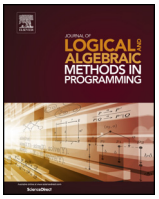
Journal of Logical and Algebraic Methods in Programming 103 (2019) 184–212

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Executable component-based semantics

L. Thomas van Binsbergen [a,*], Peter D. Mosses [b,c], Neil Sculthorpe [d]

[a] Department of Computer Science, Royal Holloway, University of London, TW20 0EX, Egham, United Kingdom
[b] Department of Computer Science, Swansea University, SA2 8PP, Swansea, United Kingdom
[c] EEMCS, Programming Languages, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, the Netherlands
[d] Department of Computing and Technology, Nottingham Trent University, NG11 8NS, Nottingham, United Kingdom

ARTICLE INFO

Article history:
Received 6 July 2018
Received in revised form 18 December 2018
Accepted 21 December 2018
Available online 4 January 2019

Keywords:
Programming languages
Formal semantics
Reuse
Components
Tool support

ABSTRACT

The potential benefits of formal semantics are well known. However, a substantial amount of work is required 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, has discouraged language developers from using formal semantics. Consequently, many major programming languages (and most domain-specific languages) do not yet have formal semantic definitions.

To improve the practicality of formal semantic definitions, the PLanCompS project has developed a component-based approach. In this approach, the semantics of a language is defined by translating its constructs (compositionally) to combinations of so-called fundamental constructs, or 'funcons'. Each funcon is defined using a modular variant of Structural Operational Semantics, and forms a language-independent component that can be reused in definitions of different languages. A substantial library of funcons has been developed and tested in several case studies. Crucially, the definition of each funcon is fixed, and does not need changing when new funcons are added to the library.

For specifying component-based semantics, we have designed and implemented a meta-language called CBS. It includes specification of abstract syntax, of its translation to funcons, and of the funcons themselves. Development of CBS specifications is supported by an integrated development environment. The accuracy of a language definition can be tested by executing the specified translation on programs written in the defined language, and then executing the resulting funcon terms using an interpreter generated from the CBS definitions of the funcons. This paper gives an introduction to CBS, illustrates its use, and presents the various tools involved in our implementation of CBS.

1. Introduction

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. Infor-

* Corresponding author.
E-mail addresses: ltvanbinsbergen@acm.org (L.T. van Binsbergen), p.d.mosses@swansea.ac.uk (P.D. Mosses), neil.sculthorpe@ntu.ac.uk (N. Sculthorpe).

https://doi.org/10.1016/j.jlamp.2018.12.004
2352-2208/

mal 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 [1]), there is generally little inclination on the part of programming language developers themselves to produce a complete formal semantics definition. 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 (see, for instance, comments by the Haskell designers [2, page 9]).

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

1.1. Component-based semantics

To improve the practicality of formal semantic definitions of larger languages, the PLanCompS project[1] has developed a component-based approach to semantics. In this approach 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 [3], a variant of Modular Structural Operational Semantics [4]. For example, the funcon **if-true-else** is defined as follows:

$$\textit{Funcon}\ \textbf{if-true-else}(\_ : \textbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T$$

$$\textit{Rule}\ \textbf{if-true-else}(\textbf{true}, X, \_) \rightsquigarrow X$$

$$\textit{Rule}\ \textbf{if-true-else}(\textbf{false}, \_, Y) \rightsquigarrow Y$$

The collection of funcons is open-ended; crucially, adding new funcons does not require 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. For example, a non-strict conjunction with Boolean-valued operands could be translated as follows:

$$\textit{Rule}\ \textit{rval}\ [\![\textit{Exp}_1\ \text{`}\&\&\text{'}\ \textit{Exp}_2]\!] = \textbf{if-true-else}(\textit{rval}\ [\![\textit{Exp}_1]\!], \textit{rval}\ [\![\textit{Exp}_2]\!], \textbf{false})$$

Many funcons can be widely reused in the definitions of different languages. An initial case study [5] gave a semantics for Caml Light [6] based on a preliminary collection of funcons. The reusability of funcons has subsequently been validated by several further case studies, some of which are available online [7], including an update of the Caml Light specification to OCaml Light. These case studies 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.

Analogous practices are widely adopted in software engineering: developers rely on reusable components in the form of packages. However, applications generally rely on the details of particular versions of packages, and problems can arise when new versions of packages are installed, requiring changes to applications that use them. In contrast, each individual funcon definition is fixed, and language definitions require changes only when the language itself evolves. For example, the above translation of the expression "$\textit{Exp}_1$ '**&&**' $\textit{Exp}_2$" would need changing if conjunction were to become strict, or if its arguments were no longer required to be Booleans, since the definition of the **if-true-else** funcon cannot change.

This article presents CBS, a unified meta-language for component-based semantics of programming languages. CBS includes 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.

1.2. Executability of CBS

One of the most important properties of a language definition is whether the defined language corresponds to the intentions of the language developers. In practice, the only way of checking this property is to take a collection of test programs, and compare the intended behaviour of each program with that determined by the language definition. Provided that the collection of test programs is sufficiently representative, and that their behaviour involves all parts of the language definition, such empirical validation can detect most discrepancies between the intended and the defined behaviour, and establish a high degree of confidence in the accuracy of the language definition.

In general, however, the program behaviour determined by a language definition is not immediately apparent. One approach is to carry out formal proofs of the relationship between programs and their behaviours, based on facts derived from the language definition. A less demanding approach is to derive an *implementation* of the language from its definition,

[1] http://plancomps.org.

and observe the behaviour of programs when run. When the derived implementation can be generated automatically from the language definition, the latter is called *executable*.

We have implemented software tools to make component-based semantic definitions in CBS executable. CBS definitions are developed and maintained within an IDE (integrated development environment) that offers syntax highlighting, clickable references between uses of funcons and their definitions, as well as access to the language-specific tools generated from CBS definitions. From a CBS definition, a user can generate a parser, a translator for translating abstract syntax trees to funcon terms, and an interpreter for executing funcon terms. In combination, these generated artifacts form an interpreter for the defined language.

1.3. Novel features of CBS

This article presents details of the CBS meta-language, and shows how it is used to formally define languages as well as an extensible library of reusable programming constructs. CBS is novel in its emphasis on reuse: language definitions are based on an extensible collection of predefined, fixed funcons, which are themselves defined in CBS. The library of reusable funcons provided with CBS is moderated, and a new funcon is added to the library only after its definition has been validated through analysis and experimentation. Language definitions reuse funcons simply by referring to their names; there are no explicit imports of modules or packages, nor any mention of the files or folders in which definitions are stored. This is possible because a particular funcon name always refers to the same definition within and across language definitions. Implemented as a Spoofax project [8], CBS provides name resolution and (clickable) references between occurrences of funcons and their definitions. The moderated collection of reusable funcons is still under development, and has been pre-released [7] to encourage discussion and review.

If necessary, language-specific funcons can be defined as part of a language definition. Such funcon definitions are modified freely, and are not intended for reuse. However, a generalisation of a language-specific funcon may reveal itself as a useful reusable component.

Funcons are highly modular: each funcon is defined independently, and its definition can be understood in isolation. Funcons inherit their modularity from the I-MSOS inference rules that define them, which involve the implicit propagation of several classes of auxiliary semantic entities. The I-MSOS rules used in CBS are based on a rich variety of such classes. Since funcons are defined with small-step style rules, it is relatively simple to give the semantics of constructs that involve interleaving, or abnormal control flow such as delimited continuations, exceptions and return statements.

1.4. Contributions and overview

This paper contributes by:

- Introducing CBS, a novel specification language with an extensible collection of fixed, highly reusable funcons for modular, component-based language definitions.
- Demonstrating how an existing language is defined using CBS, and how to extend a language definition in a modular fashion when the language evolves.
- Introducing two new classes of auxiliary semantic entities, along with illustrative examples of their use.
- Explaining how funcon definitions can be implemented as micro-interpreters, which are as modular as the definitions from which they are generated.

As a running example, we use CBS to formally define Simple, a small C-like imperative programming language [9]. The Simple definition involves funcons that are likely to be reused in the definitions of most programming languages.

The rest of this article is organised as follows. In Sect. 2 we discuss the background for CBS, and related frameworks. In Sect. 3 we show how to use CBS to define abstract syntax and translation of abstract syntax trees to funcons. In Sect. 4 we show the definitions of some of the funcons that occur in the translations of the Simple constructs, and explain the different classes of semantic entities used in these definitions. In Sect. 5, the Simple language is extended with a delimited control operator to demonstrate the evolution of CBS language definitions as well as the novel bidirectional signals. In Sect. 6 we give an overview of the Haskell framework for implementing funcon definitions. In Sect. 7 we present an IDE for developing and validating CBS language definitions.

Appendix A provides a complete CBS definition of Simple (omitting concurrency constructs) and Appendix B lists the signatures of all the funcons used in the definition. The CBS definitions of these funcons are available online [7].

2. Background and related work

In this section we discuss the background for CBS, and its relation to other current frameworks for formal semantics that have a high degree of modularity as well as executable definitions.

2.1. Theoretical foundations

As mentioned in the Introduction, CBS includes meta-notation for specifying abstract syntax, translation functions (from abstract syntax to funcons), and funcons. The details of CBS are explained and illustrated in subsequent sections; here, we give an overview of the foundations of the framework.

Grammars for specifying abstract syntax in CBS are context-free. They are expressed in a variant of Extended BNF, using standard regular expression notation for sequences, alternatives, and optional and iterated phrases. Abstract syntax trees (ASTs) correspond to finite terms constructed by operations corresponding to the alternatives for each sort of phrase. (CBS also supports disambiguation rules for parsing based on abstract syntax grammars, but that is not significant for its foundations.)

CBS allows multiple translation functions for each sort of AST, defined inductively by mutual recursion. The foundations of inductive definitions of families of functions are well known.

The rules used to specify behaviour in CBS resemble rules in conventional Structural Operational Semantics [10]. However, their interpretation is based on a combination of value-computation systems [11] and Implicitly-Modular SOS (I-MSOS) [3]:

- Value-computation systems provide a context-insensitive rewriting relation as well as a transition relation, and allow transitions to be labelled by multiple entities as in Modular SOS (MSOS) [4]. The format of the rules ensures that (strong) bisimulation is a congruence, and that it is preserved by disjoint extension. The distinction between values and computations is crucial for allowing nested patterns in source terms of transition rules (e.g., the constants in the above rules for '**if-true-else**'). Moreover, the saturation of the transition relation by the rewrite relation gives the effect of weak bisimulation [12].
- I-MSOS supports the use of conventional SOS notation for transitions, whereas its foundations are based on MSOS. Extra conciseness is obtained by allowing strictness annotations in funcon signatures, inspired by the $\mathbb{K}$ Framework [9, page 9], reflecting the distinction between values and computations. Strictness annotations in CBS are interpreted as standard congruence rules for the transition relation, avoiding the need for the reversible 'heating' and 'cooling' rewrites used in $\mathbb{K}$.

2.2. The PLanCompS project

The CBS framework has been developed by PLanCompS. This project was originally funded by EPSRC in the UK (2011–16), and it is currently continuing as an open international collaboration.

Precursors of CBS include an embedding of component-based semantics in Prolog: language syntax and its translation to funcons were specified together using Definite Clause Grammars; MSOS transition rules were written as inference rules, supporting the notation introduced in [4] for transitions with labels, and their desugaring into Prolog rules was also implemented in Prolog. A similar system, MMT [13], was implemented using the Maude framework instead of Prolog. Both systems have been used in connection with teaching operational semantics [14].

PLanCompS developed the Prolog-based meta-notation further to support value-computation rules and rewrites, as well as typing rules for static semantics. It was used to validate the component-based semantics of Caml Light [5]. PLanCompS subsequently switched from Prolog to Haskell as implementation language, as described in Sect. 6 and [15].

2.3. Other semantic frameworks

Executable semantic definitions have been developed since the early 1970s. Current frameworks supporting executable semantics[2] include the CoreASM tools [16], Ott [17], PLT Redex [18], the $\mathbb{K}$ tools [19], Maude [20] and MMT [13], Melange [21], and DynSem [22]. Most of these frameworks have a high degree of modularity. An extensive discussion relating several of these other frameworks to component-based semantics can be found in [5].

Apart from CBS, the only current framework that comes together with an extensive collection of reusable components is MMT, which is based on an embedding of MSOS in Maude. Case studies using MMT include a sublanguage of Concurrent ML, and MiniJava. However, MMT was developed before I-MSOS and value-computation systems, and lack of support for these features in MMT significantly reduces the conciseness and readability of its funcon definitions.

It is of considerable interest to develop executable definitions of collections of funcons in some of the other frameworks listed above: that would allow users of those frameworks to specify languages simply by translation to funcons, as in CBS; and it would allow CBS users to exploit features of those frameworks that have not been implemented in CBS.

We have already investigated the possibility of defining funcons in $\mathbb{K}$, and of using the $\mathbb{K}$ tools to translate languages to funcons [23]. Similarly to CBS, the $\mathbb{K}$ rules for funcons do not require modification when funcons are combined or new funcons added. However, in contrast to CBS, the $\mathbb{K}$ Framework requires configurations of entities to be defined monolithically. Different collections of funcons may require different sets of entities, so each translation of a language to funcons needs to

[2] We restrict attention to formal semantic frameworks with established foundations.

be accompanied by a configuration definition. Any reuse of parts of configurations between different language definitions is left implicit.

The DynSem meta-language [22] has been used[3] to specify a semantics for a dynamic demonstration language called SimpleLanguage.[4] One of the present authors has developed a translation of (most of) SimpleLanguage to funcons in CBS (available online [7]), and experimented with defining those funcons in DynSem. DynSem has adopted implicit propagation of semantic entities from I-MSOS. The DynSem definitions of funcons appear to be just as modular as in CBS. However, DynSem is based on big-step rules, which do not support specification of interleaving or concurrency.

3. Specifying programming languages in CBS

A component-based semantics for a programming language is a compositional translation from the abstract syntax of the language to funcon terms. CBS provides notation for context-free grammars defining abstract syntax, and notation for introducing translation functions and defining them inductively.

As a running example, we will use the Simple programming language, a small C-like imperative language. Simple was introduced by Roşu and Şerbănuţă [9] as a pedagogical language for demonstrating the $\mathbb{K}$ Semantic Framework [19]. This section presents illustrative extracts from our CBS definition of Simple, which is listed in Appendix A. For more detailed discussion and examples of the translation of programming languages to funcons (though not using CBS), see [5] or [23].

3.1. Abstract syntax definitions

Our CBS definition of the abstract syntax of Simple is presented in Fig. 1. We typeset non-terminal symbols in sans-serif and variables in *Capitalised Italic*. Terminal symbols consist of concrete syntax, and are typeset in '**bold**' and enclosed in single quotation marks. A '?' suffix applied to a symbol (or parenthesised sequence of symbols) denotes that the symbol(s) are optional. Each non-terminal may be declared with an associated variable, which can then be used in translation equations to match phrases of that non-terminal sort.

3.2. Translation functions

We specify the semantics of a programming language by defining a set of compositional *translation functions*. A translation function comprises a signature and a set of translation equations. The signature declares the sort of source language phrases that the function translates, and the sort of funcon term that it produces. For example, the signature of a translation function named *val* that translates Simple value phrases into funcon terms computing **values** is declared as follows:

$$\mathit{Semantics}\ \mathit{val}\ [\![\_ : \mathsf{value}]\!] : \Rightarrow \mathbf{values} \tag{1}$$

The convention in the funcon framework is to use the plural form for funcon type names. We typeset funcons and their types in **bold**, and translation functions in *italic*.

In Simple, the semantics of an expression depends on whether it occurs to the left or right of an assignment operator (so-called *l-expressions* and *r-expressions*). An l-expression should evaluate to an imperative variable, whereas an r-expression, if it results in a variable, should evaluate to the value currently assigned to that variable. In the grammar we have grouped the subset of expressions that are valid l-expressions into a separate non-terminal named lexp, and we define two translation functions for specifying the semantics of expressions:

$$\mathit{Semantics}\ \mathit{lval}\ [\![\_ : \mathsf{lexp}]\!] : \Rightarrow \mathbf{variables} \tag{2}$$

$$\mathit{Semantics}\ \mathit{rval}\ [\![\_ : \mathsf{exp}]\!] : \Rightarrow \mathbf{values} \tag{3}$$

A signature is accompanied by a set of translation equations. Each equation should be *compositional*: translation functions in the right-hand side of an equation should be applied to subphrases of the argument phrase in the left-hand side of the equation, for example:

$$\mathit{Rule}\ \mathit{rval}\ [\![\text{'}\mathbf{!}\text{'}\ \mathit{Exp}]\!] = \mathbf{not}(\mathit{rval}\ [\![\mathit{Exp}]\!]) \tag{4}$$

$$\mathit{Rule}\ \mathit{rval}\ [\![\mathit{Exp}_1\ \text{'}\mathbf{+}\text{'}\ \mathit{Exp}_2]\!] = \mathbf{integer\text{-}add}(\mathit{rval}\ [\![\mathit{Exp}_1]\!], \mathit{rval}\ [\![\mathit{Exp}_2]\!]) \tag{5}$$

$$\mathit{Rule}\ \mathit{rval}\ [\![\text{'}\mathbf{read}\text{'}\ \text{'}\mathbf{(}\text{'}\ \text{'}\mathbf{)}\text{'}]\!] = \mathbf{read} \tag{6}$$

The translation of an lexp phrase that occurs as an r-expression is:

$$\mathit{Rule}\ \mathit{rval}\ [\![\mathit{LExp}]\!] = \mathbf{assigned}(\mathit{lval}\ [\![\mathit{LExp}]\!]) \tag{7}$$

[3] https://github.com/MetaBorgCube/metaborg-sl.
[4] https://github.com/graalvm/simplelanguage.

$$
\begin{array}{rlll}
\mathit{Pgm} & : \text{pgm} & ::= & \text{decls} \\
\mathit{Decls} & : \text{decls} & ::= & \text{decl decls}^{?} \\
\mathit{Decl} & : \text{decl} & ::= & \text{func-decl} \mid \text{vars-decl} \\
\mathit{FuncDecl} & : \text{func-decl} & ::= & \text{`}\mathbf{function}\text{'} \text{ id `(' ids}^{?} \text{ `)' block} \\
\mathit{Ids} & : \text{ids} & ::= & \text{id (`,' ids)}^{?} \\
\mathit{VarsDecl} & : \text{vars-decl} & ::= & \text{`}\mathbf{var}\text{'} \text{ declarators `;'} \\
\mathit{Declarators} & : \text{declarators} & ::= & \text{declarator (`,' declarators)}^{?} \\
\mathit{Declarator} & : \text{declarator} & ::= & \text{id} \mid \text{id `=' exp} \mid \text{id ranks} \\
\mathit{Ranks} & : \text{ranks} & ::= & \text{`[' exps `]' ranks}^{?} \\
\mathit{Block} & : \text{block} & ::= & \text{`\{' stmts}^{?} \text{ `\}'} \\
\mathit{Stmts} & : \text{stmts} & ::= & \text{stmt stmts}^{?} \\
\mathit{Stmt} & : \text{stmt} & ::= & \text{imp-stmt} \mid \text{vars-decl} \\
\mathit{ImpStmt} & : \text{imp-stmt} & ::= & \text{block} \\
 & & \mid & \text{exp `;'} \\
 & & \mid & \text{`}\mathbf{if}\text{' `(' exp `)' block (`}\mathbf{else}\text{' block)}^{?} \\
 & & \mid & \text{`}\mathbf{while}\text{' `(' exp `)' block} \\
 & & \mid & \text{`}\mathbf{for}\text{' `(' stmt exp `;' exp `)' block} \\
 & & \mid & \text{`}\mathbf{print}\text{' `(' exps `)' `;'} \\
 & & \mid & \text{`}\mathbf{return}\text{' exp}^{?} \text{ `;'} \\
 & & \mid & \text{`}\mathbf{try}\text{' block `}\mathbf{catch}\text{' `(' id `)' block} \\
 & & \mid & \text{`}\mathbf{throw}\text{' exp `;'} \\
\mathit{Exp} & : \text{exp} & ::= & \text{`(' exp `)'} \mid \text{value} \mid \text{lexp} \\
 & & \mid & \text{lexp `=' exp} \mid \text{`++' lexp} \\
 & & \mid & \text{`}-\text{' exp} \mid \text{exp `(' exps}^{?} \text{ `)'} \\
 & & \mid & \text{`}\mathbf{sizeOf}\text{' `(' exp `)'} \mid \text{`}\mathbf{read}\text{' `(' `)'} \\
 & & \mid & \text{exp `+' exp} \mid \text{exp `}-\text{' exp} \\
 & & \mid & \text{exp `}*\text{' exp} \mid \text{exp `/' exp} \mid \text{exp `\%' exp} \\
 & & \mid & \text{exp `<' exp} \mid \text{exp `<=' exp} \\
 & & \mid & \text{exp `>' exp} \mid \text{exp `>=' exp} \\
 & & \mid & \text{exp `==' exp} \mid \text{exp `!=' exp} \\
 & & \mid & \text{`!' exp} \mid \text{exp `\&\&' exp} \mid \text{exp `||' exp} \\
\mathit{LExp} & : \text{lexp} & ::= & \text{id} \mid \text{lexp `[' exps `]'} \\
\mathit{Exps} & : \text{exps} & ::= & \text{exp (`,' exps)}^{?} \\
\mathit{V} & : \text{value} & ::= & \text{bool} \mid \text{int} \mid \text{string}
\end{array}
$$

Fig. 1. CBS specification of Simple abstract syntax (omitting concurrent constructs).

where:

$$\mathit{Rule}\ \mathit{lval}\ [\![\mathit{Id}]\!] = \mathbf{bound}(\mathit{id}\ [\![\mathit{Id}]\!]) \tag{8}$$

$$\mathit{Rule}\ \mathit{lval}\ [\![\mathit{LExp}\ \text{`}\mathbf{[}\text{'}\ \mathit{Exp}\ \text{`}\mathbf{]}\text{'}]\!] = \mathbf{checked}(\mathbf{index}(\mathbf{integer\text{-}add}(1, \mathit{rval}\ [\![\mathit{Exp}]\!]), \mathbf{vector\text{-}elements}(\mathit{rval}\ [\![\mathit{LExp}]\!]))) \tag{9}$$

Notice that dereferencing imperative variables (Rule (7)) and identifier bindings (Rule (8)) are implicit in Simple syntax, but are made explicit in the produced funcon term.

Programming languages often contain constructs that are intended to be understood as abbreviations for other constructs in the language: so-called 'syntactic sugar'. CBS allows *desugaring equations* to be defined. For example, a Simple **for**-loop can be desugared into a Simple **while**-loop as follows:

$$\mathit{Rule}\ [\![\text{`}\mathbf{for}\text{' `(' }\mathit{Stmt}\ \mathit{Exp}_1\ \text{`;'}\ \mathit{Exp}_2\ \text{`)' `\{' }\mathit{Stmts}\ \text{`\}'}]\!] : \text{stmt} = [\![\text{`\{' }\mathit{Stmt}\ \text{`}\mathbf{while}\text{' `(' }\mathit{Exp}_1\ \text{`)' `\{' `\{' }\mathit{Stmts}\ \text{`\}' }\mathit{Exp}_2\ \text{`;' `\}' `\}'}]\!] \tag{10}$$

This desugaring equation mirrors the $\mathbb{K}$ specification of Simple [9, page 26], which uses analogous "desugaring macros" to achieve the same effect. However, we deviate slightly from the K desugaring, which (presumably inadvertently) scopes any declarations in the loop body (*Stmts*) over the for-iterator (Exp_2).

Most of the equations presented so far have translated each Simple construct directly into a corresponding funcon. Translating many other Simple constructs is just as straightforward (see Appendix A). However, often the semantics of a language construct needs to be expressed by a combination of funcons. For example, here are the equations for two forms of Simple declarator:

$$\mathit{Semantics}\ \mathit{var\text{-}declare}\ [\![\_ : \text{declarator}]\!] : \Rightarrow \mathbf{environments} \tag{11}$$

Rule *var-declare* $[\![Id]\!] = \textbf{bind}(id\ [\![Id]\!], \textbf{allocate-variable}(\textbf{values}))$ (12)

Rule *var-declare* $[\![Id\ \text{`=`}\ Exp]\!] = \textbf{bind}(id\ [\![Id]\!], \textbf{allocate-initialised-variable}(\textbf{values}, rval\ [\![Exp]\!]))$ (13)

One of our aims is that the name of a funcon should be sufficiently indicative of its semantics that a reader should be able to understand the gist of a translation without referring to the funcon definitions.

3.3. Language-specific funcons

Although the funcon library contains over a hundred computational funcons (as well as over a hundred value operations), sometimes the semantics of a language construct cannot be directly expressed by a straightforward combination of existing funcons. In Simple, declarators for nested arrays are one such construct. The difficulty is that the rank of an outer array is used to determine the size of each inner array, but the expression computing that array size should only be evaluated once, with the result shared between the inner arrays. Roşu and Şerbănuţă also encounter this issue in their $\mathbb{K}$ specification [9, page 28]. They note that this case cannot be handled by desugaring, as that would duplicate the expression (and any side effects it may contain). Their solution, which they remark is not ideal, is to have their semantic rule for array declarators generate Simple code that uses a **for**-loop to iteratively allocate each sub-array. This code generation happens at run-time, after the size expression has been computed.

We cannot take this approach in the funcon framework: funcon definitions cannot refer to source-language syntax or invoke translation functions. However, CBS provides a convenient alternative: local funcons can be specified within a language definition. This is appropriate for funcons that are not considered reusable enough to be included in the funcon library, but nonetheless capture a useful concept of the language being defined. (Local funcon definitions are also useful for validating new funcons before adding them to the library.) Here, we introduce a Simple-specific funcon **allocate-nested-vectors** that precisely captures the semantics of nested array declarators, allowing us to define the translation as follows:

Rule *var-declare* $[\![Id\ Ranks]\!] = \textbf{bind}(id\ [\![Id]\!], \textbf{allocate-nested-vectors}[ranks\ [\![Ranks]\!]])$ (14)

The definition of **allocate-nested-vectors** is included in Appendix A.4.2.

3.4. Remarks

Our definition of Simple omits the concurrent aspects of the language. This is not due to any limitation of CBS or (I-)MSOS-based specifications – indeed, one author of this article has previously used MSOS to specify a sublanguage of Concurrent ML [24] – rather it is that development of the funcon library is ongoing work, and funcons specifying concurrency have not yet been added.

4. Specifying fundamental constructs in CBS

Structural Operational Semantics (SOS) [10] is a well-established framework for specifying computational behaviour, where the behaviour of programs is modelled by labelled transition systems, defined inductively by axioms and inference rules. To specify the dynamic semantics of funcons, we use small-step rules in a *modular* variant of SOS called MSOS [4]. The CBS notation for MSOS rules is based on *Implicitly-Modular SOS* (I-MSOS) [3], which provides a notational style similar to conventional SOS. CBS (or I-MSOS) rules can be understood as syntactic sugar for MSOS rules. We assume the reader is familiar with conventional SOS rules (e.g. [10]). The rules define two transition relations, for computational steps and context-free rewrites respectively. A formal account of the underlying MSOS framework can be found in [4], of the I-MSOS rules in [3], and of the two transition relations in [11].

This section presents an informal and example-driven overview of using CBS to specify funcons. The funcons in this section are selected to demonstrate different aspects of CBS specifications. Appendix B.1 lists the signatures of all funcons used in the Simple language specification; the definitions of those funcons are available online [7].

4.1. Transition relations

Consider the following CBS specification of **if-true-else**:

$$\textit{Rule}\ \frac{B \longrightarrow B'}{\textbf{if-true-else}(B, X, Y) \longrightarrow \textbf{if-true-else}(B', X, Y)} \tag{15}$$

Rule $\textbf{if-true-else}(\textbf{true}, X, \_) \rightsquigarrow X$ (16)

Rule $\textbf{if-true-else}(\textbf{false}, \_, Y) \rightsquigarrow Y$ (17)

This is a fairly conventional SOS specification of a conditional expression, except that there are two transition relations: '$\longrightarrow$' and '$\rightsquigarrow$'. The former denotes exactly one step of computation, which in general may have arbitrary side effects and be sensitive to its context. The latter denotes context-free term rewriting, which cannot involve side effects, and is reflexive

and transitive. The distinction facilitates reasoning about funcons terms. It can also be exploited by a funcon interpreter to provide a more efficient implementation of the pure transitions. The formal details of these two relations, and the interaction between them, are spelled out in [11].[5]

4.2. Funcon signatures

CBS requires each funcon to have a declared signature. For example, the signature for **if-true-else** is declared as follows:

$$\textit{Funcon}\ \textbf{if-true-else}(\_ : \textbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T \tag{18}$$

The result of the funcon, and each funcon parameter, has a *sort*. A funcon sort is either a *value sort* (a type), or a *computation sort* (a type prefixed by the '$\Rightarrow$' symbol, which is pronounced "computes"). When the result sort is a value sort, then the funcon is a value constructor, and the funcon will not have any operational rules.

Additionally, polymorphic funcons may have signatures that contain type variables. In the case of **if-true-else**, the type variable T is used to express that the second and third arguments compute values of the same type, and that the funcon as a whole also computes a value of that type.

Notice that the first parameter has a value sort. This does not mean that **if-true-else** is limited to being applied to literal Boolean values; rather, value-sort annotations play a special role in funcon signatures, as we will now describe.

When a parameter has a value sort, we say that the funcon is *strict* in that argument. If a funcon is strict in all its arguments, then we say that the funcon is strict. For each *strict* parameter in a funcon's signature, a *congruence rule* is implicitly generated for that argument.

A congruence rule has a single premise consisting of a computation step for the strict argument. The target of the rule's conclusion is the same as the source of the conclusion, except that the argument subterm is replaced with the updated subterm from the target of the premise. For example, Rule (15) is the congruence rule implicitly generated by the signature of **if-true-else** (18), and thus that rule can be omitted. A funcon with congruence rules generated for several arguments is non-deterministic with respect to the order in which these arguments are evaluated: the congruence rules admit all possible interleavings. This technique for automatically generating congruence rules was inspired by the strictness annotations in the $\mathbb{K}$ Framework [9, page 9].

4.3. Values and types

CBS provides a rich suite of primitive and composite values, and operations on those values. The available operations include arithmetic, conversions between types, and insertion/deletion from collections, among others. The signatures of all value operations used in this article are listed in Appendix B.2. Some familiar types of values are left unspecified, for example **integers**, **floats** and **maps**. In these cases the semantics of value operations is implemented manually in a library provided as part of the Haskell funcon framework (see Sect. 6.5).

The names of types are themselves funcon values, and belong to the type **types**. All values, including type names, are members of the type **values** (i.e. **values** is a supertype of **integers**, **floats**, etc.). The type **values** is frequently used when specifying funcons that can take arbitrary values as arguments.

New data types can be introduced. For example, the data types **booleans**, and **null-type** are defined as follows:

$$\textit{Datatype}\ \textbf{booleans} ::= \textbf{true} \mid \textbf{false} \tag{19}$$

$$\textit{Datatype}\ \textbf{null-type} ::= \textbf{null} \tag{20}$$

The type **null-type** is the unit type of the funcon library, and is used as the result type of commands.

When specifying a funcon, it may be necessary to restrict a rule to be only applicable when some of its arguments are values (rather than computations). This can be achieved by annotating the appropriate arguments with '$: T$' (or a more specific type).

Alternatively, arguments in rules can be restricted to values by using *patterns*. A pattern consists of a value constructor, with sub-patterns to match the arguments of the constructor (if any). Those sub-patterns may include meta-variables. For example, rules 16 and 17 have the **true** and **false** constructors as the first argument, respectively, restricting each rule to that specific argument.

Following [11], the only funcons permitted in CBS patterns are value constructors. This ensures that the internal structure of a computation cannot be observed: the only way to observe anything about a computation is to execute it. Conversely, CBS values can be observed, but must be stable: values do not perform computation steps.

[5] In [11] the relations '$\rightsquigarrow$' and '$\longrightarrow$' are written '$\Rightarrow$' and '$\rightarrow$', respectively.

4.4. Semantic entities

Formal specifications of programming-language constructs typically make use of auxiliary *semantic entities* to model computational side effects. Common examples of such entities include environments, stores or output sequences.[6] In an SOS specification, such semantic entities are explicit arguments of transition relations, and thus they must appear in every transition formula of that relation. For example, in a conventional SOS specification, Rule (15) could well have been written as follows:

$$\frac{\rho \vdash \langle B\,,\sigma\rangle \xrightarrow{\alpha} \langle B'\,,\sigma'\rangle}{\rho \vdash \langle \textbf{if-true-else}(B,X,Y)\,,\sigma\rangle \xrightarrow{\alpha} \langle \textbf{if-true-else}(B',X,Y)\,,\sigma'\rangle} \tag{21}$$

ρ, σ, σ' and α are all variables. ρ is the environment in which the term is executing, σ and σ' are the state of the store before and after the transition, and α is the output sequence being emitted. Observe that both the environment and output are the same in both the premise and conclusion. This would typically be understood as propagating the environment unmodified from the conclusion to the premise, and propagating the output unmodified from the premise to the conclusion. The use of σ and σ' would typically be understood as propagating the initial store unmodified from the source of the conclusion to the source of the premise, and propagating the resulting store from the target of the premise to the target of the conclusion. While this rule does not access or modify the store directly, the computation step in the premise may do so.

This systematic propagation of semantic entities is common to many rules in SOS specifications. The key feature of I-MSOS is that any semantic entities that are not mentioned in a rule are *implicitly propagated* between the premise(s) and conclusion, systematically following the propagation schemes described above. This allows semantic entities that are not affected by the programming construct being specified to be omitted from the rule. Thus, rather than Rule (21), in CBS we are able to write the much more concise Rule (15).

Moreover, beyond just notational convenience, implicit propagation brings a key modularity benefit: new semantic entities can later be added without requiring any change to the previously given rules. This high degree of modularity is what allows the collection of funcons to be open-ended, while still ensuring that once a funcon is added to the collection, its CBS definition need never be updated.

To allow semantic entities to be implicitly propagated, they must be classified according to their desired propagation scheme. CBS supports five classes of semantic entity, each with its own notation and system of implicit propagation: *contextual*, *mutable*, *input*, *output* and *control-flow*. The input and control-flow entities are novel: in prior publications on MSOS or funcons (e.g. [4,5,24,27]), only contextual, mutable and output entities were provided, a more limited form of control-flow was achieved by using output entities, and program input was represented by mutable entities.[7]

CBS allows multiple entities of each class, so each entity is tagged with a distinguishing name. The notation for a CBS computation step using one entity of each of the five classes is as follows (writing the classification names as placeholders for the actual entity names):

$$\textit{Rule}\ \mathsf{contextual}(\_) \vdash \langle \_\,, \mathsf{mutable}(\_)\rangle \xrightarrow{\mathsf{input?}(\_)\ \ \mathsf{control\text{-}flow}(\_)\ \ \mathsf{output!}(\_)} \langle \_\,, \mathsf{mutable}(\_)\rangle \tag{22}$$

Note that the '$\rightsquigarrow$' relation never involves semantic entities.

We will now present examples of each class of semantic entity, and of funcons whose semantics require them.

4.4.1. Contextual entities

Contextual entities model contextual information that is available to a computation step, but which cannot be modified by that computation step. In a CBS rule, unmentioned contextual entities are implicitly propagated from the transition in the conclusion to the transition in any premises.

The classic example of a contextual entity is an *environment* containing bindings for program identifiers. We declare such an entity in CBS as follows:

$$\textit{Entity}\ \mathsf{environment}(\_ : \textbf{environments}) \vdash \_ \longrightarrow \_ \tag{23}$$

Here, environment is the name of the entity being declared, **environments** is the type of value contained in the entity. The type **environments** is a mapping from **identifiers** to optional values:

$$\textit{Funcon}\ \textbf{environments} : \Rightarrow \textbf{types} \tag{24}$$

$$\textit{Rule}\ \textbf{environments} \rightsquigarrow \textbf{maps}(\textbf{identifiers}, \textbf{values}^?) \tag{25}$$

[6] The use of environments and stores in SOS dates back to 1981, when they were adopted by Plotkin [10]. But these were simply first-order versions of the corresponding entities used in denotational semantics since the early 1970s (foreshadowed in Strachey's work on CPL [25], and his lecture notes from 1967 [26]).

[7] In prior publications, *contextual*, *mutable* and *output* entities have appeared under a variety of names, most commonly *read-only*, *read-write* and *write-only*, respectively.

A '?' suffix on a funcon type (or type variable) denotes an option type; that is, zero or one values of that type. An absent value in an environment represents that a binding is present, but inaccessible.[8]

Let us consider some funcons that use this entity. The funcon **bound**(I) evaluates to the value bound to the identifier I in the current environment. A common use of this funcon is when specifying the translation of occurrences of identifiers in expressions (e.g. in Eq. (8)). The definition of **bound** for non-recursive bindings is as follows:[9]

$$\textit{Funcon}\ \textbf{bound}(\_ : \textbf{identifiers}) : \Rightarrow \textbf{values} \tag{26}$$

$$\textit{Rule}\ \frac{\textbf{lookup}(\rho, I) \rightsquigarrow V : \textbf{values}}{\text{environment}(\rho) \vdash \textbf{bound}(I : \textbf{identifiers}) \longrightarrow V} \tag{27}$$

$$\textit{Rule}\ \frac{\textbf{lookup}(\rho, I) \rightsquigarrow (\,)}{\text{environment}(\rho) \vdash \textbf{bound}(I : \textbf{identifiers}) \longrightarrow \textbf{fail}} \tag{28}$$

The conclusions of Rules (27) and (28) are computation steps, not rewrites, because they refer to a semantic entity. The notation '( )' denotes the absence of a CBS value.

The funcon **scope**(ρ, X) adds the bindings in ρ to the current environment for the purposes of executing the subterm X:

$$\textit{Funcon}\ \textbf{scope}(\_ : \textbf{environments}, \_ : \Rightarrow T) : \Rightarrow T \tag{29}$$

$$\textit{Rule}\ \frac{\textbf{map-override}(\rho_1, \rho_0) \rightsquigarrow \rho_2 \qquad \text{environment}(\rho_2) \vdash X \longrightarrow X'}{\text{environment}(\rho_0) \vdash \textbf{scope}(\rho_1 : \textbf{environments}, X) \longrightarrow \textbf{scope}(\rho_1, X')} \tag{30}$$

$$\textit{Rule}\ \textbf{scope}(\_ : \textbf{environments}, V : T) \rightsquigarrow V \tag{31}$$

Recall that a funcon signature containing any strict arguments implicitly generates congruence rules for evaluating those strict arguments. Thus the first argument to **scope** could be a funcon term that computes an environment. This is indeed the case in SIMPLE, where declarations involve allocating and initialising imperative variables (see Sect. 10).

Another example of a contextual entity is given-value:

$$\textit{Entity}\ \text{given-value}(\_ : \textbf{values}^?) \vdash \_ \longrightarrow \_ \tag{32}$$

This entity is used for a very simple form of binding and scoping. It holds only a single value. The use of given-value is sufficiently common that we give it special treatment in funcon signatures. A sort $S{\Rightarrow}T$ represents a term that, in a context where the given value has type S, computes a value of type T. To access the value, we use the funcon **given**, which is analogous to **bound**:

$$\textit{Funcon}\ \textbf{given} : T{\Rightarrow}T \tag{33}$$

$$\textit{Rule}\ \text{given-value}(V : \textbf{values}) \vdash \textbf{given} \longrightarrow V \tag{34}$$

$$\textit{Rule}\ \text{given-value}(\,) \vdash \textbf{given} \longrightarrow \textbf{fail} \tag{35}$$

To provide a value to a subterm via the given-value entity, we use the funcon **give**, which is analogous to **scope**:

$$\textit{Funcon}\ \textbf{give}(\_ : S, \_ : S{\Rightarrow}T) : \Rightarrow T \tag{36}$$

$$\textit{Rule}\ \frac{\text{given-value}(V) \vdash X \longrightarrow X'}{\text{given-value}(\_) \vdash \textbf{give}(V : S, X) \longrightarrow \textbf{give}(V, X')} \tag{37}$$

$$\textit{Rule}\ \textbf{give}(\_ : S, V : T) \rightsquigarrow V \tag{38}$$

A common use of **give** and **given** is to model passing arguments to a function, as we shall discuss in Sect. 4.5.

4.4.2. Mutable entities

Mutable entities model persistent information that can be modified by a computation step: the resulting value of a mutable entity after a step is the initial value of the mutable entity for the next step. In a CBS rule, unmentioned mutable entities are implicitly propagated unmodified.

The classic example of a mutable entity is a *store* containing the values of imperative variables. We declare such an entity in CBS as follows:

$$\textit{Entity}\ \langle \_ , \text{store}(\_ : \textbf{stores}) \rangle \longrightarrow \langle \_ , \text{store}(\_ : \textbf{stores}) \rangle \tag{39}$$

[8] For example, in C♯ a local variable is in scope in its entire enclosing block (shadowing any outer variable of the same name), but it is not accessible until after its point of declaration.

[9] See [7] for the generalisation to recursive bindings.

The type **stores** is a mapping from locations (**locations**) to optional values, where a mapping to an absent value represents a location without an assignment.

$$\textit{Funcon}\ \mathbf{stores} : \Rightarrow \mathbf{types} \tag{40}$$

$$\textit{Rule}\ \mathbf{stores} \rightsquigarrow \mathbf{maps}(\mathbf{locations}, \mathbf{values}^{?}) \tag{41}$$

A variable wraps a location and the type of values that can be assigned to that location.

$$\textit{Datatype}\ \mathbf{variables}\ ::=\ \mathbf{variable}(\_ : \mathbf{locations}, \_ : \mathbf{types}) \tag{42}$$

To assign a value to a variable in the store, we use the funcon **assign**:

$$\textit{Funcon}\ \mathbf{assign}(\_ : \mathbf{variables}, \_ : \mathbf{values}) : \Rightarrow \mathbf{null\text{-}type} \tag{43}$$

$$\textit{Rule}\ \frac{\mathbf{and}(\mathbf{is\text{-}in\text{-}set}(L, \mathbf{dom}(\sigma)), \mathbf{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow \mathbf{true} \qquad \mathbf{map\text{-}override}(\{L \mapsto V\}, \sigma) \rightsquigarrow \sigma'}{\langle \mathbf{assign}(\mathbf{variable}(L : \mathbf{locations}, T : \mathbf{types}), V : \mathbf{values}) \,, \mathsf{store}(\sigma)\rangle \longrightarrow \langle \mathbf{null} \,, \mathsf{store}(\sigma')\rangle} \tag{44}$$

$$\textit{Rule}\ \frac{\mathbf{and}(\mathbf{is\text{-}in\text{-}set}(L, \mathbf{dom}(\sigma)), \mathbf{is\text{-}in\text{-}type}(V, T)) \rightsquigarrow \mathbf{false}}{\langle \mathbf{assign}(\mathbf{variable}(L : \mathbf{locations}, T : \mathbf{types}), V : \mathbf{values}) \,, \mathsf{store}(\sigma)\rangle \longrightarrow \langle \mathbf{fail} \,, \mathsf{store}(\sigma)\rangle} \tag{45}$$

The notation $\{\_ \mapsto \_\}$ constructs map values. The value assigned to a variable is retrieved using the funcon **assigned** (e.g. in Eq. (7)):

$$\textit{Funcon}\ \mathbf{assigned}(\_ : \mathbf{variables}) : \Rightarrow \mathbf{values} \tag{46}$$

$$\textit{Rule}\ \frac{\mathbf{lookup}(\sigma \,, L) \rightsquigarrow V : \mathbf{values}}{\langle \mathbf{assigned}(\mathbf{variable}(L : \mathbf{locations}, T : \mathbf{types}) \,, \mathsf{store}(\sigma)\rangle \longrightarrow \langle V \,, \mathsf{store}(\sigma)\rangle} \tag{47}$$

$$\textit{Rule}\ \frac{\mathbf{lookup}(\sigma \,, L) \rightsquigarrow (\,)}{\langle \mathbf{assigned}(\mathbf{variable}(L : \mathbf{locations}, T : \mathbf{types}) \,, \mathsf{store}(\sigma)\rangle \longrightarrow \langle \mathbf{fail} \,, \mathsf{store}(\sigma)\rangle} \tag{48}$$

4.4.3. Output entities

Output entities model emitted output. An output entity is a sequence of values, and the sequences from adjacent computation steps are implicitly concatenated, with non-terminating computations potentially resulting in infinite output. If an output entity is not mentioned in a rule, it is implicitly propagated from the premise (if any) to the conclusion.

An example of an output entity is the standard-output channel. We declare output entities with an '!' suffix:

$$\textit{Entity}\ \_ \xrightarrow{\text{standard-out!}(\_:\mathbf{values}^*)} \_ \tag{49}$$

The standard-out entity holds a sequence of values. A funcon for printing a value to the standard-out entity can be defined as follows:

$$\textit{Funcon}\ \mathbf{print}(\_ : \mathbf{values}^*) : \Rightarrow \mathbf{null\text{-}type} \tag{50}$$

$$\textit{Rule}\ \mathbf{print}(V^{\star} : \mathbf{values}^*) \xrightarrow{\text{standard-out!}(V^{\star})} \mathbf{null} \tag{51}$$

The star in '$V^{\star}$' and the annotation ': **values***' denote that '$V^{\star}$' is a placeholder for zero or more **values**.

4.4.4. Input entities

Input entities model consumed input. Like output entities they form a sequence of values, but this represents the input *consumed* by a computation step, not the output emitted.

An example of an input entity is the standard-input channel. We declare input entities with a '?' suffix:

$$\textit{Entity}\ \_ \xrightarrow{\text{standard-in?}(\_:\mathbf{values}^*)} \_ \tag{52}$$

The standard-in entity holds a sequence of values, with **null** representing the end of input. A funcon for reading a single value from the standard-in entity can be defined as follows:

$$\textit{Funcon}\ \mathbf{read} : \Rightarrow \mathbf{values} \tag{53}$$

$$\textit{Rule}\ \mathbf{read} \xrightarrow{\text{standard-in?}(V:\sim\mathbf{null\text{-}type})} V \tag{54}$$

$$\textit{Rule}\ \mathbf{read} \xrightarrow{\text{standard-in?}(\mathbf{null})} \mathbf{fail} \tag{55}$$

The notation '$\sim$' is type complement; that is, Rule (54) is applicable when the input value is not **null**.

The Haskell funcon framework discussed in Sect. 6 supports simulated and interactive input, where the input values are provided before execution begins, or given by the user during execution, respectively.

4.4.5. Control-flow entities

Control-flow entities model changes in the control-flow of the program. In CBS rules, control-flow entities contain signals that communicate between two rules in a derivation tree. Propagation of control-flow entities is not *directed*; rather, any signals in unmentioned control-flow entities are implicitly synchronised between the premise and conclusion of a rule. Unlike input and output entities, a control-flow entity does not accumulate a sequence of values. Instead, it contains either a present signal, which carries a value, or an absent signal, which is empty.

The classic example of a control-flow change is abrupt termination of a computation. The abrupted control-flow entity indicates abrupt termination when it appears with value V in a label on a transition. The type of value is used to indicate different forms of abrupt termination, which include thrown exceptions, returning from a function, or breaking from a loop. The absence of V indicates the absence of abrupt termination.

$$\mathit{Entity}\ \_ \xrightarrow{\text{abrupted}(\_:\textbf{values}^?)} \_ \tag{56}$$

Abrupt termination is initiated using **abrupt**, which emits a signal in the abrupted entity:

$$\mathit{Funcon}\ \textbf{abrupt}(\_ : \textbf{values}) : \Rightarrow\textbf{empty-type} \tag{57}$$

$$\mathit{Rule}\ \textbf{abrupt}(V : \textbf{values}) \xrightarrow{\text{abrupted}(V)} \textbf{stuck} \tag{58}$$

Funcons for throwing exceptions, for example, are then defined as follows:

$$\mathit{Datatype}\ \textbf{throwing} ::= \textbf{thrown}(\_ : \textbf{values}) \tag{59}$$

$$\mathit{Funcon}\ \textbf{throw}(\_ : \textbf{values}) : \Rightarrow\textbf{empty-type} \tag{60}$$

$$\mathit{Rule}\ \textbf{throw}(V : \textbf{values}) \rightsquigarrow \textbf{abrupt}(\textbf{thrown}(V)) \tag{61}$$

To handle thrown exceptions, we use a funcon called **handle-thrown**:

$$\mathit{Funcon}\ \textbf{handle-thrown}(\_ : S\Rightarrow T, \_ : X\Rightarrow T) : S\Rightarrow T \tag{62}$$

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{abrupted}(\textbf{thrown}(V))} \_}{\textbf{handle-thrown}(X, H) \xrightarrow{\text{abrupted}()} \textbf{give}(V, H)} \tag{63}$$

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{abrupted}(V:\sim\textbf{throwing})} X'}{\textbf{handle-thrown}(X, H) \xrightarrow{\text{abrupted}(V)} \textbf{handle-thrown}(X', H)} \tag{64}$$

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{abrupted}()} X'}{\textbf{handle-thrown}(X, H) \xrightarrow{\text{abrupted}()} \textbf{handle-thrown}(X', H)} \tag{65}$$

$$\mathit{Rule}\ \textbf{handle-thrown}(V : T, \_) \rightsquigarrow V \tag{66}$$

The first argument of **handle-thrown** is a term to be evaluated, and the second argument is a term that serves as an exception handler. If the step on the first argument emits a thrown signal with value V, Rule (63) invokes the handler H by giving it V. Other forms of abrupt termination are not handled, and are propagated by Rule (64). Rule (65) is similar to a congruence rule for the first argument, except that the step on the argument must not emit a thrown signal. If the first argument of **handle-thrown** is a value, Rule (66) ignores the handler and returns that value.

As another example, a funcon representing returning from within a function is defined as follows:

$$\mathit{Datatype}\ \textbf{returning} ::= \textbf{returned}(\_ : \textbf{values}) \tag{67}$$

$$\mathit{Funcon}\ \textbf{return}(\_ : T) : \Rightarrow\textbf{empty-type} \tag{68}$$

$$\mathit{Rule}\ \textbf{return}(V : T) \rightsquigarrow \textbf{abrupt}(\textbf{returned}(V)) \tag{69}$$

We then define a funcon **handle-return** that 'handles' returned values by evaluating its argument and gives V if its argument terminates normally with value V, or abruptly for reason **returned**(V). This is analogous to **handle-thrown**, so we omit the rules. Further examples of using control-flow entities are in Sect. 5.

4.5. Functions and closures

Semantic entities hold *values*, not arbitrary computation terms. However, sometimes we want to record an unevaluated computation; for example, we need to represent procedural abstractions as values in the environment.

The value constructor **abstraction** takes a computation as argument and constructs a value:

$$\textit{Funcon}\ \textbf{abstraction}(\_ : S{\Rightarrow}T)\ :\ \textbf{abstractions}(S{\Rightarrow}T) \tag{70}$$

Procedural abstractions are modelled using the **functions** type defined below.

$$\textit{Datatype}\ \textbf{functions}(S, T)\ ::=\ \textbf{function}(\_ : \textbf{abstractions}(S{\Rightarrow}T)) \tag{71}$$

We do not merge **functions** and **abstractions** into a single funcon, because we also model other concepts using abstractions, including patterns (which compute bindings), thunks (delayed computations), and continuations (which we discuss in Sect. 5).

We model argument passing using the given-value entity: **given** refers to the argument (which could be a tuple of arguments) inside the function body at the definition site of the function, and **give** supplies the argument at the call site of the function. Function application is modelled by extracting the computation from the **function** value, and *giving* an argument to that computation:

$$\textit{Funcon}\ \textbf{apply}(\_ : \textbf{functions}(S, T), \_ : S) : {\Rightarrow}T \tag{72}$$

$$\textit{Rule}\ \textbf{apply}(\textbf{function}(\textbf{abstraction}(X)), V : S) \rightsquigarrow \textbf{give}(V, X) \tag{73}$$

Note that we are *not* saying that we translate occurrences of SIMPLE identifiers in a function body to the funcon **given**. Rather, we use **given** to refer to the argument when expressing the semantics of parameter binding. The bindings are then scoped over the function body, using **scope**. (See Appendix A.4.3.)

This treatment of functions leads to *dynamic scoping* for identifiers, as the **bound** funcon is defined such that it looks up the identifier in the environment where it is evaluated. To support *static scoping* of identifiers we use the funcon **closed**, which ensures that its computation argument is evaluated without any non-local bindings:

$$\textit{Funcon}\ \textbf{closed}(\_ : {\Rightarrow}T) : {\Rightarrow}T \tag{74}$$

$$\textit{Rule}\ \frac{\mathsf{environment}(\textbf{map-empty}) \vdash X \longrightarrow X'}{\mathsf{environment}(\_) \vdash \textbf{closed}(X) \longrightarrow \textbf{closed}(X')} \tag{75}$$

$$\textit{Rule}\ \textbf{closed}(V : T) \rightsquigarrow V \tag{76}$$

The closure of a computation X is an abstraction with a closed body derived from X and with the current environment available as local bindings:

$$\textit{Funcon}\ \textbf{closure}(\_ : S{\Rightarrow}T) : {\Rightarrow}\textbf{abstractions}(S{\Rightarrow}T) \tag{77}$$

$$\textit{Rule}\ \mathsf{environment}(\rho) \vdash \textbf{closure}(X) \longrightarrow \textbf{abstraction}(\textbf{closed}(\textbf{scope}(\rho, X))) \tag{78}$$

The computation X will thus be evaluated with respect to the environment where the closure is formed, not the environment where the closure is evaluated.

4.6. Remarks

We believe language definitions written with CBS are relatively easy to comprehend and accessible to a wide audience. The process of defining the syntax of a language with (a variant of Extended) BNF is ubiquitous and well-understood. Funcon translations are defined through simple translation equations familiar especially to users experienced in denotational semantics. The I-MSOS rules that define funcons are similar to standard SOS rules, with the difference that that auxiliary semantic entities may be omitted, making specifications more concise. Moreover, for understanding the behaviour of an individual funcon it is unnecessary to know the details of the implicit propagation rules of the entity classes. Most importantly, to define languages in CBS, awareness of the precise formal semantics of the used funcons is not required: an informal explanation suffices, which is provided in the form of inline documentation (available online [7]).

The library of reusable funcons contains funcons that capture many types of constructs found in programming languages across language paradigms. For example, the funcons give semantics to different evaluation strategies in procedural and functional programming, algebraic data types and patterns in functional programming, classes and inheritance in object-oriented programming, as well as GOTO-statements and other forms of abnormal control flow.

Funcons have not yet been developed for all aspects of existing programming languages. For example, the library of funcons currently contains funcons for interleaving (and its prevention), and for interactive input and output, but not for threads, synchronisation, or distributed processes. Adding appropriate entities for use in CBS rules specifying the semantics of such constructs should be straightforward, based on previous experience with using MSOS for Concurrent ML [24] and on the many examples of process algebra specifications provided in the literature (the latter relies on the close relationship between small-step MSOS and the conventional framework of transition system specifications). Deciding whether to include funcons for synchronous or asynchronous communication (or both) may be more difficult, but case studies can be carried

```
function main() {
  print(1);
  {
    print(2);
    shift k { resume k; print(3); resume k; }
    print(4);
  }
  print(5);
}
```

Fig. 2. A program that prints the sequence '1 2 4 3 4 5'.

out to inform such choices. The main difficulties seem likely to arise in connection with executability and validation: interpreter generation will need to ensure that CBS rules are applied fairly, and model checking may be needed to determine whether all intended computations are possible. The $\mathbb{K}$ framework addressed those issues [9] by annotating transition rules (essentially identifying different states); it is unclear whether we will be able to adopt a similar approach for interpreter generation based on CBS rules.

In general, if funcons for particular language constructs cannot be defined in CBS as it is presented in this paper, we may extend the current collection of entity classes. Adding new entities would not affect the definitions of existing funcons, assuming that the new entities are implicitly propagated when not mentioned in rules.

We have only considered the *dynamic* semantics of funcons here. Together with Churchill and Torrini, two of the present authors have previously explored specifying the *static* semantics of funcons [5]. Following that approach, we intend to add facilities for declaring typing rules in CBS, and to generate static analyses from those rules; but this remains as future work.

5. Case study: delimited continuations

In Sect. 3, we discussed using CBS to specify the semantics of an existing language (SIMPLE). In this section, we demonstrate the ease with which an existing CBS language specification can be extended, using the addition of a *delimited control operator* as a small case study. We also use this case study to demonstrate how advanced forms of control flow can be specified in CBS.

5.1. Extending SIMPLE with delimited continuations

First we extend SIMPLE with syntax for two new forms of statement:

ImpStmt : imp-stmt ::= ... | '**shift**' id block | '**resume**' id '**;**'

Informally, the desired semantics of '**shift** *Id Block*' is that the remainder of the enclosing block is captured as a *delimited continuation*, bound to *Id*, and scoped over *Block*. Then, *Block* is executed *instead of the remainder of the enclosing block*. During execution of *Block*, the captured continuation can be executed zero or more times, which is achieved by invoking '**resume** *Id*'. This **shift** statement is essentially Danvy and Filinski's [28] *shift* operator, except formulated as a statement rather than an expression. See Fig. 2 for an example.

The funcon repository contains several funcons for expressing delimited control (we defer the accompanying rules to Sect 5.2):

$$\mathit{Funcon}\ \mathbf{control}(\_ : \mathbf{functions}(\mathbf{continuations}(T, S), S)) : \Rightarrow T \tag{79}$$

$$\mathit{Funcon}\ \mathbf{delimit\text{-}cc}(\_ : \Rightarrow T) : \Rightarrow T \tag{80}$$

$$\mathit{Funcon}\ \mathbf{resume\text{-}continuation}(\_ : \mathbf{continuations}(S, T), \_ : S) : \Rightarrow T \tag{81}$$

The **control** and **delimit-cc** funcons correspond closely to Felleisen's [29] *control* and *prompt* operators. The only difference is that captured continuations have type **continuations**, which is distinct from the type **functions**. The **resume-continuation** funcon is analogous to **apply** (Rules (72) & (73)).

We formally define the semantics of **shift** and **resume** by translation to funcons. In (extended) SIMPLE, blocks act as delimiters when determining the scope of a captured continuation. The existing translation of blocks (from Appendix A.3) is simply to execute the statements within the block:

$$\mathit{Rule}\ \mathit{exec}\ [\![\text{'}\mathbf{\{}\text{'}\mathit{Stmts}\text{'}\mathbf{\}}\text{'}]\!] = \mathit{exec}\ [\![\mathit{Stmts}]\!] \tag{82}$$

We therefore modify this equation to also insert a continuation delimiter:

$$\mathit{Rule}\ \mathit{exec}\ [\![\text{'}\mathbf{\{}\text{'}\mathit{Stmts}\text{'}\mathbf{\}}\text{'}]\!] = \mathbf{delimit\text{-}cc}(\mathit{exec}\ [\![\mathit{Stmts}]\!]) \tag{83}$$

The **control** funcon captures the current continuation in a similar manner to SIMPLE's **shift** statement. However, SIMPLE's **shift** statement has an explicit binder as part of the syntax, whereas **control** takes a function as its argument and applies that function to the captured continuation. To bridge this mismatch, the translation of a **shift** statement to funcons explicitly constructs a function that binds the argument (the continuation) to the identifier:

$$\mathit{Rule}\ \mathit{exec}\ [\![\text{`}\mathbf{shift}\text{'}\ \mathit{Id}\ \mathit{Block}]\!] = \mathbf{control}(\mathbf{function}(\mathbf{closure}(\mathbf{scope}(\mathbf{bind}(\mathit{id}\ [\![\mathit{Id}]\!], \mathbf{given}), \mathit{exec}\ [\![\mathit{Block}]\!])))) \tag{84}$$

The signature of the **control** funcon indicates that it computes a value, and thus, in general, the captured continuation may include an enclosing expression. Consequently, the **resume-continuation** funcon requires a value to plug the hole in the expression where **control** occurred. Conversely, SIMPLE's **shift** is a *statement*, and thus does not require a value to resume. To bridge this mismatch, the translation of a **resume** statement inserts a **null** value:

$$\mathit{Rule}\ \mathit{exec}\ [\![\text{`}\mathbf{resume}\text{'}\ \mathit{Id}\ \text{`}\mathbf{;}\text{'}]\!] = \mathbf{resume\text{-}continuation}(\mathbf{bound}(\mathit{id}\ [\![\mathit{Id}]\!]), \mathbf{null}) \tag{85}$$

This serves as a representative example of how the definition of a language feature often proceeds. The funcon repository does not provide a funcon for every possible variation of every possible language feature. Rather, it provides funcons that embody core concepts, and then the translation expresses the semantics of the language feature as a combination of those funcons.

This completes the addition of delimited continuations to SIMPLE. We only needed to declare the new syntax, add translation equations for that syntax, and modify one existing translation equation. A cynical reader may consider that we have 'cheated': the complexity of the semantics of **shift** is in the definition of **control**, **delimit-cc**, and **resume-continuation**, and we have not included these definitions. But this is precisely our aim: by providing reusable components with pre-specified semantics, the effort required to formally specify a programming language is significantly reduced by moving the majority of the specification work into those reusable components.

5.2. Defining delimited control funcons

We now present the definitions of the funcons that we used to specify the semantics of SIMPLE's delimited continuations. These are based on the I-MSOS definitions of similar funcons previously presented by two authors of this article, together with Torrini [30]. However, the control-flow entities provided by CBS facilitate a simpler specification than was possible in that prior work.

First we define the **continuations** type in terms of **abstractions**, in a similar manner to **functions** (Rule (71)):

$$\mathit{Datatype}\ \mathbf{continuations}(S, T)\ ::=\ \mathbf{continuation}(\_ : \mathbf{abstractions}(()\Rightarrow T)) \tag{86}$$

Next we need to express how **control** captures the current continuation. The key idea is to use control-flow entities to communicate between **control** and its enclosing **delimit-cc**, in a similar manner to how abrupt termination is modelled (Sect. 4.4.5). We define **control** to emit a signal containing the function that is to be applied to the continuation, and leave a **hole** in the term:

$$\mathit{Rule}\ \mathbf{control}(F : \mathbf{functions}(\_, \_)) \xrightarrow{\text{control-signal}(F)} \mathbf{hole} \tag{87}$$

The **delimit-cc** funcon receives the control-signal, constructs the continuation, and applies the received function to that continuation:

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{control-signal}(F)} X'}{\mathbf{delimit\text{-}cc}(X) \xrightarrow{\text{control-signal}()} \mathbf{delimit\text{-}cc}(\mathbf{apply}(F, \mathbf{continuation}(\mathbf{closure}(X'))))} \tag{88}$$

We also need rules for when the body of **delimit-cc** does not emit a control-signal, and for when it has been evaluated to a value:

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{control-signal}()} X'}{\mathbf{delimit\text{-}cc}(X) \xrightarrow{\text{control-signal}()} \mathbf{delimit\text{-}cc}(X')} \tag{89}$$

$$\mathit{Rule}\ \mathbf{delimit\text{-}cc}(V : T) \rightsquigarrow V \tag{90}$$

The **resume-continuation** funcon takes a value as its second argument and 'plugs' it into the **hole** in the continuation. This is defined as follows:

$$\mathit{Rule}\ \frac{X \xrightarrow{\text{plug-signal}(V)} X'}{\mathbf{resume\text{-}continuation}(\mathbf{continuation}(\mathbf{abstraction}(X)), V : T) \xrightarrow{\text{plug-signal}()} X'} \tag{91}$$

$$\mathit{Funcon}\ \mathbf{hole} : \Rightarrow \mathbf{values} \tag{92}$$

$$\mathit{Rule}\ \mathbf{hole} \xrightarrow{\text{plug-signal}(V)} V \tag{93}$$

5.3. Remarks

The use of **resume-continuation** and **hole** is the main simplification relative to the specification in [30]. The key enabling feature provided by CBS is control-flow entities that allow signals to synchronise between two funcons, thereby allowing information to flow in either direction. Thus plug-signal can transmit a value into a sub-term, and control-signal can transmit a value to an enclosing delimiter. In [30], output entities were used to simulate outward flowing control-flow signals (which worked rather well), and contextual entities were used to simulate inward flowing control-flow signals (which was undesirable). The main problem with the latter is that a premise cannot directly detect whether the contents of a contextual entity are used in a computation step, so it is not straightforward to enforce that a plug-signal is received (this is necessary in the presence of non-deterministic evaluation order and multiple occurrences of **control**). This was worked around by tagging each hole with a special kind of identifier, and then maintaining an auxiliary environment that accumulated all the values to be plugged into those holes. This was circuitous, and detracted from the clarity of the specification.

6. Executing funcon terms

A CBS language definition is intended to be executable so that a prototype implementation of a language can be generated from its definition. Each of the three aspects of a CBS definition – abstract syntax grammars, semantic translation functions, and the I-MSOS rules for funcons – generates a component of the prototype implementation. In Sect. 7 we discuss the tools involved in generating and connecting the components responsible for parsing a program and translating the program to a funcon term. In this section we discuss some of the features of the funcon term interpreters generated in Haskell from CBS funcon definitions by a compiler for CBS. In particular, we explain that the generated Haskell fragments are as modular as the funcon definitions that they implement. As a result, generated Haskell modules compose freely to form interpreters for arbitrary funcon collections and can be reused without recompilation. The implementation of the reusable funcon library is provided alongside CBS, enabling interpretation for all languages specified without language-specific funcons. Users employ the CBS compiler only to generate implementations for any language-specific funcons. We also discuss a suite of values built-in to CBS and the development of implementation-specific funcons and unit-tests with configuration files.

6.1. Funcon modules

The Haskell funcon framework [31] provides three Haskell modules:

- `Funcons.EDSL` exports helper functions that are used in the implementation of funcons. For example, to modify or access semantic entities, or to test side-conditions.
- `Funcons.Tools` exports helper functions for composing *funcon modules*. A funcon module is a Haskell module exporting three collections containing information about funcon, data type, and semantic-entity definitions, respectively. Each funcon module exports an interpreter, which is aware only of the components defined in the module.
- `Funcons.Core`, a funcon module containing the implementation of the funcons in the reusable library. This includes all the funcons mentioned in this paper except the SIMPLE-specific funcon **allocate-nested-vectors**. The implementations have been generated once and for all.

Funcon modules have two noteworthy properties. Firstly, they are independent and do not need to import other funcon modules; they only need to import `Funcons.EDSL`, `Funcons.Operations` (see Sect. 6.5) and `Funcons.Tools`. Secondly, funcon modules can be freely composed: a module implementing funcons *A* and a module implementing funcons *B* are composed without restrictions to form a module implementing funcons *A* and *B*. CBS files are thus compiled individually, and the resulting modules can be composed as needed. Moreover, once a module has been compiled, it can be linked as needed, without requiring recompilation. This makes it possible to deliver the implementation of the reusable funcon library as an isolated package. Enforcing these properties has been a leading principle in the development of the Haskell funcon framework. For the purpose of illustration we have compiled a contrived CBS file containing the definitions of **environments**, **scope**, and **returning** (see Rules (25), (29), and (67)). Fig. 3 shows a fragment of the funcon module generated from this file. It omits the functions that implement the behaviour of the funcons, e.g. *stepEnvironments*.

We represent funcon terms using strings for the names of funcons in a data type, with generic constructors for funcon application, constants (funcons with no arguments), and literals (built-in values and types). A funcon module contains a *funcon library* mapping funcon names to their implementation (*funcons* in Fig. 3). Similarly, we use maps to associate the names of semantic entities with values (in Fig. 3, *entities* associates semantic entities with default values). Evaluating funcon terms can thus cause runtime errors because it involves searching in maps and there are no static guarantees that the required entries are available. By using strings in these ways we have gained the desired modularity, but lost static guarantees provided by a Haskell compiler. Instead, we rely on the tools explained in Sect. 7 to ensure that all mentioned funcons and entities are defined and that their code is indeed generated and included in the final interpreter.

An alternative approach would have been to use the *Data types à la carte* technique [32] to combine funcons defined in separate modules into a single data type with a constructor for each funcon. Having individual Haskell constructors

```
import Funcons.Tools (mkFreshInterpreter)
import Funcons.EDSL

import Funcons.Operations hiding (Values, libFromList)

import Funcons.Tools

main = mkMainWithLibraryEntitiesTypes funcons entities types

entities = [ ]

types = typeEnvFromList
  [("returning", DataTypeMembers "returning" [ ]
    [DataTypeConstructor "returned"
      [TName "defined-values"] (Just [ ])])]

funcons = libFromList
  [("environments", NullaryFuncon stepEnvironments)
  ,("scope", NonStrictFuncon stepScope)
  ,("returning", NullaryFuncon stepReturning)
  ,("returned", StrictFuncon stepReturned)]
```

Fig. 3. Excerpt from a funcon module for a contrived collection of CBS funcon specifications.

for each funcon would have provided stronger correct-by-construction guarantees about the well-formedness of funcon terms. Several authors [33,34] have used this technique for the specific purpose of defining modular programming-language constructs in a Haskell setting.

Types are either built-in to CBS (e.g. **integers** and **values**), provided as reusable components (e.g. **environments** and **identifiers**), or defined as part of a language definition. A funcon module generated from a CBS file provides a funcon implementation for each type and data-type constructor defined in the file (see *funcons* in Fig. 3). In addition, a funcon module provides a type environment which relates the data types to their constructors (*types* in Fig. 3). An interpreter uses the type environment for dynamic type checking.

6.2. Executable funcons

In Sect. 4.4 we described how each class of semantic entity is implicitly propagated in CBS rules. We achieve implicit propagation in funcon code by working in a monad, and defining its *return* and *bind* operations such that they implement the desired implicit propagation. A contextual entity corresponds to a reader monad, a mutable entity corresponds to a state monad, and an output entity corresponds to a writer monad [35]. Input entities correspond to a restricted form of state monad (input is always consumed in order, and each input can only be consumed once), and control-flow entities correspond to a combination of a reader and a writer monad (except that only one signal may be emitted in each control-flow entity at once; they do not form a monoid).

We define a data type *IMSOS* that combines the five semantic-entity classes (we omit the monad instance):

$$
\begin{aligned}
&\textbf{data}\ \mathit{IMSOS}\ a &&= \mathit{IMSOS}\ (\mathit{MReader} \to \mathit{MState} \to (\mathit{Either}\ \mathit{Exception}\ a, \mathit{MState}, \mathit{MWriter}))\\
&\textbf{data}\ \mathit{MReader} &&= \mathit{MReader}\ \mathit{Contextual}\ \mathit{ControlFlow}\\
&\textbf{data}\ \mathit{MState} &&= \mathit{MState}\ \mathit{Mutable}\ \mathit{Input}\\
&\textbf{data}\ \mathit{MWriter} &&= \mathit{MWriter}\ \mathit{ControlFlow}\ \mathit{Output}\\
&\textbf{data}\ \mathit{Exception} &&= \ldots
\end{aligned}
$$

Each semantic-entity class such as *Contextual* is implemented by a mapping from entity names to values. This avoids the need to use monad transformers [36] to add additional entities: our *IMSOS* monad is fixed. Using Haskell's native support for monads, we can generate human-readable funcon code that only refers to the entities explicitly mentioned in the funcon definition. As a consequence, the code of a funcon is as modular and compositional as its CBS definition. Moreover, the code is only regenerated if the CBS file in which the funcon is defined is regenerated. Fig. 4 gives an indication of generated funcon code. It contains the implementation of **scope**, which was omitted from Fig. 3.

The code for a funcon is formed by one sequence of statements in the *IMSOS* monad for every rule of the funcon. The code of Fig. 4 shows three rules. The rules *rewrite1* and *step1* correspond to Rules (31) and (30), respectively. The rule *step2* is the congruence rule generated because the signature of **scope** is strict in its first parameter. The statements can access or modify semantic-entity values, perform pattern matching and substitution, and test side conditions and premises. The statements generated for a rule may raise several kinds of internal exceptions, each representing a particular reason why the rule implemented by those statements is not applicable. For example, a funcon can be applied to incorrect arguments (e.g. if the first argument of **if-true-else** was not a Boolean), pattern matching may have failed, or a premise may not hold. To handle failure, the *IMSOS* monad contains the *Exception* component and backtracking is used to try an alternative rule. If no rules are applicable to a particular term, i.e. all alternatives raise an exception, then the term is stuck.

The *IMSOS* monad stores extra information, including the funcon library, runtime options, and collected meta-data; but we elide those details here. The code for a funcon forms a micro-interpreter that can be executed independently, given

```
stepScope fargs =
  evalRules [rewrite1] [step1, step2]
  where
    rewrite1 = do
      let env = emptyEnv
      env ← fsMatch fargs [PAnnotated PWildCard (TName "environments")
                          , PAnnotated (PMetaVar "V") (TName "values")] env
      rewriteTermTo (TVar "V") env
    step1 = do
      let env = emptyEnv
      env ← liftedfsMatch fargs
              [PAnnotated PWildCard (TName "environments")
              , PMetaVar "X"] env
      env ← getInhPatt "environment" (VPMetaVar "Rho0") env
      env ← liftedsideCondition (SCPatternMatch (TApp "map-override"
              [TVar "Rho1", TVar "Rho0"]) (VPMetaVar "Rho2")) env
      env ← withInhTerm "environment" (TVar "Rho2") env
              (premise (TVar "X") (PMetaVar "X'") env)
      stepTermTo (TApp "scope" (TTuple [TVar "Rho1", TVar "X'"])) env
    step2 = do
      let env = emptyEnv
      env ← liftedfsMatch fargs [PMetaVar "X1", PMetaVar "X2"] env
      env ← premise (TVar "X1") (PMetaVar "X1'") env
      stepTermTo (TApp "scope" (TTuple [TVar "X1'", TVar "X2"])) env
```

Fig. 4. Haskell code implementing **scope**.

a funcon library with entries for all funcons explicitly mentioned in the definition of the funcon. As an alternative to input/output simulation, the *IMSOS* monad also provides facilities for connecting input and output entities to real console input/output, allowing a user to run a funcon program interactively.

6.3. Context-free rewriting

CBS supports two transition relations: computation steps and rewrites (see Sect. 4.1). Computation steps may depend on semantic entities, making them sensitive to context. Furthermore, computation steps may modify semantic entities, causing observable side effects. Conversely, the rewrite relation is context insensitive and never causes observable side effects. To separate rewriting from computation steps we use a separate monad (not shown), which does not provide access to the semantic entities. This guarantees, by construction, that rewrites do not access semantic entities.

The properties of the rewrite relation are such that, operationally, it can be applied at any time during execution anywhere in a term. Thus, funcon term interpreters are free to perform rewriting whenever is most convenient or efficient. Experience has shown that the choice of strategy to determine when rewrites are performed has a tremendous influence on the efficiency of funcon term interpreters (see also Sect. 6.7). A greedy rewriting strategy performs rewrites immediately on every occurrence of a term, possibly resulting in redundant rewrites (e.g. rewriting all arguments **if-true-else**). We choose to rewrite a term just before performing a computational step, and directly afterwards on the term resulting from the step if it was successful. This strategy exhibits inefficiencies in combination with our backtracking approach. A rule may turn out to be inapplicable after some rewrites have been performed. The rewrites are forgotten when backtracking selects the next rule to try. We expect this problem is easy to avoid in a host language providing mutable references or objects. Our Haskell implementation would benefit from graph reduction, sharing the effects of rewrites [37].

6.4. Refocussing optimisation

Directly implementing a small-step operational semantics as the transitive closure of the computation-step relation is straightforward but inefficient. At each computation step, the interpreter traverses the funcon term from the root to a subterm (sometimes called the redex), 'executes' the subterm by replacing it, and reconstructs a (mostly identical) term. Simultaneously, the values held by semantic entities are maintained and possibly modified. This corresponds to constructing a derivation tree in accordance to the I-MSOS rules (including the implicit congruence rules) of a CBS language definition. The cost of each step is potentially linear in the size of the funcon term.

To overcome this inefficiency, we have added a *refocussing* optimisation [38]. The key insight of refocussing is that, in many situations, once execution of a subterm has begun, the subsequent computation steps will involve executing that same subterm, until it is reduced to a value. Therefore, rather than re-traversing the term each step, the interpreter can 'take a short-cut' and locally continue executing the subterm. This optimisation is easily implemented and significantly improves the efficiency of the interpreter.

However, it is not always valid to 'take a short-cut'. In general, deciding the next subterm to be executed requires checking the conditions of the inference rules. The execution of a computational step, which involves replacing some subterm

r with r', may be accompanied by a modification to the value held by some semantic entity. Because of the latter, the conditions for choosing r as the subterm to be executed may no longer hold, in which case the next subterm to be executed is not r', invalidating refocussing. For example, when a control-flow signal is emitted, execution of the current subterm should *not* continue, as that signal needs to be handled further up the term. Typically, the handler for a control-flow signal will then change the subterm that is next to be executed (e.g. Rule (63) for **handle-thrown** in Sect. 4.4.5). Our interpreter checks for emitted signals and additional output to determine whether to apply refocussing. To guarantee the validity of refocussing in all cases, we place restrictions on the way semantic entities are used in inference rules, as suggested by Bach Poulsen [39].

6.5. Values and value operations

The CBS framework provides a large suite of values and operations on those values (see Appendix B.2 for examples). Some operations have not been specified in CBS, for example operations on integers and floating-point numbers. In these cases the operations are manually implemented and exported by the module `Funcons.Operations` provided by a separate library [40]. The library is separate from the the Haskell funcon framework and can be used by other semantic frameworks implemented in Haskell.

Some built-in values are native to Haskell, such as floating-point numbers and ASCII characters. Other types of values are provided by libraries, such as vectors and multisets.[10] Most of the value operations are directly implemented by a function provided by an existing Haskell library. In other cases, the operation is easy to define in terms of existing operations or functions. Creating funcon modules that export the corresponding funcons is done manually by unwrapping values from the funcon term data type, applying the corresponding operation from `Funcons.Operations`, and wrapping the result. This work is done once and for all, as the suite of values and operations is fixed (exported by `Funcons.Core`).

6.6. Configuration

The interpreters exported by funcon modules are easily configured by command line flags or configuration files. For example, a user can restrict the number of computational steps performed. This is useful to determine which steps lead up to unwanted behaviour (arising from an invalid program or an incorrect specification). The interpreters can print debugging information such as the values of semantic entities, the number of computational steps performed, or the number of rewrites performed. The documentation of `Funcons.Tools` [31] contains an exhaustive enumeration of all configuration options.

Configuration files offer a limited method for defining funcons. The primary use case is to define *implementation-specific* funcons representing a particular implementation choice. Language manuals often leave options for language implementers regarding certain aspects of the language, e.g., the number of bits used in the binary representation of different forms of numbers. The following excerpt shows the definition of some implementation-specific funcons in a configuration file:

```
funcons {
    implemented-floats-format   = binary64;
    implemented-integers-width  = 31;
    implemented-characters      = unicode-characters;
}
```

Configuration files also offer a simple but effective method for defining unit tests. A configuration file for a unit test contains an initial term to be evaluated and the expected outcome of evaluating the term. The expected outcome consists of a result term, as well as the values of some semantic entities. The excerpt below shows an example configuration file containing a unit test.

```
general{
  funcon-term:  give(read,sequential(print(given),print(given)));
}
inputs{
  standard-in:  2,3;   // provides 2 and 3 as input
}
tests{
  result-term:  null;  // expects null as the result value
  standard-out: 2,2;   // expects 2 is printed twice
  standard-in:  3;     // expects 3 as remaining input
}
```

Differences between expected and actual outcome are reported to the standard output. A successful unit test is silent. Entities not mentioned in a unit test are treated differently depending on the class of the entity. For example, any value for store is considered acceptable outcome, if the tests does not mention store. However, a raised abrupted signal is considered a violation if the test did not mention the entity abrupted.

[10] The `vector` and `multiset` packages on Hackage.

Table 1
Runtimes in seconds of the generated SIMPLE interpreter on several test programs.

Test program	Generated interpreter		𝕂 tool
	Unoptimised	Refocussing enabled	
`exception tests 1 to 15`	11.4	1.67	30.3
`div-nondet`	0.2	0.06	1.9
`factorial`	2.8	0.18	1.9
`collatz`	11.3	0.53	1.9
running total	25.6	2.43	35.9
`higher-order`	–	3.02	10.1
`matrix`	–	5.84	2.2
`sortings`	–	6.11	2.9
running total	–	17.4	51.1

6.7. Evaluation

The Haskell funcon framework is developed together with a CBS compiler that generates funcon modules from CBS specifications. Using these tools, an interpreter is obtained in two steps. Firstly, the CBS compiler is used to generate code for any language-specific funcons defined in the specification. Secondly, using helper functions in `Funcons.Tools` of the framework, the funcon modules generated for the language are combined with `Funcons.Core`, the module containing the implementations of the reusable funcons. In the case of SIMPLE, the resulting funcon module exports an interpreter capable of executing all funcon terms produced by translating SIMPLE programs. The interpreter for SIMPLE is available on Hackage [41]. If a specification does not define language-specific funcons, then no code needs to be generated and the interpreter provided by `Funcons.Core` can be used directly.

We have tested the SIMPLE interpreter on some of the SIMPLE test programs that come bundled with distributions of the 𝕂 tool [42]. Table 1 presents the runtimes of the interpreter for several of these test programs, with and without the refocussing optimisation enabled (see Sect. 6.4). For reference, we have included the runtimes of the same test programs executed with Version 5 of the 𝕂 tool. These results were produced on a virtual private server with four 2.2 GHz virtual CPUs and 8GiB of RAM, under Ubuntu 18.04. The Haskell funcon modules were compiled using GHC 8.0.2. The 𝕂 specification of SIMPLE was compiled with the Java back end of the 𝕂 tool and loaded in server-mode to avoid compilation overhead on each run. Several runs of 'JVM warm up' have been performed. The numbers are the average runtimes of 10 runs.

Without refocussing enabled, the tests `higher-order`, `matrix` and `sortings` cause the memory to overflow. These tests involve a relatively large number of function calls or loop-iterations, which result in considerable growth of the funcon term under evaluation, thus increasing the overhead of decomposing and recomposing the term at each step of the small-step evaluation. Decomposition involves potentially a lot of backtracking and undoing context-free rewrites unnecessarily, as discussed in Sect. 6.3. With refocussing, these programs execute in three to six seconds.

This experiment shows that the generated SIMPLE interpreter is adequate for testing the SIMPLE specification by running basic programs.

7. Integrated development environment for CBS

CBS language definitions can be developed, maintained, and tested in Eclipse. We have used SPOOFAX [8] to develop an Eclipse-based editor for CBS language definitions, with many useful features for syntax-aware browsing and editing.

- The CBS editor has menu actions for generating language editors, web pages, and funcon indexes from CBS language definitions; outdated files are detected and regenerated when changes have been made.
- A generated language editor parses programs in the specified language (flagging any text that has syntax errors or ambiguities) and provides basic syntax highlighting; there is also a menu action for translating programs into funcon terms.
- The CBS compiler generates modules for any language-specific funcons from CBS language definitions. These are composed with `Funcons.Core` to form a language-specific funcon term interpreter. The CBS compiler and the interpreter are added as external tools so that they can be used from within Eclipse.

The rest of this section discusses these features in more detail.

7.1. Editing CBS files

The CBS editor analyses all CBS source files in an Eclipse project, updating the analysis in the background whenever the user edits the text. It checks that the text conforms to the context-free grammar of the CBS meta-language, adds syntax

highlighting, and flags as erroneous any parts of the text that cannot be parsed. It also creates hyperlinks from uses of symbols to the locations where the symbols are introduced, flagging missing symbols as errors.

The CBS analysis is independent of the division of language and funcon definitions into files and folders. For example, we may divide language definitions into sections in the same way as reference manuals, and put each section in a separate file. The reusable funcons are currently grouped together in files according to the semantic entities involved, to facilitate browsing of closely related funcons. The hyperlinks in the edited files allow navigation and browsing of CBS specifications without awareness of the underlying file system. The analysis was implemented by specifying it in the current Spoofax meta-language for static semantics, NaBL2 [43], based on the concepts of scope graphs [44].

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 language syntax and its translation to funcons. If a new funcon is needed, it can be added to the language specification for local use (like **allocate-nested-vectors**). New funcons that are language-independent may subsequently be submitted for inclusion in the library of reusable funcons.

The screenshot in Fig. 5 shows several files open in our integrated development environment (IDE) during the development of the component-based semantics of Simple. The top-left pane is browsing the abstract syntax and semantics of variable declarations. In the bottom-left pane, a CBS rule defining the translation of variable-declaration statements to funcons is being edited; the red mark in the margin flags an error in the name of the translation function for declarations. The colours and fonts distinguish the names of syntax non-terminals (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 CBS file containing the definition of the reusable funcon **scope**.

Name analysis and arity checking were specified in NaBL2: the editor resolves all references to names (with separate namespaces for funcons, syntax sorts, translation functions, and meta-variables) to their declarations, flagging any missing or duplicate declarations as errors. However, full type checking for CBS cannot be specified in NaBL2, which does not support structural subtyping.

7.2. Generating funcon terms

A generated language editor uses a scannerless generalised LR (SGLR) parser generated from the abstract syntax grammar of a CBS language definition. An abstract syntax grammar is usually highly ambiguous; to support disambiguation, disambiguation rules are currently written inside comment blocks in CBS specifications. The language editor has a menu action for translating the abstract syntax trees of disambiguated programs to funcon terms, executing code generated from the semantic equations. The two panes on the lower right of Fig. 5 show a small test program and part of its translation to funcons.

7.3. Evaluating funcon terms

Funcon term interpreters can be added to Eclipse as 'external tools'. This makes it possible to execute funcon term files and configuration files with unit tests from within Eclipse, showing the output of the interpreter in Eclipse's console. If a language specification defines any language-specific funcons, then the CBS files that define these funcons need to be compiled with the CBS compiler to produce interpreter code. The CBS compiler can be added to Eclipse as an external tool as well. The CBS compiler generates modular interpreters, as described in Sect. 6. If a CBS file containing a language-specific funcon is modified, only that file needs to be recompiled. The CBS compiler is not needed for any projects that use only funcons from the reusable library.

8. Conclusion

We have presented CBS, a unified meta-language for the formal specification of programming languages. Our intent is that a language designer will use CBS to define the syntax and semantics of a source language, using context-free grammars and translation functions that produce funcon terms (Sect. 3). The funcons themselves are defined in CBS and are provided as part of a library of over 100 pre-defined funcons that can be reused in the development of different programming languages. The reuse of formally specified components is the key novelty compared to other specification frameworks. If required, language-specific funcons can be defined, and new reusable funcons can be added to the main (open-ended) funcon library; this is possible because of the *modular* nature of the underlying formal framework (MSOS).

CBS definitions are *executable*. This allows the definitions of funcons and programming languages to be validated through testing, and for prototype interpreters to be generated from CBS definitions of new languages. We provide a suite of tools to support interactive development of funcon and language definitions (Sect. 7) as well as generating funcon term interpreters (Sect. 6).

Our hope is that with the release of over 100 validated funcons, supported by several case studies, CBS could 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; domain experts could be empowered to design, specify and implement domain-specific languages themselves; and researchers and students would be provided with a dedicated portal and online open-access repository for language and funcon definitions.

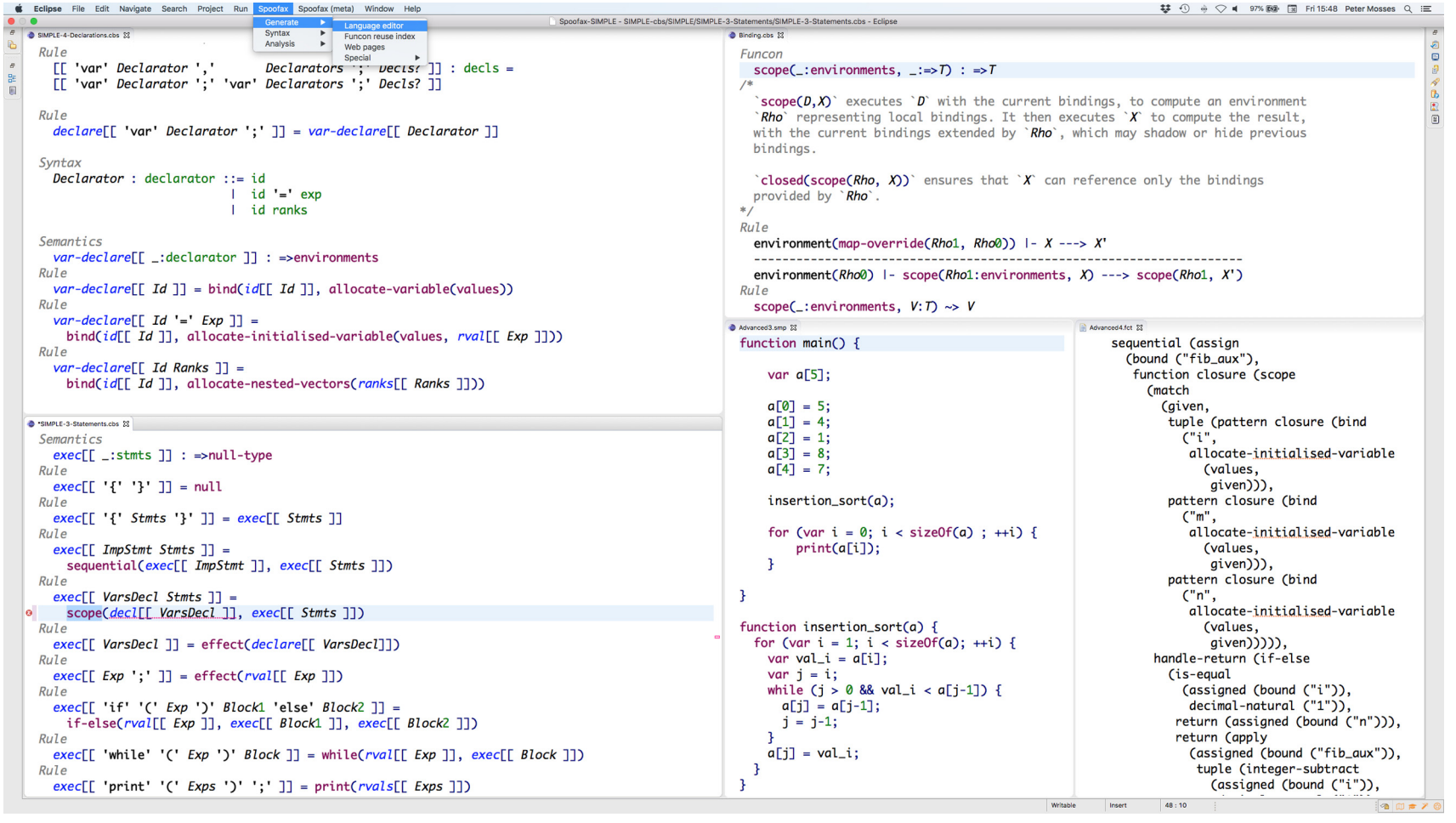


Fig. 5. The IDE for CBS in action. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

Acknowledgements

The authors are grateful to the anonymous reviewers for helpful comments and suggestions for improvement. 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).

Appendix A. Simple language definition

This appendix presents our CBS definition of the Simple language. The top-level translation function is *run*. We omit the lexical grammar for the non-terminals bool, int, string and id, and the equations for corresponding translation functions *val* and *id*, but they are available to be viewed online [7].

A.1. Values

Syntax V : value ::= bool | int | string

Semantics val ⟦_ : value⟧ : ⇒**values**

Semantics id ⟦_ : id⟧ : ⇒**ids**

A.2. Expressions

Syntax Exp : exp ::= '(' exp ')' | value | lexp | lexp '**=**' exp | '**++**' lexp
| '**−**' exp | exp '(' exps$^?$ ')' | '**sizeOf**' '(' exp ')' | '**read**' '(' ')'
| exp '**+**' exp | exp '**−**' exp | exp '*****' exp | exp '**/**' exp | exp '**%**' exp
| exp '**<**' exp | exp '**<=**' exp | exp '**>**' exp | exp '**>=**' exp
| exp '**==**' exp | exp '**!=**' exp | '**!**' exp | exp '**&&**' exp | exp '**||**' exp

Rule ⟦'(' *Exp* ')'⟧ : exp = ⟦*Exp*⟧

Semantics rval ⟦_ : exp⟧ : ⇒**values**

Rule rval ⟦*V*⟧ = *val* ⟦*V*⟧

Rule rval ⟦*LExp*⟧ = **assigned**(*lval* ⟦*LExp*⟧)

Rule rval ⟦*LExp* '**=**' *Exp*⟧ = **give**(*rval* ⟦*Exp*⟧, **sequential**(**assign**(*lval* ⟦*LExp*⟧, **given**), **given**))

Rule rval ⟦*LExp* '**++**'⟧ =
give(*lval* ⟦*LExp*⟧, **sequential**(**assign**(**given**, **integer-add**(**assigned**(**given**), 1)), **assigned**(**given**)))

Rule rval ⟦'**−**' *Exp*⟧ = **integer-negate**(*rval* ⟦*Exp*⟧)

Rule rval ⟦*Exp* '(' *Exps*$^?$ ')'⟧ = **apply**(*rval* ⟦*Exp*⟧, **tuple**(*rvals* ⟦*Exps*$^?$⟧))

Rule rval ⟦'**sizeOf**' '(' *Exp* ')'⟧ = **length**(**vector-elements**(*rval* ⟦*Exp*⟧))

Rule rval ⟦'**read**' '(' ')'⟧ = **read**

Rule rval ⟦Exp_1 '**+**' Exp_2⟧ = **integer-add**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**−**' Exp_2⟧ = **integer-subtract**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '*****' Exp_2⟧ = **integer-multiply**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**/**' Exp_2⟧ = **checked**(**integer-divide**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧))

Rule rval ⟦Exp_1 '**%**' Exp_2⟧ = **checked**(**integer-modulo**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧))

Rule rval ⟦Exp_1 '**<**' Exp_2⟧ = **is-less**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**<=**' Exp_2⟧ = **is-less-or-equal**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**>**' Exp_2⟧ = **is-greater**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**>=**' Exp_2⟧ = **is-greater-or-equal**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule rval ⟦Exp_1 '**==**' Exp_2⟧ = **is-equal**(*rval* ⟦Exp_1⟧, *rval* ⟦Exp_2⟧)

Rule *rval* $[\![Exp_1$ '**!=**' $Exp_2]\!]$ = **not**(**is-equal**(*rval* $[\![Exp_1]\!]$, *rval* $[\![Exp_2]\!]$))

Rule *rval* $[\![$'**!**' $Exp]\!]$ = **not**(*rval* $[\![Exp]\!]$)

Rule *rval* $[\![Exp_1$ '**&&**' $Exp_2]\!]$ = **if-true-else**(*rval* $[\![Exp_1]\!]$, *rval* $[\![Exp_2]\!]$, **false**)

Rule *rval* $[\![Exp_1$ '**||**' $Exp_2]\!]$ = **if-true-else**(*rval* $[\![Exp_1]\!]$, **true**, *rval* $[\![Exp_2]\!]$)

Syntax *Exps* : exps ::= exp ('**,**' exps)$^?$

Semantics *rvals* $[\![\_$: exps$^?]\!]$: (⇒**values**)*

Rule *rvals* $[\![\ ]\!]$ = ( )

Rule *rvals* $[\![Exp]\!]$ = *rval* $[\![Exp]\!]$

Rule *rvals* $[\![Exp$ '**,**' $Exps]\!]$ = *rval* $[\![Exp]\!]$, *rvals* $[\![Exps]\!]$

Syntax *LExp* : lexp ::= id | lexp '**[**' exps '**]**'

Rule $[\![LExp$ '**[**' Exp '**,**' $Exps$ '**]**'$]\!]$: lexp = $[\![LExp$ '**[**' Exp '**]**' '**[**' $Exps$ '**]**'$]\!]$

Semantics *lval* $[\![\_$: lexp$]\!]$: ⇒**variables**

Rule *lval* $[\![Id]\!]$ = **bound**(*id* $[\![Id]\!]$)

Rule *lval* $[\![LExp$ '**[**' Exp '**]**'$]\!]$ = **checked**(**index**(**integer-add**(1, *rval* $[\![Exp]\!]$), **vector-elements**(*rval* $[\![LExp]\!]$)))

A.3. Statements

Syntax *Block* : block ::= '**{**' stmts$^?$ '**}**'

Syntax *Stmts* : stmts ::= stmt stmts$^?$

Syntax *Stmt* : stmt ::= imp-stmt | vars-decl

Syntax *ImpStmt* : imp-stmt ::= block | exp '**;**'

| '**if**' '(' exp ')' block ('**else**' block)$^?$ | '**while**' '(' exp ')' block

| '**for**' '(' stmt exp '**;**' exp ')' block | '**print**' '(' exps ')' '**;**' | '**return**' exp$^?$ '**;**'

| '**try**' block '**catch**' '(' id ')' block | '**throw**' exp '**;**'

Rule $[\![$'**if**' '(' Exp ')' $Block]\!]$: stmt = $[\![$'**if**' '(' Exp ')' $Block$ '**else**' '**{**' '**}**'$]\!]$

Rule $[\![$'**for**' '(' $Stmt\ Exp_1$ '**;**' Exp_2 ')' '**{**' $Stmts$ '**}**'$]\!]$: stmt = $[\![$'**{**' $Stmt$ '**while**' '(' Exp_1 ')' '**{**' '**{**' $Stmts$ '**}**' Exp_2 '**;**' '**}**' '**}**'$]\!]$

Semantics *exec* $[\![\_$: stmts$]\!]$: ⇒**null-type**

Rule *exec* $[\![$'**{**' '**}**'$]\!]$ = **null**

Rule *exec* $[\![$'**{**'*Stmts*'**}**'$]\!]$ = *exec* $[\![Stmts]\!]$

Rule *exec* $[\![ImpStmt\ Stmts]\!]$ = **sequential**(*exec* $[\![ImpStmt]\!]$, *exec* $[\![Stmts]\!]$)

Rule *exec* $[\![VarsDecl\ Stmts]\!]$ = **scope**(*declare* $[\![VarsDecl]\!]$, *exec* $[\![Stmts]\!]$)

Rule *exec* $[\![VarsDecl]\!]$ = **effect**(*declare* $[\![VarsDecl]\!]$)

Rule *exec* $[\![Exp$ '**;**'$]\!]$ = **effect**(*rval* $[\![Exp]\!]$)

Rule *exec* $[\![$'**if**' '(' Exp ')' $Block_1$ '**else**' $Block_2]\!]$ = **if-true-else**(*rval* $[\![Exp]\!]$, *exec* $[\![Block_1]\!]$, *exec* $[\![Block_2]\!]$)

Rule *exec* $[\![$'**while**' '(' Exp ')' $Block]\!]$ = **while**(*rval* $[\![Exp]\!]$, *exec* $[\![Block]\!]$)

Rule *exec* $[\![$'**print**' '(' $Exps$ ')' '**;**'$]\!]$ = **print**(*rvals* $[\![Exps]\!]$)

Rule *exec* $[\![$'**return**' Exp '**;**'$]\!]$ = **return**(*rval* $[\![Exp]\!]$)

Rule *exec* $[\![$'**return**' '**;**'$]\!]$ = **return**(**null**)

Rule *exec* $[\![$'**try**' $Block_1$ '**catch**' '(' Id ')' $Block_2]\!]$ =

handle-thrown(*exec* ⟦*Block*$_1$⟧, **scope**(**bind**(*id* ⟦*Id*⟧, **allocate-initialised-variable**(**values**, **given**)), *exec* ⟦*Block*$_2$⟧))

Rule *exec* ⟦'**throw**' *Exp* '**;**'⟧ = **throw**(*rval* ⟦*Exp*⟧)

A.4. Declarations

Syntax *Decl* : decl ::= vars-decl | func-decl

Semantics *declare* ⟦_ : decl⟧ : ⇒**environments**

A.4.1. Variable declarations

Syntax *VarsDecl* : vars-decl ::= '**var**' declarators '**;**'

Syntax *Declarators* : declarators ::= declarator ('**,**' declarators)$^?$

Rule ⟦'**var**' *Declarator* '**,**' *Declarators* '**;**' *Stmts*$^?$⟧ : stmts = ⟦'**var**' *Declarator* '**;**' '**var**' *Declarators* '**;**' *Stmts*$^?$⟧

Rule ⟦'**var**' *Declarator* '**,**' *Declarators* '**;**' *Decls*$^?$⟧ : decls = ⟦'**var**' *Declarator* '**;**' '**var**' *Declarators* '**;**' *Decls*$^?$⟧

Rule *declare* ⟦'**var**' *Declarator* '**;**'⟧ = *var-declare* ⟦*Declarator*⟧

Syntax *Declarator* : declarator ::= id | id '**=**' exp | id ranks

Semantics *var-declare* ⟦_ : declarator⟧ : ⇒**environments**

Rule *var-declare* ⟦*Id*⟧ = **bind**(*id* ⟦*Id*⟧, **allocate-variable**(**values**))

Rule *var-declare* ⟦*Id* '**=**' *Exp*⟧ = **bind**(*id* ⟦*Id*⟧, **allocate-initialised-variable**(**values**, *rval* ⟦*Exp*⟧))

Rule *var-declare* ⟦*Id Ranks*⟧ = **bind**(*id* ⟦*Id*⟧, **allocate-nested-vectors**(*ranks* ⟦*Ranks*⟧))

A.4.2. Array declarations

Syntax *Ranks* : ranks ::= '**[**' exps '**]**' ranks$^?$

Rule ⟦'**[**' *Exp* '**,**' *Exps* '**]**' *Ranks*$^?$⟧ : ranks = ⟦'**[**' *Exp* '**]**' '**[**' *Exps* '**]**' *Ranks*$^?$⟧

Semantics *ranks* ⟦_ : ranks⟧ : (⇒**nats**)$^+$

Rule *ranks* ⟦'**[**' *Exp* '**]**'⟧ = *rval* ⟦*Exp*⟧

Rule *ranks* ⟦'**[**' *Exp* '**]**' *Ranks*⟧ = *rval* ⟦*Exp*⟧, *ranks* ⟦*Ranks*⟧

Funcon **allocate-nested-vectors**(_ : **nats**$^+$) : ⇒**variables**

Rule **allocate-nested-vectors**(*N* : **nats**) ⇝

allocate-initialised-variable(**vectors**(**variables**),

vector(**left-to-right-repeat**(**allocate-variable**(**values**), 1, *N*)))

Rule **allocate-nested-vectors**(*N* : **nats**, N^+ : **nats**$^+$) ⇝

allocate-initialised-variable(**vectors**(**variables**),

vector(**left-to-right-repeat**(**allocate-nested-vectors**(N^+), 1, *N*)))

A.4.3. Function declarations

Syntax *FuncDecl* : func-decl ::= '**function**' id '(' ids$^?$ ')' block

Rule *declare* ⟦'**function**' *Id* '(' *Ids*$^?$ ')' *Block*⟧ = **bind**(*id* ⟦*Id*⟧, **allocate-variable**(**functions**(**tuples**(**values***), **values**)))

Semantics *initialise* ⟦_ : decl⟧ : ⇒**null-type**

Rule *initialise* ⟦'**var**' *Declarators* '**;**'⟧ = **null**

Rule *initialise* ⟦'**function**' *Id* '(' *Ids*$^?$ ')' *Block*⟧ =

assign(**bound**(*id* ⟦*Id*⟧), **function**(**closure**(

scope(**match**(**given**, **tuple**(*patts* ⟦*Ids*$^?$⟧)), **handle-return**(*exec* ⟦*Block*⟧)))))

Syntax *Ids* : ids ::= id (‘,’ ids)$^{?}$

Semantics *patts* ⟦_ : ids$^{?}$⟧ : patterns*

Rule *patts* ⟦ ⟧ = ( )

Rule *patts* ⟦*Id*⟧ = **pattern**(**closure**(**bind**(*id* ⟦*Id*⟧, **allocate-initialised-variable**(**values**, **given**))))

Rule *patts* ⟦*Id* ‘,’ *Ids*⟧ = *patts* ⟦*Id*⟧, *patts* ⟦*Ids*⟧

A.5. Programs

Syntax *Pgm* : pgm ::= decls

Semantics *run* ⟦_ : pgm⟧ : ⇒**values**

Rule *run* ⟦*Decls*⟧ = **scope**(**collateral**(*declarations* ⟦*Decls*⟧),
sequential(*initialisations* ⟦*Decls*⟧, **apply**(**assigned**(**bound**(**“main”**)), **tuple**())))

Syntax *Decls* : decls ::= decl decls$^{?}$

Semantics *declarations* ⟦_ : decls⟧ : (⇒**environments**)$^{+}$

Rule *declarations* ⟦*Decl*⟧ = *declare* ⟦*Decl*⟧

Rule *declarations* ⟦*Decl Decls*⟧ = *declare* ⟦*Decl*⟧, *declarations* ⟦*Decls*⟧

Semantics *initialisations* ⟦_ : decls⟧ : (⇒**null-type**)$^{+}$

Rule *initialisations* ⟦*Decl*⟧ = *initialise* ⟦*Decl*⟧

Rule *initialisations* ⟦*Decl Decls*⟧ = *initialise* ⟦*Decl*⟧, *initialisations* ⟦*Decls*⟧

Appendix B. Funcon signatures

This appendix lists the signatures of all funcons and value operations that appear in this article.

B.1. Funcons

abrupt(_ : **values**) : ⇒**empty-type**

allocate-initialised-variable(*T* : **types**, _ : *T*) : ⇒**variables**

allocate-nested-vectors(_ : **nats**$^{+}$) : ⇒**variables**

allocate-variable(_ : **types**) : ⇒**variables**

apply(_ : **functions**(*S*, *T*), _ : *S*) : ⇒*T*

assign(_ : **variables**, _ : **values**) : ⇒**null-type**

assigned(_ : **variables**) : ⇒**values**

bind(_ : **identifiers**, _ : **values**) : ⇒**environments**

bound(_ : **identifiers**) : ⇒**values**

checked(_ : (*T*)$^{?}$) : ⇒*T*

closed(_ : ⇒*T*) : ⇒*T*

closure(_ : *S*⇒*T*) : ⇒**abstractions**(*S*⇒*T*)

collateral(_ : **environments***) : ⇒**environments**

control(_ : **functions**(**continuations**(*T*, *S*), *S*)) : ⇒*T*

effect(_ : *T**) : ⇒**null-type**

delimit-cc(_ : ⇒*T*) : ⇒*T*

fail : $\Rightarrow$**empty-type**

give(_ : S, _ : $S \Rightarrow T$) : $\Rightarrow T$

given : $T \Rightarrow T$

handle-return(_ : $\Rightarrow T$) : $\Rightarrow T$

handle-thrown(_ : $S \Rightarrow T$, _ : $X \Rightarrow T$) : $S \Rightarrow T$

hole : $\Rightarrow$**values**

if-true-else(_ : **booleans**, _ : $\Rightarrow T$, _ : $\Rightarrow T$) : $\Rightarrow T$

left-to-right-repeat(_ : **integers**$\Rightarrow T$, _ : **integers**, _ : **integers**) : $\Rightarrow (T)^*$

match(_ : **values**, _ : **values**) : $\Rightarrow$**environments**

print(_ : **values***) : $\Rightarrow$**null-type**

read : $\Rightarrow$**values**

resume-continuation(_ : **continuations**(S, T), _ : S) : $\Rightarrow T$

return(_ : T) : $\Rightarrow$**empty-type**

scope(_ : **environments**, _ : $\Rightarrow T$) : $\Rightarrow T$

sequential(_ : ($\Rightarrow$**null-type**)*, $\Rightarrow T$) : $\Rightarrow T$

stuck : $\Rightarrow$**empty-type**

throw(_ : T) : $\Rightarrow$**empty-type**

while(_ : $\Rightarrow$**booleans**, _ : $\Rightarrow$**null-type**) : $\Rightarrow$**null-type**

B.2. Value operations

abstraction(_ : $S \Rightarrow T$) : **abstractions**$(S \Rightarrow T)$

continuation(_ : **abstractions**$(() \Rightarrow T)$) : **continuations**(S, T)

function(_ : **abstractions**$(S \Rightarrow T)$) : **functions**(S, T)

index(_ : **nats**, _ : **values***) : $\Rightarrow$**values**$^?$

integer-add(_ : **integers***) : $\Rightarrow$**integers**

integer-divide(_ : **integers**, _ : **integers**) : $\Rightarrow$**integers**$^?$

integer-modulo(_ : **integers**, _ : **integers**) : $\Rightarrow$**integers**$^?$

integer-multiply(_ : **integers***) : $\Rightarrow$**integers**

integer-negate(_ : **integers**) : $\Rightarrow$**integers**

integer-subtract(_ : **integers**, _ : **integers**) : $\Rightarrow$**integers**

is-in-set(_ : T, _ : **sets**(T)) : $\Rightarrow$**booleans**

is-in-type(_ : **values**, _ : **types**) : $\Rightarrow$**booleans**

is-less(_ : **integers**, _ : **integers**) : $\Rightarrow$**booleans**

is-less-or-equal(_ : **integers**, _ : **integers**) : $\Rightarrow$**booleans**

is-greater(_ : **integers**, _ : **integers**) : $\Rightarrow$**booleans**

is-greater-or-equal(_ : **integers**, _ : **integers**) : $\Rightarrow$**booleans**

is-equal(_ : **values**, _ : **values**) : $\Rightarrow$**booleans**

length(_ : **values***) : $\Rightarrow$**nats**

lookup(_ : **maps**$(S, T^?)$, _ : S) : $\Rightarrow (T^?)^?$

map-override(_ : **maps**$(S, T)^*$) : $\Rightarrow$**maps**(S, T)

not(_ : **booleans**) : ⇒**booleans**

null : **null-type**

pattern(_ : **abstractions**(**values**⇒**environments**)) : **patterns**

returned(_ : **values**) : **returning**

thrown(_ : **values**) : **throwing**

tuple(_ : T^*) : **tuples**(T^*)

variable(_ : **locations**, _ : **types**) : **variables**

vector(_ : $(T)^*$) : **vectors**(T)

vector-elements(_ : **vectors**(T)) : ⇒$(T)^*$

References

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

[2] P. Hudak, J. Hughes, S.L. Peyton Jones, P. Wadler, A history of Haskell: being lazy with class, in: Third Conference on History of Programming Languages, ACM, 2007, pp. 12:1–12:55.

[3] P.D. Mosses, M.J. New, Implicit propagation in structural operational semantics, in: Fifth Workshop on Structural Operational Semantics, in: Electronic Notes in Theoretical Computer Science, vol. 229(4), Elsevier, 2009, pp. 49–66.

[4] P.D. Mosses, Modular structural operational semantics, J. Log. Algebraic Methods Program. 60–61 (2004) 195–228, https://doi.org/10.1016/j.jlap.2004.03.008.

[5] M. Churchill, P.D. Mosses, N. Sculthorpe, P. Torrini, Reusable components of semantic specifications, in: Transactions on Aspect-Oriented Software Development XII, in: Lecture Notes in Computer Science, vol. 8989, Springer, 2015, pp. 132–179.

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

[7] PLanCompS project, CBS and funcons beta release, GitHub, Online, https://plancomps.github.io/CBS-beta, 2018. (Accessed 17 December 2018).

[8] L.C.L. Kats, E. Visser, The Spoofax language workbench: rules for declarative specification of languages and IDEs, in: International Conference on Object Oriented Programming Systems Languages and Applications, ACM, 2010, pp. 444–463.

[9] G. Roşu, T.F. Şerbănuţă, $\mathbb{K}$ overview and SIMPLE case study, Electronic Notes in Theoretical Computer Science 304 (2014) 3–56, https://doi.org/10.1016/j.entcs.2014.05.002.

[10] G.D. Plotkin, A structural approach to operational semantics, J. Log. Algebraic Methods Program. 60–61 (2004) 17–139, https://doi.org/10.1016/j.jlap.2004.05.001, reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981.

[11] M. Churchill, P.D. Mosses, Modular bisimulation theory for computations and values, in: 16th International Conference on Foundations of Software Science and Computation Structures, in: Lecture Notes in Computer Science, vol. 7794, Springer, 2013, pp. 97–112.

[12] P.D. Mosses, F. Vesely, Weak bisimulation as a congruence in MSOS, in: Logic, Rewriting, and Concurrency, in: Lecture Notes in Computer Science, vol. 9200, Springer, 2015, pp. 519–538.

[13] F. Chalub, C. Braga, Maude MSOS tool, Electronic Notes in Theoretical Computer Science 176 (4) (2007) 133–146, https://doi.org/10.1016/j.entcs.2007.06.012.

[14] P.D. Mosses, Teaching semantics of programming languages with Modular SOS, in: Teaching Formal Methods 2006: Practice and Experience, Electronic Workshops in Computing, BCS, 2006, pp. 275–286, https://ewic.bcs.org/content/ConWebDoc/9093.

[15] 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.

[16] R. Farahbod, V. Gervasi, U. Glässer, CoreASM: an extensible ASM execution engine, Fundam. Inform. 77 (1–2) (2007) 71–103.

[17] P. Sewell, F.Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, R. Strniša, Ott: effective tool support for the working semanticist, J. Funct. Program. 20 (1) (2010) 71–122, https://doi.org/10.1017/S0956796809990293.

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

[19] G. Roşu, T.F. Şerbănuţă, An overview of the K semantic framework, J. Log. Algebraic Methods Program. 79 (6) (2010) 397–434, https://doi.org/10.1016/j.jlap.2010.03.012.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J.F. Quesada, Maude: specification and programming in rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 187–243, https://doi.org/10.1016/S0304-3975(01)00359-0.

[21] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: a meta-language for modular and reusable development of DSLs, in: Eighth International Conference on Software Language Engineering, ACM, 2015, pp. 25–36.

[22] V. Vergu, P. Neron, E. Visser, DynSem: a DSL for dynamic semantics specification, in: 26th International Conference on Rewriting Techniques and Applications, in: Leibniz International Proceedings in Informatics, vol. 36, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 365–378.

[23] P.D. Mosses, F. Vesely, FunKons: component-based semantics in K, in: 10th International Workshop on Rewriting Logic and Its Applications, in: Lecture Notes in Computer Science, vol. 8663, Springer, 2014, pp. 213–229.

[24] P.D. Mosses, A Modular SOS for ML Concurrency Primitives, BRICS Research Series RS-99-57. Department of Computer Science, Aarhus University, 1999, http://www.brics.dk/RS/99/57/.

[25] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, C. Strachey, The main features of CPL, Comput. J. 6 (2) (1963) 134–143, https://doi.org/10.1093/comjnl/6.2.134.

[26] C. Strachey, Fundamental concepts in programming languages, High.-Order Symb. Comput. 13 (1) (2000) 11–49, https://doi.org/10.1023/A:1010000313106.

[27] P.D. Mosses, Pragmatics of modular SOS, in: International Conference on Algebraic Methodology and Software Technology, in: Lecture Notes in Computer Science, vol. 2422, Springer, 2002, pp. 21–40.

[28] O. Danvy, A. Filinski, A Functional Abstraction of Typed Contexts, Tech. Rep. 89/12, DIKU, University of Copenhagen, 1989, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.4822.

[29] M. Felleisen, The theory and practice of first-class prompts, in: 15th Symposium on Principles of Programming Languages, ACM, 1988, pp. 180–190.

[30] N. Sculthorpe, P. Torrini, P.D. Mosses, A modular structural operational semantics for delimited continuations, in: Post-Proceedings of the 2015 Workshop on Continuations, in: Electronic Proceedings in Theoretical Computer Science, vol. 212, Open Publishing Association, 2016, pp. 63–80.

[31] L.T. van Binsbergen, N. Sculthorpe, The Haskell funcon framework, Hackage, Online, https://hackage.haskell.org/package/funcons-tools-0.2.0.5, 2018. (Accessed 17 December 2018).
[32] W. Swierstra, Data types à la carte, J. Funct. Program. 18 (4) (2008) 423–436, https://doi.org/10.1017/S0956796808006758.
[33] L.E. Day, G. Hutton, Towards modular compilers for effects, in: 12th International Symposium on Trends in Functional Programming, in: Lecture Notes in Computer Science, vol. 7193, Springer, 2012, pp. 49–64.
[34] N. Wu, T. Schrijvers, R. Hinze, Effect handlers in scope, in: 2014 Symposium on Haskell, ACM, 2014, pp. 1–12.
[35] M.P. Jones, Functional programming with overloading and higher-order polymorphism, in: First International School on Advanced Functional Programming, in: Lecture Notes in Computer Science, vol. 925, Springer, 1995, pp. 97–136.
[36] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: 22nd Symposium on Principles of Programming Languages, ACM, 1995, pp. 333–343.
[37] S.L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice Hall, 1987.
[38] O. Danvy, 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/.
[39] C. Bach Poulsen, Extensible Transition System Semantics, Ph.D. thesis, Swansea University, 2016.
[40] L.T. van Binsbergen, N. Sculthorpe, Funcons values and value operations, Hackage, Online, https://hackage.haskell.org/package/funcons-values-0.1.0.3, 2018. (Accessed 17 December 2018).
[41] L.T. van Binsbergen, N. Sculthorpe, Funcon interpreter for the SIMPLE language, Hackage, Online, https://hackage.haskell.org/package/funcons-simple-0.1.0.3, 2018. (Accessed 17 December 2018).
[42] The $\mathbb{K}$ framework, GitHub, Online, https://github.com/kframework/k5, 2018. (Accessed 17 December 2018).
[43] H. van Antwerpen, P. Néron, A.P. Tolmach, E. Visser, G. Wachsmuth, A constraint language for static semantic analysis based on scope graphs, in: 2016 Workshop on Partial Evaluation and Program Manipulation, ACM, 2016, pp. 49–60.
[44] P. Néron, A.P. Tolmach, E. Visser, G. Wachsmuth, A theory of name resolution, in: 24th European Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 9032, Springer, 2015, pp. 205–231.