Funcons
reusable components of language specifications

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

LangDev Meet-Up at CWI, Amsterdam, 8–9 March 2018
## Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>3</td>
</tr>
<tr>
<td>Funcons</td>
<td>7</td>
</tr>
<tr>
<td>Component-based semantics</td>
<td>13</td>
</tr>
<tr>
<td>Tool support</td>
<td>22</td>
</tr>
<tr>
<td>Demo</td>
<td>25</td>
</tr>
<tr>
<td>Conclusion</td>
<td>35</td>
</tr>
<tr>
<td>References</td>
<td>47</td>
</tr>
</tbody>
</table>

*Note: Slides 31–34 and 40–47 were not presented*
Specification vs implementation

Suppose you were developing a new software language…

‣ would *you* specify a *formal semantics* of it?

**Yes:** a few languages from the 80s and 90s

‣ **ADA, SCHEME, STANDARD ML, CONCURRENT ML**

**No:** all other major programming languages

‣ **HASKELL, OCAML, SCALA, JAVA, C#, … (and most DSLs)**
Funcons

Make formal semantics *easier* than BNF!

Encourage language developers to use formal semantics for:

- *documentation*
  - language features, design decisions
- *implementation*
  - rapid prototyping, exploration of design alternatives
Conjecture

Using a *component-based* semantic meta-language can significantly reduce the effort of language specification.
Meta-language engineering

Meta-language requirements

- clear, concise, expressive *notation*
- solid *foundations*
- *tool support* for browsing, checking, validating
- ease of *co-evolution* of languages and specifications
- *reusable components*
Funcons
Funcons

- correspond to *fundamental programming concepts*
- language-*independent*
- have *fixed* behaviour
- specified *independently*
- *new* funcons can be added
Kinds of funcons

**Computations**

- **Normal:** flowing, giving, binding, storing, linking, generating, interacting, …

- **Abnormal:** failing, throwing, returning, controlling, …

- (Concurrent: not yet specified)
Kinds of function

Values *(some types are built-in)*

- **Primitive**: atoms, bools, ints, floats, chars, strings
- **Composite**: algebraic datatypes, tuples, lists, vectors, sets, multisets, maps, pointers, references, variants, …
- **Abstractions**: closures, thunks, functions, patterns, …
- none : no-value
Granularity

Funcons are *individual* programming constructs

- *not* deltas
- *not* language features
- *not* language extensions

Funcons can be *freely* combined

- *independent, unordered, no constraints*
CBS: component-based semantics
Component-based semantics

Language semantics by reduction

- **translation**: language constructs → funcon terms, *hence:*

- **derivation**: funcon semantics → language semantics

Reusable components of language specifications

- funcon specifications
Component-based semantics

programming languages → evolving

translation

funcons

stable reusable components

open-ended repository
Conjecture

Using a component-based semantic meta-language can significantly reduce the effort of language specification.
Language specification in CBS

Syntax

\[ E : \ exp :::= \ldots \ \mid \ 'let' \ id \ '!=' \ exp \ 'in' \ exp \ \mid \ldots \]

Semantics

\[ \text{eval}[[ \_ : \ exp ]] : = \Rightarrow \text{values} \]

Rule

\[ \text{eval}[[ \ 'let' \ I \ '!=' \ E1 \ 'in' \ E2 \ ]] = \]

\[ \text{scope} ( \ \text{bind} ( \ I, \ \text{eval}[[E1]] ), \ \text{eval}[[E2]] ) \]
Language specification in CBS

Syntax

\[
S : \text{stm} ::= \ldots \mid \text{'while'} \text{ '('	ext{exp}'\text{'})'} \text{stm} \mid \ldots
\]

Semantics

\[
\text{exec}[[\_ : \text{stm}]] : \Rightarrow \text{no-value}
\]

Rule

\[
\text{exec}[[\text{'while'} \text{ '('	ext{E}'\text{'})'} \text{S}]] = \\
\quad \text{while-true (}
\quad \quad \text{eval}[[E]],
\quad \quad \text{exec}[[S]]
\quad )
\]
Language specification in CBS co-evolution

Syntax

\[ S : \text{stm} ::= \ldots \mid \text{'}while\text{'}\ (\text{'}\text{exp}\text{'})\ \text{stm} \mid \ldots \]

Semantics

\[ \text{exec}[[\_ : \text{stm}]] : \Rightarrow \text{no-value} \]

Rule

\[ \text{exec}[[\text{'}while\text{'}\ (\text{'}E\text{'})\ S\ ]] = \]

\[ \text{while-true (} \]

\[ \text{not is-eq (} \text{eval}[[E]], 0 ), \]

\[ \text{exec}[[S]] \text{)} \]
Language specification in CBS
co-evolution

Syntax

\[ S : \text{stm} ::= ... | 'while' '(' \text{exp} ')' \text{stm} | \text{'break'} | ... \]

Semantics

\[ \text{exec}[[ _ : \text{stm} ]] : = \text{no-value} \]

Rule

\[ \text{exec}[[ 'while' '(' \text{E} ')' \text{S} ]] = \text{handle-break while-true (} \not \text{is-eq} (0, \text{eval}[[\text{E}]]), \text{exec}[[\text{S}]] \) \]

Rule

\[ \text{exec}[[ \text{'break'} ]] = \text{break} \]
Version control?

Funcons never change!

- no versioning
- optimal reusability

Languages evolve

- ordinary version control
- no support for language reuse/extension/composition
Tool support
CBS tool support
implemented in SPOOFAX (v2.4)

**CBS-Editor**

- for developing language and funcon specifications
- CBS parser (in SDF3)
- CBS name resolution, arity-checking (in NABL2)
- HTML generation (in STRATEGO, CSS)
- language-editor generation (in STRATEGO)
CBS tool support
main features

Generated Language-Editors

‣ program parsing and translation to funcons

Integrated external Haskell tools

‣ generation of funcon interpreters

Internal DynSem tools

‣ generation of funcon interpreters
Demo
Demo of current CBS tool

**Browsing/editing CBS specifications**

- languages and funcons

**Translating programs to funcons**

- using generated STRATEGO code

‘Running’ programs by interpreting funcons

- using generated HASKELL or specified DYNSEM
Demo: language specification

/*
  The SimpleLanguage, abbreviated "SL", is a dynamic demonstration language. It was built using Truffle for the GraalVM at Oracle Labs [https://github.com/graalvm/simplelanguage].

  A DynSem specification of SL in Spoofax has been given by Vlad Vergu [https://github.com/MetaBorgCube/metaborg-sl/].

  This CBS specification of SL has been prototyped using a DynSem specification of the required funcons. The CBS grammar of SL is annotated with the SDP3 productions used for the DynSem of SL.

  The SL programs used to test the DynSem of SL are in SL-Tests/examples, together with their expected output and the funcon terms produced by the CBS of SL. Running the funcon terms using the DynSem of the funcons gives the same results as running the SL programs using the DynSem of SL, except for programs involving the following built-in SL functions, which have not yet been implemented:
  - defineFunction (the argument is a string in SL, requiring parsing)
  - nanoTime
  - stackTrace

  Moreover, some of the literal numbers in the examples have been reduced (mostly by a factor of 10) to avoid potential stack overflow.

  Acknowledgement: Vlad Vergu provided helpful advice regarding SL and DynSem.
*/

#1 Function definitions
#2 Expressions
#3 Statements
#A Disambiguation

Syntax
Program : program
::= fun-def* // Program.Program = [ [[FunDef "\n"]+] ]

Semantics
Demo: funcon reference
Demo: funcon specification
Demo: funcon index

```cbl
dataflow

## Computations

```cbl
dataflow

### Types of computation

```cbl
dataflow

<table>
<thead>
<tr>
<th>Funcon</th>
<th>computation-types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>entities</td>
</tr>
<tr>
<td>Funcon</td>
<td>entity</td>
</tr>
</tbody>
</table>

### Normal computation

```cbl
dataflow

### Flowing

```cbl
dataflow

<table>
<thead>
<tr>
<th>Funcon</th>
<th>interleave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datatype</td>
<td>yielding</td>
</tr>
<tr>
<td>Funcon</td>
<td>signal</td>
</tr>
<tr>
<td>Funcon</td>
<td>yielded</td>
</tr>
<tr>
<td>Funcon</td>
<td>yield</td>
</tr>
<tr>
<td>Funcon</td>
<td>yield-on-value</td>
</tr>
<tr>
<td>Funcon</td>
<td>yield-on-abrupt</td>
</tr>
<tr>
<td>Funcon</td>
<td>atomic</td>
</tr>
<tr>
<td>Funcon</td>
<td>left-to-right</td>
</tr>
<tr>
<td>Funcon</td>
<td>right-to-left</td>
</tr>
<tr>
<td>Funcon</td>
<td>sequential</td>
</tr>
<tr>
<td>Funcon</td>
<td>if-then-else</td>
</tr>
<tr>
<td>Funcon</td>
<td>while</td>
</tr>
<tr>
<td>Funcon</td>
<td>do-while</td>
</tr>
</tbody>
</table>

### Giving

```cbl
dataflow

<table>
<thead>
<tr>
<th>Entity</th>
<th>given-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Funcon</td>
<td>initialise-giving</td>
</tr>
<tr>
<td>Funcon</td>
<td>give</td>
</tr>
<tr>
<td>Funcon</td>
<td>given</td>
</tr>
<tr>
<td>Funcon</td>
<td>no-given</td>
</tr>
<tr>
<td>Funcon</td>
<td>left-to-right-map</td>
</tr>
<tr>
<td>Funcon</td>
<td>interleave-map</td>
</tr>
<tr>
<td>Funcon</td>
<td>left-to-right-filter</td>
</tr>
</tbody>
</table>
```
Demo: translation to funcons
Demo: running funcons

```haskell
function main() {
    println("Hello World!");
}
```
Demo: resulting behaviour

```plaintext
function main() {
  println("Hello World!");
}
```
Demo: (re)generating
Conclusion
Language specification in CBS

programming languages

translation

funcons

stable reusable components

open-ended repository
Conjecture

Using a component-based semantic meta-language can significantly reduce the effort of language specification.
Proving the conjecture

Case studies

- Various small example languages (IMP, SIMPLE, SL, MJ, …)
- **CAML LIGHT**
  - *Reusable components of semantic specifications*
- C# (v1.2)
  - ongoing…
Funcons beta-release (imminent…😊)

2018(Q1): Funcons-beta to be made available for review

- some details may change !
- preliminary tool support, minimal documentation

2018(Q3/Q4): Funcons to be released for reuse

- details will not be allowed to change !!
- usable tool support, user-level documentation
PLANCOMPS
www.plancomps.org

“Programming Language Components and Specifications”

Funded project 2011–16: EPSRC

- at Swansea, Royal Holloway (RHUL), City, Newcastle

Current participants:

- A. Johnstone, E.A. Scott, L.T. van Binsbergen (RHUL)
  N. Sculthorpe (NTU), C. Bach Poulsen, PDM (Delft)

New participants are welcome!
Current and future work

- **modular static semantics** for funcons
  - *modular* type soundness proofs?
- improved *tool support*
- funcons for *threads and concurrency*
- completing a *major case study: C#*
Appendix
Funcon specification in CBS

Normal computation: **flowing**

**Funcon**

\[
\text{while-true ( } \_ : \Rightarrow \text{booleans}, \_ : \Rightarrow \text{no-value} \): \Rightarrow \text{no-value}
\]

**Rule**

\[
\text{while-true ( } X, Y \)
\]

\[
\Rightarrow \text{if-then-else ( } X, \text{seq ( } Y, \text{while ( } X, Y \), \text{none) }
\]

43
Normal computation: **flowing**

**Funcon**

```plaintext
if-then-else ( _ : bools, _ : =>T, _ : =>T ) : =>T
```

**Rule**

```plaintext
if-then-else ( true, X, Y ) => X
```

**Rule**

```plaintext
if-then-else ( false, X, Y ) => Y
```
Funcon specification in CBS

Normal computation: **flowing**

**Funcon**

\[
\text{seq}(\_ : \text{no-value}, \_ : =>T) : =>T
\]

**Rule**

\[
\text{seq}(\text{none}, Y) \rightsquigarrow Y
\]
Funcon specification in CBS

Normal computation: **binding**

---

**Funcon**

```
scope(_, envs, _: =>T) : =>T
```

**Rule**

```
env(map-override(Rho1, Rho0)) |- X  ----> X'
```

```
env(Rho0) |- scope(Rho1:envs, X)  ----> scope(Rho1, X')
```

**Rule**

```
scope(_, envs, V:T) ~> V
```
References

  https://doi.org/10.1016/j.jlap.2004.03.008

  https://doi.org/10.1016/j.entcs.2009.07.073

• PLanCompS project home page. 
  http://www.plancomps.org


• M. Churchill, PDM: Modular bisimulation for computations and values. In *FoSSaCS’13*. 
  https://doi.org/10.1007/978-3-642-37075-5_7

• F. Vesely, PDM. Funkons – Component-based semantics in K. In *WRLA’14*. 
  https://doi.org/10.1007/978-3-319-12904-4_12

  https://doi.org/10.1007/978-3-662-46734-3_4

• V.A. Vergu, P. Neron, E. Visser: DynSem – A DSL for dynamic semantics specification. In *RTA’15*. 
  https://doi.org/10.4230/LIPIcs.RTA.2015.365

  http://doi.acm.org/10.1145/2892664.2893464