# Tool Support for Component-Based Semantics

L. Thomas van Binsbergen[1], Peter D. Mosses[2], and Neil Sculthorpe[2]

[1] Department of Computer Science, Royal Holloway University of London, UK
ltvanbinsbergen@acm.org
[2] Department of Computer Science, Swansea University, UK
p.d.mosses@swansea.ac.uk, n.a.sculthorpe@swansea.ac.uk

**Abstract**

The PLanCompS project has developed a component-based approach to formal semantics. Here, we present the tools we have implemented to support component-based language definitions, including semantics-based program execution. The talk includes a demonstration of the use of the tools.

## 1 Introduction and Background

The benefits of formal semantics are well known at NWPT. However, it requires a lot of work to produce a complete and accurate formal semantics for a major language; and when the language evolves, large-scale revision of the semantics may be needed to reflect the changes. The investment of effort needed to produce an initial definition, and subsequently to revise it, can discourage language developers from using formal semantics [3].

To improve the practicality of formal semantic definitions of larger languages, the PLanCompS project [9] proposes to base them on a collection of reusable components, and to implement tool support for their development and testing. Analogous practices are widely adopted in software engineering: developers rely on reusable components in the form of packages, and on IDEs when coding and testing.

**Component-based semantics.** 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 a modular variant of SOS [6, 7]. The collection of funcons is open-ended; crucially, adding new funcons never requires changes to the definition or use of previous funcons.

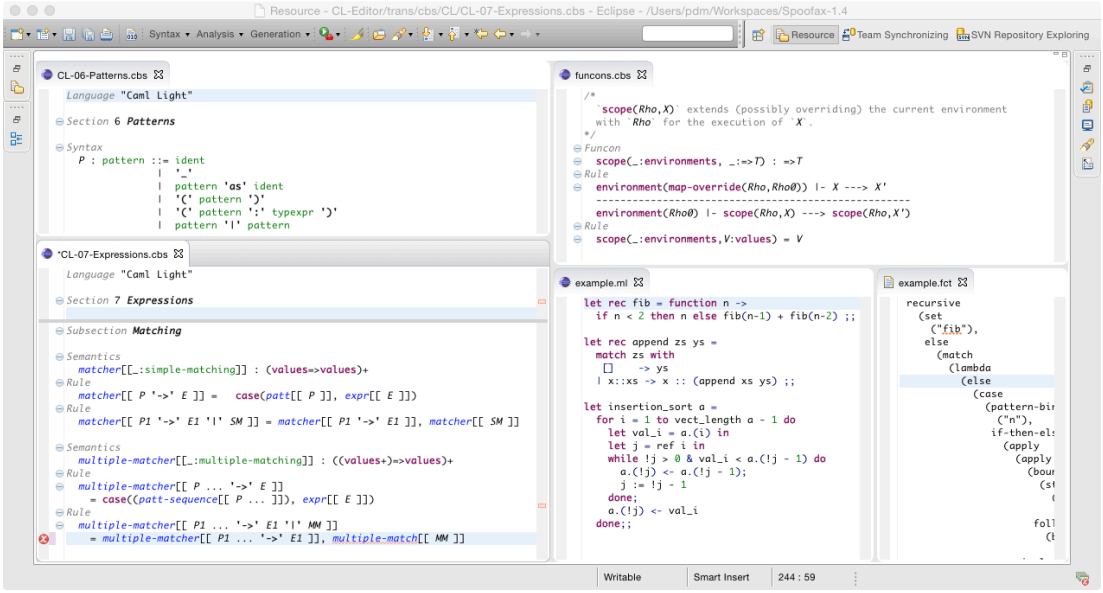
A component-based semantics of a programming language is defined by translating its constructs to funcons. The expectation is that many funcons can be widely reused in the definitions of different languages. An initial case study [1] gave a semantics for Caml Light [5] based on a preliminary collection of funcons; after completing a further case study (C♯), the funcons used in the two language definitions are to be finalised and made freely available in a digital library.

**Contributions.** We introduce a unified meta-notation called CBS for defining abstract syntax of programming languages, translations from language constructs to funcons, and the semantics of the funcons themselves. To accompany CBS, we provide a complete tool chain for executing translation functions and running the resulting funcon terms. A further contribution is illustrating the usefulness of Spoofax [4] for generating IDEs to support semantic frameworks.

**Related work.** Other tools supporting development of semantic definitions and/or semantics-based program execution include the ASM tools, Ott, PLTRedex, the K tools, Maude, Melange, and DynSem. Some of the semantic frameworks supported by these tools have a high degree of modularity, but we are not aware of any that provide a collection of reusable components, apart from an exploratory definition of a modest collection of funcons in K [8].

## 2 Developing and Executing Language Definitions

We have implemented an IDE in Eclipse to support development and testing of component-based semantics. We have used Spoofax [4] 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 screenshot shows several files open during the development of the component-based semantics of Caml Light (CL). The top left pane is browsing the CL abstract syntax in the CBS definition of CL patterns. In the lower part of the bottom left pane, a CBS rule defining the translation of CL multiple matchings to funcons is being edited; the red mark in the margin flags an undeclared symbol. The colours and fonts distinguish the names of syntax nonterminals (green), funcons (red), semantic functions (blue italic), and variables (black italic).

Clicking on a name in a CBS editor shows its definition in a separate pane. The top right pane shows the definition of the operational semantics of the funcon for scoping declarations. The two panes on the lower right show a small CL test program and part of its translation to funcons. When the focus is on a CBS file specifying the semantics of CL, there is a button to (re)generate an executable translator from CL to funcons. While editing a CL program, the same button translates it to a funcon term (which can then be executed, see Sect. 3). On rebuilding the project, any open files with the results of translating test programs to funcons are updated accordingly. Entire test suites can be translated from a shell command line.

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. Each semantic equation in CBS generates a corresponding Stratego rule, e.g.:

```
Rule                                                    to-funcons:
  matcher[[ P '->' E ]] =          generates              |[ matcher[: (P)->(E) :] ]| ->
    case(patt[[ P ]], expr[[ E ]])                        |[ case(patt[: (P) :], expr[: (E) :]) ]|
```

The generated SDF3 grammars provide the syntax for the semantic functions and metavariables that occur in the generated Stratego rules.

# 3 Executing Funcon Terms

We execute funcon terms using an interpreter written in Haskell. The interpreter provides an implementation of I-MSOS [7] specifications, the modular variant of SOS that CBS uses to specify funcons and semantic entities. The interpreter can be invoked from within Eclipse with its output printed to Eclipse's console .

The defining feature of I-MSOS is the implicit propagation of entities, and this is achieved in Haskell by using a monad in the implementation of the small-step evaluation function. The Haskell code corresponding to the CBS specifications of the individual funcons and semantic entities is systematically derived from the CBS rules. The use of a monad allows the resulting code to be as modular as I-MSOS rules: adding a new funcon or semantic entity requires no modification to the code for the existing funcons or semantic entities. Deriving the Haskell code is currently performed manually, although our aim is automate this process.

The CBS language includes a fixed universe of value types, and a set of operations on those types; these are provided 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.

Dynamic errors are handled gracefully by the interpreter, which reports the immediate cause of the error along with the current contents of the semantics entities and funcon term remaining to be executed. The interpreter also includes a parser and pretty printer for funcon terms, and an optional refocusing-based optimisation [2] that provides a more efficient evaluation strategy.

# 4 Conclusion

In a full version of this paper, we will explain how the CBS meta-notation supports modular specifications of funcons and semantic entities, and how these specifications can be translated to modular Haskell code. We will also explain how by categorising our semantic entities, we allow for the modular addition of new entities without concern for the order in which those entities are added, in contrast to a conventional approach using monad transformers.

# References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] P. D. Mosses. Modular structural operational semantics. *J. LAP*, 60-61:195–228, 2004.

[7] 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.

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

[9] PLanCompS: Programming language components and specifications. `http://www.plancomps.org`.