The CBS Framework

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

IFIP WG 2.11 Meeting, Kyoto, June 2018
CBS: Component-Based Semantics

Main goal:

Make formal semantics as popular as BNF!

Encourage language developers to use formal semantics:

- documentation of language features, design decisions
- generation of (prototype) implementations
CBS: Component-Based Semantics

Reusable components: ‘funcons’

- specifications of *fundamental programming constructs*
- independent of particular programming languages

Language semantics

- *translation* from language constructs to funcon terms
- language semantics *derived* from funcon semantics
Conjecture

Using *component-based* semantics can significantly reduce the effort of language specification
Component-based semantics

programming languages

translation

stable reusable components

open-ended repository
Language specification in CBS

Syntax

\[
E : \text{exp} ::= \ldots \mid \text{'let' id '=' exp 'in' exp} \mid \ldots
\]

Semantics

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

Rule

\[
\text{eval}[\[\text{'let' I '=' E1 'in' E2}\]] = \text{scope ( bind-value ( I, eval[[E1]] ), eval[[E2]] )}
\]
Language specification in CBS
c o-e volution

Syntax

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

Semantics

\[
\text{exec}\llbracket \_ : \text{stm} \rrbracket \Rightarrow \text{null-type}
\]

Rule

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

Syntax

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

Semantics

\[ \text{exec}[\_ : \text{stm}]] : = \Rightarrow \text{null-type} \]

Rule

\[ \text{exec}[[\text{'while' '(' E ')' S}]] = \]
\[ \text{while-true ( not is-eq( \_0, \text{eval}[[E]] ), \text{exec}[[S]] )} \]
Language specification in CBS
colo-evolution

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

Semantics
\[ \text{exec}[[ \_ : \text{stm} ]] : \Rightarrow \text{null-type} \]

Rule
\[ \text{exec}[[ 'while' '(' \text{E} ')' \text{S} ]] = \]
\[ \text{handle-break} ( \text{while-true} ( \text{eval}[[\text{E}]], \text{exec}[[\text{S}]] ) ) \]

Rule
\[ \text{exec}[[ 'break' ]] = \text{break} \]
Funcons
Funcons

Fundamental programming constructs

- language-\textit{independent}
- (mostly) \textit{specified independently}
- have \textit{fixed behaviour}
- \textit{extensible}
Computations

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

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

- (Concurrent: not yet specified)
Funcons

**Values** *(some types are built-in)*

- **Primitive**: atoms, booleans, integers, floats, characters, null, …

- **Composite**: algebraic datatypes, tuples, lists, vectors, sets, multisets, maps, pointers, references, variants, …

- **Abstractions**: closures, thunks, functions, patterns, …
Funcon specification in CBS

Normal computation: **flowing**

*Funcon*

```plaintext
while-true ( _: =>booleans, _: =>null-type ) : =>null-type
```

*Rule*

```plaintext
while-true ( X, Y ) ~> if-true-else ( X, sequential ( Y, while-true ( X, Y ) ), null-value )
```
Funcon specification in CBS

Normal computation: flowing

Funcon

\[
\text{if-true-else ( \_ : booleans, \_ : } \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T
\]

Rule

\[
\text{if-true-else ( true, X, Y ) } \Rightarrow X
\]

Rule

\[
\text{if-true-else ( false, X, Y ) } \Rightarrow Y
\]
Funcon specification in CBS
Normal computation: flowing

Funcon

\[
\text{funcon} \quad \text{sequential} \ ( \_ : \text{null-type}, \_ : \Rightarrow T ) : \Rightarrow T
\]

Rule

\[
\text{rule} \quad \text{sequential} \ ( \text{null-value}, Y ) \Rightarrow Y
\]
Funcon specification in CBS

Normal computation: binding

Type
  \texttt{envs} \rightarrow \texttt{maps}(\texttt{ids, values})

Funcon
  \texttt{bind-value} ( \_ : \texttt{ids}, \_ : \texttt{values} ) : \rightarrow \texttt{envs}

Rule
  \texttt{bind-value} ( I : \texttt{ids}, V : \texttt{values} ) \rightarrow \{ I \rightarrow V \}
Funcon specification in CBS

Normal computation: binding

**Entity**

```
env(_: envs) |- _ ---> _
```

**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
```
CBS foundations
main features

- **I-MSOS**: SOS notation, but interpreted as *Modular SOS*
- **strictness annotations** in signatures ($\_ : T$ vs $\_ : \Rightarrow T$)
- **value-computation systems**
  - transitions (→>) and rewriting (~>)
- **bisimulation congruence format** for rules
- **bisimulation preserved** by disjoint extension
CBS beta-release
plancomps.github.io/CBS-beta

Funcons-beta made available for review

- changes possible until final release (autumn 2018)

Languages-beta illustrates CBS using Funcons-beta

- simple languages: IMP, SIMPLE, SL
- sub-languages: MiniJava, OCaml Light
- may need to change before final release
CBS beta-release
plancomps.github.io/CBS-beta

**Demonstrated:**

- benefits of funcons: reuse, simplicity
- generation of editors, prototype interpreters

**Not yet demonstrated:**

- co-evolution, scaling-up, DSLs, modeling languages
- funcon algebra
Conclusion
Language specification in CBS

programming languages

translation

stable reusable components

funcons

open-ended repository
Conjecture

Using *component-based semantics* can significantly reduce the effort of language specification.
Static semantics questions

- **modular static semantics** for funcons?
  - unrestricted effects, co-effects

- **type-directed translation** of language constructs?
  - separate conventional static semantics?
  - exploitation of scope graphs?
"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 !**