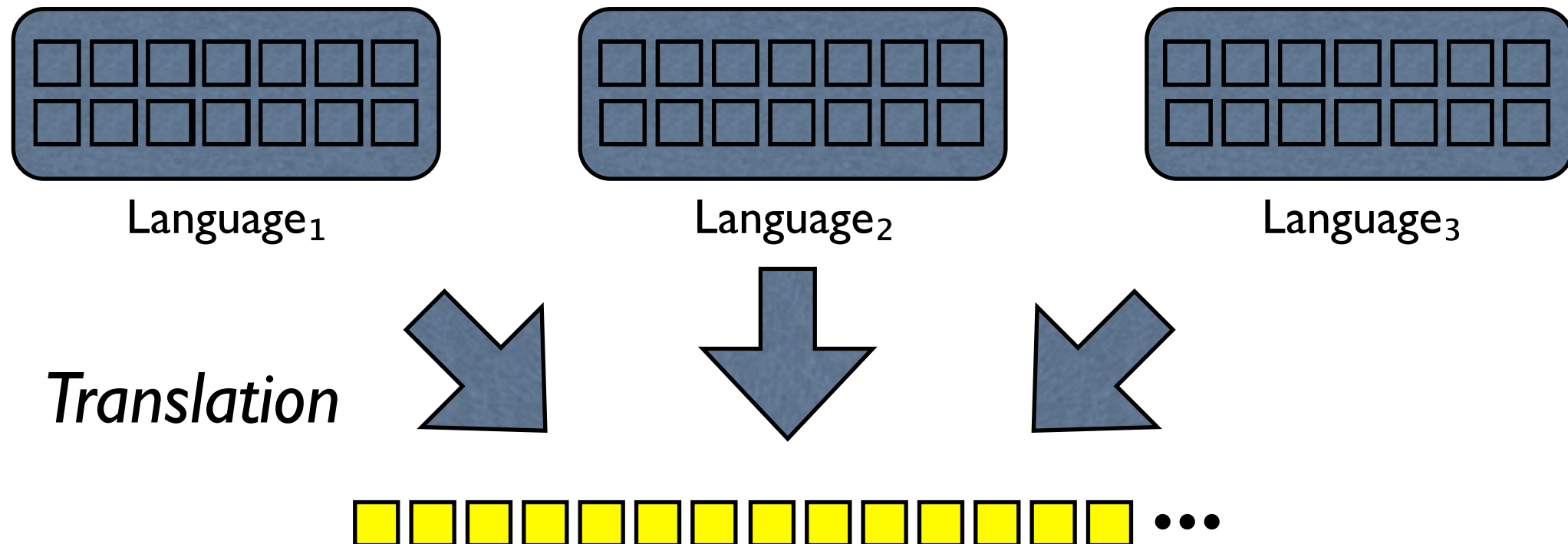
reuse !

- ▶ Funcons (fundamental programming constructs)
- ▶ Language specifications

# Component-based semantics

Reusable components of language definitions

- ▶ ***language*** constructs?
- ▶ ***kernel language*** constructs?
- ▶ ***fundamental*** programming constructs!

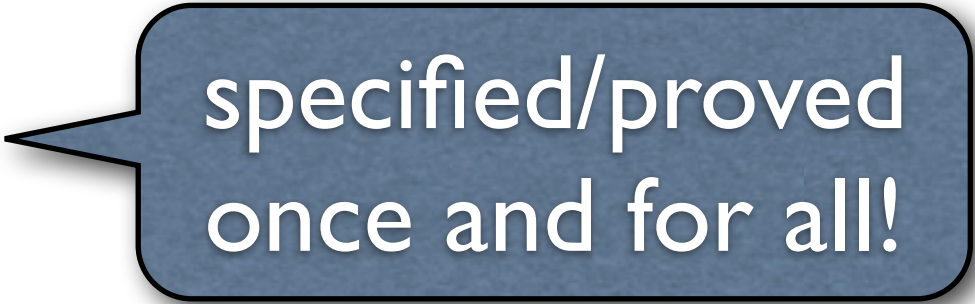


# Funcons

# Reusable components

## Fundamental constructs (funcons)

- ▶ correspond to **individual** programming constructs
  - each funcon is a separate component
- ▶ have (*when validated and released*)
  - **fixed** notation
  - **fixed** behaviour
  - **fixed** algebraic properties



specified/proved  
once and for all!



# Sorts of funcons

## Notation

### ▶ *commands*

- $C$  : computes ( )

### ▶ *declarations*

- $D$  : computes environments (mapping ids  $I$  to values  $V$  )

### ▶ *expressions*

- $E$  : computes values

## Generic funcons

- $X$  : could be commands, declarations, expressions

# Funcons: binding

- ▶ **bound-val**( $l$ )
  - gives the value bound to  $l$  in the current env
- ▶ **bind-val**( $l, E$ )
  - computes an env binding  $l$  to the value of  $E$
- ▶ **scope**( $D, X$ )
  - localises the declarations  $D$  to the execution of  $X$

# I-MSOS: **bound-val**( $I$ )

## Funcon

$\text{bound-val}(I:\text{ids}):\text{values}$

## Rule

$\text{env}(\rho) \vdash \text{bound-val}(I) \rightarrow V \quad \text{if } \rho(I) = V$

# I-MSOS: **bind-val**( $I, E$ )

## Value

$\text{bind-val}(I : \text{ids}, V : \text{values}) : \text{envs}$

## Rule

$\text{bind-val}(I, V) \rightarrow \{I \mapsto V\}$

## *implicit :*

### Funcon

$\text{bind-val}(I : \text{ids}, E : \Rightarrow \text{values}) : \text{envs}$

### Rules

$$\frac{E \rightarrow E'}{\text{bind-val}(I, E) \rightarrow \text{bind-val}(I, E')}$$

$\text{bind-val}(I, V) \rightarrow \{I \mapsto V\}$

# I-MSOS: **scope**( $D, X$ )

## Funcon

$\text{scope}(\rho : \text{envs}, X : \Rightarrow T) : T$

## Rules

$$\frac{\text{env}(\rho_0[\rho_1]) \vdash X \rightarrow X'}{\text{env}(\rho_0) \vdash \text{scope}(\rho_1, X) \rightarrow \text{scope}(\rho_1, X')}$$

$\text{scope}(\rho_1, V) \rightarrow V$

***implicit :***

## Funcon

$\text{scope}(D : \Rightarrow \text{envs}, X : \Rightarrow T) : T$

## Rules

$$\frac{D \rightarrow D'}{\text{scope}(D, X) \rightarrow \text{scope}(D', X)}$$

...

# Language specification

## *Language constructs:*

- ▶  $e ::= x \mid \text{let } x = e \text{ in } e \mid \dots$

## *Translation to funcons:*

*eval*  $\llbracket e \rrbracket$  : **expressions**

- ▶  $\text{eval } \llbracket x \rrbracket = \text{bound-val}( \text{id } \llbracket x \rrbracket )$
- ▶  $\text{eval } \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket =$   
 $\quad \text{scope}( \text{bind-val}( \text{id } \llbracket x \rrbracket, \text{eval } \llbracket e_1 \rrbracket ),$   
 $\quad \text{eval } \llbracket e_2 \rrbracket )$
- ▶ ...

# Overview

## Modular semantics

- ▶ SOS (Structural Operational Semantics)
- ▶ MSOS (Modular SOS)
- ▶ I-MSOS (Implicitly-MSOS)
- ▶ Bisimulation

## Component-based semantics

- ▶ Funcons (fundamental programming constructs)
- ▶ Language specifications

**More examples ? see Appendix**

# Preliminary tool support

## SPOOFAX/ECLIPSE

- ▶ parsing programs (SDF3, disambiguation, AST creation)
- ▶ translating ASTs to funcon terms (SDF3, STRATEGO)
- ▶ browsing and editing component-based specifications (SDF3, NABL, STRATEGO)

## PROLOG

- ▶ translating MSOS rules for funcons to PROLOG
  - *currently migrating to STRATEGO*
- ▶ running funcon terms



# PLANCOMPS project (2011-2015)

## *Foundations*

- ▶ component-based semantics [Swansea]
- ▶ GLL parsing, disambiguation [RHUL]

## *Case studies*

- ▶ CAML LIGHT, C#, JAVA [Swansea]

## *Tool support*

- ▶ IDE, funcon interpreter/compiler [RHUL, Swansea]

## *Digital library*

- ▶ interface [City], historic documents [Newcastle]

# Conclusion

**Reusable components: funcons**

to reduce the **initial** effort

**High modularity: I-MSOS**

to reduce the effort of **extension**

**Tool support: IDE**

to reduce the effort of **getting it right!**

work in progress !

# Appendix

# Fundamental constructs (funcons)

Funcons ***normally compute values***

- ▶ values compute themselves

Funcon computations may also:

- ▶ ***terminate abruptly***
  - signalling some value as the reason
  - failure is a special case
- ▶ ***never terminate***
- ▶ ***have effects***

# Funcons

Funcons specify ***computational behaviour***, including:

- ▶ normal termination
- ▶ abrupt termination
- ▶ non-termination
- ▶ effects

sort of funcon	computes	abstractions
expressions	values	functions
declarations	environments	patterns
commands	null	procedures

# Values

## *Universe*

- ▶ ***primitive*** (booleans, numbers, characters, symbols)
- ▶ ***composite*** (sequences, maps, sets, variants)
- ▶ ***types*** (names for sets of values)
- ▶ ***abstractions*** (encapsulating funcons)

*New types of values are defined in terms of old ones*

# Funcon ‘aspects’

***(Mostly) independent concerns***

- ▶ control flow
- ▶ data flow
- ▶ binding
- ▶ storing
- ▶ interacting

***each funcon has  
a primary  
‘aspect’***

# Fundamental construct design

## *Universe*

- ▶ **computes**( $T^*$ )
  - **expressions** = computes(values)
  - **declarations** = computes(envs)
  - **commands** = computes( )
  - ...



# Examples of funcons

## ***Control flow***

seq, null, enact, if-true-else, while-true,  
either, else, fail, handle-thrown, throw

## ***Data flow***

value operations, give-val, given, abs, apply

## ***Binding***

scope, bind-val, bound-val, close, override, unite, accumulate,  
recursive, match-val, case

## ***Storing***

alloc, release, assign, current-val

# Control flow

## *Normal*

- ▶ **seq**( $X_1, \dots$ )
  - left to right sequencing
  - concatenates computed values
- ▶ **null** is the empty sequence ( )
  - unit for **seq**( $X_1, X_2$ )

# Control flow

## *Conditional*

- ▶ **if-true-else**( $E, X_1, X_2$ )
  - $E$  has to be boolean-valued
- ▶ **while-true**( $E, C$ )
  - doesn't handle break or continue

## *Call*

- ▶ **enact**( $E$ )
  - evaluates  $E$  to an abstraction value **abs**( $X$ )
  - executes  $X$

# Control flow

## *Alternatives*

- ▶ **either**( $X_1, \dots$ )
  - unordered alternatives
- ▶ **else**( $X_1, \dots$ )
  - left to right alternatives
- ▶ **fail**
  - unit for **either**( $X_1, X_2$ ) and **else**( $X_1, X_2$ )
- ▶ **when-true**( $E, X$ ), **check-true**( $E$ )
  - **fail** when  $E$  false

# Data flow

**Default: arguments are pre-evaluated (strict)**

- ▶ **Funcon** *F* ( $\_ : T_1, \dots, \_ : T_N$ ) : *T*
  - *F* ( $E_1, \dots, E_N$ ) – argument computations *interleaved*
  - *F seq* ( $E_1, \dots, E_N$ ) – argument computations *left-to-right*

**Explicit: arguments not pre-evaluated**

- ▶ **Funcon** *seq* ( $\_ : \Rightarrow T_1, \dots, \_ : \Rightarrow T_N$ ) : ( $T_1, \dots, T_N$ )
  - like *call-by-name* parameters in SCALA

# Control and data flow

## *Giving a value*

- ▶ **give-val**( $E, X$ )
  - first evaluates  $E$  to a value  $V$
  - then executes  $X$ , with the funcon **given** referring to  $V$
- ▶ **given**

## *Discarding a value*

- ▶ **effect**( $X$ )
  - executes  $X$ , but computes ( )

# Control and data flow

## *Abstraction*

- ▶ **abs**( $X$ )
  - procedural abstraction
  - a value, so context-independent

## *Application*

- ▶ **apply**( $E_1, E_2$ )
  - evaluates  $E_1$  to an abstraction **abs**( $X$ ), and evaluates  $E_2$  to a value  $V$
  - then executes  $X$ , with the funcon **given** referring to  $V$

# Control and data flow

## *Exception handling*

- ▶ **handle-thrown**( $X_1, X_2$ )
  - try to handle abrupt termination of  $X_1$  by giving the thrown value to the execution of  $X_2$
- ▶ **throw-val**( $E$ )
  - terminates abruptly, throwing the value of  $E$

## *Continuations*

- ▶ see the paper by Neil Sculthorpe et al. at the ETAPS 2015 *Workshop on Continuations*



# Binding

## Scopes

- ▶ **scope**( $D, X$ )
  - localises the bindings computed by  $D$  to  $X$
- ▶ **bind-val**( $I, E$ )
  - computes the binding of the id  $I$  to the value of  $E$
- ▶ **bound-val**( $I$ )
  - inspects the current binding of the id  $I$

# Binding

## Scopes

- ▶ **override**( $D_1, D_2$ )
- ▶ **unite**( $D_1, D_2$ )
- ▶ **accumulate**( $D_1, D_2$ )
- ▶ **recursive**( $lset, D$ )
  - various ways of composing declarations

# Binding

## *Scopes in abstractions*

- ▶ **close**( $E$ )
  - evaluates  $E$  to an abstraction **abs**( $X$ )
  - returns the *closure* incorporating the current bindings

## *Patterns*

- ▶ *simple*: abstractions **abs**( $D$ )
- ▶ *composite*: formed using value *constructors*
  - structure required to be identical when matching

# Binding

## *Pattern matching*

- ▶ **match-val**( $E_1, E_2$ )
  - evaluates  $E_1$  to a pattern  $P$  and  $E_2$  to a value  $V$
  - matching  $P$  to  $V$  computes bindings or fails
- ▶ **case**( $E, X$ )
  - evaluates  $E$  to a pattern  $P$ ,  
then matches  $P$  to a given value
  - the scope of the computed bindings is  $X$
  - equivalent to **scope**(**match**( $E$ , **given**),  $X$ )

# Storing

## ***Variables***

- ▶ *simple*: representing independent storage locations
  - for storing values of a fixed type
  - monolithic update
- ▶ *composite*: formed using value *constructors*
  - component variables can be independently updated
  - structure required to be identical when updating

# Storing

## *Variable allocation*

### ▶ **alloc**( $E_1, E_2$ )

- evaluates  $E_1$  to a type  $T$ , and  $E_2$  to a value  $V$
- allocates a simple or composite variable for storing values of type  $T$
- assigns  $V$  to the variable

### ▶ **release**( $E$ )

- evaluates  $E$  to a variable
- terminates the allocation of the variable

# Storing

## ***General assignment***

- ▶ **assign**( $E_1, E_2$ )
  - evaluates  $E_1$  to  $V_1$ , and  $E_2$  to  $V_2$
  - when  $V_1$  and  $V_2$  have the same structure, updates the stored values of any simple variables in  $V_1$  by the corresponding component values of  $V_2$
- ▶ **current-val**( $E$ )
  - evaluates  $E$  to  $V$
  - gives the value formed by replacing any simple variables in  $V$  by their stored values

# Funcon reuse

## *Language construct:*

$s ::= \text{while}(e)s$

## *Translation to funcons:*

$\text{exec } \llbracket \text{while}(e)s \rrbracket =$   
 $\text{while-true}(\text{current-val}(\text{eval } \llbracket e \rrbracket), \text{exec } \llbracket s \rrbracket)$

## *For languages with break statements:*

$\text{exec } \llbracket \text{while}(e)s \rrbracket =$   
 $\text{handle-thrown}(\text{while-true}(\text{current-val}(\text{eval } \llbracket e \rrbracket), \text{exec } \llbracket s \rrbracket),$   
 $\text{case}(\text{'break'}, \text{null}))$



# Funcon reuse

## *Language construct:*

►  $e ::= e \text{ ? } e : e$

## *Translation to funcons:*

►  $\text{eval } \llbracket e_1 \text{ ? } e_2 : e_3 \rrbracket =$   
 $\text{if-true-else } ( \text{eval } \llbracket e_1 \rrbracket, \text{eval } \llbracket e_2 \rrbracket, \text{eval } \llbracket e_3 \rrbracket )$

## *For languages with non-Boolean tests:*

►  $\text{eval } \llbracket e_1 \text{ ? } e_2 : e_3 \rrbracket =$   
 $\text{if-true-else } ( \text{not } ( \text{equal } ( \text{eval } \llbracket e_1 \rrbracket, 0 ) ),$   
 $\text{eval } \llbracket e_2 \rrbracket, \text{eval } \llbracket e_3 \rrbracket )$

# Funcon reuse

## *Language construct:*

- ▶  $e ::= \text{if } e \text{ then } e \text{ else } e$

## *Translation to funcons:*

- ▶  $\text{eval } \llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket =$   
 $\text{if-true-else } ( \text{eval } \llbracket e_1 \rrbracket, \text{eval } \llbracket e_2 \rrbracket, \text{eval } \llbracket e_3 \rrbracket )$

## *For languages with non-Boolean tests:*

- ▶  $\text{eval } \llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket =$   
 $\text{if-true-else } ( \text{not } ( \text{equal } ( \text{eval } \llbracket E_1 \rrbracket, 0 ) ),$   
 $\text{eval } \llbracket E_2 \rrbracket, \text{eval } \llbracket E_3 \rrbracket )$

# Funcon reuse

## *Language construct:*

►  $s ::= \text{if}(e) \ s \ \text{else} \ s$

## *Translation to funcons:*

►  $\text{exec} \llbracket \text{if}(e_1) \ s_2 \ \text{else} \ s_3 \rrbracket =$   
 $\text{if-true-else} \ ( \text{eval} \llbracket e_1 \rrbracket, \text{exec} \llbracket s_2 \rrbracket, \text{exec} \llbracket s_3 \rrbracket )$

## *For languages with non-Boolean tests:*

►  $\text{exec} \llbracket \text{if}(e_1) \ s_2 \ \text{else} \ s_3 \rrbracket =$   
 $\text{if-true-else} \ ( \text{not} \ ( \text{equal} \ ( \text{eval} \llbracket e_1 \rrbracket, 0 ) ),$   
 $\text{exec} \llbracket s_2 \rrbracket, \text{exec} \llbracket s_3 \rrbracket )$

# Funcon reuse

## *Language construct:*

- ▶  $s ::= \text{if}(e) \ s$ 
  - |  $s \ s \ \dots$
  - |  $\{ \}$

## *Translation to funcons:*

- ▶  $\text{exec} \llbracket \text{if}(e) \ s \rrbracket = \text{exec} \llbracket \text{if}(e) \ s \ \text{else} \ \{ \} \rrbracket$
- ▶  $\text{exec} \llbracket s_1 \ s_2 \ \dots \rrbracket = \text{seq} \ ( \text{exec} \llbracket s_1 \rrbracket, \text{exec} \llbracket s_2 \ \dots \rrbracket )$
- ▶  $\text{exec} \llbracket \{ \} \rrbracket = \text{null}$

# Funcon reuse

## Language construct:

►  $s ::= \{ d \ s \}$   
     $\mid i = e ;$   
 $e ::= i$

## Translation to funcons:

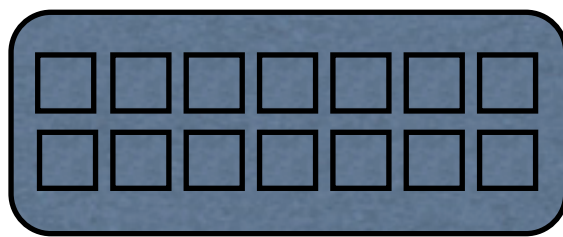
- $\text{exec } \llbracket \{ d \ s \} \rrbracket = \text{scope } ( \text{elab } \llbracket d \rrbracket, \text{exec } \llbracket s \rrbracket )$
- $\text{exec } \llbracket i = e ; \rrbracket =$   
     $\text{assign } ( \text{bound-val } ( \text{id } \llbracket i \rrbracket ), \text{eval } \llbracket e \rrbracket )$
- $\text{eval } \llbracket i \rrbracket =$   
     $\text{current-val } ( \text{bound-val } ( \text{id } \llbracket i \rrbracket ) )$

# High modularity

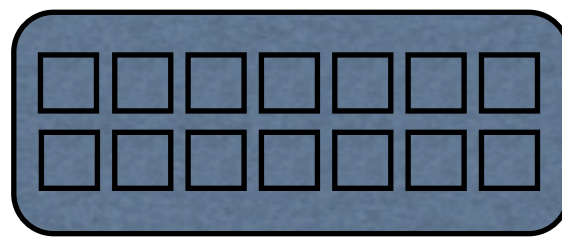
# Component-based semantics

Reusable components of language definitions

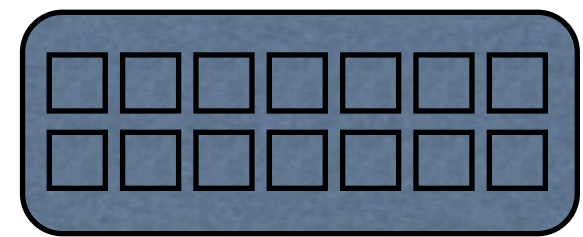
- ***fundamental*** programming constructs



Language<sub>1</sub>

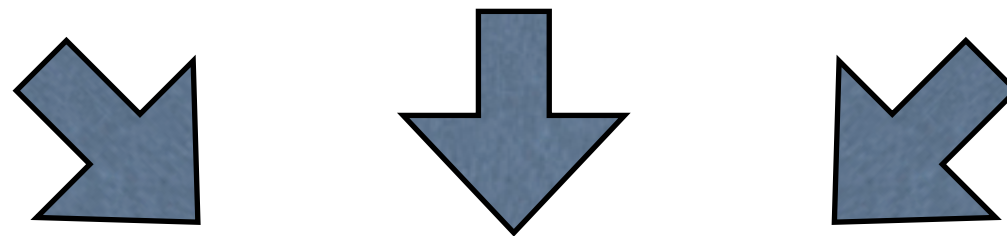



Language<sub>2</sub>



Language<sub>3</sub>

*Translation*



***Flat structure***  ***open-ended***

***Moderated – no versioning!***

# Tool support



# Preliminary tool support

## SPOOFAX/ECLIPSE

- ▶ parsing programs (SDF3, disambiguation, AST creation)
- ▶ translating ASTs to funcon terms (SDF3, STRATEGO)
- ▶ browsing and editing component-based specifications (SDF3, NABL, STRATEGO)

## PROLOG

- ▶ translating MSOS rules for funcons to PROLOG
  - *currently migrating to STRATEGO*
- ▶ running funcon terms

# Future tool support

ESOP'14:

- ▶ refocusing  
small-step  
(M)SOS rules

## Deriving Pretty-Big-Step Semantics from Small-Step Semantics

Casper Bach Poulsen and Peter D. Mosses

Department of Computer Science, Swansea University, Swansea, UK,  
`cscbp@swansea.ac.uk`, `p.d.mosses@swansea.ac.uk`

**Abstract.** Big-step semantics for languages with abrupt termination and/or divergence suffer from a serious duplication problem, addressed by the novel ‘pretty-big-step’ style presented by Charguéraud at ESOP’13. Such rules are less concise than corresponding small-step rules, but they have the same advantages as big-step rules for program correctness proofs. Here, we show how to automatically derive pretty-big-step rules directly from small-step rules by ‘refocusing’. This gives the best of both worlds: we only need to write the relatively concise small-step specifications, but our reasoning can be big-step as well as small-step. The use of strictness annotations to derive small-step congruence rules gives further conciseness.

# Alternative tool support

WRLA'14:

- using the K framework and tools

## FunKons: Component-Based Semantics in K

Peter D. Mosses and Ferdinand Vesely(✉)

Swansea University, Swansea SA2 8PP, UK  
`{p.d.mosses,csfvesely}@swansea.ac.uk`

**Abstract.** Modularity has been recognised as a problematic issue of programming language semantics, and various semantic frameworks have been designed with it in mind. Reusability is another desirable feature which, although not the same as modularity, can be enabled by it. The K Framework, based on Rewriting Logic, has good modularity support, but reuse of specifications is not as well developed.

The PPlanCompS project is developing a framework providing an open-ended collection of reusable components for semantic specification. Each component specifies a single fundamental programming construct, or ‘funcon’. The semantics of concrete programming language constructs is given by translating them to combinations of funcons. In this paper, we show how this component-based approach can be seamlessly integrated with the K Framework. We give a component-based definition of CinK (a small subset of C++), using K to define its translation to funcons as well as the (dynamic) semantics of the funcons themselves.

# Alternative tool support

RTA'15:

- ▶ using a big-step variant of I-MSOS

## DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology  
Delft, The Netherlands

`v.a.vergu@tudelft.nl`, `p.j.m.neron@tudelft.nl`, `visser@acm.org`

---

### Abstract

The formal definition the semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules*. DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.