Software meta-language engineering and CBS

Mosses, Peter D.

DOI
10.1016/j.jvlc.2018.11.003

Publication date
2019

Document Version
Accepted author manuscript

Published in
Journal of Computer Languages

Software meta-language engineering and CBS

Peter D. Mosses[1]

Department of Computer Science, Swansea University,
Computational Foundry, Bay Campus, Swansea SA1 8EN, United Kingdom

Abstract

The SLE conference series is devoted to the engineering principles of software languages: their design, their implementation, and their evolution. This paper is about the role of language specification in SLE. A precise specification of a software language needs to be written in a formal meta-language, and it needs to co-evolve with the specified language. Moreover, different software languages often have features in common, which should provide opportunities for reuse of parts of language specifications. Support for co-evolution and reuse in a meta-language requires careful engineering of its design.

The author has been involved in the development of several meta-languages for semantic specification, including action semantics and modular variants of structural operational semantics (MSOS, I-MSOS). This led to the PLanCompS project, and to the design of its meta-language, CBS, for component-based semantics. CBS comes together with an extensible library of reusable components called 'funcons', corresponding to fundamental programming constructs. The main aim of CBS is to optimise co-evolution and reuse of specifications during language development, and to make specification of language semantics almost as straightforward as context-free syntax specification.

The paper discusses the engineering of a selection of previous meta-languages, assessing how well they support co-evolution and reuse. It then gives an introduction to CBS, and illustrates significant features. It also considers whether other current meta-languages might also be used to define an extensible library of funcons for use in component-based semantics.

Keywords: semantics of programming languages, meta-languages, modularity

1. Introduction

In general, it is good engineering practice to produce a full design specification of a new artefact before starting its construction. If the design needs to be adjusted during the construction, or a new version of the artefact is subsequently required, the design specification is updated accordingly. Moreover, a design often makes extensive use of pre-existing components that have precisely specified properties.

In software language engineering, however, developers seldom produce *complete and precise* language design specifications. This seems to be at least partly because of the effort required to specify a major software language in full detail, and subsequently co-evolve the specification together with the specified language. Perhaps a component-based approach could reduce the effort, and encourage language developers to specify the designs of new languages before implementing them?

The rest of this section recalls some general features of formal language specification, and discusses the relationship between formality and co-evolution. Section 2 examines some previous meta-languages, pointing out issues with co-evolution and reuse. Section 3 introduces CBS, a component-based framework for language specification; it illustrates how CBS facilitates co-evolution, then gives an overview of the initial library of reusable components provided with CBS. Section 4 indicates the current status of CBS and plans for its further development.

Email address: p.d.mosses@swansea.ac.uk (Peter D. Mosses)

[1]Present address: EEMCS, Programming Languages, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands

Preprint submitted to JVLC, accepted 19 November 2018. *DOI:* https://doi.org/10.1016/j.jvlc.2018.11.003

This article is based on the author's keynote at SLE 2017, extending [1]. Its contribution is an analysis of the support for co-evolution and reuse in selected meta-languages, together with an explanation of relevant CBS features; it does not present previously unpublished research results.

1.1. Formal language specification

A language specification defines requirements on implementations: which texts an implementation is to accept as well-formed, and what behaviour should be exhibited when executing such texts.[2] For conventional high-level programming languages, well-formedness may be divided into lexical syntax, context-free phrase structure, and context-sensitive constraints, all to be checked before program execution starts; the behavioural requirements generally include the relation between input and output, but exclude properties such as how much time or space program execution should take. Context-sensitive constraints are also referred to as static semantics, and behavioural requirements as dynamic semantics.

A precise specification of a software language needs to be written in a formal *meta-language* (a language for specifying languages) and validated for consistency and completeness: natural language, however carefully formulated, is inherently imprecise, and not amenable to validation, hence unsuitable as a meta-language. Language reference manuals often specify lexical and context-free syntax using formal grammars, written in some variant of the BNF meta-language; validation is supported by tools that generate parsers from grammars.

Developers of major software languages have themselves produced complete formal specifications of semantics in only a few cases. An early example was Ada 83: the US Department of Defense required the developers to deliver a formal specification of the language semantics together with the implementation [2]. Another example was Standard ML, where the language developers voluntarily specified its semantics [3, 4]. But for Haskell, the developers wrote [5]:

> *One of our explicit goals was to produce a language that had a formally defined type system and semantics. We were strongly motivated by mathematical techniques in programming language design. We were inspired by our brothers and sisters in the ML community, who had shown that it was possible to give a complete formal definition of a language [. . .]. Nevertheless, we never achieved this goal. [. . .] No one undertook the work, and in practice the language users and implementers seemed to manage perfectly well without it.*

In fact the developers did make use of formal semantics for *fragments* of the language (ibid.):

> *[. . .] at many times during the design of Haskell, we resorted to denotational semantics to discuss design options, as if we all knew what the semantics of Haskell should be, even if we didn't write it all down formally.*

A recent example where language developers have found it worthwhile to produce a complete formal specification is WebAssembly [6]: the language documentation uses formal meta-languages to specify both the static and dynamic semantics. However, WebAssembly is quite small, and not intended as a programming language; the developers of most major programming languages (C, C++, C#, Java, OCaml, Scala, etc.) appear to be as reluctant as the Haskell developers to produce complete design specifications.

1.2. Co-evolution of languages and specifications

Major software languages always evolve – some slowly, others quite rapidly. New versions often reflect significant changes. For example, Ada 95 added object-orientation to Ada 83; further evolution led to new versions in 2005 and 2012. Standard ML, originally released in 1990, was revised only once, in 1997. Also the original version of Haskell was released in 1990, but by 1997 there had already been four further versions of the language design; the Haskell 98 Report was published in 1999 (and with minor revisions in 2002). Starting with Haskell 2010 [7], the language was supposed to evolve in small, agreed steps, with annual revisions of the Haskell Report, but somehow that did not happen – a new version is currently planned for 2020.

How about co-evolution of languages and their specifications? It appears this was not even attempted for Ada: only Ada 83 was formally speci-

[2] Software languages and meta-languages can both be textual and/or graphical; we here consider purely textual languages, for simplicity.

fied. (The developers specified the sequential sublanguage themselves [8], but left the concurrency constructs to other projects [9].)

The developers of Standard ML successfully coevolved their specification with the language revision in 1997 [4]:

> *In the 1990 Definition it was predicted that further versions of the Definition would be produced as the language develops, with the intention to minimise the number of versions. This is the first revised version, and we foresee no others.*

There were no changes at all to the parts of the specification concerned with constructs that were unaffected by the language revision.

The Haskell developers regarded the need for coevolution of specifications with language revisions as a hindrance to evolution [5]:

> *[. . .] we always found it a little hard to admit that a language as principled as Haskell aspires to be has no formal definition. But that is the fact of the matter, and it is not without its advantages. In particular, the absence of a formal language definition does allow the language to* evolve *more easily, because the costs of producing fully formal specifications of any proposed change are heavy, and by themselves discourage changes.*

Simon Peyton Jones expanded on that last point in a subsequent interview [10, page 196]:

> *We keep changing Haskell. If I have to formalize every aspect of that change, that is quite a big brake on the changes in the language, and that's actually happened to ML. It's quite hard to change ML, precisely because it has a formal description.*

When the SML/NJ implementation of Standard ML was extended with concurrency primitives, it turned out to be impossible to co-evolve the language definition to include their specification [11, §5], and the definition of the Concurrent ML language involved a complete reformulation of the dynamic semantics of Standard ML constructs [12, 13]. However, that was due to the (big-step) relational style of semantics used in the Definition of Standard ML, rather than the formality of the specification: when using a (small-step) transitional style, a functional language can in fact be extended with concurrency constructs – even without requiring any changes at all to the formal specification of the original constructs [14, 15].

2. Some previous meta-languages

In this section, we recall the main features of several meta-languages used for specifying dynamic semantics, and discuss their support for co-evolution and reuse. Section 2.1 considers some particular styles of denotational semantics; Sect. 2.2 focuses on structural variants of operational semantics; and Sect. 2.3 is about a hybrid of denotational and operational semantics called action semantics.

We illustrate each meta-language with a simple running example: let-expressions, with the following syntax:

$$E ::= \texttt{let}\ I\texttt{=}E\ \texttt{in}\ E \mid \ldots$$

The meta-variable E ranges over the set *Expr* of all expressions. Informally, the intended behaviour of evaluating '$\texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2$' is first to evaluate E_1 to some value v, then evaluate E_2 with the identifier I bound to v. Such let-expressions allow multiple references to the value of an expression without its re-evaluation. Expression evaluation could have side-effects, such as assignment to imperative variables (as in ML) or spawning concurrent processes (as in Concurrent ML); the development of a new language might start with simple expressions that have no side-effects, but introduce them at a later stage.

The current bindings of identifiers can be represented mathematically by an *environment* $\rho : Env$ mapping identifiers to values determined by their respective declarations: constants, variables, procedures, etc. Similarly, the current assignments to mutable variables can be represented by a store $\sigma : S$ mapping locations of variables to their values. (The distinction between identifiers and locations is motivated not only by fundamental concepts of programming languages, but also by allowing a simple specification of aliasing: different identifiers can be mapped by the environment to the same location.) The environment $\rho[v/I]$ maps the identifier I to v, and other identifiers as ρ.

2.1. Denotational semantics

The main general feature of the meta-language used in denotational semantics is the inductive def-

inition of families of semantic functions $\mathcal{F}_i$ that map phrases of programs to mathematical entities (usually continuous functions on Scott-domains) representing their behaviour. The definition of each $\mathcal{F}_i$ is presented as a set of semantic equations of the form:

$$\mathcal{F}_i[\![\ldots t_1 \ldots t_n \ldots]\!] = \ldots \mathcal{F}_{i_1}[\![t_1]\!] \ldots \mathcal{F}_{i_n}[\![t_n]\!] \ldots$$

Each equation specifies how the denotations of phrases are composed from the denotations of their subphrases. The lack of dependence on the form of the subphrases ensures *compositionality*: replacing a sub-phrase by one with the same denotation does not affect the denotation of the enclosing phrase.

For example, when the semantic function $\mathcal{E}$ maps expressions E_i to their denotations, the denotations of let-expressions are specified by an equation of the form:

$$\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!] = \ldots \mathcal{E}[\![E_1]\!] \ldots \mathcal{E}[\![E_2]\!] \ldots$$

The notation used on the right usually employs some variant of λ-notation, where '$\lambda x. \ldots x \ldots$' expresses a function of an argument x. Note that 'fxy' is grouped as '$(fx)y$', and '$\lambda x.fy$' as '$\lambda x.(fy)$'.

Scott–Strachey style. In the original style proposed by Dana Scott and Christopher Strachey [16], the denotation $\mathcal{E}[\![E]\!]$ of an expression E in a pure functional language could be an element of the domain $Env \to V$, where the environment $\rho : Env$ determines the values of identifiers used in E. Then the semantics of let-expressions can be defined thus:

$$\begin{array}{l}\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!] = \\ \quad \lambda\rho.\mathsf{strict}(\lambda v.\mathcal{E}[\![E_2]\!](\rho[v/I]))(\mathcal{E}[\![E_1]\!]\rho)\end{array} \tag{1}$$

where $\mathsf{strict}f$ is the function whose value is undefined when its argument is undefined, and otherwise as given by f.

In fact Scott and Strachey (ibid.) considered a language that included mutable variables and assignment statements. Suppose that expressions can inspect the values assigned to variables. We can easily change the definition of the domain of denotations to $Env \to S \to V$, but the semantic equations for *all* expressions then need reformulating accordingly. For example, (1) becomes:

$$\begin{array}{l}\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!] = \\ \quad \lambda\rho.\lambda\sigma.\mathsf{strict}(\lambda v.\mathcal{E}[\![E_2]\!](\rho[v/I])\sigma)(\mathcal{E}[\![E_1]\!]\rho\sigma)\end{array} \tag{2}$$

If expressions can also assign to variables, we can change the domain to $Env \to S \to (V \times S)$, where the resulting store reflects any side-effects of expression evaluation – the denotation of a constant simply returns the pair of its value and the unchanged store. Scott and Strachey defined an infix operation '$*$' such that $(f * g)(\sigma) = f(v)(\sigma')$ when $g(\sigma) = (v, \sigma')$, otherwise undefined. Using this operation, (2) can be reformulated thus:

$$\begin{array}{l}\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!] = \\ \quad \lambda\rho.((\lambda v.\mathcal{E}[\![E_2]\!](\rho[v/I])) * \mathcal{E}[\![E_1]\!]\rho)\end{array} \tag{3}$$

For any ρ and σ, when $\mathcal{E}[\![E_1]\!]\rho\sigma = (v, \sigma')$ and $\mathcal{E}[\![E_2]\!](\rho[v/I])\sigma' = (v', \sigma'')$ we have $\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!]\rho\sigma = (v', \sigma'')$.

Continuation-passing style. For languages where the flow of control can change abruptly (e.g., due to jumps to labels), Strachey and Christopher Wadsworth proposed using continuations as denotations [17]. In this style, the denotation of an expression could be an element of $Env \to K \to C$, where the domain $K = (V \to C)$ represents continuations dependent on values of expressions, and the domain $C = (S \to S)$ represents continuations that simply map stores to stores.

The denotation of a constant with value v, argument continuation κ, and current store σ, returns the store $\kappa v\sigma$ determined by the continuation κ, which corresponds to "the rest of the program"; the denotation of an expression that is to terminate the whole program abruptly can simply ignore κ and return σ. The semantics of let-expressions (3) should now be reformulated as follows:

$$\begin{array}{l}\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!] = \\ \quad \lambda\rho.\lambda\kappa.\mathcal{E}[\![E_1]\!]\rho(\lambda v.\mathcal{E}[\![E_2]\!](\rho[v/I])\kappa)\end{array} \tag{4}$$

If $\mathcal{E}[\![E_1]\!]\rho\kappa_1\sigma = \kappa_1 v\sigma'$, where $\kappa_1 = \lambda v.\mathcal{E}[\![E_2]\!](\rho[v/I])\kappa$, and $\kappa_1 v\sigma' = \kappa v'\sigma''$, we have $\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!]\rho\kappa\sigma = \kappa v'\sigma''$.

Monadic style. Eugenio Moggi proposed using elements of monads as denotations [18]. In this style, the denotation of an expression could be an element of the monad $EnvT\ Env\ (StateT\ S\ Id)\ V$ where $EnvT\ Env\ M$ transforms a monad M into one that supports dependence on an environment $\rho : Env$, $StateT\ S\ M$ transforms M into a monad that supports mutable stores $\sigma : S$, and Id is the monad for pure dataflow. The semantic equation for let-expressions can then be written:

$$\begin{array}{l}\mathcal{E}[\![\texttt{let } I\texttt{=}E_1 \texttt{ in } E_2]\!]\rho = \\ \quad \mathcal{E}[\![E_1]\!] \ \mathsf{then} \\ \quad \lambda v.\mathsf{rdEnv}(\lambda\rho.\mathsf{inEnv}(\rho[v/I])(\mathcal{E}[\![E_2]\!]))\end{array} \tag{5}$$

We have $EnvT\ Env\ (StateT\ S\ Id)\ V = Env \to S \to (V \times S)$; the functions '$\mathsf{then}$', '$\mathsf{rdEnv}$', and '$\mathsf{inEnv}$' are defined by $(f\ \mathsf{then}\ g)\rho = (g\rho) * (f\rho)$ (using Scott and Strachey's '$*$' operation), $\mathsf{rdEnv}(f) = f$, and $\mathsf{inEnv}(\rho')(f)\rho = f\rho'$.

Co-evolution: With both the original Scott–Strachey style and the continuation passing style, changes or extensions to the specified language may require extra components in the domains of denotations, which entails modification to all semantic equations that involve those denotations. The patterns of λ-notation used to express denotations of language constructs generally depend on the structure of the domains, and become ill-formed when that structure changes.

Also changing from the original Scott–Strachey style to continuation-passing style requires reformulation of all semantic equations. Adding concurrency constructs would require the use of domains that represent power sets, and significant further reformulation of the semantic equations.

In contrast, the monadic style supports co-evolution rather well: adding an extra monad transformer to provide a richer monad of denotations, or transforming to a monad for continuations, does not require any changes at all to the semantic equations. However, operations such as 'rdEnv' may need to be redefined, to lift them through the new monad layer (or operations of the new monad through the environment layer, depending on the order of composition).

Reuse: With the Scott–Strachey and continuation-passing styles, the only reusable items are those provided by Scott-domains: constructors for sums, products, and (continuous) function domains, and an extended λ-notation for expressing elements of those domains.

Using the monadic style, collections of monad transformers could be made available for reuse. The names and notation for monad transformers, and for the operations that it supports, might vary between collections, so language specifications reusing monad transformers would need to refer to a specific collection, or copy and paste the required definitions.

2.2. Structural operational semantics

An operational semantics for a software language usually represents program behaviour by a set of states and a transition relation, and a computation consists of a sequence of transitions between states. A state records previously computed values, together with the syntax of the program being executed (possibly discarding those parts of the program which have already been executed). The potential flow of control through the program may be represented directly (e.g., by moving a pointer through the program, as in Abstract State Machines) or left implicit in a set of rules for locating which part of the program is next to be executed.

In *structural* operational semantics (SOS), the potential transitions for a phrase of the program are determined by inference rules, and they may depend only on the potential transitions for subphrases. This dependency determines the potential flow of control without use of an explicit program pointer. In reduction semantics, the potential flow of control is specified by a context-free grammar for reduction contexts.

Gordon Plotkin introduced structural operational semantics for programming languages in a lecture course in 1981. The lecture notes (eventually published in a journal [19]) illustrate the specification of a wide range of programming constructs using unlabelled transitions; Plotkin has also given an SOS for Communicating Sequential Processes using labelled transitions.

In SOS, environments ρ represent the current bindings, and stores σ represent the values of mutable variables (as in denotational semantics, except that in SOS, bound or stored values are required to be finite entities). In a language without mutable variables, stores are not needed, and the SOS of let-expressions can be specified using transition formulae $\rho \vdash E \to E'$, holding when a state consisting of E and ρ can make a transition to E' (ρ is implicitly preserved). The meta-variables V_1, V_2 used below range over expressions that correspond to computed values (e.g., Boolean constants and literal numbers), which are terminal states with no transitions.

$$\frac{\rho \vdash E_1 \to E_1'}{\rho \vdash \mathtt{let}\ I\mathtt{=}E_1\ \mathtt{in}\ E_2 \to \mathtt{let}\ I\mathtt{=}E_1'\ \mathtt{in}\ E_2} \tag{6}$$

$$\frac{\rho[V_1/I] \vdash E_2 \to E_2'}{\rho \vdash \mathtt{let}\ I\mathtt{=}V_1\ \mathtt{in}\ E_2 \to \mathtt{let}\ I\mathtt{=}V_1\ \mathtt{in}\ E_2'} \tag{7}$$

$$\rho \vdash \mathtt{let}\ I\mathtt{=}V_1\ \mathtt{in}\ V_2 \to V_2 \tag{8}$$

If mutable variables can be inspected but not assigned by expressions, transition formulae can be changed to $\rho \vdash \langle E, \sigma\rangle \to E'$, to reflect dependence on the current store σ, and the specified rules for evaluating let-expressions need to be reformulated thus:

$$\frac{\rho \vdash \langle E_1, \sigma\rangle \to E_1'}{\rho \vdash \langle \texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2, \sigma\rangle \to \texttt{let}\ I\texttt{=}E_1'\ \texttt{in}\ E_2} \quad (9)$$

$$\frac{\rho[V_1/I] \vdash \langle E_2, \sigma\rangle \to E_2'}{\rho \vdash \langle \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2, \sigma\rangle \to \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2'} \quad (10)$$

$$\rho \vdash \langle \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ V_2, \sigma\rangle \to V_2 \quad (11)$$

When expressions can also assign to variables, transition formulae become $\rho \vdash \langle E, \sigma\rangle \to \langle E', \sigma'\rangle$, and all the rules need reformulating again:

$$\frac{\rho \vdash \langle E_1, \sigma\rangle \to \langle E_1', \sigma'\rangle}{\rho \vdash \langle \texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2, \sigma\rangle \to \langle \texttt{let}\ I\texttt{=}E_1'\ \texttt{in}\ E_2, \sigma'\rangle} \quad (12)$$

$$\frac{\rho[V_1/I] \vdash \langle E_2, \sigma\rangle \to \langle E_2', \sigma'\rangle}{\rho \vdash \langle \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2, \sigma\rangle \to \langle \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2', \sigma'\rangle} \quad (13)$$

$$\rho \vdash \langle \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ V_2, \sigma\rangle \to \langle V_2, \sigma\rangle \quad (14)$$

Adding concurrency constructs to expressions would require further components of transition formulae (usually written as labels above the arrows), with corresponding reformulation of all the rules; similarly for abrupt flow of control.

Co-evolution: As with the original Scott–Strachey style of denotational semantics, changes or extensions to the specified language may require changes to transition formulae, and modification of all inference rules that involve those formulae. However, specification of non-determinism in SOS is straightforward, as SOS is based on relations (rather than functions), and evolving from a deterministic language to a non-deterministic one does not require reformulation of previous rules.

Reduction semantics using evaluation contexts generally supports co-evolution better than SOS, and appears to be particularly well-suited to specifying obscure control flow, e.g., delimited continuations. However, changing from SOS to reduction semantics usually involves definition of notation for substitution (e.g., $[E_1/I]E_2$), and the elimination of environments. For example, a reduction semantics for let-expressions might specify the following rule:

$$\texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2 \to [V_1/I]E_2 \quad (15)$$

together with a grammar for evaluation contexts allowing reduction of E_1 (but not E_2) in $\texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2$.

Reuse: The inference rules that specify the transitions for a particular programming language construct are generally independent of the rules for other constructs, and might be considered as reusable when specifying other languages that include the same construct, provided that the states have the same components. However, the (abstract or concrete) syntax grammar for constructs that have the same behaviour often varies between languages, which can prevent verbatim reuse.

Modular SOS. The author has developed a variant of SOS with a high degree of modularity [14]. The original motivation was that the original SOS definition of the action notation (AN) used in action semantics (see Sect. 2.3) had poor modularity. Keith Wansbrough and John Hamer had given a highly modular denotational semantics for AN in the monadic style [20]; unless the SOS of AN could be made equally modular, there would be strong pressure to switch to a denotational semantics for AN. Fortunately, a solution was found, and the resulting Modular SOS (MSOS) meta-language was used to give a revised operational definition of AN [21].

MSOS is based the use of labelled transitions

$$E \xrightarrow{\{\cdots\}} E'.$$

The MSOS notation for labels introduced in [22] uses '$\cdots$' to stand for the unmentioned entities, and '$-$' to indicate that a transition does not have any effect on the unmentioned entities. A complete label is enclosed in braces, e.g., '$\{\rho, \cdots\}$', with entities separated by commas.

Evaluation of let-expressions (regardless of whether expressions can inspect or assign to mutable variables) can be specified as follows in MSOS:

$$\frac{E_1 \xrightarrow{\{\cdots\}} E_1'}{\texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2 \xrightarrow{\{\cdots\}} \texttt{let}\ I\texttt{=}E_1'\ \texttt{in}\ E_2} \tag{16}$$

$$\frac{E_2 \xrightarrow{\{\rho=\rho_0[V_1/I],\cdots\}} E_2'}{\texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2 \xrightarrow{\{\rho=\rho_0,\cdots\}} \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2'} \tag{17}$$

$$\texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ V_2 \xrightarrow{\{-\}} V_2 \tag{18}$$

Superficially, the only differences from SOS are that states are now purely syntactic (enriched with computed values) and that entities (environments, stores, etc.) can now be written only in the labels on transitions. To ensure modularity, however, labels should not mention entities that are irrelevant to the construct being executed. For example, the transition formulae in (17) above necessarily involve environments, but it is irrelevant to let-expressions whether expression evaluation might inspect or update the current store, so no mention is made of σ; and in (16) and (18), there is no mention of any entities at all.

In SOS, the labels on adjacent transitions in a computation are generally unconstrained. In MSOS, in contrast, they are required to be *composable*:[3] entities like environments have to be identical; and entities such as pairs of stores (representing potential changes to assigned values) have to be joined-up.

Co-evolution: MSOS fully supports co-evolution: adding new entities for use in the specification of new constructs does not require any changes at all to the rules for existing constructs. For example, when a pure functional language is extended with mutable variables and/or concurrency primitives, the specified MSOS transition rules do not change [14, 15].[4]

Reuse: The inference rules that specify the transitions for a particular programming language construct are more reusable than in SOS, since ignored entities do not need to be made explicit. However, the (abstract or concrete) syntax for constructs that have the same behaviour often varies between languages, which can prevent verbatim reuse.

Implicitly-modular SOS. The I-MSOS metalanguage, developed in collaboration with Mark New [23], makes a significant notational improvement on MSOS. In I-MSOS, rules are written using the conventional SOS notation for transition formulae, but interpreted as the corresponding MSOS rules. This provides formal foundations for the common convention of leaving entities implicit in SOS transition rules when they are simply propagated between premises and conclusions. I-MSOS also eliminates the somewhat clumsy label notation used in MSOS. The rules for let-expressions now look as simple as possible:

$$\frac{E_1 \to E_1'}{\texttt{let}\ I\texttt{=}E_1\ \texttt{in}\ E_2 \to \texttt{let}\ I\texttt{=}E_1'\ \texttt{in}\ E_2} \tag{19}$$

$$\frac{\rho[V_1/I] \vdash E_2 \to E_2'}{\rho \vdash \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2 \to \texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ E_2'} \tag{20}$$

$$\texttt{let}\ I\texttt{=}V_1\ \texttt{in}\ V_2 \to V_2 \tag{21}$$

Rule (19) above merely specifies that E_1 is evaluated before E_2; it could be implied by a so-called strictness annotation in the grammar of let-expressions, following the $\mathbb{K}$ framework [24], or by a grammar for evaluation contexts [25].

Support for co-evolution and reuse in I-MSOS is exactly as in MSOS: the only drawback is the dependence of transition rules on the grammar of the specified language, which can prevent their verbatim reuse.

2.3. Action semantics

The action semantics framework was initially developed in collaboration with David Watt [26, 27]. Here, the behaviour of program phrases is defined by translation to actions, expressed in a notation that provides a rich collection of action primitives and combinators (including dataflow, control flow, binding, storing, nondeterminism, process spawning, and asynchronous message-passing). The intended interpretation of action notation was defined operationally, using a combination of SOS and algebraic equations.

[3] Formally: each entity corresponds to a morphism of a category, labels are morphisms of indexed product categories, and traces of computations are required to be paths in the label category.

[4] The cited references use a less perspicuous notation for labels than that illustrated above, but the difference does not affect co-evolution.

For example, the infix action combinator 'A_1 then A_2' passes the value computed by A_1 to A_2, where it can be referenced as 'the given value'; the composite action 'furthermore A_1 hence A_2' overrides the current bindings for A_2 by those computed by A_1. The action semantics of let-expressions is specified as follows.

$$\begin{aligned}&\text{evaluate}[\![\text{“let”}\ I{:}\text{Id}\ \text{“=”}\ E_1{:}\text{Expr}\ \text{“in”}\ E_2{:}\text{Expr}]\!] = \\ &\qquad \text{furthermore} \\ &\qquad (\text{evaluate}\ E_1\ \text{then bind}\ I\ \text{to the given value}) \\ &\qquad \text{hence evaluate}\ E_2\end{aligned} \tag{22}$$

Co-evolution: Action semantics was developed to support co-evolution. The above specification of the action semantics of let-expressions (22) necessarily makes use of a combinator that (implicitly) affects the current bindings. But it is independent of whether expressions might inspect or update mutable variables, send or receive messages, or spawn processes. The type 'value' can include whatever types of data are expressible in the specified language.

Reuse: The definitions of the actions and data types provided by the framework are implicitly reused in all language specifications. However, action notation is not extensible: an action semantics for a language has to translate it to the pre-defined action primitives and combinators.

'Constructive' action semantics was subsequently developed in collaboration with Mark van den Brand and Jørgen Iversen [28]. Here, programming languages are translated to BAS, an open-ended collection of 'basic abstract syntax' constructors. The semantics of each constructor is defined using action notation, independently of any programming language. Constructive action semantics was the direct precursor of CBS, which uses a variant of I-MSOS (instead of action notation) to define fundamental programming constructs, as explained in the next section.

3. CBS: Component-based semantics

The CBS meta-language for component-based semantics has been developed by the PLanCompS project [29]. CBS comes together with an extensible library of reusable components called 'funcons', corresponding to fundamental programming constructs. Languages are specified in CBS by translating their constructs to (combinations of) funcons. When a language evolves, co-evolution requires updating only the translation of its affected constructs. The specifications of the funcons are fixed, and do not change when funcons are used together.

Conjecture. *A* component-based *semantic meta-language such as CBS can significantly reduce the effort of formally specifying languages and their evolution.*

The CBS meta-language was engineered to optimise co-evolution and reuse, and to make language semantics almost as straightforward as context-free syntax. Careful comparison of CBS examples with specifications of the same languages in other frameworks could provide solid evidence for the above conjecture; but that has not yet been carried out, and we have to leave evaluation of the credibility of the conjecture to the reader's judgement.

The notion of 'component' in CBS is somewhat simpler than in component-based software engineering (CBSE) or component-based development (CBD). The interface for using a funcon is just its signature, determining how many arguments it can take, their types, and the type of values that it may compute. Well-formed funcon terms can be freely composed: no 'contracts' are specified regarding requirements and guarantees on the potential effects when funcons are executed.[5]

Note especially that groups of funcons are *not* encapsulated in modules (although related funcons are generally specified in the same file, and files may be divided into sections, for convenience of browsing). Moreover, CBS has no explicit import mechanism: all funcons are globally visible. Crucially, each funcon is to have a *fixed* definition in the library, so occurrences of the same funcon name always refer to the same definition; this is possible because of the inherent modularity of operational specifications in CBS, which are based on I-MSOS.

In Sect. 3.1, we give some simple examples to illustrate how CBS supports language specification and co-evolution, and how the same funcon can be used in the specification of different language

[5] Behavioural properties of funcons (e.g., associativity of sequential composition) can be proved from their definitions [30].

```
Syntax
  E : expr ::= 'let' id '=' expr 'in' expr | ...

Semantics
  eval[[ _ : expr ]] : =>values

Rule
  eval[[ 'let' I '=' E1 'in' E2 ]] =
    scope(bind-value(I, eval[[E1]]), eval[[E2]])
```

Table 1: CBS of let-expressions

```
Syntax
  S : stmt ::= 'while' '(' expr ')' stmt | ...

Semantics
  exec[[ _ : stmt ]] : =>null-type

Rule
  exec[[ 'while' '(' E ')' S ]] =
    while-true(eval[[E]], exec[[S]])
```

Table 2: CBS of simple while-statements with Boolean conditions

```
Rule
  exec[[ 'while' '(' E ')' S ]] =
    while-true(not is-equal(0, eval[[E]]), exec[[S]])
```

Table 3: Co-evolution of CBS of simple while-statements for numeric conditions

constructs. In Sect. 3.2, we introduce the library of funcons developed by the PLanCompS project. Section 3.3 illustrates how funcons are specified in CBS. In Sect. 3.4, we consider whether other meta-languages could be used to specify funcons.

Further examples of CBS specifications of languages (including MiniJava and OCaml Light), are available online at `https://plancomps.github.io/CBS-beta`, together with an initial library of funcons. The CBS of OCaml Light is an update of a specification of the Caml Light language in an earlier version of CBS, presented in [31]. OCaml Light and Caml Light are substantial languages in the ML family; their specifications in CBS demonstrate how translation to funcons scales up for medium-sized languages. A further case study (C#) has been initiated to test how well CBS can cope with a major programming language.

After the current review period for the beta-release of CBS and the initial library of funcons, the notation and specifications of those funcons are to be fixed, to provide a solid basis for their reuse in further language specifications. Further funcons can subsequently be developed, but they will be added to the library only after rigorous validation of their specifications and reusability.

3.1. Language specification in CBS

The example in Table 1 illustrates how let-expressions can be specified in CBS. '*Syntax*' introduces one or more grammar productions for the context-free syntax of the language, together with meta-variables ranging over the associated sorts of phrases. The notation corresponds closely to that used for abstract syntax in other meta-languages: *E* is the stem for meta-variables of sort '`expr`', and terminal symbols are quoted.

'*Semantics*' introduces a declaration of a semantic function that translates language phrases to funcon terms. The notation '`=>values`' here specifies that the funcon terms produced by the semantic function '*eval*' may compute any values; in a complete language specification, '`values`' could be replaced by a language-specific type of expressible values.

'*Rule*' introduces an equation defining a translation function on phrases matching a specified pattern. The symbols '`scope`' and '`bind`' are names of funcons defined in the PLanCompS library. Different meta-variables of the same sort are distinguished by suffixed digits, e.g., E_1, E_2.

To illustrate co-evolution and reuse in CBS, consider the CBS specification of simple while-statements in Table 2. When execution of a statement terminates normally, it computes '`null-value`' of type '`null-type`', which represents the lack of an ordinary value.

The signature of the funcon '`while-true`' requires its first argument to compute only Boolean values; the result of the translation of a while-statement with a numerical test expression would be an ill-typed funcon term. If a language developer wanted to change the semantics of while-statements to have numerical test expressions, the specification of the translation rule would be updated as shown in Table 3. The language specification of the while-statement has thus co-evolved, whereas the specification of the '`while-true`' funcon is reused unchanged. An alternative would be to define a new funcon '`while-non-zero`' corresponding directly to the desired semantics of while-statements, and change the original translation rule to use this new funcon instead of '`while-true`'.

A more radical evolution of the while-statement is to allow a break-statement in its body to terminate iteration abruptly. Such behaviour could be specified using general funcons for throwing and handling exceptions, but funcons specifically for

```
Syntax
  S : stmt ::= 'while' '(' expr ')' stmt | 'break' | ...

Semantics
  exec[[ _ : stmt ]] : =>null-type

Rule
  exec[[ 'break' ]] = break
Rule
  exec[[ 'while' '(' E ')' S ]] =
    handle-break(while-true(eval[[E]], exec[[S]]))
```

Table 4: Co-evolution of CBS of while-statements with break-statements

breaks are already provided by the PLanCompS library, allowing the particularly simple co-evolution of the specification of the translation shown in Table 4. It would be equally straightforward to add a continue-statement. Specifying the required semantics using such combinations of funcons in translations seems preferable to naming the combinations and adding them to the funcons library.

The PLanCompS library of funcons is strictly curated: before a funcon can be added, it has to be carefully specified, and the specification validated by unit tests and by examples of its use in at least one validated language specification.

Language developers are free to define funcons for their own use within a particular language specification, but these are local, and cannot be referenced from other language specifications: the only way to reuse them is by copy-and-paste. This is because language specifications are allowed to evolve, and thus do not provide a solid basis for reuse. A funcon defined in a language specification might change, or even disappear, thereby undermining any other language specification that referred to it. A local funcon may also be introduced merely to abbreviate a particular combination of funcons that is required for the semantics of many constructs in the same language; such funcons are unlikely to be useful in other language specifications.

3.2. Funcons

This section gives an overview of the initial library of funcons developed by the PLanCompS project.

Funcons correspond to fundamental programming concepts. They are language-independent, and each funcon has fixed behaviour. New funcons can be introduced without affecting the behaviour of the existing funcons.

Computations. The CBS library provides funcons for expressing and composing various kinds of computations. The following classification of funcons reflects whether they are concerned with normal or abnormal flow of control, and the kinds of entities generally required by their computations.

Normal computation: Funcons compute (sequences of) values on normal termination. The funcons provided for expressing normal computation correspond to fundamental programming constructs for normal flow of control, data flow, binding, storing, and interactive input and output. (Funcons for threads and processes are currently under development.)

Abnormal computation: Funcons may terminate abruptly, causing abnormal flow of control until a corresponding handler is encountered. Particular cases of abrupt termination include failing, throwing exceptions, returning values, breaking, and continuing. The handlers for failure and thrown exceptions execute specified funcon terms when activated, whereas the others merely terminate normally. Funcons using control signals support specification of delimited continuations and related constructs.

Values. The CBS library provides funcons for expressing and computing various kinds of values.

Primitive values: Booleans, integers, floats, characters, and a null value.

Composite values: algebraic datatypes, including tuples, lists (strings are lists of characters), vectors, trees, references (pointers or null), records, variants, classes, objects, bit-vectors; and built-in collection types, including sets, maps, multisets, and (directed) graphs.

Abstraction values: general abstractions, thunks, functions, and patterns.

Types: values representing sets of values.

Further types of values are provided in connection with particular kinds of computation (e.g., simple and composite variables for storing values).

Funcons can take sequences of values and computations as arguments, and compute sequences of values as results. A trivial value sequence of length one is equivalent to its only element. The empty value sequence represents the lack of a well-defined result (e.g., the head of an empty list).

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

Rule
  while-true(B, X)
    ~> if-true-else(B, sequential(X, while-true(B, X)), null-value)
```

Table 5: CBS of a funcon for while-statements

```
Funcon
  if-true-else(_:booleans, _:=>T, _:=>T) : =>T

Rule
  if-true-else(true, X, Y) ~> X
Rule
  if-true-else(false, X, Y) ~> Y
```

Table 6: CBS of a funcon for conditional choice

```
Rule
                    B ---> B'
  ------------------------------------------------
  if-true-else ( B, X, Y ) ---> if-true-else ( B', X, Y )
```

Table 7: A CBS rule implied by an argument type

```
Funcon
  sequential(_:null-type, _:=>T) : =>T

Rule
  sequential(null-value, Y) ~> Y
```

Table 8: CBS of a funcon for sequential execution

3.3. Funcon specification in CBS

This section shows how funcons are specified in CBS. The examples include all the computational funcons used in the illustrations of language specifications given in Sect. 3.1.

Consider the specification of the funcon for while-loops with Boolean tests in Table 5.

'*Funcon*' introduces a declaration of a fresh funcon name together with its signature. The signature of the '`while-true`' funcon specifies that the funcon takes a sequence of two computation arguments. The first argument is required to compute a value of type '`booleans`', whereas the second argument corresponds to a statement executed only for its effect, always computing '`null-value`' of type '`null-type`' on normal termination.

'*Rule*' here introduces a formula or inference rule defining the operational behaviour of a funcon. A formula written with infix '`~>`' is a context-independent rewrite, whereas general transition formulae are written with '`--->`', possibly involving explicit entities such as environments and stores (using conventional SOS notation). As in I-MSOS, entities are implicitly propagated when omitted in transition formulae.

Only one rule is needed to specify the '`while-true`' funcon. Atypically, the rule involves two other funcons, which are specified independently of '`while-true`', as shown in Tables 6 and 8. The signature of '`if-true-else`' specifies that the funcon takes a sequence of three arguments. In contrast to '`while-true`', the first argument is here specified to be a *value* of type '`booleans`', rather than a *computation* of type '`=>booleans`'. Conceptually, a value type '*T*' in a signature specifies a call-by-value argument, whereas a computation type '`=>`*T*' specifies a call-by-name argument (as in the Scala language).

The intended semantics of '`if-true-else`(*B*, *X*, *Y*)' is to compute either *X* or *Y*, depending on the value of *B*, so the second and third arguments have computation types '`=>`*T*'. Here, *T* is a type variable ranging over arbitrary types of values. When *T* is instantiated with an ordinary value type, this funcon can be used to specify the semantics of conditional expressions; with *T* as '`null-type`', it corresponds to conditional statements.

Rules for evaluating arguments having value types are left implicit. The rule shown in Table 7 does not have to be specified, since the first argument of '`if-true-else`' has a value type. If the signature of '`if-true-else`' had been specified with the computation type '`=>booleans`' for the first argument, the above rule would have to be specified.

Leaving the rules for computing value arguments implicit significantly increases the conciseness of CBS specifications. For example, Table 8 shows a *complete* CBS specification of a binary funcon that computes its arguments sequentially, discarding the value computed by the first argument.

For funcons whose signature specifies more than one argument with a value type, the implicit rules allow interleaving the computations of those arguments. For example, the signature of a binary logical conjunction funcon '`and`' specifies both arguments with value type '`booleans`'.

```
Funcon
  left-to-right(X*:(=>T)*) : =>(T)*

Rule
                                  Y ---> Y'
  ------------------------------------------------------------
  left-to-right(V*:(T)*, Y, Z*) ---> left-to-right(V*, Y', Z*)
Rule
  left-to-right(V*:(T)*) ~> V*
```

Table 9: CBS of a funcon for sequential argument evaluation

```
Type
  environments ~> maps(identifiers, values?)

Entity
  environment(_:environments) |- _ ---> _

Funcon
  bind-value(I:identifiers, V:values) : =>environments
    -> { I |-> V }

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:T) ~> V
```

Table 10: CBS of some funcons for binding

```
Datatype
  breaking ::= broken

Funcon
  break : =>empty-type ~> abrupt(broken)

Funcon
  handle-break(_:=>null-type) : =>null-type
Rule
                X --abrupted( )-> X'
  ------------------------------------------------
  handle-break(X) --abrupted( )-> handle-break(X')
Rule
                X --abrupted(broken)-> _
  ---------------------------------------
  handle-break(X) --abrupted( )-> null-value
Rule
                X --abrupted(V:~breaking)-> X'
  ------------------------------------------------
  handle-break(X) --abrupted(V)-> handle-break(X')
Rule
  handle-break(null-value) ~> null-value
```

Table 11: CBS of the funcons for breaks

The funcon '`left-to-right`', specified in Table 9 can be combined with any funcon to compute the arguments in the order in which they are written. The postfix '`*`' specifies that sequences of any length are allowed.[6] For example, the funcon term '`and`(`left-to-right`(B_1, B_2))' expresses the sequential variant of logical conjunction; conditional conjunction, which does not compute its second argument when the value of the first argument is '`false`', is expressed by '`if-true-else`(B_1, B_2, `false`)'.

The CBS specifications of the remaining funcons used in Sect. 3.1 are shown in Tables 10 and 11; explanatory comments are provided on the PLanCompS CBS-beta website.

3.4. Funcon specification in other meta-languages

Perhaps other meta-languages than CBS could be used to specify an extensible library of reusable funcons? A specification of the translation of a language to funcons defines a component-based semantics of the language in any meta-language that defines the required funcons. The following meta-languages appear to be promising candidates for specifying funcons.

- The DynSem meta-language [32] supports implicit propagation of entities in rules, much as in I-MSOS, so it might appear to be a particularly relevant alternative to CBS.

  However, DynSem is primarily for specifying big-step operational rules. Although many funcons can be specified using big-step rules, DynSem treats the premises as sequential, so it does not provide the intended semantics for

[6] The PLanCompS library specifies a slightly more general signature for this funcon, allowing each argument to compute a sequence of values.

funcons whose CBS rules allow interleaving. Specifying funcons for threads and processes might not be possible in DynSem.

Nevertheless, it would still be useful to specify a sequential sub-library of funcons in DynSem, as many languages are themselves sequential, and their translations do not require interleaving of computations. The integrated support for DynSem in the Spoofax Language Workbench [33] makes DynSem especially attractive, since tool support for generating editors and translators from CBS language specifications have also been implemented in Spoofax.

- The $\mathbb{K}$ Framework [24] is based on small-step term rewriting, and supports specification of interleaving and concurrency. Entities stored in the state are implicitly propagated by rules. However, rules are unconditional, so environments need to be explicitly stacked and unstacked in connection with nested scopes.

  Ferdinand Vesely and the author have experimented with specifying a translation from the SIMPLE language to funcons in $\mathbb{K}$, along with the semantics of the required funcons [34]. The experiment was successful, and it would be interesting to try extending the specifications to the rest of the funcons library.

- PLT/Redex [25] is based on specifying (small-step) reduction rules. As in CBS, mutable entities can be omitted in reduction rules for constructs that do not affect or inspect them. PLT/Redex supports specification of interleaving, and includes tools to explore nondeterministic semantics.

  The main potential difficulty with defining a library of independently specified funcons in PLT/Redex is specifying the required grammars for reduction contexts. Moreover, the supporting tools appear to rely on specifying the semantics of programming language constructs directly, rather than by translation to another language.

- Specifying funcons using monads and monad transformers should be possible. One potential issue is that composing monad transformers in different orders can give different semantics, whereas funcon specifications are not combined in a specific order. Another issue could be the need for widespread use of so-called resumptions to specify interleaving.

Further meta-languages for dynamic semantics with a high degree of modularity include abstract state machines [35] and algebraic effects [36]; investigation of the possibility of using them to define libraries of funcons is left to future work.

4. Conclusion

In CBS, the dynamic semantics of programming language constructs is specified by translating them to funcon terms. This appears to be significantly less effort than specifying their semantics directly using other meta-languages. When the specified language evolves, co-evolution of the translation affects only the specifications of the constructs concerned: the difference between two versions of a translation pinpoints what has changed, which (in conjunction with version control) should be useful as documentation during language development.

The definitions of individual funcons are highly reusable components of language specifications. After funcons have been made available for reuse, their definitions cannot be changed, so proposals for adding new funcons to a public library need to be carefully validated. (A language specification can include tentative local definitions of unvalidated new funcons, but these cannot be referenced from other language specifications.) Crucially, the modular variant of structural operational semantics used in CBS allows new funcons to be defined without changing previous funcon definitions.

The CBS meta-language for component-based semantics has been under development by the PLanCompS project [29] since 2011, although the main principles were envisaged already in 2004 [37]. A beta-release of the meta-language, together with an initial library of funcon specifications and some examples of language descriptions, was made available in July 2018 at `https://plancomps.github.io/CBS-beta`, to solicit comments and suggestions for improvement before the full release (currently planned for early 2019). Changes to the meta-language and the funcon specifications are possible during the beta-release period; after the full release, the specifications of the funcons in the initial library will be fixed, and made freely available for anybody to use. Tool support for using CBS is to be released at the same time; see [38] for a description of an early version.

CBS appears to be well suited for specifying dynamic semantics of funcons (and thereby of programming language constructs via translation to funcons). However, the signatures of funcons specify well-formedness for funcon terms only with regard to the types of argument sequences: properties of funcons that depend on bindings (such as declaration before use) are not specified in signatures, and not implied by well-formedness. Conventional typing rules for funcons have previously been specified [31], ensuring the expected properties of bindings, but it was unclear how to add (and implicitly propagate) checks for other aspects of static semantics (e.g., definite assignment). When a satisfactory approach to modular static semantics has been developed, it should be straightforward to add the appropriate rules to the funcon specifications, without affecting their dynamic semantics.

This article has focused entirely on the semantics of textual programming languages. However, a component-based approach might also be applicable to other kinds of software languages, such as domain specific languages and modelling languages with graphical elements. The current funcons for specifying flow of data and control should be useful for specifying any language that has an operational interpretation.

PLanCompS is an open international collaboration, and welcomes new participants interested in contributing to the development of CBS, its library of funcons, or examples of language specifications using CBS. Developers of other meta-languages could contribute by providing their own definitions of the PLanCompS funcons (or of alternative libraries), thereby allowing their users to exploit the proposed component-based approach to language specification.

Acknowledgements. The author is grateful to the SLE 2017 PC chairs for the invitation to give a presentation at the conference and submit an article to this special issue, and to the anonymous reviewers for helpful comments and constructive suggestions for improvement.

The development of CBS has been supported by EPSRC [grant number EP/I032495/1].

References

[1] P. D. Mosses, Engineering meta-languages for specifying software languages (keynote), in: SLE 2017, ACM, 2017, p. 1. doi:10.1145/3136014.3148041.

[2] W. A. Whitaker, Ada—the project: The DoD High Order Language Working Group, in: HOPL-II, ACM, New York, NY, USA, 1993, pp. 299–331. doi:10.1145/154766.155376.

[3] R. Harper, R. Milner, M. Tofte, The Definition of Standard ML, MIT Press, Cambridge, MA, USA, 1990.

[4] R. Milner, M. Tofte, R. Harper, D. MacQueen, The Definition of Standard ML (Revised), MIT Press, Cambridge, MA, USA, 1997.

[5] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, A history of Haskell: Being lazy with class, in: HOPL III, ACM, New York, NY, USA, 2007, pp. 12.1–12.55. doi:10.1145/1238844.1238856.

[6] WebAssembly, Home page, Accessed November 2018. https://webassembly.org.

[7] S. Marlow, Haskell 2010 language report, 2010. https://www.haskell.org/definition/haskell2010.pdf.

[8] V. Donzeau-Gouge, G. Kahn, B. Lang, On the formal definition of ADA, in: Semantics-Directed Compiler Generation, volume 94 of *Lecture Notes in Computer Science*, Springer, 1980, pp. 475–489. doi:10.1007/3-540-10250-7_34.

[9] E. Astesiano, G. Reggio, SMoLCS-driven concurrent calculi, in: TAPSOFT'87, volume 249 of *Lecture Notes in Computer Science*, Springer, 1987, pp. 169–201. doi:10.1007/3-540-17660-8_55.

[10] F. Biancuzzi, S. Warden, Masterminds of Programming: Conversations with the Creators of Major Programming Languages, O'Reilly, 2009.

[11] D. Berry, R. Milner, D. N. Turner, A semantics for ML concurrency primitives, in: POPL'92, ACM Press, 1992, pp. 119–129. doi:10.1145/143165.143191.

[12] J. H. Reppy, Concurrent ML: Design, application and semantics, Springer, 1993, pp. 165–198. doi:10.1007/3-540-56883-2_10.

[13] J. H. Reppy, Concurrent Programming in ML, Cambridge University Press, New York, NY, USA, 1999.

[14] P. D. Mosses, Foundations of modular SOS, in: MFCS'99, volume 1672 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 70–80. doi:10.1007/3-540-48340-3_7.

[15] P. D. Mosses, A modular SOS for ML concurrency primitives, BRICS Report Series 6 (1999). doi:10.7146/brics.v6i57.20127.

[16] D. Scott, C. Strachey, Toward a Mathematical Semantics for Computer Languages, Technical Report PRG-6, Oxford University Computing Lab., 1971.

[17] C. Strachey, C. P. Wadsworth, Continuations: A Mathematical Semantics for Handling Full Jumps, Technical Report PRG-11, Oxford University Computing Lab., 1974.

[18] E. Moggi, Computational lambda-calculus and monads, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science, IEEE Press, NJ, USA, 1989, pp. 14–23. doi:10.1109/LICS.1989.39155.

[19] G. D. Plotkin, A structural approach to operational semantics, J. Log. Algebr. Program. 60-61 (2004) 17–139.

[20] K. Wansbrough, J. Hamer, A modular monadic action semantics, in: DSL'97, USENIX, 1997. URL: http://www.usenix.org/publications/library/proceedings/dsl97/wansbrough.html.

[21] P. D. Mosses, A modular SOS for action notation, BRICS Report Series 6 (1999). doi:10.7146/brics.v6i56.20126.

[22] P. D. Mosses, Modular structural operational semantics, J. Log. Algebr. Program. 60-61 (2004) 195–228. doi:10.1016/j.jlap.2004.03.008.

[23] P. D. Mosses, M. J. New, Implicit propagation in structural operational semantics, Electr. Notes Theor. Comput. Sci. 229 (2009) 49–66. doi:10.1016/j.entcs.2009.07.073.

[24] G. Roşu, T. F. Şerbănuţă, An overview of the K semantic framework, J. Log. Algebr. Program. 79 (2010) 397–434. doi:10.1016/j.jlap.2010.03.012.

[25] M. Felleisen, R. B. Findler, M. Flatt, Semantics Engineering with PLT Redex, The MIT Press, 2009.

[26] P. D. Mosses, Action Semantics, volume 26 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, 1992.

[27] P. D. Mosses, Theory and practice of action semantics, in: MFCS'96, volume 1113 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 37–61. doi:10.1007/3-540-61550-4_139.

[28] M. van den Brand, J. Iversen, P. D. Mosses, An action environment, Sci. Comput. Program. 61 (2006) 245–264. doi:10.1016/j.scico.2006.04.005.

[29] PLanCompS, Project home page, 2011. http://plancomps.org.

[30] M. Churchill, P. D. Mosses, Modular bisimulation theory for computations and values, in: FOSSACS 2013, volume 7794 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 97–112. doi:10.1007/978-3-642-37075-5_7.

[31] M. Churchill, P. D. Mosses, N. Sculthorpe, P. Torrini, Reusable components of semantic specifications, Trans. Aspect-Oriented Software Development 12 (2015) 132–179. doi:10.1007/978-3-662-46734-3_4.

[32] V. A. Vergu, P. Neron, E. Visser, Dynsem: A DSL for dynamic semantics specification, in: RTA 2015, volume 36 of *LIPIcs*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 365–378. doi:10.4230/LIPIcs.RTA.2015.365.

[33] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, G. Konat, A language designer's workbench: A one-stop-shop for implementation and verification of language designs, in: Onward! 2014, ACM, New York, NY, USA, 2014, pp. 95–111. doi:10.1145/2661136.2661149.

[34] P. D. Mosses, F. Vesely, Funkons: Component-based semantics in K, in: WRLA 2014, volume 8663 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 213–229. doi:10.1007/978-3-319-12904-4_12.

[35] E. Börger, The abstract state machines method for modular design and analysis of programming languages, J. Log. Comput. 27 (2017) 417–439. doi:10.1093/logcom/exu077.

[36] M. Pretnar, An introduction to algebraic effects and handlers. invited tutorial paper, Electr. Notes Theor. Comput. Sci. 319 (2015) 19–35. doi:10.1016/j.entcs.2015.12.003.

[37] P. D. Mosses, Modular language descriptions, in: GPCE 2004, volume 3286 of *Lecture Notes in Computer Science*, Springer, 2004, p. 489. doi:10.1007/978-3-540-30175-2_27.

[38] L. T. van Binsbergen, N. Sculthorpe, P. D. Mosses, Tool support for component-based semantics, in: Companion Proceedings of the 15th International Conference on Modularity, ACM, 2016, pp. 8–11. doi:10.1145/2892664.2893464.