Reusable Components of Semantic Specifications

Martin Churchill, Peter Mosses, Paolo Torrini
Swansea University, UK

MODULARITY'14: 22–25 April 2014, Lugano, Switzerland

MODULARITY – A Good Thing!

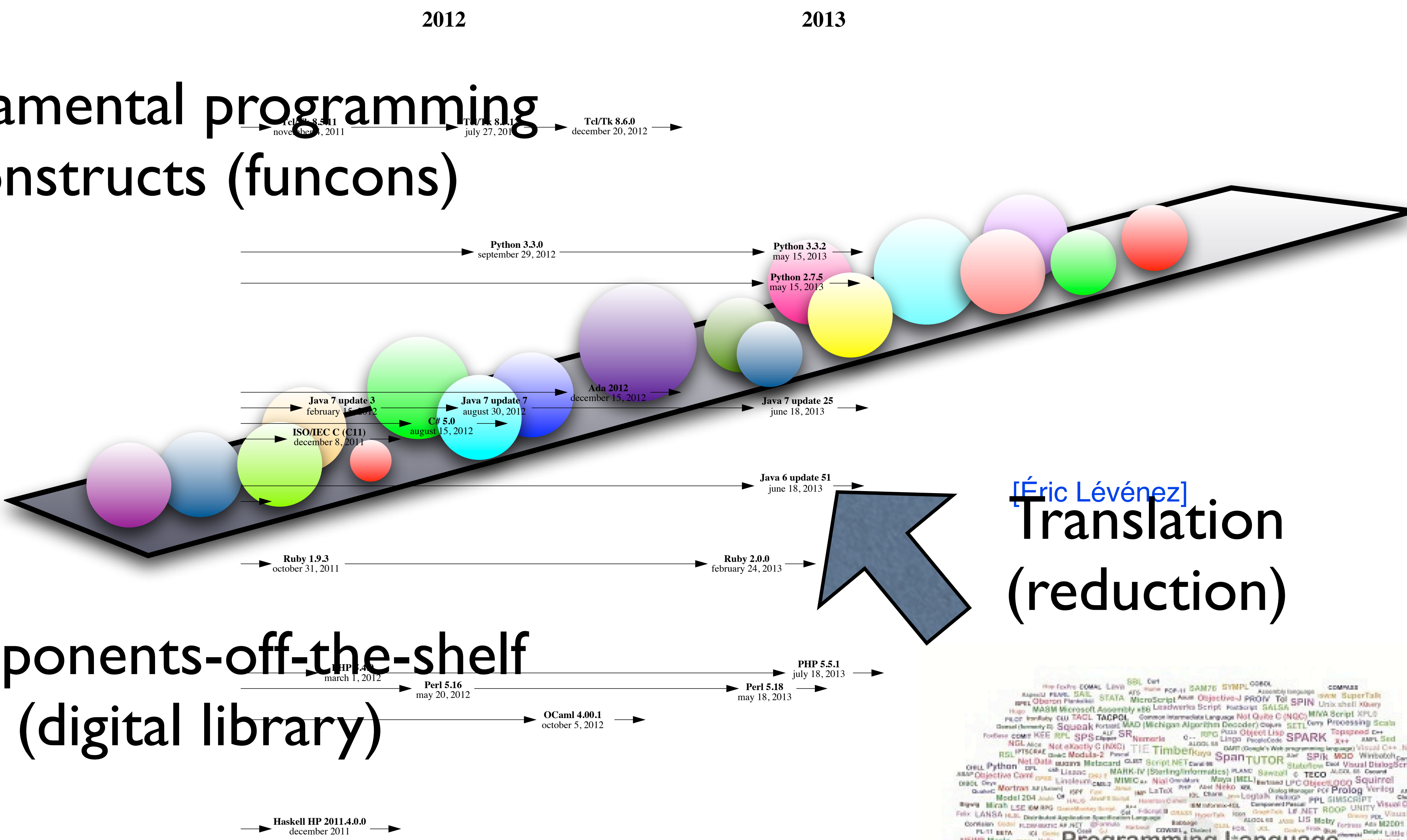
Our paper

- **modular framework**: component-based semantics
- **preliminary case study**: CAML LIGHT

Our project

- PLANCOMPS [www.plancomps.org]
  - *Programming Language Components and Specifications*
- **testing component reusability**
  - *major case studies*: C#, JAVA, …
- developing a **language <u>specifier</u>'s workbench**

Component-based semantics



Fundamental programming constructs (funcons)

Components-off-the-shelf (digital library)

[Éric Lévénez]

Translation (reduction)

Evolving languages

Reusable components

Fundamental constructs (funcons)

- correspond to programming constructs
  - ***directly*** (**if-true**), or
  - ***special case*** (**apply**), or
  - ***implicit*** (**bound-value**)
- and have *(when validated and released)*
  - ***fixed*** notation, and
  - ***fixed*** behaviour, and
  - ***fixed*** algebraic properties

specified/proved once and for all!

Component reuse

Language construct:

- $exp ::= exp \mathbf{\ ?\ } exp \mathbf{\ :\ } exp$

Translation to funcons:

- $expr[\![E_1 \mathbf{\ ?\ } E_2 \mathbf{\ :\ } E_3]\!] =$

  if-true($expr[\![E_1]\!]$, $expr[\![E_2]\!]$, $expr[\![E_3]\!]$)

For languages with non-Boolean tests:

- $expr[\![E_1 \mathbf{\ ?\ } E_2 \mathbf{\ :\ } E_3]\!] =$

  if-true(**not**(**equal**($expr[\![E_1]\!]$, 0)),

  $expr[\![E_2]\!]$, $expr[\![E_3]\!]$)

Component reuse

Language construct:

- $\mathit{stm} ::= \texttt{if}\,\texttt{(}\mathit{exp}\texttt{)}\ \mathit{stm}\ \texttt{else}\ \mathit{stm}$

Translation to funcons:

- $\mathit{comm}[\![\ \texttt{if}\,\texttt{(}E_1\texttt{)}\ S_2\ \texttt{else}\ S_3\ ]\!] =$
  $\textbf{if-true}(\mathit{expr}[\![\ E_1\ ]\!], \mathit{comm}[\![\ S_2\ ]\!], \mathit{comm}[\![\ S_3\ ]\!])$

For languages with non-Boolean tests:

destructive change

- $\mathit{comm}[\![\ \texttt{if}\,\texttt{(}E_1\texttt{)}\ S_2\ \texttt{else}\ S_3\ ]\!] =$
  $\textbf{if-true}(\textbf{not}(\textbf{equal}(\mathit{expr}[\![\ E_1\ ]\!], 0)),$
  $\mathit{comm}[\![\ S_2\ ]\!], \mathit{comm}[\![\ S_3\ ]\!])$

Component specification

modular extension

Notation

$$\textbf{if-true}(\text{boolean}, \text{comp}(T), \text{comp}(T)) : \text{comp}(T)$$

Static semantics

$$\frac{E : \textbf{boolean}, \quad X_1 : T, \quad X_2 : T}{\textbf{if-true}(E, X_1, X_2) : T}$$

specified once and for all!

Dynamic semantics

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \rightarrow X_1$$

$$\textbf{if-true}(\textbf{false}, X_1, X_2) \rightarrow X_2$$

This talk

Reusable components:

- ***fundamental constructs (funcons)***
  - notation
  - semantics

Component-based semantics:

- ***translation to funcons***
  - illustrative examples
  - introduction to CAML LIGHT case study

Funcon notation – examples

Sorts of funcons

- **comm** = **comp(skip)**
- **decl** = **comp(env)**
- **expr** = **comp(value)**
- **...**

T <: comp(T)

- **comp(T)** – *funcons computing values of type T*
  – SCALA: ⇒T

Funcon notation – examples

Types of values

- **boolean**, **int**, **atom**, …
- **list**(*S*), **map**(*S*, *T*), …
- **array**, **record**, **tuple**, …
- **abs**(*S*, *T*)
  - **func** = **abs**(value, env), **patt** = **abs**(value, env), …

Abstract types (language-dependent)

- **value**, **env**, **var**, **store**, …

Funcon notation – examples

Control flow funcons – comm = comp(skip)

- **seq**(skip, comp(*T*)) : comp(*T*)
- **skip** : skip
- **if-true**(boolean, comp(*T*), comp(*T*)) : comp(*T*)
- **while-true**(comp(boolean), comm) : comm

value sorts

Funcon notation – examples

Binding and scoping funcons – decl = comp(env)

- **scope**(env, comp(*T*)) : comp(*T*)
- **bind-value**(id, value) : env
- **bound-value**(id) : expr

Function abstraction and application

- **abs**(patt, expr) : func
- **apply**(func, value) : expr
- **close**(func) : comp(func)

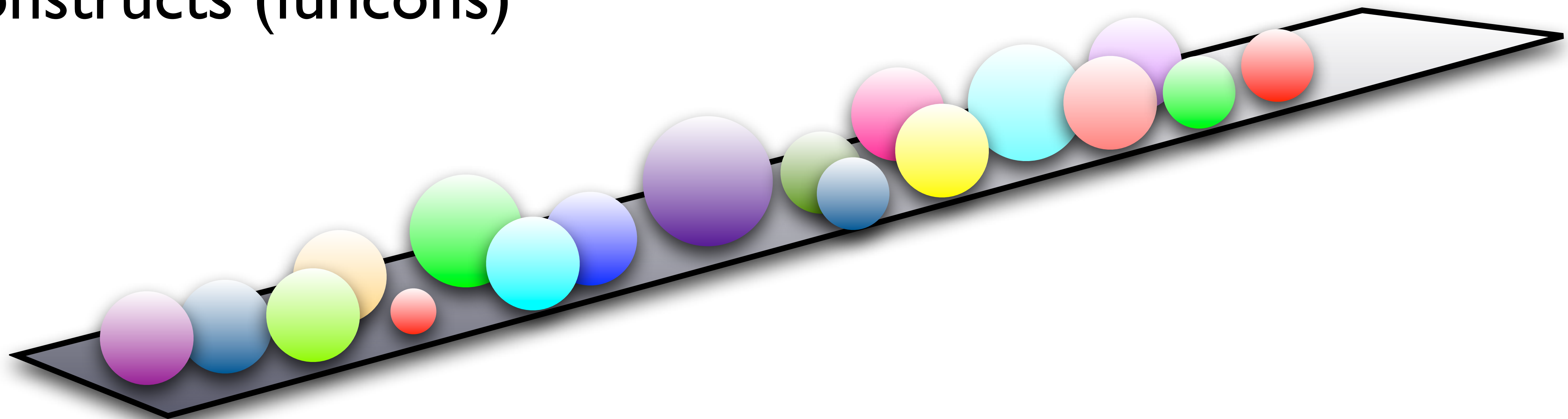
Funcon notation – examples

Storing funcons

- **allocate**(value) : comp(var)
- **assigned-value**(var) : expr
- **assign**(var, value) : comm

Funcon notation

Fundamental programming constructs (funcons)



This talk

Reusable components:

- ***fundamental constructs (funcons)***
  - ✓ notation
  - ➡ semantics

Component-based semantics:

- ***translation to funcons***
  - illustrative examples
  - introduction to CAML LIGHT case study

Funcon semantics – format

Notation *(algebraic signature):*

$$\textbf{Funcon}(Sort_1, \ldots) : \; Sort$$

Static semantics *(context-sensitive)*

$$\frac{Var_1 : Type_1, \quad \ldots}{\textbf{Funcon}(Var_1, \ldots) : Type}$$

Dynamic semantics *(transition system)*

$$\frac{Var \quad \rightarrow \quad Var'}{\textbf{Funcon}(Term_1, \ldots) \quad \rightarrow \quad Term'}$$

Funcon semantics – features

Aims:

- stable
- concise
- modular

Means:

- ***I-MSOS*** – implicit propagation of auxiliary entities
- ***lifting*** – implicit rules for computing expression values
- ***rule format*** – bisimulation congruence, preservation

Funcon semantics – examples

$\textbf{if-true}(\text{boolean}, \text{comp}(T), \text{comp}(T)) : \text{comp}(T)$

$$\frac{E : \text{boolean}, \quad X_1 : T, \quad X_2 : T}{\textbf{if-true}(E, X_1, X_2) : T}$$

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \rightarrow X_1$$

$$\textbf{if-true}(\textbf{false}, X_1, X_2) \rightarrow X_2$$

$$\frac{E \rightarrow E'}{\textbf{if-true}(E, X_1, X_2) \rightarrow \textbf{if-true}(E', X_1, X_2)}$$

Implicit!

Funcon semantics – examples

$\textbf{seq}(\text{skip}, \text{comp}(T)) : \text{comp}(T)$

$$\frac{C : \textbf{comm}, \quad X : T}{\textbf{seq}(C, X) : T}$$

$$\textbf{seq}(\textbf{skip}, X) \rightarrow X$$

$$\frac{C \rightarrow C'}{\textbf{seq}(C, X) \rightarrow \textbf{seq}(C', X)}$$

Implicit!

Funcon semantics – examples

bound-value(id) : expr

env $\Gamma \vdash$ **bound-value**(I) : $\Gamma(I)$

env $\rho \vdash$ **bound-value**(I) $\rightarrow$ $\rho(I)$

Funcon semantics – examples

$\mathbf{scope}(\text{env}, \text{comp}(T)) : \text{comp}(T)$

$$\frac{\text{env } \Gamma \vdash D : \Gamma_1, \qquad \text{env } (\Gamma_1/\Gamma) \vdash X : T}{\text{env } \Gamma \vdash \mathbf{scope}(D, X) : T}$$

$$\frac{\text{env } (\rho_1/\rho) \vdash X \to X'}{\text{env } \rho \vdash \mathbf{scope}(\rho_1, X) \to \mathbf{scope}(\rho_1, X')}$$

$$\frac{D \to D'}{\mathbf{scope}(D, X) \to \mathbf{scope}(D', X)}$$

Implicit!

This talk

Reusable components:

- ***fundamental constructs (funcons)***
  - ✓ notation
  - ✓ semantics

Component-based semantics:

- ***translation to funcons***
  - ➡ illustrative examples
  - \- introduction to CAML LIGHT case study

Language specifications

Syntax

- context-free
- concrete ↔ abstract

Semantics

- *translation*⟦ *abstract syntax sort* ⟧ : **funcon sort**
- specified inductively by equations
- induces both static and dynamic semantics
  - *relationship adjustable by adding 'static funcons'*

Component-based semantics – examples

Translation function

- *comm*⟦ *stm* ⟧ : **comm**

Translation equations

- *stm* ::= **{ }**
  - *comm*⟦ **{ }** ⟧ = **skip**
- *stm* ::= *stm* stm^+
  - *comm*⟦ S_1 S_2 … ⟧ = **seq**(*comm*⟦ S_1 ⟧, *comm*⟦ S_2 … ⟧)

Component-based semantics – examples

Translation functions

- $comm⟦ stm ⟧$: **comm**
- $expr⟦ exp ⟧$: **expr**

Translation equations

- *stm* ::= **`if`**`(`*exp*`)` *stm* **`else`** *stm*
  - $comm⟦$ **`if`**`(`E`)` S_1 **`else`** S_2 $⟧$ =
    if-true($expr⟦ E ⟧$, $comm⟦ S_1 ⟧$, $comm⟦ S_2 ⟧$)
- *stm* ::= **`if`**`(`*exp*`)` *stm*
  - $comm⟦$ **`if`**`(`E`)` S $⟧$ = $comm⟦$ **`if`**`(`E`)` S **`else {}`** $⟧$

Component-based semantics – examples

Translation functions

- *comm*⟦ *stm* ⟧ : **comm**
- *expr*⟦ *exp* ⟧ : **expr**

Translation equations

- *stm* ::= *id* **=** *exp* **;**
  - *comm*⟦ *I* **=** *E* **;** ⟧ =
    assign(**bound-value**(*I*), *expr*⟦ *E* ⟧)
- *exp* ::= *id*
  - *expr*⟦ *I* ⟧ = **assigned-value**(**bound-value**(*I*))

This talk

Reusable components:

- ***fundamental constructs (funcons)***
  - ✓ notation
  - ✓ semantics

Component-based semantics:

- ***translation to funcons***
  - ✓ illustrative examples
  - ➡ introduction to CAML LIGHT case study

Case study: CAML LIGHT

A pedagogical functional programming language

- a sub-language of CAML
  - *some constructs differ a bit from OCAML*
- similar to the Core of STANDARD ML
  - *except for order of evaluation!*
- higher-order, polymorphic, pattern-matching, …
- references, mutable arrays, mutable record fields, …
- abstract syntax defined in the reference manual

Case study: CAML LIGHT

Introduction

- section 3 of the paper

Full specification

- available online [www.plancomps.org/churchill2014]

(Incomplete) validation using test programs

- parser generated from abstract syntax grammar (in SDF2)
- translation to funcons implemented (in ASF+SDF)
- interpreter (in PROLOG) generated from I-MSOS rules

Needs polishing and further testing…

Conclusion

Funcons – A Good Thing!

- **reusable components** of semantic specifications
- each funcon **specified once and for all**
  - *I-MSOS, lifting, implicit rules*
- optimal(?) **abstraction level**
  - *simple translations*
  - *simple rules*

But further case studies are needed to prove it

- C#, JAVA, DSLs, **...**

Appendix

Funcon semantics – examples

assigned-value(var) : expr

$$\frac{E : \textbf{var}(T)}{\textbf{assigned-value}(E) : T}$$

$$(\textbf{assigned-value}(V), \text{store } \sigma) \to (\sigma(V), \text{store } \sigma)$$

$$\frac{E \to E'}{\textbf{assigned-value}(E) \to \textbf{assigned-value}(E')}$$

Implicit!

Funcon semantics – examples

assign(var, value) : expr

$$\frac{E_1 : \mathbf{var}(T), \qquad E_2 : T}{\mathbf{assign}(E_1, E_2) : \mathbf{comm}}$$

$$(\mathbf{assign}(V_1, V_2), \text{store } \sigma) \rightarrow (\mathbf{comm}, \text{store } \sigma[V_1 \mapsto V_2])$$

$$\frac{E_1 \rightarrow E_1'}{\mathbf{assign}(E_1, E_2) \rightarrow \mathbf{assign}(E_1', E_2)}$$

Implicit!

$$\frac{E_2 \rightarrow E_2'}{\mathbf{assign}(E_1, E_2) \rightarrow \mathbf{assign}(E_1, E_2')}$$

Implicit!

Funcon notation – examples

Data flow funcons

- value <: expr – *computed values*
- *lifted value operations*
  - **not**(boolean) : boolean ➦
    not(expr) : expr
  - **equal**(boolean, boolean) : boolean ➦
    equal(expr, expr) : expr
- *use of previously computed value*
  - **supply**(expr, comp(*X*)) : comp(*X*)
  - **given** : expr