

Component-Based Semantics

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

IFIP WG 2.2 Meeting, Bordeaux, September 2017

Component-based semantics


Aims

- ▶ encourage language developers to use formal semantics

Means

- ▶ modularity
- ▶ reusable components
- ▶ tool support

Related work

- ▶ Action Semantics
- ▶ ASM: Abstract State Machines
- ▶ PLT Redex / Racket 
- ▶ \mathbb{K} -framework
- ▶ **{S}** spoofax
- ▶ PLANCOMPS project, collaborators:



Neil Sculthorpe



Thomas van
Binsbergen

Component-based semantics

Components are ***fundamental constructs*** (***‘funcons’***)

- defined operationally, e.g.:

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

$$\frac{\rho_1 / \rho_0 \vdash X \rightarrow X'}{\rho_0 \vdash \mathbf{scope}(\rho_1, X) \rightarrow \mathbf{scope}(\rho_1, X')}$$

$$\mathbf{scope}(\rho_1, V) \rightsquigarrow V$$

Component-based semantics

Translation of language constructs to funcons

- ▶ e.g. (Caml Light) :

```

$$E : \text{expr} ::= \dots \mid \text{value-defs in expr} \mid \dots$$

$$\text{eval } \llbracket E:\text{expr} \rrbracket : \Rightarrow \text{values}$$

$$\dots$$

$$\text{eval } \llbracket VD \text{ in } E \rrbracket = \mathbf{scope}(\text{decl } \llbracket VD \rrbracket, \text{eval } \llbracket E \rrbracket)$$

$$\dots$$

$$VD : \text{value-defs} ::= \dots$$

$$\text{decl } \llbracket VD:\text{value-defs} \rrbracket : \Rightarrow \text{envs}$$

$$\dots$$

```

Funcons: **Characteristics**

- ▶ correspond to *fundamental programming concepts*
- ▶ language-*independent*
- ▶ have *fixed* behaviour
- ▶ *new* ones can be added

Funcons: **Foundations**

Modular SOS (MSOS)

- *Proc. MFCS*, 1999; *J.LAP*, 2004

Implicitly-Modular SOS (I-MSOS)

- *Proc. SOS*, 2008 (with M.New)

Value-computation specifications, bisimulation congruence format

- *Proc. FoSSaCS*, 2013 (with M.Churchill)

Signatures with strictness annotations

- *Trans. AOSD*, 2015 (with M.Churchill, N.Sculthorpe, P.Torrini)

Funcons: **Foundations**

Structural operational semantics (SOS)

$$\frac{\rho_0 \vdash \langle D, \sigma \rangle \rightarrow \langle D', \sigma' \rangle}{\rho_0 \vdash \langle \mathbf{scope}(D, X), \sigma \rangle \rightarrow \langle \mathbf{scope}(D', X), \sigma' \rangle}$$
$$\frac{\rho_1 / \rho_0 \vdash \langle X, \sigma \rangle \rightarrow \langle X', \sigma' \rangle}{\rho_0 \vdash \langle \mathbf{scope}(\rho_1, X), \sigma \rangle \rightarrow \langle \mathbf{scope}(\rho_1, X'), \sigma' \rangle}$$
$$\rho_0 \vdash \langle \mathbf{scope}(\rho_1, V), \sigma \rangle \rightarrow \langle V, \sigma \rangle$$

Funcons: **Foundations**

Implicitly-Modular SOS (I-MSOS)

$$\frac{D \rightarrow D'}{\mathbf{scope}(D, X) \rightarrow \mathbf{scope}(D', X)}$$
$$\frac{\rho_1 / \rho_0 \vdash X \rightarrow X'}{\rho_0 \vdash \mathbf{scope}(\rho_1, X) \rightarrow \mathbf{scope}(\rho_1, X')}$$
$$\mathbf{scope}(\rho_1, V) \rightarrow V$$

Funcons: **Foundations**

Value-computation specifications, bisimulation congruence format

$$\frac{D \rightarrow D'}{\mathbf{scope}(D, X) \rightarrow \mathbf{scope}(D', X)}$$
$$\frac{\rho_1 / \rho_0 \vdash X \rightarrow X'}{\rho_0 \vdash \mathbf{scope}(\rho_1, X) \rightarrow \mathbf{scope}(\rho_1, X')}$$
$$\mathbf{scope}(\rho_1, V) \rightsquigarrow V$$

Funcons: **Foundations**

Signatures with strictness annotations

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

$$\frac{\rho_1 / \rho_0 \vdash X \rightarrow X'}{\rho_0 \vdash \mathbf{scope}(\rho_1, X) \rightarrow \mathbf{scope}(\rho_1, X')}$$

$$\mathbf{scope}(\rho_1, V) \rightsquigarrow V$$

Funcons: **Values**

Universe

- ***algebraic data types:*** booleans, lists, tuples, ...
- ***built-in types:*** numbers, sets, maps, types, ...
- ***none*** : no-value – represents the *lack* of an ordinary value

Types

- ***Boolean algebra:*** union, intersection, complement, $S <: T$

Funcons: **Computations**

Control flow

- sequencing, interleaving, choosing, iterating, ...

Data flow

- giving, binding, storing, interacting, generating, ...
- throwing / handling, delimited continuations ...

Funcons: **Abstractions**

Values encapsulating computations

- ▶ thunks, functions, procedures, methods, patterns, ...
- ▶ open, closures
- ▶ forcing, applying, composing, ...

Funcon descriptions of
programming concepts

— Examples —

Examples of funcon descriptions

Operand evaluation order

Funcon **and**(B₁: booleans, B₂: booleans) : booleans

- ▶ *interleaved*: **and**(B₁, B₂)
- ▶ *sequential*: **and**(**left-to-right**(B₁, B₂))
- ▶ *explicit*: **give**(B₂, **and**(B₁, **given**))
- ▶ *conditional*: **if-then-else**(B₁, B₂, **false**)

Examples of funcon descriptions

Unbounded and bounded arithmetic

Funcon **integer-add**(N_1 : integers, N_2 : integers) : integers
 integer-subtract(N_1 : integers, N_2 : integers) : integers
 ...

Funcon **short-integer**(N : integers) : bounded-integers(...,...)

- ▶ **short-integer**(**integer-add**(N_1 , N_2))
 short-integer(**integer-subtract**(N_1 , N_2))
 ...

Examples of funcon descriptions

Partial arithmetic operations

Funcon **integer-divide**(N_1 : integers, N_2 : integers) :
integers|no-value

Funcon **definitely**(V : T|no-value) : $\Rightarrow T$

Rule **definitely**(V : values) $\leadsto V$

Rule **definitely**(none) \leadsto **fail**

- ▶ **integer-add**(1, **definitely**(**integer-divide**(N_1 , N_2)))

Examples of funcon descriptions

Declarations compute environments

Type envs \leadsto maps(ids, values|no-value)

Funcon **bind**(l: ids, V: values) : envs

bound(l: ids) : \Rightarrow values

scope(p: envs, X: \Rightarrow T) : \Rightarrow T

- ▶ *local declaration*: **scope**(**bind**(l, E), ...**bound**(l)...))

Examples of funcon descriptions

Recursive and forward declarations

Funcon **recursively-bind**(l : ids, E : \Rightarrow values) : \Rightarrow environments

- ▶ bind l to a fresh link
- ▶ in the scope of that binding:
 - bind l to the value of E
 - set the link to refer to the value of E

Examples of funcon descriptions

Abstractions with static or dynamic binding

Funcon **abstraction**($X: \Rightarrow T$) : $\text{abstractions}(T)$

close($A: \text{abstractions}(T)$) : $\Rightarrow \text{abstractions}(T)$

- ▶ *dynamic bindings*: **bind**(l , **abstraction**(X))
- ▶ *static bindings*: **bind**(l , **close**(**abstraction**(X)))

Funcon **closure**($X: \Rightarrow T$) : $\Rightarrow \text{abstractions}(T)$

\leadsto **close**(**abstraction**(X))

Examples of funcon descriptions

Abstractions with a call-by-value or -name argument

Funcon **force**(A: abstractions(T)) : $\Rightarrow T$

- ▶ *call-by-value*: ...**bind**(l, **closure**(...**given**...))...
...**apply**(**bound**(l), E)...
- ▶ *call-by-name*: ...**bind**(l, **closure**(...**force**(**given**)...))...
...**apply**(**bound**(l), **closure**(E))...

Examples of funcon descriptions

Further programming concepts

- ▶ patterns
- ▶ variables
- ▶ input/output
- ▶ handling abrupt termination
- ▶ delimited continuations

Tool support

Prototype, implemented in Spoofax and Haskell

- ▶ editing and parsing
- ▶ checking translation and transition rules
- ▶ navigating and browsing
- ▶ generating parsers and interpreters
- ▶ running test programs

Preliminary tool support for CBS

[Van Binsbergen et al: Modularity 2016]

The screenshot displays the Funcon-Interpreter tool interface, which is divided into four main panels showing code snippets from different files.

IMP-3.cbs (Language "IMP" Section 3 Statements and blocks):

- Syntax**
`Stmt : stmt ::= block`
| `id '=' aexp ';' ;`
| `'if' '(' bexp ')' block ('else' block)?`
| `'while' '(' bexp ')' block`
| `stmt stmt`
- Syntax**
`Block : block ::= '{' stmt? '}'`
- Rule**
`[['if' '(' BExp ')' Block]] : stmt =`
`[['if' '(' BExp ')' Block 'else' '{' '}']]`
- Semantics**
`execute[[_:stmt]] : =>()`
- Rule**
`execute[[I '=' AExp ';']] =`
`assign(bound(id[[I]]), eval-arith[[AExp]])`
- Rule**
`execute[['if' '(' BExp ')' Block1 'else' Block2]] =`
`if-then-else(eval-bool[[BExp]],`

IMP-4.cbs (Language "IMP" Section 4 Programs and variable declarations):

- Syntax**
`Pgm : pgm ::= 'int' idlist ';' stmt`
- Semantics**
`run[[_:pgm]] : =>()`
- Rule**
`run[['int' IL ';' Stmt]] =`
`scope(map-unite(declare-int-vars[[IL]],`

Funcons-Library.cbs (scope.cbs):

- `/*`
``scope(Rho,X)` extends (possibly overriding) the current environment`
`with `Rho` for the execution of `X`.`
`*/`
- Funcon**
`scope(_:environments, _:=>T) : =>T`
- Rule**
`environment(map-override(Rho1,Rho0)) |- X ----> X'`

`environment(Rho0) |- scope(Rho1:environments,X) ----> scope(Rho1,X')`
- Rule**
`scope(_:environments, V:values) = V`

primes.imp:

- `int i, m, n, q, r, s, t, x, y, z;`
`m = 10; n = 2;`
`while (n <= m) {`
 // checking primality of n and write
 `i = 2; q = n/i; t = 1;`
 `while (i <= q && 1 <= t) {`
 `x = i;`
 `y = q;`
 // fast multiplication (base 2)
 `z = 0;`
 `while (!(x <= 0)) {`
 `q = x/2;`
 `r = q+q+1;`
 `if (r <= x) { z = z+y; } else`
 `x = q;`
 `y = y+y;`
 } // end fast multiplication
 `if (n <= z) { t = 0; } else { i`
 } // end checking primality
 `if (1 <= t) { s = s+1; } else {`

primes.fct:

- `assign`
 `(bound ("q"),`
 `integer-divide`
 `(assigned (bound ("n")),`
 `assigned (bound ("i"))))`,
 `assign`
 `(bound ("t"),`
 `decimal-natural ("1"))`,
 `while`
 `(if-then-else`
 `(is-less-or-equal`
 `(assigned (bound ("i")),`
 `assigned (bound ("q"))`,
 `is-less-or-equal`
 `(decimal-natural ("1"),`
 `assigned (bound ("t"))`),
 `false`),
 `sequential`
 `(sequential`
 ...

The bottom status bar shows "Writable", "Smart Insert", and "81 : 17".

Current and future work

- ▶ ***modular static semantics*** for funcons
 - *modular* type soundness proofs?
- ▶ ***improved tool support***
- ▶ adding funcons for ***threads and concurrency***
- ▶ completing a ***major case study: C#***

Funcons

- ▶ correspond to *fundamental programming concepts*
- ▶ language-*independent*
- ▶ have *fixed* behaviour
- ▶ *new* funcons can be added