Reuse and co-evolution in CBS language specifications

Peter Mosses
Swansea University and TU Delft

pdmosses.github.io

LangDev Meetup at Amazon Amsterdam, 21–22 March 2019
Formality of language specifications

Complete language specifications produced by language developers themselves

- syntax
  - reasonably formal 😊

### OCAML:

```
OCAML:
```

```
7.7 Expressions
expr ::= value-path
| constant
| expr |
| tuple expr end
| [ expr , expr ]
| match expr
| func expr

expr ( expr )

expr ( expr : typexpr )

expr { expr , expr }

expr [ expr ]

expr [ expr ]

expr # method-name

expr inst-var-name

expr inst-var-name = expr

expr ( expr :> typexpr )

expr ( expr :> typexpr )

{ inst-var-name = expr { inst-var-name = expr } { inst-var-name = expr } }
```

```
```
Formality of language specifications

Complete language specifications produced by language developers themselves

- **syntax**
  - *reasonably formal* 😊

- **semantics** (static and dynamic)
  - *completely informal* 😞

- **a few exceptions:** ADA, SCHEME, STANDARD ML, WEBASSEMBLY

---

**OCAML:**

```ocaml
if we ignore labels, which will only be meaningful at function application, this is equivalent to

```
Reuse and co-evolution

CBS: component-based semantics

› semantics: language → funcons
  - context-free, compositional

› funcons (fundamental constructs)
  - open-ended library of fixed items

Claim: CBS can significantly reduce the effort of formal semantics!

Developed by the PLANCOMPS project

› EPSRC funding 2011–16; now an open collaboration
Reusable components

Funcons – not languages!

- familiar programming concepts
- simpler than language constructs
- fixed definitions
- open-ended library
- unbiased to any language class

Example:

Funcon
\[
\text{sequential}(\_:(\Rightarrow \text{null-type})^*, \_:=\Rightarrow T) : \Rightarrow T
\]

Rule
\[
X \longrightarrow X'
\]

\[
\text{sequential}(X, Y+) \longrightarrow \text{sequential}(X', Y+)
\]

Rule
\[
\text{sequential}(\text{null-value}, Y+) \Rightarrow \text{sequential}(Y+)
\]

Rule
\[
\text{sequential}(Y) \Rightarrow Y
\]
Co-evolution of languages and specifications

Translations

- **language** → funcons
  - dependence on language syntax
- **context-free** translation
  - compositional
  - specified by equations

**Examples:**

**Semantics**

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

**Rule**

\[
\text{eval}[\[ E_1 '=' E_2 \]] = \\
\quad \text{assign}( \text{eval}[\[ E_1 \]], \text{eval}[\[ E_2 \]])
\]

**Rule**

\[
\text{eval}[\[ '! E \]] = \text{assigned}( \text{eval}[\[ E \]])
\]

**Rule**

\[
\text{eval}[\[ E_1 ';' E_2 \]] = \\
\quad \text{sequential}( \text{effect}( \text{eval}[\[ E_1 \]], \text{eval}[\[ E_2 \]]))
\]

**Rule**

\[
\text{eval}[\[ \text{while} E_1 \text{ do} E_2 \]] = \\
\quad \text{while-true}( \text{eval}[\[ E_1 \]], \text{eval}[\[ E_2 \]])
\]
Tool support for CBS specifications

Creating, editing, browsing

- grammars, funcons, translations

CBS workbench

- based on SPOOFAX

Generating prototypes

- language parser

- funcon interpreter

- translator : language $\rightarrow$ funcons

- hence language interpreter

Fig. 5. The IDE for CBS in action. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)
Incremental specification

- LD: a demo language
  - literals
  - lambda-calculus
  - arithmetic
  - references
  - threads x

No changes to previous rules!

Grammar:

```
Syntax E::= int

// Call-by-value lambda-calculus:
| 'lambda' id '.' exp
| exp exp
| 'let' id '=' exp 'in' exp
| '(' exp ')'

// Arithmetic and Boolean expressions:
| exp '+' exp
| exp '*' exp
| exp '/' exp
| exp '<=' exp
| exp '&&' exp
| 'if' exp 'then' exp 'else' exp

// References and imperatives:
| 'ref' exp
| exp '===' exp
| '!' exp
| exp ';' exp
| '(' ')
| 'while' exp 'do' exp

// Multithreading:
| 'spawn' exp
| 'join' exp
```
Current status

**CBS-beta** [plancomps.github.io/CBS-beta]

- Funcons-beta
- Languages-beta
  - *toy*: IMP, SIMPLE, MINIJAVA
  - *medium*: OCAML-LIGHT, SL
  - *pending*: IMP++, SIMPLE-THR
    - multithreading

**CBS-Editor**

- SPOOFAX/ECLIPSE plugin
- *under development…*

**Funcons.Tools**

- HASKELL package
- *generates interpreters for funcons from their definitions*
Conclusion

CBS: component-based semantics framework [plancomps.github.io]

- unified specification language with solid theoretical foundations
- support for reuse and co-evolution
- library of funcon definitions

CBS language workbench

- creating, editing, browsing specifications
- generating editors, translators, interpreters

<table>
<thead>
<tr>
<th>CBS</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBS-beta</td>
<td>July 2018</td>
</tr>
<tr>
<td>threads</td>
<td>April 2019</td>
</tr>
<tr>
<td>CBS workbench</td>
<td>June 2019 ?</td>
</tr>
<tr>
<td>major case studies</td>
<td>2020 ?</td>
</tr>
</tbody>
</table>
Recent references

Executable component-based semantics

Software meta-languages and CBS