Tool Support for Component-Based Semantics

L. Thomas van Binsbergen Neil Sculthorpe
Department of Computer Science,
Royal Holloway, University of London, UK
ltvanbinsbergen@acm.org, neil.sculthorpe@rhul.ac.uk

Peter D. Mosses
Department of Computer Science,
Swansea University, UK
p.d.mosses@swansea.ac.uk

Abstract

The developers of a programming language need to document its intended syntax and semantics, and to update the documentation when the language evolves. They use formal grammars to define context-free syntax, but usually give only an informal description of semantics. Use of formal semantics could greatly increase the consistency and completeness of language documentation, support rapid prototyping, and allow empirical validation.

Modularity of semantics is essential for practicality when scaling up to definitions of larger languages. Component-based semantics takes modularity to the highest possible level. In this approach, the semantics of a language is defined by equations translating its constructs (compositionally) to combinations of so-called fundamental constructs, or 'funcons'. The definition of each funcon is a small, highly reusable component. The PLanCompS project has defined a substantial library of funcons, and shown their reusability in several case studies.

We have designed a meta-language called CBS for component-based semantics, and an IDE to support development, rapid prototyping, and validation of definitions in CBS. After introducing and motivating CBS, we demonstrate how the IDE can be used to browse and edit the CBS definition of a toy language, to generate a prototype implementation of the language, and to parse and run programs.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

Keywords Programming language semantics, modularity, reusability, tool support, rapid prototyping

1. Introduction and Background

New programming languages and domain-specific languages are continually being introduced, as are new versions of existing languages. Each language needs to be carefully specified, to determine the syntax and semantics of its programs.

Context-free aspects of syntax are usually specified, precisely and succinctly, using formal grammars; in contrast, semantics (including static checks and disambiguation) is generally specified only informally, without use of precise notation.[1] Informal specifications are often incomplete or inconsistent, and open to misinterpretation; formal specifications can avoid such issues. Moreover, completely formal definitions of programming languages may be used to generate prototype implementations, and as a basis for proving properties of languages and of individual programs.

Although there is broad agreement as to the benefits of formality in language definitions, and although there are a few examples of successful individual projects (notably the definition of Standard ML (Milner et al. 1997)), there is generally little inclination on the part of programming language developers themselves to use formal semantics. For instance, even Haskell, a language designed with an emphasis on its mathematical structure, does not have a formal semantics (Hudak et al. 2007). It appears that this is at least partly due to the effort required when scaling up to larger languages, and when updating a formal semantics to reflect language evolution.

New languages typically include a large number of constructs from previous languages, presenting a major opportunity for reuse of specification components. In the absence of a suitable collection of reusable components, however, each language would have to be specified from scratch – a huge effort.

Component-based semantics. To improve the practicality of formal semantic definitions of larger languages, the PLanCompS project[2] proposed to base them on a collection of reusable components, and to implement tool support for development and testing. Analogous practices are widely adopted in software engineering: developers rely on reusable components in the form of packages, and on IDEs (integrated development environments) when coding and testing.

In the PLanCompS approach, a reusable component of language definitions corresponds to a fundamental programming construct: a so-called 'funcon', which has a fixed operational interpretation. The formal semantics of each funcon is defined independently, using I-MSOS (Mosses and New 2009), a variant of modular structural operational semantics (Mosses 2004). The collection of funcons is open-ended; crucially, adding new funcons never requires changes to the definition or use of previous funcons.

A component-based semantics of a programming language is defined by translating its constructs to funcons. Many funcons can be widely reused in the definitions of different languages. An initial medium-scale case study (Churchill et al. 2015) gave a semantics for Caml Light (Leroy 1997) based on a preliminary collection of funcons; after completion of a major case study (C#), the validated funcons are to be finalised and made freely available in a digital library, for reuse in future language definitions.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...
http://dx.doi.org/10.1145/2892664.2893464

[1] Metamodels allow also context-sensitive constraints to be specified formally, but they may be considerably less succinct than formal grammars.

[2] `http://www.plancomps.org`

The CAML LIGHT case study originally used a mixture of meta-languages: the lexical and context free syntax of CAML LIGHT were defined in SDF3; its translation to funcons was defined by transformation rules written in STRATEGO; and the static and operational semantics of funcons were defined in CSF, a plain-text version of I-MSOS, enriched with rules for the value-computation transition systems introduced in (Churchill and Mosses 2013). As mentioned in connection with the CAML LIGHT case study (Churchill et al. 2015, Sect. 4.4), a unified specification language called CBS was to be developed, to replace the combination of SDF, STRATEGO and CSF. The aim was to provide considerably greater notational consistency, and improve the readability of the specifications.

We have now completed the development of CBS, and we are using it for our case studies. CBS supports specification of abstract syntax grammars (essentially BNF with regular expressions), the signatures and equations for functions translating language constructs to funcons, and the signatures and rules for defining funcons. Component-based semantics and funcons can now be defined without the use of SDF3,[3] STRATEGO, and CSF. Use of CBS also improves the conciseness of component-based semantics.

Tool support. We have used SPOOFAX (Kats and Visser 2010) to generate a CBS editor with many useful features, including syntax highlighting, syntax error recovery, hyperlinks from uses of symbols to their definitions, and flagging of undefined symbols. The CBS editor can be used to browse, navigate and update both language definitions and funcon definitions. It provides buttons for generating (SDF3 and STRATEGO) code, which can subsequently be built to produce an editor for the defined language. The generated editor is immediately available for use in the same instance of ECLIPSE as the CBS editor, and can be customised without editing the generated files.

To support development and validation of definitions in CBS, we provide a complete tool chain for executing translation functions and running the resulting funcon terms. This allows component-based language specifications to be easily prototyped and tested. Generation of funcon interpreter components in HASKELL is a new feature of the current version of the tool support for CBS.

A further contribution is illustrating the usefulness of SPOOFAX for generating IDEs to support semantic frameworks. Moreover, users of CBS can enjoy the benefits of the SPOOFAX platform without direct use of its built-in meta-notations (SDF3, NABL, TS, DYNSEM, STRATEGO).

Potential impact. The successful completion of the major C# case study, and subsequent publication of over 100 validated funcon definitions, might have significant transformative effects: language developers could be encouraged to make use of formal semantics and experiment with generated prototype implementations during the design process; DSL users may be empowered to design, specify and implement languages themselves; and researchers and students will be provided with an online open-access repository for language and funcon definitions. A digital library could support information retrieval in formal semantics using mathematical knowledge management techniques.

Related work. Other currently available tools supporting development of semantic definitions and semantics-based program execution[4] include the COREASM tools,[5] OTT,[6] PLTREDEX,[7] the K tools,[8] MAUDE[9] and MMT,[10] MELANGE,[11] and DYNSEM (Vergu et al. 2015). Most of the semantic frameworks supported by these tools have a high degree of modularity. Some of the tools have also been used to develop and validate semantic definitions of major programming languages such as C, C#, and JAVA – albeit not by the language developers themselves.

The main distinguishing feature of the CBS framework is the dramatic reduction of effort required to formulate and maintain semantic definitions due to *reuse* of funcon definitions. Our tool support for CBS enhances practicality by providing sophisticated editors and semantics-based program execution.

The only other framework which comes together with an extensive collection of reusable components is MMT (a direct precursor of CBS, based on MSOS rather than on I-MSOS). The possibility of defining funcons in the K framework, and of using the K tools to translate languages to funcons, is explored in (Mosses and Vesely 2014).

2. Developing and Executing Language Definitions

The screenshot in Fig. 1 shows several files opened during the development and testing of the component-based semantics of IMP, a toy C-like imperative language (Roşu and Şerbănuţă 2010).

The CBS files in the two leftmost panes specify the abstract syntax and semantics of IMP statements, blocks, programs, and variable declarations; the rest of the language (comprising arithmetic and Boolean expressions) is specified in separate files. The function *execute* maps IMP statements to funcon terms that represent their semantics. For instance, it maps an assignment statement '*I* = *AExp* ;' to a term formed from the translations of *I* and *AExp* using the funcons **assign** (updating a variable with a value) and **bound** (returning whatever is currently bound to an identifier – here, it can only be a variable).

Clicking on a name in a CBS editor shows its definition, opening the file containing it (if needed). The upper right pane shows the CBS definition of the operational semantics of the funcon **scope**, which is used in IMP to make integer variable declarations local to a statement. This funcon is declared to be strict in its first argument (allowing the rule for computing it to be elided) and generic in its second argument (so it can be used for expressions as well as statements).

When the focus is on a CBS file specifying the semantics of (part or all of) a language, clicking on Generation produces or updates (part or all of) an executable translator from the language to funcon terms. Generation of funcon interpreters from CBS definitions of funcons is currently done by a shell script, but is to be incorporated in the CBS editor.

The two panes on the lower right show a small IMP test program and part of its translation to a funcon term. When the focus is on a complete program, clicking on Generation translates it to the corresponding funcon term, which can then be executed using the Funcon-Interpreter button shown at the top of the screen (see

[3] To define concrete syntax, an abstract syntax grammar usually needs to be supplemented by disambiguation rules and layout productions. CBS does not support metamodelling.

[4] We restrict attention here to formal semantic frameworks with established theoretical foundations.

[5] `https://www.uni-ulm.de/in/pm/forschung/projekte/coreasm`

[6] `http://www.cl.cam.ac.uk/~pes20/ott`

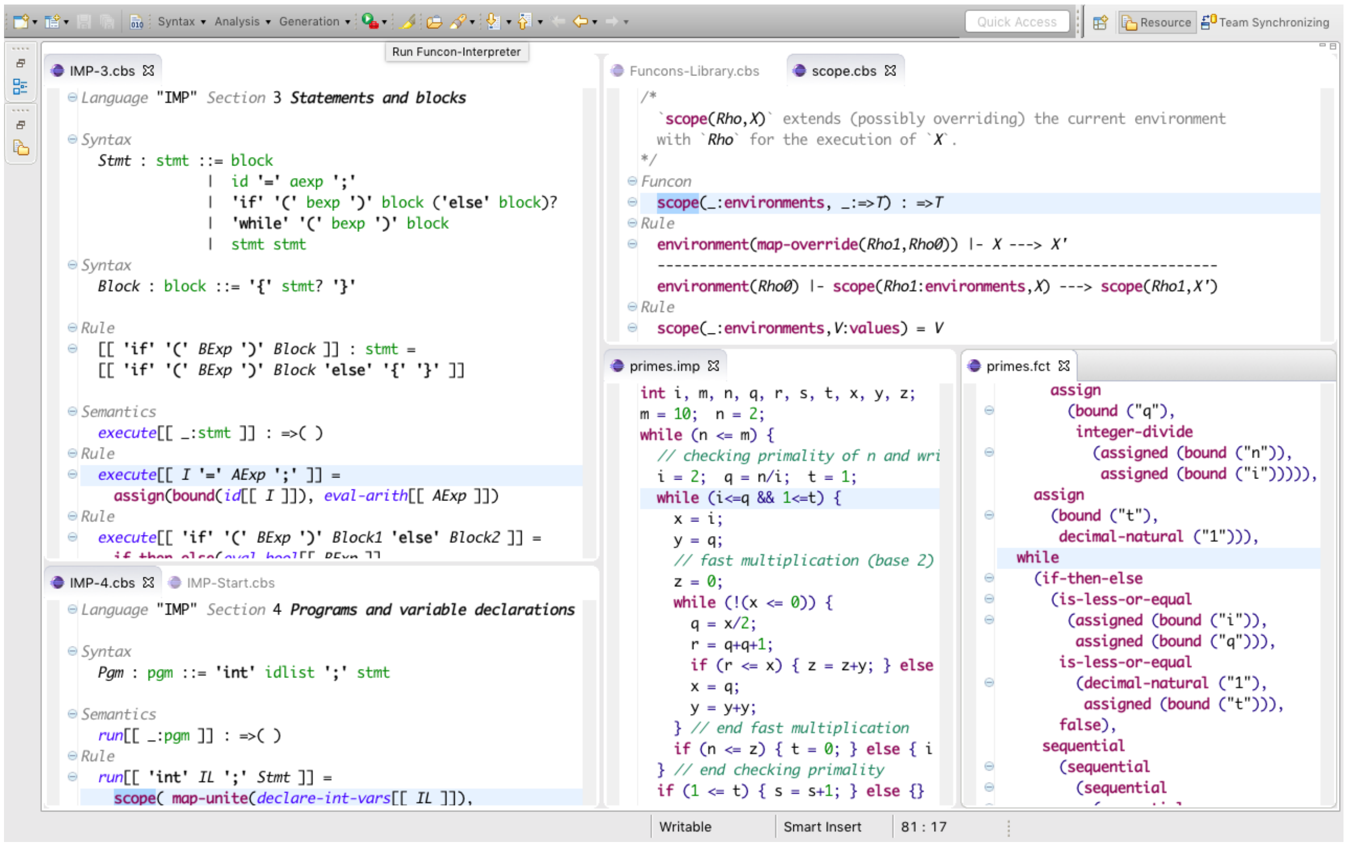
[7] `http://redex.racket-lang.org`

[8] `http://www.kframework.org`

[9] `http://maude.cs.uiuc.edu`

[10] `https://github.com/fcbr/mmt`

[11] `http://melange-lang.org`



```
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 ]] =
```

```
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 ]]),
```

```
/*
  `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
```

```
int i, m, n, q, r, s, t, x, y, z;
m = 10;  n = 2;
while (n <= m) {
  // checking primality of n and wri
  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 {}
```

```
        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
           (sequential
```

Figure 1. Editing and testing the CBS definition of IMP

Sect. 3). Shell commands allow entire test suites to be translated and executed automatically.

The implementation of the CBS editor in SPOOFAX involved writing an SDF3 grammar for the CBS language, some small files specifying the various editor services (highlighting, name resolution, menus, folding), and STRATEGO code to generate SDF3 grammars and STRATEGO rules from the ASTs of CBS specifications. The generated SDF3 grammars provide the syntax for the semantic functions and metavariables that occur in the generated STRATEGO rules, as well as the abstract syntax of the programming language. The screenshot in Fig. 2 shows part of the current collection of funcons; the CBS file for the funcons library provides hyperlinks to all funcons, which allows browsing them without navigating down through the ECLIPSE directory structure shown on the left. The screenshot also shows generation of SDF3 and STRATEGO code for all sections of the CBS for IMP, which includes generation of SDF3 for the required funcons.

When a programming language evolves, the syntax and/or semantics of its constructs can change, new constructs may be added, and existing constructs may be removed. This is achieved by editing the CBS files that define the abstract syntax and its translation to funcons, and regenerating the code for the changed files. If new funcons are needed, they are added to the collection in separate CBS files. Existing funcon definitions never change, so they do not need version control. However, we use SUBVERSION (SVN) both to track changes to language definitions and to check the lack of changes to the funcon definitions. Moreover, SVN external links facilitate sharing the entire collection of funcons between definitions of different languages.

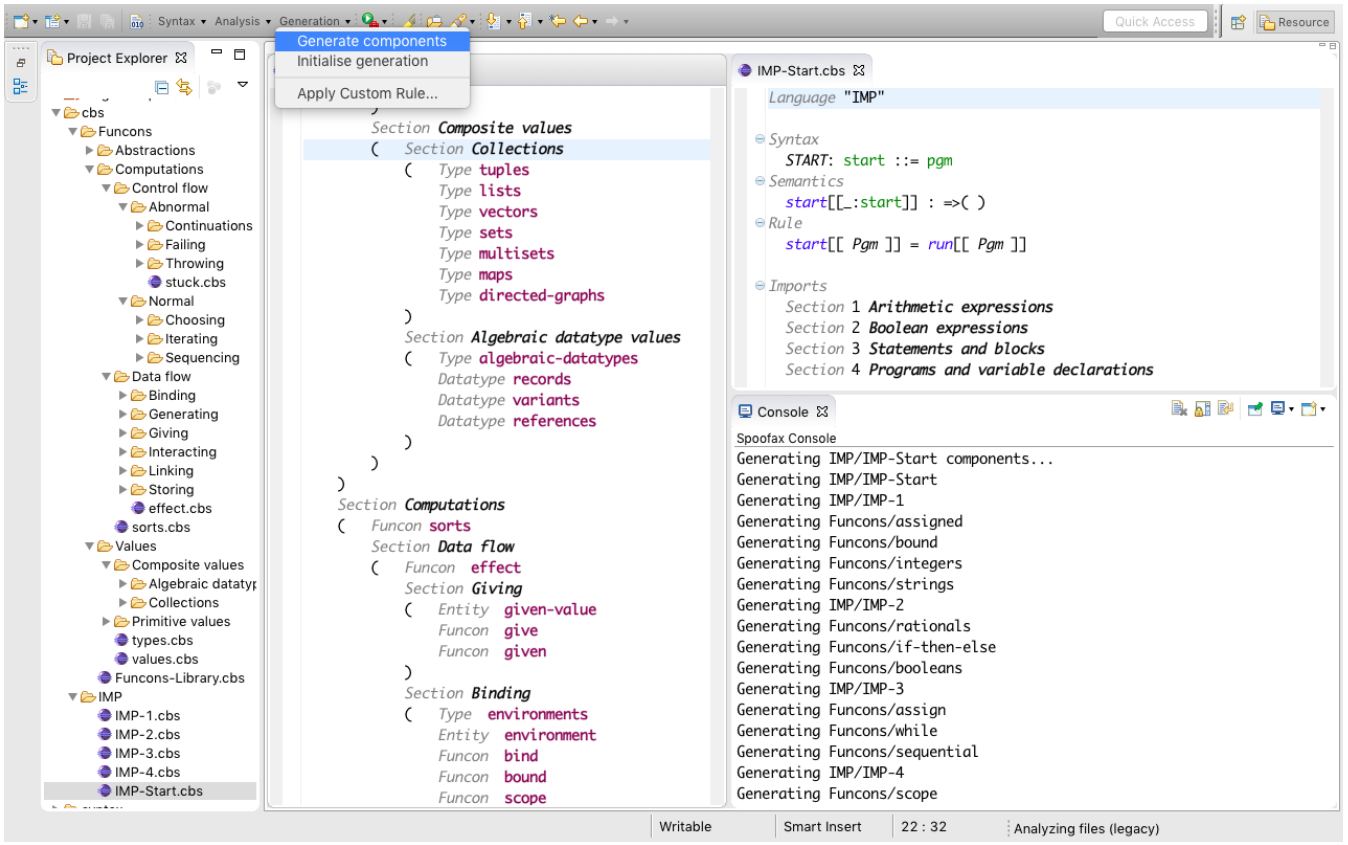
3. Executing Funcon Terms

We execute funcon terms using a HASKELL library where funcons are specified in a style similar to I-MSOS (Mosses and New 2009), the modular variant of SOS supported by CBS. The interpreter for funcon terms can be invoked from ECLIPSE, with output printed to ECLIPSE's console.

The defining feature of I-MSOS is the implicit propagation of auxiliary entities, and this is achieved in HASKELL by using a monad in the implementation of the small-step evaluation relation. The resulting code is as modular as I-MSOS rules: adding a new funcon or auxiliary entity requires no modification to the code for the existing funcons. The HASKELL code defining the individual funcons can either be written manually, or generated by compiling CBS funcon specifications to HASKELL code using our CBS compiler (also written in HASKELL).

The CBS language includes a fixed universe of value types, and a set of built-in operations on those types; these are supported by binding them to HASKELL's data types and library functions. For nearly all cases, direct counterparts of the CBS value types and operations are available in the HASKELL standard library. Further value operations are defined as funcons by I-MSOS rules.

Dynamic errors are handled gracefully by the interpreter, which reports the immediate cause of the error along with the current contents of the auxiliary entities and the funcon term remaining to be executed. The interpreter also includes a parser and pretty printer for funcon terms, and an optional refocusing-based optimisation (Danvy and Nielsen 2004) that provides a more efficient evaluation strategy.



```
        Section Composite values
        (   Section Collections
            (   Type tuples
                Type lists
                Type vectors
                Type sets
                Type multisets
                Type maps
                Type directed-graphs
            )
            Section Algebraic datatype values
            (   Type algebraic-datatypes
                Datatype records
                Datatype variants
                Datatype references
            )
        )
    )
    Section Computations
    (   Funcon sorts
        Section Data flow
        (   Funcon  effect
            Section Giving
            (   Entity  given-value
                Funcon  give
                Funcon  given
            )
            Section Binding
            (   Type  environments
                Entity  environment
                Funcon  bind
                Funcon  bound
                Funcon  scope
```

```
Language "IMP"

Syntax
  START: start ::= pgm
Semantics
  start[[_:start]] : =>( )
Rule
  start[[ Pgm ]] = run[[ Pgm ]]

Imports
  Section 1 Arithmetic expressions
  Section 2 Boolean expressions
  Section 3 Statements and blocks
  Section 4 Programs and variable declarations
```

```
Spoofax Console
Generating IMP/IMP-Start components...
Generating IMP/IMP-Start
Generating IMP/IMP-1
Generating Funcons/assigned
Generating Funcons/bound
Generating Funcons/integers
Generating Funcons/strings
Generating IMP/IMP-2
Generating Funcons/rationals
Generating Funcons/if-then-else
Generating Funcons/booleans
Generating IMP/IMP-3
Generating Funcons/assign
Generating Funcons/while
Generating Funcons/sequential
Generating IMP/IMP-4
Generating Funcons/scope
```

Figure 2. Browsing funcons, and generating component code from the CBS definition of IMP

4. Tool Download and Installation

The tool support for CBS will be released for general use when the major C# case study has been completed, and our present funcon notation polished and finalised. To allow Modularity'16 participants (and other readers) to test the tools demonstrated at the conference, however, we have made the current version, together with the definition of the toy language used in the demonstration and the current collection of funcon definitions, freely available for download at:

`http://www.plancomps.org/modularity2016`.

Installation of the tools involves downloading a version of ECLIPSE with SPOOFAX pre-installed, then importing and building two SPOOFAX editor projects. For executing funcon terms, some HASKELL packages need to be installed.

Acknowledgments

We thank the referees for their helpful comments and suggestions. The reported work was supported by EPSRC grants for the PLANCOMPS project at Swansea University (EP/I032495/1) and Royal Holloway, University of London (EP/I032509/1).

References

M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *FoSSaCS 2013*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.

M. Churchill, P. D. Mosses, N. Sculthorpe, and P. Torrini. Reusable components of semantic specifications. In *Trans. AOSD XII*, volume 8989 of *LNCS*, pages 132–179. Springer, 2015.

O. Danvy and L. R. Nielsen. Refocusing in reduction semantics. BRICS Research Series RS-04-26, Dept. of Computer Science, Aarhus Univ., 2004. `http://www.brics.dk/RS/04/26/`.

P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL-III*, pages 12:1–12:55. ACM, 2007.

L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *OOPSLA '10*, pages 444–463. ACM, 2010.

X. Leroy. Caml Light manual, 1997. `http://caml.inria.fr/pub/docs/manual-caml-light`.

R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

P. D. Mosses. Modular structural operational semantics. *J. Log. Algebraic Program.*, 60-61:195–228, 2004.

P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *SOS '08*, volume 229(4) of *ENTCS*, pages 49–66. Elsevier, 2009.

P. D. Mosses and F. Vesely. FunKons: Component-based semantics in K. In *WRLA '14*, volume 8663 of *LNCS*, pages 213–229. Springer, 2014.

G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *J. Log. Algebraic Program.*, 79(6):397–434, 2010.

V. A. Vergu, P. Neron, and E. Visser. DynSem: A DSL for dynamic semantics specification. In *RTA 2015*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015.