Executable Component-Based Semantics

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

[a] *Department of Computer Science, Royal Holloway University of London, UK*
[b] *Department of Computer Science, Swansea University, UK*

Abstract

The potential benefits of formal semantics are well known. However, it requires a lot of work to produce a complete and accurate formal semantics for a major language; and when the language evolves, large-scale revision of the semantics may be needed to reflect the changes. The investment of effort needed to produce an initial definition, and subsequently to revise it, 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. CBS 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. This paper gives an introduction to CBS, illustrates its use, and presents the various tools involved in our implementation of CBS.

Keywords: programming languages, formal semantics, reuse, components, tool support

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. Informal specifications are often incomplete or inconsistent, and open to misinterpretation; formal specifications can avoid such issues. Moreover, completely formal definitions of programming languages may be used to generate prototype implementations, and as a basis for proving properties of languages and of individual programs.

Although there is broad agreement as to the benefits of formality in language definitions, and although there are a few examples of successful individual projects (notably the definition of Standard ML [1]), there is generally little inclination on the part of programming language developers themselves to use formal semantics. For instance, even Haskell, a language designed with an emphasis on its mathematical structure, does not have a formal semantics [2]. 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.

Moreover, 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] decided to base them on a collection of reusable components. In the PLanCompS approach, a reusable component of language definitions corresponds to a *fundamental programming construct*: a so-called '*funcon*', which has a fixed operational interpretation. The formal semantics of each funcon is defined independently, using I-MSOS [3], a variant of modular structural operational semantics [4]. For example, the funcon **if-then-else** is defined as follows:

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

Rule **if-then-else**(**true**, X, _) = X

Rule **if-then-else**(**false**, _, Y) = Y

The collection of funcons is open-ended; crucially, adding new funcons never requires changes to the definition or use of previous funcons.

[1] `http:www.plancomps.org`

A component-based semantics of a programming language is defined by translating its constructs to funcons. For example, consider the following translation of non-strict conjunction for a language where its operands are Boolean-valued:

Rule $\mathit{rexp}\,[\![\mathit{Exp}_1\ \text{'\&\&'}\ \mathit{Exp}_2]\!] = \textbf{if-then-else}(\mathit{rexp}\,[\![\mathit{Exp}_1]\!], \mathit{rexp}\,[\![\mathit{Exp}_2]\!], \textbf{false})$

Many funcons can be widely reused in the definitions of different languages. An initial medium-scale case study [5] gave a semantics for CAML LIGHT [6] based on a preliminary collection of funcons; after completion of a major case study (C♯), the validated funcons are to be finalised and made freely available in a digital library, for reuse in future language definitions.

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 "Exp_1 '&&' Exp_2" would need changing if conjunction were to become strict, or if its arguments were no longer required to be Booleans, but the definition of the **if-then-else** funcon would not 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. The CAML LIGHT case study originally used a mixture of meta-languages: the syntax of CAML LIGHT was defined in SDF3 [7]; its translation to funcons was defined by transformation rules written in STRATEGO [8]; and the semantics of funcons was defined in CSF (a plain-text version of I-MSOS [3], enriched with rules for the value-computation transition systems introduced in [9]). Use of CBS provides notational uniformity, and significantly improves the conciseness of component-based semantics.

In Sect. 2 we show how to use CBS to define abstract syntax and translation of abstract syntax trees to funcons. For illustration, we take constructs from SIMPLE, a small C-like imperative programming language [10]. Their translation involves funcons that are likely to be reused in definitions of most imperative languages. Appendix A provides a complete CBS definition of SIMPLE (omitting concurrency constructs).

In Sect. 3 we show the definitions of the funcons used in the translations of the SIMPLE constructs considered in Sect. 2. Appendix B lists the signatures of all the funcons used in Appendix A; we provide their CBS definitions in the supplementary material accompanying this paper [11].

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, and observe the behaviour of programs when run on it. 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. Our tools generate SDF3 and Stratego code from the CBS definition of a translation from ASTs to funcon terms, which is then built as a Spoofax editor. Using a parser for the defined language, the generated editor can then be used to compile programs to funcon terms. From the CBS definitions of the operational semantics of the individual funcons used in the translation, our tools generate Haskell code that can be imported by our Haskell-embedded funcon interpreter. Combining these two generated artefacts with a parser for programs, we obtain an interpreter for programs written in the defined language.

In Sect. 5 we explain our technique for writing funcon interpreters that preserve the modularity of funcon definitions in CBS. In Sect. 6 we discuss how we compile CBS funcon definitions to modular Haskell code that can be executed by our Haskell-embedded funcon interpreter.

1.3. Integrated Development Environment for CBS

We have implemented an IDE to support development and use of CBS. The IDE includes a CBS editor with many useful features, including syntax highlighting, syntax error recovery, hyperlinks from uses of symbols to their definitions, and flagging of undefined symbols. The CBS editor can be used to browse, navigate and update language definitions and funcon definitions. To support development and validation of definitions in CBS, we provide a complete tool chain within Eclipse for executing translation functions and running the resulting funcon terms. In Sect. 4 we illustrate the use of the IDE.

1.4. Related Work

In Sect. 7 we discuss current frameworks for formal semantics that have a high degree of modularity as well as executable definitions.

2. 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 [10] as a pedagogical language for demonstrating the K Semantic Framework [12]. Roşu and Şerbănuţă define two variants of SIMPLE, one typed and one untyped; here we consider the untyped variant.

2.1. Abstract Syntax Definitions

Our CBS definition of the abstract syntax of SIMPLE is presented in Figure 1. We typeset non-terminal symbols in sans-serif, terminal symbols in **bold**, meta-variables in *Capitalised Italic*, and keywords in *Grey Capitalised Italic*. CBS uses concrete syntax, enclosed in single quotation marks, for terminal symbols. However, CBS translation functions are defined over abstract syntax trees; to support this, each non-terminal may be declared with an associated meta-variable that can be used in translation equations to match phrases of that non-terminal sort.

CBS supports iteration of symbols in grammars directly, with the suffixes '$^?$', '*' and '$^+$' denoting 'zero or one', 'zero or more', and 'one or more' iterations, respectively. These suffixes may be applied to multiple symbols at once, by using parentheses for grouping.

2.2. Translation Function Signatures

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 **values** is declared as follows:

Semantics *val* ⟦_ : value⟧ : **values** (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*). In the grammar we have grouped the subset of expressions that are valid l-expressions into a separate non-terminal named lexp. An l-expression should evaluate to a *variable*, whereas an r-expression, if it evaluates a variable, should result in the value currently assigned to that variable. We therefore

```
Syntax  Prog  : program               ::= global-declaration+
Syntax  GDecl : global-declaration    ::= 'function' id '(' ids? ')' block
                                        | variable-declaration
Syntax  Ids   : ids                   ::= id (',' id)*
Syntax  VDecl : variable-declaration  ::= 'var' declarator (',' declarator)* ';'
Syntax  Decl  : declarator            ::= id | id '=' exp | id ranks
Syntax  Ranks : ranks                 ::= '[' exps ']' ranks?
Syntax  Block : block                 ::= '{' stmt* '}'
Syntax  Stmt  : stmt                  ::= variable-declaration
                                        | 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 ';'
Syntax  Exp   : exp                   ::= '(' exp ')'
                                        | value
                                        | '++' lexp
                                        | lexp
                                        | exp '(' exps? ')'
                                        | '-' exp
                                        | '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
                                        | lexp '=' exp
Syntax  LExp  : lexp                  ::= id | lexp '[' exps ']'
Syntax  Exps  : exps                  ::= exp (',' exp)*
Syntax  V     : value                 ::= bool | int | string
```

Figure 1: CBS specification of SIMPLE abstract syntax.

require two translation functions for specifying the semantics of expressions, with differing signatures:

Semantics *lexp* ⟦_ : lexp⟧ : ⇒**variables**(**values**) (2)

Semantics *rexp* ⟦_ : exp⟧ : ⇒**values** (3)

Note that the funcon type **variables** takes as a parameter the type of values the variable can store, which in this case is arbitrary **values**.

Observe that the result sort in these signatures is prefixed with the symbol '⇒'. This is pronounced "computes", and means that the resulting funcon term denotes a *computation* that will evaluate to a value. In general, computations may perform arbitrary side-effects, such as input/output, throwing exceptions or mutating a store.

The '⇒' notation was inspired by the Scala notation for call-by-name parameters; the similarity with call-by-name parameter passing will become more apparent when we consider funcon specifications in Sect. 3. It can also be viewed as somewhat analogous to the monadic *IO* type in a pure functional language such as Haskell.

2.3. Translation Equations

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. Here are some example equations of the *rexp* function:

Rule *rexp* ⟦'**!**' *Exp*⟧ = **not**(*rexp* ⟦*Exp*⟧) (4)

Rule *rexp* ⟦Exp_1 '+' Exp_2⟧ = **integer-add**(*rexp* ⟦Exp_1⟧, *rexp* ⟦Exp_2⟧) (5)

Rule *rexp* ⟦'**read**' '(' ')'⟧ = **read** (6)

The meta-variable *Exp* was declared with the non-terminal exp (Figure 1), and consequently ranges over exp phrases. Meta-variables of the same sort may be given a numeric suffix to differentiate between them, as in Eq. 5.

Using meta-variables that range over abstract syntax allows us to restrict the left-hand side of an equation without using concrete syntax. For example, the translation of an lexp phrase that occurs in an r-expression is as follows:

Rule *rexp* ⟦*LExp*⟧ = **assigned**(*lexp* ⟦*LExp*⟧) (7)

The equations of the *lexp* function are as follows:

Rule *lexp* ⟦*Id*⟧ = **bound**(*id* ⟦*Id*⟧) (8)

Rule *lexp* ⟦*LExp* '[' *Exp* ']'⟧ = **vector-index**(*rexp* ⟦*LExp*⟧, *rexp* ⟦*Exp*⟧) (9)

2.4. Sequences

Grammars can contain iterated phrases, as expressed by the $^?$, * and $^+$ suffixes on symbols (or groups of symbols). Translation functions may also produce sequences of funcon terms, which can be expressed by annotating the funcon sort with the appropriate iteration suffix. For example, the following function translates non-empty comma-separated sequences of SIMPLE expressions (the exps non-terminal) into non-empty sequences of funcon terms:

$$\textit{Semantics}\ \textit{rexps}\,[\![\_ : \mathsf{exps}]\!] : (\Rightarrow \textbf{values})^+ \tag{10}$$

$$\textit{Rule}\ \textit{rexps}\,[\![\textit{Exp}]\!] = \textit{rexp}\,[\![\textit{Exp}]\!] \tag{11}$$

$$\textit{Rule}\ \textit{rexps}\,[\![\textit{Exp}_1\ \text{`,'}\ \textit{Exp}_2\ \ldots]\!] = \textit{rexp}\,[\![\textit{Exp}_1]\!], \textit{rexps}\,[\![\textit{Exp}_2\ \ldots]\!] \tag{12}$$

CBS uses the '...' notation *formally* as a meta-variable that ranges over any iterated sequence of phrases.

Sequences of funcon terms can be combined using parentheses in the enclosing translation function. CBS supports several forms of parentheses, each corresponding to a particular funcon collection type ('[ ]' for lists, '( )' for tuples, '{ }' for sets). For example, the translation of SIMPLE's function application combines the argument expressions into a tuple:

$$\textit{Rule}\ \textit{rexp}\,[\![\textit{Exp}\ \text{`(`}\ \textit{Exps}\ \text{`)'}]\!] = \textbf{apply}(\textit{rexp}\,[\![\textit{Exp}]\!], (\textit{rexps}\,[\![\textit{Exps}]\!])) \tag{13}$$

The signature of a translation function can also explicitly state that it takes a sequence of arguments. For example, the function *stmts* takes a non-empty sequence of SIMPLE statements as its arguments:

$$\textit{Semantics}\ \textit{stmts}\,[\![\_ : \mathsf{stmt}^+]\!] : \Rightarrow() \tag{14}$$

Note that SIMPLE statements are commands that are executed for their side-effects—they do not compute a result value. However, in the funcon framework, all funcons sorts indicate the type of value they compute. The funcon convention is to model commands as computations that evaluate to the empty tuple, the type of which appears in the result sort of *stmts* here.

As an alternative to the "..." notation, meta-variables may be suffixed with an iteration symbol to have that meta-variable range over corresponding iterated phrases of the same sort as the meta-variable. For example, the translation of a declaration followed by one or more statements is specified as follows:

$$\textit{Rule}\ \textit{stmts}\,[\![\text{`}\textbf{var}\text{'}\ \textit{Decl}\ \text{`;'}\ \textit{Stmt}^+]\!] = \textbf{scope}(\textit{decl}\,[\![\textit{Decl}]\!], \textit{stmts}\,[\![\textit{Stmt}^+]\!]) \tag{15}$$

When a sequence starts with a statement other than a declaration, that statement should be executed before the rest of the sequence. We would like to define this once, rather than giving a separate rule for each other form of statement. This is achieved using *Otherwise* equations, which allow overlapping equations to be declared, giving the *Otherwise* equation lower priority. For example, we translate all other sequences of at least two statements as follows:

$$\textit{Otherwise}\ \textit{stmts}\,[\![\textit{Stmt}\ \textit{Stmt}^+]\!] = \textbf{sequential}(\textit{stmts}\,[\![\textit{Stmt}]\!], \textit{stmts}\,[\![\textit{Stmt}^+]\!]) \tag{16}$$

The translations of other forms of statement can then be given in isolation, for example:

Rule *stmts*⟦**'if'** **'('** *Exp* **')'** $Block_1$ **'else'** $Block_2$⟧ =
if-then-else(*rexp*⟦*Exp*⟧, *block*⟦$Block_1$⟧, *block*⟦$Block_2$⟧) (17)

Rule *stmts*⟦**'while'** **'('** *Exp* **')'** *Block*⟧ = **while**(*rexp*⟦*Exp*⟧, *block*⟦*Block*⟧) (18)

Rule *stmts*⟦**'print'** **'('** *Exps* **')'** **';'**⟧ = **print-list**[*rexps*⟦*Exps*⟧] (19)

Rule *stmts*⟦**'throw'** *Exp* **';'**⟧ = **throw**(*rexp*⟦*Exp*⟧) (20)

2.5. Desugaring

Programming languages often contain constructs that are intended to be understood as abbreviations for other constructs in the language; so-called "syntactic sugar". While we could translate the sugared syntax directly to funcons (likely involving some overlap of specification), CBS also allows *desugaring equations* to be defined.

For example, in a SIMPLE conditional statement, the **else** clause is optional. However, Eq. 17 only handles the case when it is present. Rather than add another translation equation for the absent case, we instead define a desugaring equation that inserts an **else** clause containing an empty block:

Rule ⟦**'if'** **'('** *Exp* **')'** *Block*⟧ : stmt = ⟦**'if'** **'('** *Exp* **')'** *Block* **'else'** **'{'** **'}'**⟧ (21)

Unlike translation equations, desugaring equations are not named. Whereas translation functions are explicitly applied in translation equations, desugaring equations are instead implicitly applied before every translation equation (if applicable) when executing a translation, and hence naming them is redundant.

To give a more interesting example, a SIMPLE **for** loop can be desugared into a SIMPLE **while** loop as follows:

Rule ⟦**'for'** **'('** *Stmt* Exp_1 **';'** Exp_2 **')'** **'{'** $Stmt^*$ **'}'**⟧ : stmt =
⟦**'{'** *Stmt* **'while'** **'('** Exp_1 **')'** **'{'** $Stmt^*$ Exp_2 **';'** **'}'** **'}'**⟧ (22)

Our use of desugaring equations here mirrors the K specification of SIMPLE [10], which uses analogous "desugaring macros" to achieve the same effect.

2.6. Readability

The equations presented so far have translated each SIMPLE construct directly into a corresponding funcon. Indeed, translating many 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:

Semantics *decl*⟦_ : declarator⟧ : ⇒**environments** (23)

Rule *decl*⟦*Id*⟧ = **bind**(*id*⟦*Id*⟧, **allocate-variable**(**values**)) (24)

Rule *decl*⟦*Id* **'='** *Exp*⟧ =
bind(*id*⟦*Id*⟧, **allocate-initialised-variable**(**values**, *rexp*⟦*Exp*⟧)) (25)

One of our aims is that the name of a funcon should be sufficiently indicative of its semantics that, in most cases, a reader should be able to understand the gist of a translation without referring to the funcon definitions. In this case, Eq. 24 says that a variable for storing arbitrary **values** (as this is the untyped variant of SIMPLE) should be allocated, and bound to the identifier. Eq. 25 is similar, except that the allocated variable should be initialised with the result of the expression.

2.7. 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 K specification [10, page 27]. 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 is to have their semantic rule for array declarators generate SIMPLE code that uses a **for** loop to iteratively allocate each sub-array.

We cannot take this approach in the funcon framework. The funcons are reusable components, specified independently of any particular source language—funcon definitions cannot refer to source-language syntax or invoke translation functions. However, CBS provides an alternative: local funcons can be specified within a language definition. This facility is for funcons that are not considered reusable enough to be part of the funcon library, but nonetheless capture a useful concept of the language being defined. 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:

$$\textit{Rule}\ \textit{decl}[\![\textit{Id}\ \textit{Ranks}]\!] = \\ \textbf{bind}(\textit{id}[\![\textit{Id}]\!], \textbf{allocate-nested-vectors}[\textit{ranks}[\![\textit{Ranks}]\!]]) \tag{26}$$

The definition of **allocate-nested-vectors** is included in Appendix A.4.

2.8. Remarks

This section has presented representative extracts from our CBS definition of SIMPLE. Our full definition is listed in Appendix A. For more detailed discussion and examples of the translation of programming languages to funcons (though not using the CBS language), see [5] or [13].

Our definition of SIMPLE does not include the concurrent aspects of the language, because the funcon library does not yet contain funcons expressing concurrent behaviour. This is not due to any fundamental limitation of (I-)MSOS-based specifications—indeed, one author of this article has previously used MSOS to specify a sublanguage of Concurrent ML [14]—rather it is that developing the funcon library is ongoing work.

3. Specifying Fundamental Constructs in CBS

Structural operational semantics (SOS) [15] 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. [15]).

3.1. Transition Relations

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

$$\textit{Rule}\ \frac{B \longrightarrow B'}{\textbf{if-then-else}(B, X, Y) \longrightarrow \textbf{if-then-else}(B', X, Y)} \tag{27}$$

$$\textit{Rule}\ \textbf{if-then-else}(\textbf{true}, X, \_) = X \tag{28}$$

$$\textit{Rule}\ \textbf{if-then-else}(\textbf{false}, \_, Y) = Y \tag{29}$$

This is a fairly conventional SOS specification of a conditional expression, except that there are two relations: '$\longrightarrow$' and '='. The former is a transition relation denoting a step of computation, which in general may have arbitrary side effects and be sensitive to its context. The latter is a transition relation denoting context-free term rewriting, which cannot involve side effects. (Note that this use of the symbol '=' is unrelated to our use of '=' in translation equations.) The formal details of these two relations, and the connection between them, are spelled out in [9][2]; here we shall just give an informal overview.

Implicitly, a computation step may be preceded or followed by any number of rewrites. Rewriting is reflexive and transitive, and is a pre-congruence for all funcon terms. Considering the rules operationally, this means that a rewrite can be applied at any time during execution of a funcon term, anywhere within the term. However, the CBS rule format (following [9]) ensures that any such transition is not observable.

Inference rules for either relation may have any number of premises. However, because we use small-step rules to specify computation steps, it is rare for a funcon rule to have more than one computation-step premise. It is fairly common to have multiple rewrite premises.

The two relations may only appear in the same inference rule if the conclusion of the rule is a computation step. Formally, any rewrites that appear among the premises of such a rule can be understood as side conditions on the rule; if the only premises of a computation-step rule are rewrites, then the rule is considered to be an axiom of the computation-step relation.

[2] In [9] the relations '=' and '$\longrightarrow$' were written '$\Rightarrow$' and '$\rightarrow$', respectively.

3.2. Funcon Signatures

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

$$\textit{Funcon}\ \textbf{if-then-else}(\_ : \textbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T \tag{30}$$

Funcon signatures are similar to the signatures of translation functions, except that the arguments have funcon sorts. As with the results of translation functions, a funcon sort is either a *value sort* (a type), or a *computation sort* (a type prefixed by the '⇒' symbol). Additionally, polymorphic funcons may have signatures that contain meta-variables in place of types[3]. In the case of **if-then-else**, the meta-variable T is used to express that the second and third arguments compute values of the same type, and the funcon as a whole also computes a value of that type.

Notice that the first argument is value sort. This does not mean that **if-then-else** is limited to being applied to literal Boolean values; rather, value-sort annotations play a special role in funcon signatures.

When an argument has a value sort, we say the funcon is *strict* in that argument. When an argument has a computation sort, we say the funcon is *non-strict* in that argument. If a funcon is strict in all its arguments, then we say that the funcon is strict. If a funcon has any non-strict arguments, then we say that the funcon is non-strict.

For each *strict* argument 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, and the target of rule's conclusion is the same as the source of the rule, except that the argument subterm is replaced with the updated subterm in the target of the premise. For example, Rule 27 is the congruence rule implicitly generated by the signature of **if-then-else** (30), and thus that rule can be omitted. This technique for automatically generating congruence rules was inspired by the strictness annotations in the K Framework [10, page 9].

Each signature containing strict arguments implicitly gives rise to a *lifted* signature, in which each value sort is replaced by the corresponding computation sort. This lifted signature is essentially the "real" signature of the funcon, which can be used for checking well-formedness of funcon terms.

3.3. Values and Types

The CBS language provides a rich suite of primitive and composite values, and operations on those values. Table 1 presents a selection of the value types available. 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.

[3] Meta variables range over types, not sorts. Thus an argument in a signature of the form '$\_ : T$' has a value sort (for some type T), not an arbitrary computation sort.

CBS Type	Represented Values
algebraic-datatypes	Algebraic data types.
booleans	Booleans.
binders	String identifiers, optionally annotated with a namespace.
bits(N : **naturals**)	Fixed-width bit vectors of width N.
environments	Maps from binders to values.
ieee-floats(_ : **ieee-formats**)	IEEE floating-point numbers in various formats.
integers	Integers.
links(T : **types**)	Write-once links to values of type T.
lists(T : **types**)	Finite lists with elements of type T.
maps(S : **types**, T : **types**)	Finite maps with keys of type S and elements of type T.
multisets(T : **types**)	Finite multi-sets with elements of type T.
naturals	Natural numbers.
pointers(T : **types**)	Pointers are either null, or refer to a value of type T.
rationals	Rational numbers.
sets(T : **types**)	Finite sets with elements of type T.
stores	Maps from variables to values.
strings	Strings of Unicode characters.
thunks(_ : **sorts**)	An unevaluated funcon term wrapped as a value.
tuples(T^* : **types***)	Tuples of any finite size, with elements of types T^*.
types	The names of all CBS value types.
unicode-characters	Unicode characters.
variables(T : **types**)	Imperative variables for storing values of type T.
vectors(T : **types**)	Finite vectors with elements of type T.

Table 1: A selection of CBS value types.

The names of types are themselves funcon values, and belong to the type **types**. All values, whether types or not, are members of the type **values**, which is frequently used when specifying funcons that can take arbitrary values as arguments. New types can be defined as synonyms for combinations of existing types; for example, **environments** is a synonym for **maps**(**binders**, **values**). Additionally, new data types can be introduced as algebraic data types. For example, the data type **booleans** is defined as follows:

$$\textit{Datatype}\ \textbf{booleans}\ ::=\ \textbf{false} \mid \textbf{true} \tag{31}$$

The constructors of an algebraic data types may also have arguments, in which case the constructor gives rise to a strict funcon with the implicit congruence rules as its only evaluation rules.

When specifying a funcon, it may be necessary to restrict a rule to be only applicable when one or more of its arguments is a value. This can be achieved in CBS by annotating the argument with ": **values**". More generally, any type can be written instead of **values**, if it is required to restrict the rule to only certain types of value.

Alternatively, instead of writing a meta-variable as the argument of a funcon in a rule, a *pattern* can be written instead. CBS patterns are inspired by pattern matching from functional programming; a pattern may be a literal value, a

constructor of an algebraic data type (with argument patterns if the constructor has arguments), or a tuple or list pattern. For example, rules 28 and 29 have an algebraic data type constructor for the first argument, restricting the rule to the argument values **true** and **false**, respectively.

Following [9], CBS does *not* permit patterns to include non-value funcons. In practice, this means 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. Furthermore, CBS restricts the format in [9] such that any pattern implicitly requires the argument to be a fully evaluated value; consequently a pattern containing the constructor of an algebraic data type implicitly requires all arguments of that constructor to be values. If a form of lazy data structures are desired, then the specifier should explicitly use thunks (see Sect. 3.5).

3.4. Semantic Entities

Formal specifications of programming-language constructs typically make use of auxiliary *semantic entities*, such as environments, stores or emitted signals. In an SOS specification, the presence of such semantic entities is part of the meta-syntax of the relations, and thus they must appear in every transition formula of that relation. For example, in a conventional SOS specification, Rule 27 could well have been written as follows:

$$\frac{\rho \vdash \langle B\ , \sigma \rangle \xrightarrow{\alpha} \langle B'\ , \sigma' \rangle}{\rho \vdash \langle \textsf{if-then-else}(B, X, Y)\ , \sigma \rangle \xrightarrow{\alpha} \langle \textsf{if-then-else}(B', X, Y)\ , \sigma' \rangle} \tag{32}$$

ρ, σ, σ' and α are all meta-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 any signal that is being emitted. Observe that both the environment and signal are the same in both the premise and conclusion, and otherwise are not referred to in the rule. This would typically be understood as propagating the environment unmodified from the conclusion to the premise, and propagating the signal unmodified from the premise to the conclusion (usually, environments are inherited inwards, while signals are emitted outwards). 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 entities is common to many rules in a typical SOS specification. Only a small number of language constructs will typically directly interact with an entity, either to access or modify it. 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, following the systematic propagation scheme described above. This allows entities that do not directly interact with the programming construct being specified to be omitted

from the rule. Thus, rather than Rule 32, in CBS we are able to write the much more concise Rule 27.

Moreover, in addition to the notational convenience, this facility brings a key modularity benefit: new semantic entities can later be added, without requiring any change to the definitions of existing funcons. This high degree of modularity is what allows the funcon repository to be open-ended, while still ensuring that once a funcon is added to the repository, its definition need never be updated.

To allow semantic entities to be implicitly propagated, we must classify them according to the way that they are propagated. CBS supports five classes of entity, each with its own syntax and system of implicit propagation: *inherited*, *mutable*, *input*, *output* and *control-flow*.[4] Because there can be multiple entities of each class, each entity is tagged with a name to distinguish it from any other entity of the same class. The syntax for a CBS rule using one entity of each of the five classes is as follows (writing the classification names as placeholders for the actual entity names):

$$\textit{Rule}\ \mathsf{inherited}(\_) \vdash \langle \_ \,, \mathsf{mutable}(\_) \rangle \xrightarrow{\mathsf{input?}(\_)\ \ \mathsf{control\text{-}flow}(\_)\ \ \mathsf{output!}(\_)} \langle \_ \,, \mathsf{mutable}(\_) \rangle \qquad (33)$$

If multiple entities of the same class are needed in a rule, they can be written in sequence in the appropriate position. However, note that funcons are designed to represent small self-contained programming concepts, and consequently it is rare for the definition of a funcon to involve more than one semantic entity.

We will now present examples of each class of semantic entity, and of funcons that use them.

3.4.1. Inherited Entities

Inherited entities are propagated from the root of a derivation tree to the axioms. The contents of an inherited entity may be modified in the premise of an inference rule, but any such modifications are local to that premise.

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

$$\textit{Entity}\ \mathsf{environment}(\textbf{map-empty} : \textbf{environments}) \vdash \_ \longrightarrow \_ \qquad (34)$$

Here, environment is the name of the entity being declared, **environments** is the type of value contained in the entity, and **map-empty** is the initial value of the entity at the root of a derivation.

Let us consider some funcons that use this entity. The funcon **bound**(B) evaluates to the value bound to the binder B in the current environment. A common use of this funcon is when specifying the meaning of occurrences of

[4] In previous specifications of funcons (e.g. [5]), program output and control-flow signals were both handled by a single class of *emitted* entity, and program input was not addressed. In prior publications, inherited, mutable and emitted entities have often appeared under the names *read-only*, *read-write* and *write-only*, respectively.

identifiers in expressions (e.g. in Eq. 8). The definition of **bound** is as follows:

$$\textit{Funcon}\ \textbf{bound}(\_ : \textbf{binders}) : \Rightarrow \textbf{values} \tag{35}$$

$$\textit{Rule}\ \frac{\textbf{lookup}(B, \rho) = V : \textbf{values}}{\mathsf{environment}(\rho) \vdash \textbf{bound}(B) \longrightarrow V} \tag{36}$$

The conclusion of Rule 36 is a computation step, not a rewrite, because it interacts with a semantic entity (and hence is context-sensitive). The premise requires that looking up the binder in the environment evaluates to a value V (which would not hold if the binder were not in present in the environment).

The funcon **scope**(ρ, X) adds the bindings in ρ to the current environment for the purposes of executing the subterm X (e.g. in Eq. 15). The definition of **scope** is as follows:

$$\textit{Funcon}\ \textbf{scope}(\_ : \textbf{environments}, \_ : \Rightarrow T) : \Rightarrow T \tag{37}$$

$$\textit{Rule}\ \frac{\mathsf{environment}(\textbf{map-override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \textbf{scope}(\rho_1 : \textbf{environments}, X) \longrightarrow \textbf{scope}(\rho_1, X')} \tag{38}$$

$$\textit{Rule}\ \textbf{scope}(\_ : \textbf{environments}, V : \textbf{values}) = V \tag{39}$$

Observe that in the premise of Rule 38, the environment entity is updated with the bindings in ρ_1.

Recall that a funcon with any strict arguments is implicitly lifted to a funcon where all arguments have computation sorts, with implicit congruence rules for evaluating the 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 variables (see Eq. 15 and Sect. 2.6). Notice that in rules 38 and 39, the first argument is annotated "**: environments**", thereby ensuring that that these rules cannot apply before the congruence rule has evaluated that argument. In Rule 39, the "**: values**" annotation ensures that the enclosing scoped environment is not discarded until the second argument has been evaluated.

Another example of an inherited entity is given-value:

$$\textit{Entity}\ \mathsf{given\text{-}value}(() : \textbf{values}) \vdash \_ \longrightarrow \_ \tag{40}$$

This entity is used for a simple form of binding and scoping. The single value it contains (initially the empty tuple '()') can be viewed as a degenerate environment—it holds only a single value, and as such the value does not need to be named. To access the value, we use the funcon **given**, which is analogous to **bound**:

$$\textit{Funcon}\ \textbf{given} : T \Rightarrow T \tag{41}$$

$$\textit{Rule}\ \mathsf{given\text{-}value}(V) \vdash \textbf{given} \longrightarrow V \tag{42}$$

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

$$\textit{Funcon}\ \textbf{give}(\_ : S, \_ : S \Rightarrow T) : \Rightarrow T \tag{43}$$

$$\textit{Rule}\ \frac{\textsf{given-value}(V) \vdash X \longrightarrow X'}{\textsf{given-value}(\_) \vdash \textbf{give}(V : \textbf{values}, X) \longrightarrow \textbf{give}(V, X')} \tag{44}$$

$$\textit{Rule}\ \textbf{give}(\_ : \textbf{values}, V : \textbf{values}) = V \tag{45}$$

In language definitions, one common use of **give** and **given** is for passing arguments to functions, as we shall discuss in Sect. 3.5. They are also often used in an auxiliary capacity to share intermediate values in funcon definitions. Indeed, the use of the 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. Thus the signature for **given** (41) states that **given** computes a value of the same type as the given value, and the signature for **give** (43) requires that the second argument must be able to compute a value of type T in a context where the given value has the same type as the first argument.

3.4.2. Mutable Entities

Mutable entities are passed along between computation steps, with the resulting value of a mutable entity in one step being its initial value in the next step. In contrast to inherited entities, if a mutable entity is modified in the premise of a rule, the modified entity is accessible outside of that premise via the entity's result value.

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:

$$\textit{Entity}\ \langle \_ \,, \textsf{store}(\textbf{map-empty} : \textbf{stores}) \rangle \longrightarrow \langle \_ \,, \textsf{store}(\_ : \textbf{stores}) \rangle \tag{46}$$

The type **stores** is a mapping from **variables** to **values**.

To assign a value to a variable in the store, we use the funcon **assign**[5]:

$$\textit{Funcon}\ \textbf{assign}(\_ : \textbf{variables}(T), \_ : T) : \Rightarrow () \tag{47}$$

$$\textit{Rule}\ \frac{\textbf{map-override}(\{\, \mathit{Var} \mapsto \mathit{Val} \}, \sigma) = \sigma'}{\langle \textbf{assign}(\, \mathit{Var}, \mathit{Val}) \,, \textsf{store}(\sigma) \rangle \longrightarrow \langle () \,, \textsf{store}(\sigma') \rangle} \tag{48}$$

The notation $\{\_ \mapsto \_\}$ is CBS syntax for constructing map values. To retrieve

[5]Note that in the funcon repository, the definitions of **assign** and **assigned** (and similarly **bound**) have some additional details, such as for handling the case when the variable is not present in the store. We have omitted those details here for brevity, but the full definitions are available online [11].

the value assigned to a variable, we use the funcon **assigned** (e.g. in Eq. 7):

$$\textit{Funcon}\ \textbf{assigned}(\_ : \textbf{variables}(T)) : \Rightarrow T \tag{49}$$

$$\textit{Rule}\ \frac{\textbf{lookup}(\mathit{Var}, \sigma) = \mathit{Val} : \textbf{values}}{\langle \textbf{assigned}(\mathit{Var})\ , \mathsf{store}(\sigma)\rangle \longrightarrow \langle \mathit{Val}\ , \mathsf{store}(\sigma)\rangle} \tag{50}$$

3.4.3. Output Entities

Output entities propagate information upwards towards the root of a derivation tree. An output entity contains a list of values, and the outputs from adjacent computation steps are implicitly concatenated, forming the overall output of the program.

The classic example of an output entity is the standard-output channel. We declare such an entity in CBS as follows:

$$\textit{Entity}\ \_ \xrightarrow{\mathsf{standard\text{-}out}!(\_:\textbf{lists}(\textbf{values}))} \_ \tag{51}$$

A funcon for printing a value to the standard-out entity can be defined as follows:

$$\textit{Funcon}\ \textbf{print}(\_ : \textbf{values}) : \Rightarrow () \tag{52}$$

$$\textit{Rule}\ \textbf{print}(V : \textbf{values}) \xrightarrow{\mathsf{standard\text{-}out}![V]} () \tag{53}$$

Observe that we enclose the value to be printed in square brackets, forming a singleton list, because the contents of an output entity must be a list. A similar funcon, **print-list**, takes a list of values as its argument, and hence does not need to use the square bracket notation in its definition.

$$\textit{Funcon}\ \textbf{print-list}(\_ : \textbf{lists}(\textbf{values})) : \Rightarrow () \tag{54}$$

$$\textit{Rule}\ \textbf{print-list}(L : \textbf{lists}(\textbf{values})) \xrightarrow{\mathsf{standard\text{-}out}!(L)} () \tag{55}$$

Of the two funcons, we chose to use **print-list** in the SIMPLE definition (Eq. 19), as it corresponds more closely to SIMPLE's variadic **print** function.

3.4.4. Input Entities

Input entities are the mirror image of output entities. They likewise contain a list of values, but it represents the input *consumed* by a computation step, not the output emitted.

The classic example of an input entity is the standard-input channel. We declare such an entity in CBS as follows:

$$\textit{Entity}\ \_ \xrightarrow{\mathsf{standard\text{-}in}?(\_:\textbf{lists}(\textbf{values}))} \_ \tag{56}$$

A funcon for reading a value from the standard-in entity can be defined as follows:

$$\textit{Funcon}\ \textbf{read} : \Rightarrow \textbf{values} \tag{57}$$

$$\textit{Rule}\ \textbf{read} \xrightarrow{\mathsf{standard\text{-}in}?[V]} V \tag{58}$$

3.4.5. Control-flow Entities

Control-flow entities propagate signals upwards towards the root of a derivation tree, similarly to output entities. However, unlike output entities, a control-flow entity does not accumulate a list of values, but instead contains either a single value or the absence of a value. The presence of a value denotes that a signal has been emitted, carrying that value. Unlike output entities, signals from adjacent computation steps are not combined: instead, each emitted signal must be handled by a rule of an enclosing term within the same step.

The classic use of a control-flow entity is for throwing and handling exceptions. We declare an entity for thrown exceptions as follows:

$$\textit{Entity}\ \_ \xrightarrow{\mathsf{thrown}(\_ : \mathbf{values}^?)} \_ \tag{59}$$

The question mark here denotes that the entity contains zero or one **values**, and should not be confused with the question mark that appears after the name of an input entity.

To model throwing exceptions we use a funcon called **throw** (e.g. in Eq. 20), which emits a signal in the thrown entity:

$$\textit{Funcon}\ \mathbf{throw}(\_ : \mathbf{values}) : \Rightarrow T \tag{60}$$

$$\textit{Rule}\ \mathbf{throw}(V : \mathbf{values}) \xrightarrow{\mathsf{thrown}(V)} \mathbf{stuck} \tag{61}$$

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

$$\textit{Funcon}\ \mathbf{handle\text{-}thrown}(\_ : S \Rightarrow T, \_ : X \Rightarrow T) : S \Rightarrow T \tag{62}$$

$$\textit{Rule}\ \frac{E \xrightarrow{\mathsf{thrown}()} E'}{\mathbf{handle\text{-}thrown}(E, H) \xrightarrow{\mathsf{thrown}()} \mathbf{handle\text{-}thrown}(E', H)} \tag{63}$$

$$\textit{Rule}\ \frac{E \xrightarrow{\mathsf{thrown}(V)} \_}{\mathbf{handle\text{-}thrown}(E, H) \xrightarrow{\mathsf{thrown}()} \mathbf{give}(V, H)} \tag{64}$$

$$\textit{Rule}\ \mathbf{handle\text{-}thrown}(V : \mathbf{values}, \_) = V \tag{65}$$

The **handle-thrown** funcon deserves some explanation. 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. Rule 63 allows the term E to be evaluated while no thrown signal is emitted. We denote the absence of a signal using an empty tuple (simply omitting the thrown entity would just cause the signal to be implicitly propagated). If a signal is emitted, then Rule 64 discards the term E and gives the thrown value V to the handler H. Notice that no signal is emitted in the conclusion of Rule 64, as it has now been handled and does not need to continue to be propagated upwards. If the first argument of **handle-thrown** is a value, Rule 65 discards the handler and returns that value.

The above definition is actually a slight simplification. We also allow the handler computation to fail, in which case we rethrow the exception. This allows

a funcon term to contain multiple uses of **handle-thrown**, each handling only certain types of exception. Thus Rule 64 should be replaced with:

$$\textit{Rule}\ \frac{E \xrightarrow{\mathsf{thrown}(V)} \_}{\textbf{handle-thrown}(E, H) \xrightarrow{\mathsf{thrown}()} \textbf{else}(\textbf{give}(V, H), \textbf{throw}(V))} \tag{66}$$

The **else** funcon is itself a form of handler, which executes its second argument if it detects failure in its first argument.

In the SIMPLE specification, we also use the thrown entity for returning values from functions: the **return** statement is translated to a use of **throw**, and function bodies are translated with an enclosing **handle-thrown** to catch the returned value (see Appendix A).

Beyond exceptions, control-flow entities can be used to represent more complex forms of control flow. For example, together with Torrini, two authors of this paper have previously shown how they can be used to model Scheme's *call/cc* operator and delimited control operators [16].

3.5. Thunks, Functions and Closures

The contents of semantic entities are required to always be *values*. However, it is sometimes necessary to be able to store an unevaluated computation in an entity; for example, SIMPLE allows (global) function variables to be declared, and each function variable needs to store the funcon term corresponding to the function body, without executing that term until the function is called.

CBS supports this by providing a special kind of value called *thunks*. In this context, a thunk is a wrapper around an unevaluated funcon term that causes the term to be treated as a value, hence allowing it be stored in a data structure or semantic entity. A thunk can subsequently be *forced*, exposing the funcon term and allowing it to be executed.

We mediate between terms and thunks using the funcons **thunk** and **force**:

$$\textit{Funcon}\ \textbf{thunk}(\_ : S \Rightarrow T) : \textbf{thunks}(S \Rightarrow T)$$
$$:= \ldots \tag{67}$$
$$\textit{Funcon}\ \textbf{force}(\_ : \textbf{thunks}(S \Rightarrow T)) : S \Rightarrow T$$
$$:= \ldots \tag{68}$$

The **thunks** type is parameterised on the sort of computation it contains.

These two funcons are CBS primitives (indicated by the ":= ..." notation after the signature), and have no accompanying rules. There is no way to specify these funcons as rules in CBS, because these funcons are the *only* way to convert between values and thunks (all value constructors are implicitly strict and evaluate their arguments). Any other funcons that manipulate thunks are defined in terms of **thunk** and **force**.

We model functions (or procedures) that take an argument value by using the given-value entity for the argument. This is achieved by using **given** to refer to the argument inside the thunk at the definition site of the function, and

give to supply the function argument outside the thunk at the call-site of the function. For this common situation, the funcon repository provides the funcon **apply**, which combines **give** and **force** in this way:

$$\begin{aligned} &\textit{Funcon}\ \mathbf{apply}(F : \mathbf{thunks}(S \Rightarrow T), V : S) : \Rightarrow T \\ &\quad := \mathbf{give}(V, \mathbf{force}(F)) \end{aligned} \tag{69}$$

Note that CBS allows a funcon defined by a single rewrite rule to be defined instead by merging the rule with the signature using ':=' notation, as here.

This treatment of thunks leads to *dynamic scoping* for the **bound** funcon, as it is defined such that it looks up the binder in the environment where it is evaluated. To support *static scoping* of binders we use the funcon **close**, which modifies a thunk by capturing the current environment and encapsulating it with the thunked funcon term:

$$\textit{Funcon}\ \mathbf{close}(\_ : \mathbf{thunks}(S \Rightarrow T)) : \Rightarrow \mathbf{thunks}(S \Rightarrow T) \tag{70}$$

$$\textit{Rule}\ \mathsf{environment}(\rho) \vdash \mathbf{close}(F : \mathbf{values}) \longrightarrow \mathbf{thunk}(\mathbf{closure}(\mathbf{force}(F), \rho)) \tag{71}$$

The definition of **closure** is fairly similar to the definition of **scope**, except that it completely replaces the current environment, rather than adding to it.

$$\textit{Funcon}\ \mathbf{closure}(\_ : \Rightarrow T, \_ : \mathbf{environments}) : \Rightarrow T \tag{72}$$

$$\textit{Rule}\ \frac{\mathsf{environment}(\rho) \vdash X \longrightarrow X'}{\mathsf{environment}(\_) \vdash \mathbf{closure}(X, \rho : \mathbf{environments}) \longrightarrow \mathbf{closure}(X', \rho)} \tag{73}$$

$$\textit{Rule}\ \mathbf{closure}(V : \mathbf{values}, \_ : \mathbf{environments}) = V \tag{74}$$

That is, the computation inside the closure is evaluated with respect to the environment in the closure, not the environment where the closure is evaluated.

3.6. Sequences

The funcons we have presented thus far have all taken a fixed number of arguments. This need not be the case in general. For example, the **sequential** funcon, which we used in the SIMPLE translation to combine statement sequences (Eq. 16), can be applied to any number of arguments. The definition of **sequential** is as follows:

$$\begin{aligned} &\textit{Funcon}\ \mathbf{sequential}(X^* : (\Rightarrow \mathbf{values})^*) : \Rightarrow (\mathbf{values}^*) \\ &\quad := \mathbf{discard\text{-}empty\text{-}tuples}(\mathbf{left\text{-}to\text{-}right}(X^*)) \end{aligned} \tag{75}$$

The * suffix is used to denote sequences of funcon terms, as in translation functions. The signature expresses that **sequential** takes a sequence of computations as its argument, and that it computes a sequence of values (combined into a tuple). Meta-variables with a * suffix range over sequences of funcon terms; in this case X^* is instantiated to the entire sequence of arguments.

The definition of **sequential** uses **left-to-right**, which evaluates a sequence of arguments in left-to-right order, and **discard-empty-tuples**, which is a value operation that discards any arguments that are the empty tuple. Recall that we model commands as terms that compute the empty tuple; evaluating a sequence of commands using just **left-to-right** would thus result in a sequence of empty tuples, which is why we also use **discard-empty-tuples**.

The definition of **left-to-right** is as follows:

$$\textit{Funcon}\ \textbf{left-to-right}(\_ : (\Rightarrow\textbf{values})^*) : \Rightarrow(\textbf{values}^*) \tag{76}$$

$$\textit{Rule}\ \textbf{left-to-right}(V^* : \textbf{values}^*) = (V^*) \tag{77}$$

$$\textit{Rule}\ \frac{Y \longrightarrow Y'}{\textbf{left-to-right}(V^* : \textbf{values}^*, Y, Z^*) \longrightarrow \textbf{left-to-right}(V^*, Y', Z^*)} \tag{78}$$

The annotation '$: \textbf{values}^*$' restricts the meta-variable V^* to sequences of value terms. Thus Rule 77 says that if all the arguments are values, then those values are combined into a tuple and returned. Rule 78 expresses that the first non-value argument may make a step.

3.7. Remarks

This section has presented an informal and example-driven overview of using CBS to specify funcons. A formal account of the underlying MSOS framework can be found in [4], and of the two transition relations in [9]. The funcons in this section were selected to demonstrate different aspects of CBS specifications. Appendix B.1 lists the signatures of all funcons used in the SIMPLE language specification, and the definitions of those funcons are available online [11].

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.

4. Integrated Development Environment for CBS

We have used Spoofax [17] to generate an Eclipse-based editor for CBS specifications, with many helpful features. The editor analyses all CBS source files in the user's workspace, updating the analysis in the background whenever the user enters new 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: language definitions are typically divided into sections in the same way as reference manuals, whereas funcon definitions are in

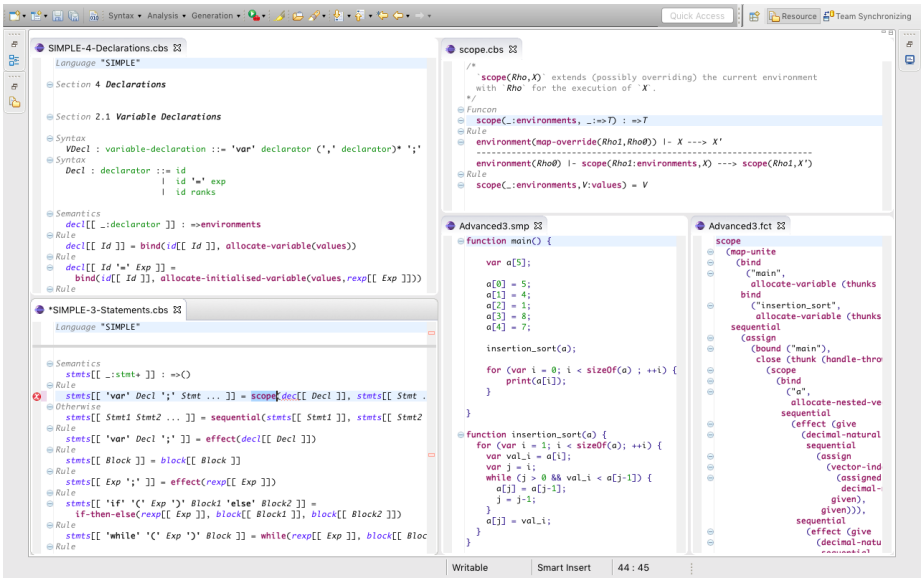

Figure 2: The IDE for CBS in action

separate files, grouped into folders according to the entities involved. The hyperlinks in the edited files facilitate navigation and browsing of CBS specifications without awareness of the underlying file system.

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

The CBS editor provides a button to generate executable code from a single CBS file (or from all CBS files in the enclosing folder and any sub-folders). Spoofax uses the code generated from a complete language definition to create an editor for programs in the defined language. This editor parses programs using a scannerless generalised LR (SGLR) parser generated from their context-free grammar for abstract syntax, which is usually highly ambiguous. The editor has a button for translating the abstract syntax trees of disambiguated programs to funcon terms; these can then be run using our Haskell-based funcon interpreter (described in Sect. 5), which we access as an external tool in Eclipse. CBS does not include notation for specifying disambiguation rules, but these can be provided in separate SDF3 files.

The screenshot in Figure 2 shows several files open in our integrated de-

velopment 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 definition of the funcon for scoping declarations. The two panes on the lower right show a small test program and part of its translation to funcons.

The implementation of the CBS editor in SPOOFAX involved writing an SDF3 grammar for the CBS language, some small files specifying the various editor services (highlighting, name resolution, menus, folding), and STRATEGO code to generate SDF3 grammars and STRATEGO rules from the ASTs of CBS specifications. Each semantic equation in CBS generates a corresponding STRATEGO rule. The generated SDF3 grammars provide the syntax for the semantic functions and meta-variables that occur in the generated STRATEGO rules.

5. Executing Fundamental Constructs

Once a programming language has been defined in CBS, we can use our IDE to translate programs of that language into funcon terms. One way to validate a language definition is to execute those terms and observe their behaviour. This requires an implementation of I-MSOS, the framework on which CBS funcon rules are based.

In earlier work with Bach Poulsen, one author of the present paper discussed synthesising executable interpreters by generating Prolog clauses from MSOS rules [18]. In this section, we discuss our new approach: a Haskell implementation of (the CBS variant of) I-MSOS.

We use monads to implement the implicit propagation of semantic entities. The benefits of Haskell as an implementation language include its native support for writing monadic code, and its strong static type system. Other benefits are its immutable data structures and explicit side effects: we need no extra effort to implement CBS' immutable meta-variables, and it becomes easy to implement backtracking of computational effects when searching for an applicable rule. The result is a fully modular interpreter that is easily extended with new funcon definitions.

This section is aimed at readers interested in the implementation of modular interpreters using functional languages generally, and the execution of I-MSOS rules specifically. Familiarity with Haskell and monads is assumed. An introduction to the use of monads in functional programming can be found in [19], and for an overview of related work we refer the reader to [20].

We have implemented a Haskell library for defining funcons in a style corresponding to CBS rules, and for interpreting those rules. The library provides

```
data Value = Nat Integer
           | String String
           | Thunk Funcon
           | ADT Name [Value]
           | Type Type
           | ...

data Type = Naturals
          | Strings
          | Thunks Sort
          | AlgebraicDatatypes Name [Type]
          | Types
          | ...

data Funcon = FName Name
            | FApp Name Funcon
            | FValue Value
            | FTuple [Funcon]
            | FList [Funcon]
            | ...
```

Figure 3: Representation of funcon terms as Haskell data types.

a parser for funcon terms, a pretty-printer, and an interpreter with a number of configuration options. The interpreter includes the definitions of all funcons from the CBS funcon library and is easily extended with new language-specific or general-purpose funcons. New funcon definitions can be added to the library by hand, or generated from CBS funcon definitions (discussed in Sect. 6).

This section gives an overview of our implementation approach, and the design decisions we made.

5.1. Modularity

Our funcon interpreter is designed to support modular development: a funcon can be defined in its own module independently of any other funcon or any semantic entity with which it does not directly interact. The former is achieved by representing funcon terms using string names, rather than Haskell data types directly, and then combining the entire funcon library into a dictionary with those names as keys. The latter is achieved by using monads to propagate the semantic entities.

5.1.1. Representation of funcon terms

The interpreter uses an internal representation of terms divided into funcons, values and types, as shown in Figure 3. Their mutual hierarchy is implemented via *constructor subtyping* [21]. CBS's built-in values (discussed in Sect. 3.3) are represented by the *Value* and *Type* data types, with a constructor for each built-in value or type. The representation of funcon terms (*Funcon*) has generic constructors *FName* and *FApp* for representing funcons by a name (a string). The funcon library is implemented as a map from funcon names to an evaluation function corresponding to their I-MSOS definitions.

```
type FunconLibrary = Map Name EvalFunction
```

An interpreter for a source language with language-specific funcons is obtained by extending the main *FunconLibrary* with those funcons. In a similar manner, each class of semantic entities is represented as a map from entity names to the initial value of that entity, and algebraic data types are represented as a map from type names to their constructors.

An alternative approach would have been to use the *Data types à la carte* technique [22] to combine funcons defined in separate modules into a single data type with a constructor for each funcon. Having individual Haskell constructors for each funcon would have provided stronger correct-by-construction guarantees about the well-formedness of funcon terms. Briefly, the technique involves defining a signature functor for each funcon (or group of funcons defined together), combining those functors as coproducts, and then applying a type-level fixed point operator to construct a recursive data type representing all funcon terms. Indeed, Day and Hutton [23] have used this technique for the specific purpose of defining modular programming-language constructs in a Haskell setting. Like us, they aim for modular compilers and interpreters for those constructs, and they use monads to propagate computational effects. We did not take the *Data types à la carte* approach here for several reasons. One reason is that we have an IDE to use as a front-end for CBS definitions, which we intend to use for well-formedness checking. Another is that many funcons do not have a fixed arity, but take an arbitrary length sequence of arguments. Furthermore, CBS permits strict funcons of any arity n to take a single funcon term that computes an n-tuple as its result (though we did not use this capability when defining SIMPLE). Consequently, we decided the extra complexity of *Data types à la carte* would not provide sufficient well-formedness guarantees to be worthwhile in this case.

5.1.2. Implicit propagation of Semantic Entities

In Sect. 3.4 we described how each class of semantic entity is implicitly propagated in CBS rules. We achieve this implicit propagation in our Haskell-embedded funcon definitions by working in a monad, and defining its *return* and *bind* operations such that they implement the desired implicit propagation. Briefly, an inherited entity correspond to a reader monad, a mutable entity corresponds to a state monad, and an output entity corresponds to a writer monad. 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 variation of a writer monad (only one signal may be emitted in each control-flow entity at once; they do not form a monoid).

We define a data type *MSOS* that combines the five entity classes (we omit the obvious monad instance):

$$\begin{aligned}
&\textbf{data}\ \mathit{MSOS}\ a\ = \mathit{MSOS}\ (\mathit{MReader} \to \mathit{MState} \to (a, \mathit{MState}, \mathit{MWriter}))\\
&\textbf{data}\ \mathit{MReader} = \mathit{MReader}\ \mathit{Inherited}\\
&\textbf{data}\ \mathit{MState}\ \ = \mathit{MState}\ \mathit{Mutable}\ \mathit{Input}\\
&\textbf{data}\ \mathit{MWriter} = \mathit{MWriter}\ \mathit{ControlFlow}\ \mathit{Output}
\end{aligned}$$

Each class such as *Inherited* stores a map from entity names to their contents.

This avoids the need to use monad transformers [20] to add additional entities: our MSOS monad is fixed. Using Haskell's native support for monads, this allows us to define funcons without mentioning entities that are unrelated to the funcon.

In our implementation the *MSOS* monad also stores extra information, including the funcon library, runtime options, and collected meta-data; but we elide those details here. It also provides facilities for connecting input and output entities to real console input/output, allowing a user to run a funcon program interactively.

5.2. Evaluating Funcon Terms

CBS supports two transition relations: computation steps and rewrites (see Sect. 3.1). Computation steps may depend on semantic entities, making them sensitive to context. Furthermore, computation steps may modify 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 introduce a separate monad called *Rewrite*, which does not provide access to the semantic entities. This guarantees, by construction, that rewrites do not access semantic entities.

5.2.1. Transition Relations

As described in Sect. 3.1, the properties of the rewrite relation are such that operationally, it can be applied at any time during execution, anywhere in a term. Thus, our interpreter is free to perform rewriting whenever is most convenient (or efficient). With this in mind, we define a function *rewriteFuncon* that attempts to rewrite a term as much as possible. The result of rewriting will then either be a value, or a funcon term that needs to execute a computation step to evaluate further. We express this by introducing a data type called *Rewritten*:

```
data Rewritten = ValTerm Value | CompTerm (MSOS Funcon)
rewriteFuncon :: Funcon → Rewrite Rewritten
```

To represent the computation-step relation, we introduce a *stepFuncon*, which first rewrites its argument to a *CompTerm* (throwing an exception if it rewrites to a value), and then performs the computation step:

```
stepFuncon :: Funcon → MSOS Funcon
```

(Note that the *Rewrite* monad can be lifted to the *MSOS* monad, but not vice-versa.)

We also define a variant of *stepFuncon* called *evalFuncon*, which differs in that if its argument rewrites to a value, then that value is simply returned in the MSOS monad. At the top-level, we execute a funcon program by iterating *evalFuncon*, stopping either when a value is returned, or an uncaught internal exception is raised (see Sect. 5.4). This corresponds to the reflexive transitive closure of the computation-step relation.

5.2.2. Refocusing 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, our interpreter traverses the funcon term from the root to the subterm being executed, updating semantic entities as required on the way. This corresponds to constructing a derivation tree for the step from the CBS rules (including the implicit congruence rules). Thus the cost of each step is potentially linear in the size of the funcon term.

To overcome this inefficiency, we have added an optional *refocusing* optimisation [24] to our interpreter. The key insight of refocusing 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, care must be taken as this optimisation is not valid in all situations. Most notably, 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 64 for **handle-thrown** in Sect. 3.4.5). Thus, when refocusing is enabled, our interpreter checks for emitted signals before each refocused step.

A similar issue arises for input, output and mutable entities. However, this is not an issue for inherited entities because modifications to inherited entities are always local to a subterm, and thus cannot influence an enclosing rule. Fortunately, interactions with inherited entities are by far the most common class of computation steps during the execution of a typical funcon term, and so the optimisation proves effective in practice.

5.2.3. Representing I-MSOS Rules

We represent the operational behaviour of a funcon as a Haskell function from a list of arguments to a funcon term. The function corresponds to one transition rule (of either of the two transition relations), and the resulting term corresponds to the right-hand side of the conclusion of the rule. Our representation of funcon behaviour also takes account of the strictness information from a CBS funcon signature: for each strict argument a congruence rule is implicitly generated (see Sect. 3.2), and any explicit rules for the funcon must be correspondingly strict in those arguments.

Concretely, we represent an evaluation function for a funcon as follows:

```
data EvalFunction
       = NullaryFuncon                  (Rewrite Rewritten)
       | StrictFuncon                   ([Value]    → Rewrite Rewritten)
       | NonStrictFuncon [Strictness] ([Funcon] → Rewrite Rewritten)
data Strictness = Strict | NonStrict
```

As all arguments of a strict funcon are implicitly evaluated by congruence rules, the evaluation function takes a list of *Value* arguments. For non-strict funcons, we provide a list of strictness values, which tells the interpreter which arguments to evaluate before applying the evaluation function.

5.3. Haskell-embedded Funcon Definitions

We will now present some example funcon definitions. In Sect. 6 we will discuss generating such definitions by compiling CBS funcon specifications.

5.3.1. Axioms

Recall the funcon **print** (Sect. 3.4.3), which was defined by a single rewriting rule (Rule 53). A corresponding evaluation function can be defined as follows:

```
funconPrint :: EvalFunction
funconPrint = StrictFuncon eval
where eval [v] = return (CompTerm (
          do writeOut "standard-out" [v]   -- output v on standard-out
             return (FTuple [])))          -- the rule's target (empty tuple)
```

The function *writeOut* is a helper function for writing values to an output entity.

```
writeOut :: Name → [Value] → MSOS ()
```

The evaluation function for **if-then-else** pattern-matches on the argument values to determine which branch to select:

```
funconIf_Then_Else :: EvalFunction
funconIf_Then_Else = NonStrictFuncon [Strict, NonStrict, NonStrict] eval
where eval [FValue b, x, y] = case b of
          (ADT "true"  []) → rewriteFuncon x
          (ADT "false" []) → rewriteFuncon y
```

(Recall that CBS Booleans are defined as an algebraic data type with two nullary constructors.) The strictness annotation for the first argument tells the interpreter to evaluate that argument before applying the evaluation function, so we do not need to explicitly encode the congruence rule as part of the definition.

5.3.2. Inference Rules

We will now consider defining the funcon **scope** from Sect. 3.4.1. This is more complicated than the previous examples, because **scope** is defined by multiple rules, one of which has a premise. We define the evaluation function as follows:

```
funconScope :: EvalFunction
funconScope = NonStrictFuncon [Strict, NonStrict] eval
where eval [FValue _, FValue v] = rewriteFuncon (FValue v)
      eval [FValue (Map ρ1), x] = return (CompTerm (
        do Map ρ0 ← getInh "environment"
           x' ← withInh "environment" (Map (union ρ1 ρ0)) (stepFuncon x)
           return (FApp "scope" (FTuple [FValue (Map ρ1), x'])))
```

The interpreter implicitly evaluates the first argument and rewrites the second argument, before applying the *eval* function.

The *getInh* and *withInh* helper functions are used to access and locally modify the environment.

$$\begin{aligned} &\mathit{getInh} &&:: \mathit{Name} \to \mathit{MSOS}\ \mathit{Values} \\ &\mathit{withInh} &&:: \mathit{Name} \to \mathit{Values} \to \mathit{MSOS}\ \mathit{Funcons} \to \mathit{MSOS}\ \mathit{Funcons} \end{aligned}$$

The interpreter provides similar functions for providing access to all classes of semantic entities.

This example demonstrates an advantage of propagating entities using immutable data, rather than using imperative variables. The environment (ρ_1) is still available to be used unmodified in the target of the rule, even though it has been extended with new bindings for the purposes of evaluating the premise.

5.4. Interpreter Exceptions

Various kinds of exceptions can arise during execution of a funcon term. A funcon could be applied to incorrect arguments (e.g. if the first argument of **if-then-else** was not a Boolean), or a built-in value value operation may be partial (e.g. division by zero). If an evaluation function attempts to perform a computation step on a subterm, that then may fail (e.g. if the subterm is a value, or if the stepping that subterm throws an exception itself).

To handle these cases, the *Rewrite* and *MSOS* monads also contain an exception-monad component. When searching for applicable transitions, the interpreter uses these exceptions to support backtracking (discussed in Sect. 6.1.1) The interpreter also remembers the current term and subterm under evaluation. Together with the current contents of semantic entities, this allows the interpreter to give detailed error messages to the user.

5.5. Remarks

The library presented in this section supports fully modular definitions of funcons that correspond to I-MSOS/CBS rules. Following this approach, we have manually defined all the funcons used in the CBS definition Caml Light (a previous case study [5]), suggesting that manual definition of funcons with the library is practical. The library is available online as a Haskell package [25], and is easily integrated into existing Haskell projects.

The main result is the degree of modularity with which funcons are defined and semantic entities are accessed. Funcons can be isolated in their own modules, which do not need to be recompiled when new funcons or semantic entities are added. Interpreters are obtained by importing all the desired funcon modules and combining the funcons in a dictionary. As a result, even interpreters are compositional. For example, our Haskell package for interpreting funcon terms generated from the CBS definition of SIMPLE contains only the funcons specific to SIMPLE, and our Haskell package for interpreting funcon terms generated from our CBS definition of Caml Light contains only the funcons specific

to Caml Light. All the other funcons are imported from the main Haskell package, which contains the funcon interpreter and main funcon library. Indeed, the main funcon library could be separated into a separate package if desired.

A user who did not wish to use CBS or its supporting tool chain could develop a funcon-based interpreter for a source language entirely in Haskell; she would just need to provide a parser for the source language, and a (Haskell) translation function from the parsed source language to funcon terms. If necessary, language-specific funcons can be added to the library manually, as discussed in Sect. 5.3.

6. Compiling CBS Funcon Definitions

In Sect. 5.3 we presented some Haskell-embedded funcon definitions. In this section, we will explain how we compile such definitions from CBS funcon definitions.

6.1. Compiler-oriented interpreter extensions

The examples in Sect. 5.3 correspond to CBS funcon definitions that do not use patterns, type annotations, or rewriting premises. Haskell code corresponding to rules with these features is less straightforward. While Haskell patterns are convenient for a human writing funcon definitions, and adequate in most cases, in general they cannot directly express all cases of CBS pattern matching. Similarly, Haskell's conditional branching cannot directly express the choice between different rules in general, because each rule may have premises that depend on the semantic entities in arbitrary ways, which may require backtracking in the general case. Therefore, to allow compilation of CBS funcon definitions, we first need to extend the library with backtracking facilities and a general method for pattern matching.

6.1.1. Backtracking

MSOS rules are not ordered, and thus operationally any rule may be applied when evaluating a funcon term. However, CBS funcon signatures with strict arguments place do restrict the order of evaluation slightly: the implicit congruence rules must be applied first. It would be valid to apply the remaining rules in any order, and our interpreter follows the strategy of attempting each rewriting rule one by one, and then each computation-step rule one by one, until one succeeds. This evaluation strategy is provided by the *evalRules* function:

$$evalRules :: [Rewrite\ Rewritten] \to [MSOS\ Funcon] \to Rewrite\ Rewritten$$

To support backtracking, the exceptions in our interpreter are divided into those that represent a rule failing to match, and those that represent other sources of internal error. The *evalRules* function acts as a handler for the former type of exceptions, backtracking and attempting the next rewrite or computation step when one is detected. Other types of exceptions are propagated to the top-level, and reported to the user.

6.1.2. Pattern matching

MSOS rules contain meta-variables, both as arguments and in funcon terms. CBS furthermore allows sequence variables, which are beyond the scope of Haskell's pattern-matching facilities. Our library therefore needs to provide an implementation of CBS pattern matching, rather than using Haskell's pattern matching directly.

We define an additional data type *FTerm*, which is essentially the same as *Funcon* except that it has an additional constructor for meta-variables.[6] We also define a meta-environment *Env* (not to be confused with the environment inherited entity), which maps meta-variables to funcon terms.

```
type Env = M.Map MetaVar LevelledTerm
data LevelledTerm = ValueTerm     Value
                  | ValuesTerm    [Value]
                  | FunconTerm    Funcon
                  | FunconsTerm [Funcon]
```

Meta-variables can match both funcons and sequences of funcons, and some patterns restrict the arguments to be values. Consequently our meta-environment can contain several different forms of funcon term (the *LevelledTerm* data type).

Substitution thus has the following type:

```
substitute :: FTerm → Env → Rewrite Funcons
```

Pattern matching (potentially) modifies the meta-environment by adding new meta-variable bindings. We provide two functions for this, one for matching sequences of values, and one for matching arbitrary funcon sequences:

```
vsMatch :: [Value]   → [Pattern] → Env → Rewrite Env
fsMatch :: [Funcon] → [Pattern] → Env → Rewrite Env
```

Recall that CBS allows sequence meta-variables such as X^*, X^+ and $X^?$ (see Sect. 3.6). Such meta-variables are the main complication of pattern-matching.

First consider the simple case of matching a sequence of terms $x_1, \ldots, x_n$ to a sequence of patterns $p_1, \ldots, p_m$, when none of those patterns can match a sequence. If we have a procedure to match a single term x_i to a single pattern p_i, we can match the sequence by checking whether $n = m$ and whether x_i matches p_i for all $0 \leqslant i \leqslant n$. However, if p_i can match a subsequence of $x_1, \ldots, x_n$, we need a more elaborate matching strategy. Moreover, meta-variables of the form X^* and X^+ may match multiple subsequences, introducing potential ambiguity.

Our solution is similar to Wadler's "list of successes" method for pattern-matching [27]. Lists capture that a single pattern can give rise to zero or more

[6] Axelsson and Vezzosi [26] have shown how to implement term rewriting using data types with and without meta-variables (or other terms), while avoiding duplication of constructors (using the *Data types à la carte* technique [22]). However, we do not believe the added complexity would make this worthwhile in our case.

```
funconHandle_thrown :: EvalFunction
funconHandle_thrown fargs = NonStrictFuncon [NonStrict, NonStrict] eval
  where eval = evalRules [rewrite1] [step1, step2]
        rewrite1 = do let env = emptyEnv
                      ...   -- rewrite statements omitted
        step1 = do let env = emptyEnv
                   ...   -- step statements omitted
        step2 = do let env = emptyEnv
                   ...   -- step statements omitted
```

Figure 4: Example of the scaffolding of a generated evaluation function.

successful matches. For meta-variables of the form X^*, we implemented a function for arbitrary repetition (similar to Wadler's *rep*). By parameterising this function, it can also handle meta-variables of the forms X^+ and $X^?$.

Disambiguation is achieved by a variant of longest-match. Straightforward longest match, in which a pattern always returns the largest possible match, does not suffice. For example, consider the sequence of patterns 'X^*, **true**'. With straightforward longest match, X^* fully matches any sequence of terms, including any term **true**. Therefore the pattern 'X^*, **true**' could not match any sequence of terms. Instead, we allow a pattern to return all possible matches, one of which is selected by a backtracking procedure. The procedure finds the largest local match that leads to a successful global match (if possible).

6.2. Compiling CBS Rules

We now have the infrastructure to need to overview the compilation scheme from CBS funcons to Haskell-embedded funcons.

6.2.1. Strictness

A funcon in Haskell comprises a *Name* and an *EvalFunction*. The signature of a funcon determines the strictness of the funcon, and hence which constructor of the *EvalFunction* data type (Sect. 5.2.3) to use. For non-strict funcons, the list of strictness annotations is also generated.

6.2.2. Evaluation functions

Figure 4 shows a fragment of the Haskell code generated for **handle-thrown** (Sect. 3.4.5), eliding the code details for the individual rules. All generated evaluation functions are of this form. In general, the funcon's arguments (*fargs*) can be of type [*Funcon*] or [*Value*], depending on whether the funcon is strict. For nullary funcons *fargs* is not available.

The rewrite rule (*rewrite*$_1$) and step rules (*step*$_1$, *step*$_2$) are defined as a sequence of statements in the *Rewrite* monad and *MSOS* monad, respectively. The statements are invocations of *rewriteFuncon* and *stepFuncon* (generated from premises), applications of the functions *fsMatch* and *vsMatch*, predicates for checking other side conditions, and functions for accessing and modifying

entities. The statements propagate, and potentially extend, an environment binding meta-variables to funcon terms. The functions for accessing or modifying semantic entities perform pattern matching or substitution respectively.

6.2.3. Rewrites

CBS rewrite rules are specified according to the following template:

$$\frac{C_1 \quad \ldots \quad C_k}{f(P) = T}$$

Here f is the name of the funcon, P is a pattern, T is a term, and C_i is either a rewrite or side condition. Haskell code is generated for rewrite rules according to the following template:

```
R = do let env = emptyEnv
       env    ← fsMatch fargs P env
       env    ← sideCondition C_1 env
       ...
       env    ← sideCondition C_k env
       target ← substitute T env
       rewriteFuncon target
```

The function *sideCondition* handles both rewrites and other conditions such as type membership checks. In the actual generated code P, T and C_i are replaced by generated code and R by a unique name for the rule. If f is a strict funcon, *fsMatch* is replaced with *vsMatch*.

6.2.4. Computation Steps

A rule for a computation step may additionally access or modify semantic entities. Moreover, they may also have computational steps as premises.

Consider the following template of the computation-step relation:

$$\mathsf{inherited}(\gamma) \vdash \langle f(T_1)\ , \mathsf{mutable}(\sigma)\rangle \xrightarrow{\mathsf{input?}(\iota)\ \mathsf{control}(\alpha)\ \mathsf{output!}(o)} \langle T_2\ , \mathsf{mutable}(\sigma')\rangle \quad (79)$$

For simplicity, we consider the case when there is exactly one of each class of entity. Note that T_1, γ, σ and ι are patterns, while T_2, α, o and σ' are terms.

Haskell code for a rule with a computation-step of the form of Template 79 as its conclusion, is generated as follows:

```
S = do let env = emptyEnv
       env ← fsMatch fargs T_1 env          -- pattern match the source
       env ← getInhPatt inherited γ env     --
       env ← consumeInputPatt input ι env   --
       env ← getMutPatt mutable σ env       --
       ...                                  -- side conditions & premises
       putMutTerm mutable σ' env            --
       raiseSignalTerm control α env        --
       writeOutTerm output o env            --
       lift2MSOS (substitute T_2 env)       -- Rewrite lifted to MSOS monad
```

In the actual generated code S is replaced by a unique name generated for the rule. The entity class names are replaced by entity names, T_1, γ, σ and ι by patterns, and T_2, α, o and σ' by terms.

If a computation-step appears in a premise, then we generate a variation of the preceding code in which the patterns and terms are inverted. (Intuitively, we *receive* the initial value of a mutable entity in the conclusion of a rule, but *provide* it with a value in the premise of the rule; the other entities are similarly inverted.) Thus, for a premise of the form of Template 79, Haskell code is generated as follows:

$$
\begin{array}{l}
\mathit{putMutTerm}\ \mathit{mutable}\ \sigma\ \mathit{env} \\
\mathit{env} \leftarrow \mathit{withInputTerm}\ \mathit{input}\ \iota\ \mathit{env} \\
\qquad (\mathit{withInhTerm}\ \mathit{inherited}\ \gamma\ \mathit{env} \\
\qquad\quad (\mathit{readOutPatt}\ \mathit{output}\ o \\
\qquad\qquad (\mathit{receiveSignalPatt}\ \mathit{control}\ \alpha \\
\qquad\qquad\quad (\mathit{stepTerm}\ T_1\ T_2\ \mathit{env})))) \\
\mathit{getMutPatt}\ \mathit{mutable}\ \sigma'\ \mathit{env}
\end{array}
$$

Reading from control-flow and output entities is structured slightly differently to writing to them, as the helper functions have to be applied to an *MSOS* computation, in a similar manner to *withInhTerm* (from Sect. 5.3.2).

6.3. Remarks

We have overviewed a compilation scheme for compiling CBS funcon definitions to Haskell, allowing us to validate CBS funcon definitions through testing. Our compiler is available online [11] as a Haskell package. For a comprehensive overview of the features and interface we refer to the package's documentation.

Efficiency was not our primary concern when developing the interpreter and CBS compiler. However, the interpreters we obtain from generating Haskell considerably outperform the generated Prolog interpreters described in [18], and refocusing (discussed in Sect. 5.2.2) significantly improves the interpreter's efficiency further. As we are compiling the CBS specifications rather than interpreting them, a number of optimisations are possible, though a thorough investigation of this remains as future work. For example, left-factorisation could be applied to sets of rules in order to decrease the cost of backtracking [28].

7. Related Work

Executable semantic definitions have been developed since the early 1970s. Current frameworks supporting executable semantics[7] include the CoreASM

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

tools[8], Ott[9], PLTRedex[10], the K tools[11], Maude[12], MMT[13], Melange[14], and DynSem[28]. Most of these frameworks have a high degree of modularity. However, the only one which comes together with an extensive collection of reusable components is MMT, which is a direct precursor of CBS, based on MSOS. CBS has the advantage of being based on the more conventional-looking I-MSOS variant of MSOS, and of having tool support based on Spoofax and Eclipse. An extensive discussion relating several of these other frameworks to component-based semantics can be found in [5], which we will not repeat here.

Like CBS, DynSem achieves modularity by adopting the implicit propagation of semantic entities from I-MSOS. Unlike CBS, DynSem does not provide fixed transition relations, instead allowing the user to define the desired relations. Reuse of semantic definitions between languages is not a goal of DynSem, although it does includes *native operators*, analogous to our value operations, and *meta-functions* for abstracting over a common operation within a language definition.

The possibility of defining a collection of funcons in K, and of using the K tools to translate languages to funcons, is explored in [13]. Similarly to CBS, the K rules for funcons do not require modifying when funcons are combined, or new funcons added. However, in contrast to CBS, the 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 be accompanied by a configuration definition. Any reuse of parts of configurations between different language definitions is left implicit.

Introduced by Knuth in 1968 [29], the Attribute Grammar formalism was among the first proposals for specifying the semantics of a programming language. Attribute grammars achieve modularity by extending the formalism with *copy rules*: omitted semantic definitions for attributes are automatically generated, based on whether the attributes are *inherited*, *synthesised* or *chained*. The resulting implicit propagation corresponds to the propagation of the CBS' inherited, output and mutable entities, respectively.

Together with Churchill and Torrini, two of the present authors have previously used Spoofax and Prolog to implement executable component-based semantics [5]. The tool support was adequate for the Caml Light case study, but its combination of several meta-languages hindered further development. The design of CBS was based on the experience acquired during that work; the implementation is essentially completely new.

[8] `https://www.uni-ulm.de/in/pm/forschung/projekte/coreasm`
[9] `http://www.cl.cam.ac.uk/~pes20/ott`
[10] `http://redex.racket-lang.org`
[11] `http://www.kframework.org`
[12] `http://maude.cs.uiuc.edu`
[13] `https://github.com/fcbr/mmt`
[14] `http://melange-lang.org`

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 compositional translation functions that produce funcon terms (Sect. 2). The funcons themselves are also defined in CBS (Sect. 3), but we intend to provide a library of over 100 pre-specified 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 highly 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. 4), execution of funcon terms (Sect. 5), and compilation of CBS funcon definitions (Sect. 6).

Our hope is that the after successful completion of the C$^\sharp$ case study, and subsequent publication of over 100 validated funcons, 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.

Acknowledgements

The reported work was supported by EPSRC grants for the PLanCompS project at Swansea University (EP/I032495/1) and Royal Holloway, University of London (EP/I032509/1).

References

[1] R. Milner, M. Tofte, D. Macqueen, The Definition of Standard ML, MIT Press, 1997.

[2] P. Hudak, J. Hughes, S. P. Jones, P. Wadler, A history of Haskell: Being lazy with class, in: Conference on History of Programming Languages III, ACM, 2007, pp. 12:1–12:55. doi:10.1145/1238844.1238856.

[3] P. D. Mosses, M. J. New, Implicit propagation in structural operational semantics, in: Fifth Workshop on Structural Operational Semantics, Vol.

229(4) of Electronic Notes in Theoretical Computer Science, Elsevier, 2009, pp. 49–66. doi:10.1016/j.entcs.2009.07.073.

[4] P. D. Mosses, Modular structural operational semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 195–228. doi: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, Vol. 8989 of Lecture Notes in Computer Science, Springer, 2015, pp. 132–179. doi:10.1007/978-3-662-46734-3_4.

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

[7] T. Vollebregt, L. C. L. Kats, E. Visser, Declarative specification of template-based textual editors, in: 12th Workshop on Language Descriptions, Tools, and Applications, ACM, 2012, pp. 1–7. doi:10.1145/2427048.2427056.

[8] E. Visser, Stratego: A language for program transformation based on rewriting strategies, in: 12th International Conference on Rewriting Techniques and Applications, Vol. 2051 of Lecture Notes in Computer Science, Springer, 2001, pp. 357–361. doi:10.1007/3-540-45127-7_27.

[9] M. Churchill, P. D. Mosses, Modular bisimulation theory for computations and values, in: 16th International Conference on Foundations of Software Science and Computation Structures, Vol. 7794 of Lecture Notes in Computer Science, Springer, 2013, pp. 97–112. doi:10.1007/978-3-642-37075-5_7.

[10] G. Roşu, T. F. Şerbănuţă, K overview and SIMPLE case study, Electronic Notes in Theoretical Computer Science 304 (2014) 3–56. doi:10.1016/j.entcs.2014.05.002.

[11] L. T. van Binsbergen, P. D. Mosses, N. Sculthorpe, Executable component-based semantics: Supplementary material (2016).
URL http://www.plancomps.org/jlamp2016

[12] G. Roşu, T. F. Şerbănuţă, An overview of the K semantic framework, Journal of Logic and Algebraic Programming 79 (6) (2010) 397–434. doi:10.1016/j.jlap.2010.03.012.

[13] P. D. Mosses, F. Vesely, FunKons: Component-based semantics in K, in: 10th International Workshop on Rewriting Logic and Its Applications, Vol. 8663 of Lecture Notes in Computer Science, Springer, 2014, pp. 213–229. doi:10.1007/978-3-319-12904-4_12.

[14] P. D. Mosses, A modular SOS for ML concurrency primitives, BRICS Research Series RS-99-57, Department of Computer Science, Aarhus University (1999).
URL http://www.brics.dk/RS/99/57/

[15] G. D. Plotkin, A structural approach to operational semantics, Journal of Logic and Algebraic Programming 60–61 (2004) 17–139, reprint of Technical Report FN-19, DAIMI, Aarhus University, 1981. doi:10.1016/j.jlap.2004.05.001.

[16] N. Sculthorpe, P. Torrini, P. D. Mosses, A modular structural operational semantics for delimited continuations, in: 2015 Workshop on Continuations, Electronic Proceedings in Theoretical Computer Science, 2016, to appear.

[17] 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. doi:10.1145/1869459.1869497.

[18] C. Bach Poulsen, P. D. Mosses, Generating specialized interpreters for modular structural operational semantics, in: 23rd International Symposium on Logic-Based Program Synthesis and Transformation, Vol. 8901 of Lecture Notes in Computer Science, Springer, 2014, pp. 220–236. doi:10.1007/978-3-319-14125-1_13.

[19] P. Wadler, The essence of functional programming, in: 19th Symposium on Principles of Programming Languages, ACM, 1992, pp. 1–14. doi:10.1145/143165.143169.

[20] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: 22nd Symposium on Principles of Programming Languages, ACM, 1995, pp. 333–343. doi:10.1145/199448.199528.

[21] G. Barthe, M. J. Frade, Constructor subtyping, in: 8th European Symposium on Programming Languages and Systems, Vol. 1576 of Lecture Notes in Computer Science, Springer, 1999, pp. 109–127. doi:10.1007/3-540-49099-X_8.

[22] W. Swierstra, Data types à la carte, Journal of Functional Programming 18 (4) (2008) 423–436. doi:10.1017/S0956796808006758.

[23] L. E. Day, G. Hutton, Towards modular compilers for effects, in: 12th International Symposium on Trends in Functional Programming, Vol. 7193 of Lecture Notes in Computer Science, Springer, 2012, pp. 49–64. doi:10.1007/978-3-642-32037-8_4.

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

[25] L. T. van Binsbergen, N. Sculthorpe, The funcons-tools package (2016). URL `https://hackage.haskell.org/package/funcons-tools`

[26] E. Axelsson, A. Vezzosi, Lightweight higher-order rewriting in haskell, in: 16th International Symposium on Trends in Functional Programming, Lecture Notes in Computer Science, Springer, to appear.

[27] P. Wadler, How to replace failure by a list of successes, in: Second Conference on Functional Programming Languages and Computer Architecture, Vol. 201 of Lecture Notes in Computer Science, Springer, 1985, pp. 113–128. `doi:10.1007/3-540-15975-4_33`.

[28] V. Vergu, P. Neron, E. Visser, DynSem: A DSL for dynamic semantics specification, in: 26th International Conference on Rewriting Techniques and Applications, Vol. 36 of Leibniz International Proceedings in Informatics, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 365–378. `doi:10.4230/LIPIcs.RTA.2015.365`.

[29] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145. `doi:10.1007/BF01692511`.

Appendix A. SIMPLE Language Definition

This appendix presents our CBS definition of the SIMPLE language. The top-level translation function is *program*. 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 [11].

Appendix A.1. Values

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

Semantics *val* ⟦_ : value⟧ : **values**
Semantics *id* ⟦_ : id⟧ : **ids**

Appendix A.2. Expressions

Syntax *Exp* : exp ::= '(' exp ')' | value | '++' lexp | lexp | exp '(' exps? ')' | '−' exp
| '**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 | lexp '=' exp

Rule ⟦'(' *Exp* ')'⟧ : exp = ⟦*Exp*⟧

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

Rule *rexp* ⟦*V*⟧ = *val* ⟦*V*⟧

Rule *rexp* ⟦'++' *LExp*⟧ =

  give(

    lexp ⟦*LExp*⟧,

    sequential(**assign**(**given**, **integer-add**(**assigned**(**given**), 1)), **assigned**(**given**)))

Rule *rexp* ⟦*LExp*⟧ = **assigned**(*lexp* ⟦*LExp*⟧)

Rule *rexp* ⟦*Exp* '(' ')'⟧ = **force**(*rexp* ⟦*Exp*⟧)

Rule *rexp* ⟦*Exp* '(' *Exps* ')'⟧ = **apply**(*rexp* ⟦*Exp*⟧, (*rexps* ⟦*Exps*⟧))

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

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

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

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

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

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

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

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

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

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

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

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

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

Rule *rexp* ⟦Exp_1 **'!='** Exp_2⟧ = **not**(**is-equal**(*rexp* ⟦Exp_1⟧, *rexp* ⟦Exp_2⟧))

Rule *rexp* ⟦**'!'** *Exp*⟧ = **not**(*rexp* ⟦*Exp*⟧)

Rule *rexp* ⟦Exp_1 '&&' Exp_2⟧ = **if-then-else**(*rexp* ⟦Exp_1⟧, *rexp* ⟦Exp_2⟧, **false**)

Rule *rexp* ⟦Exp_1 '||' Exp_2⟧ = **if-then-else**(*rexp* ⟦Exp_1⟧, **true**, *rexp* ⟦Exp_2⟧)

Rule *rexp* ⟦*LExp* '=' *Exp*⟧ =

  give(*rexp* ⟦*Exp*⟧, **sequential**(**assign**(*lexp* ⟦*LExp*⟧, **given**), **given**))

Syntax *Exps* : exps ::= exp (',' exp)*

Semantics *rexps* ⟦_ : exps⟧ : (⇒**values**)+

Rule *rexps* ⟦*Exp*⟧ = *rexp* ⟦*Exp*⟧

Rule *rexps* ⟦Exp_1 ',' Exp_2 ...⟧ = *rexp* ⟦Exp_1⟧, *rexps* ⟦Exp_2 ...⟧

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

Rule ⟦*LExp* '[' Exp_1 ',' Exp_2 ... ']'⟧ : lexp = ⟦*LExp* '[' Exp_1 ']' '[' Exp_2 ... ']'⟧

Semantics *lexp* ⟦_ : lexp⟧ : ⇒**variables**(**values**)

Rule *lexp* ⟦*Id*⟧ = **bound**(*id* ⟦*Id*⟧)

Rule *lexp* ⟦*LExp* '[' *Exp* ']'⟧ = **vector-index**(*rexp* ⟦*LExp*⟧, *rexp* ⟦*Exp*⟧)

Appendix A.3. Statements

Syntax *Block* : block ::= '{' stmt* '}'
Semantics *block*⟦_ : block⟧ : ⇒()
Rule *block*⟦'{' '}'⟧ = ()
Rule *block*⟦'{' $Stmt^+$ '}'⟧ = *stmts*⟦$Stmt^+$⟧

Syntax *Stmt* : stmt ::= variable-declaration | 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 ')' '{' $Stmt^*$ '}'⟧ : stmt =
⟦'{' *Stmt* 'while' '(' Exp_1 ')' '{' $Stmt^*$ Exp_2 ';' '}' '}'⟧
Rule ⟦'var' $Decl_1$ ',' $Decl_2$ ... ';' $Stmt^*$⟧ : $stmt^+$ =
⟦'var' $Decl_1$ ';' 'var' $Decl_2$ ... ';' $Stmt^*$⟧

Semantics *stmts*⟦_ : $stmt^+$⟧ : ⇒()
Rule *stmts*⟦'var' *Decl* ';' $Stmt^+$⟧ = **scope**(*decl*⟦*Decl*⟧, *stmts*⟦$Stmt^+$⟧)
Otherwise *stmts*⟦*Stmt* $Stmt^+$⟧ = **sequential**(*stmts*⟦*Stmt*⟧, *stmts*⟦$Stmt^+$⟧)
Rule *stmts*⟦'var' *Decl* ';'⟧ = **effect**(*decl*⟦*Decl*⟧)
Rule *stmts*⟦*Block*⟧ = *block*⟦*Block*⟧
Rule *stmts*⟦*Exp* ';'⟧ = **effect**(*rexp*⟦*Exp*⟧)
Rule *stmts*⟦'if' '(' *Exp* ')' $Block_1$ 'else' $Block_2$⟧ =
if-then-else(*rexp*⟦*Exp*⟧, *block*⟦$Block_1$⟧, *block*⟦$Block_2$⟧)
Rule *stmts*⟦'while' '(' *Exp* ')' *Block*⟧ = **while**(*rexp*⟦*Exp*⟧, *block*⟦*Block*⟧)
Rule *stmts*⟦'print' '(' *Exps* ')' ';'⟧ = **print-list**[*rexps*⟦*Exps*⟧]
Rule *stmts*⟦'return' *Exp* ';'⟧ = **throw**(**variant**("return", *rexp*⟦*Exp*⟧))
Rule *stmts*⟦'return' ';'⟧ = **throw**(**variant**("return", **null**))
Rule *stmts*⟦'try' $Block_1$ 'catch' '(' *Id* ')' $Block_2$⟧ =
handle-thrown(
block⟦$Block_1$⟧,
scope(
bind(*id*⟦*Id*⟧, **allocate-initialised-variable**(**values**, **given**)),
block⟦$Block_2$⟧))
Rule *stmts*⟦'throw' *Exp* ';'⟧ = **throw**(*rexp*⟦*Exp*⟧)

Appendix A.4. Declarations

Syntax *VDecl* : variable-declaration ::= 'var' declarator (',' declarator)* ';'

Syntax *Decl* : declarator ::= id | id '=' exp | id ranks
Semantics *decl*⟦_ : declarator⟧ : ⇒environments
Rule *decl*⟦*Id*⟧ = bind(*id*⟦*Id*⟧, allocate-variable(values))
Rule *decl*⟦*Id* '=' *Exp*⟧ = bind(*id*⟦*Id*⟧, allocate-initialised-variable(values, *rexp*⟦*Exp*⟧))
Rule *decl*⟦*Id* *Ranks*⟧ = bind(*id*⟦*Id*⟧, allocate-nested-vectors[*ranks*⟦*Ranks*⟧])

Funcon allocate-nested-vectors(_ : lists(naturals)) : ⇒vectors(variables(values))

Rule allocate-nested-vectors[N] =
  allocate-initialised-variable(vectors(variables(values)), allocate-vector(values, N))

Rule allocate-nested-vectors[N, N^+] =
  allocate-initialised-variable(
    vectors(variables(values)),
    vector-map(allocate-nested-vectors[N^+], vector-repeat(N, null)))

Syntax *Ranks* : ranks ::= '[' exps ']' ranks$^?$
Rule ⟦'[' Exp_1 ',' Exp_2 ... ']' $Ranks^?$⟧ : ranks = ⟦'[' Exp_1 ']' '[' Exp_2 ... ']' $Ranks^?$⟧
Semantics *ranks*⟦_ : ranks⟧ : (⇒naturals)$^+$
Rule *ranks*⟦'[' *Exp* ']'⟧ = *rexp*⟦*Exp*⟧
Rule *ranks*⟦'[' *Exp* ']' *Ranks*⟧ = *rexp*⟦*Exp*⟧, *ranks*⟦*Ranks*⟧

Syntax *GDecl* : global-declaration ::= 'function' id '(' ids$^?$ ')' block

Semantics *global-decl*⟦_ : global-declaration⟧ : ⇒environments
Rule *global-decl*⟦'var' *Decl* ';'⟧ = *decl*⟦*Decl*⟧
Rule *global-decl*⟦'function' *Id* '(' *Ids*$^?$ ')' *Block*⟧ =
  bind(*id*⟦*Id*⟧, allocate-variable(thunks(values⇒values)))

Semantics *initialise-func*⟦_ : global-declaration⟧ : ⇒()
Rule *initialise-func*⟦'var' *Decl* ';'⟧ = ()
Rule *initialise-func*⟦'function' *Id* '(' ')' *Block*⟧ =
  assign(bound(*id*⟦*Id*⟧), close(thunk(*function-block*⟦*Block*⟧)))
Rule *initialise-func*⟦'function' *Id* '(' *Ids* ')' *Block*⟧ =
  assign(bound(*id*⟦*Id*⟧), close(thunk(scope(
                                    match(given, (*id-patterns*⟦*Ids*⟧)),
                                    function-block⟦*Block*⟧)))))

Semantics *function-block* ⟦*Block* : block⟧ : ⇒**values**
:= **handle-thrown**(
block ⟦*Block*⟧,
case(**variant**(**"return"**, **pattern-any**), **variant-value**(**given**)))

Syntax *Ids* : ids ::= id (',' id)*
Semantics *id-patterns* ⟦_ : ids⟧ : patterns+
Rule *id-patterns* ⟦*Id*⟧ = **thunk**(**bind**(*id* ⟦*Id*⟧, **allocate-initialised-variable**(**values**, **given**)))
Rule *id-patterns* ⟦Id_1 ',' Id_2 ...⟧ = *id-patterns* ⟦Id_1⟧, *id-patterns* ⟦Id_2 ...⟧

Appendix A.5. Programs

Syntax *Prog* : program ::= global-declaration+
Rule ⟦'**var**' $Decl_1$ ',' $Decl_2$ ... ';' $GDecl^*$⟧ : program =
⟦'**var**' $Decl_1$ ';' '**var**' $Decl_2$ ... ';' $GDecl^*$⟧

Semantics *program* ⟦*Prog* : program⟧ : ⇒**values**
:= **scope**(
map-unite(*global-decls* ⟦*Prog*⟧),
sequential(*initialise-funcs* ⟦*Prog*⟧, **force**(**assigned**(**bound**(**"main"**)))))

Semantics *global-decls* ⟦_ : program⟧ : (⇒**environments**)+
Rule *global-decls* ⟦*GDecl*⟧ = *global-decl* ⟦*GDecl*⟧
Rule *global-decls* ⟦*GDecl* $GDecl^+$⟧ = *global-decl* ⟦*GDecl*⟧, *global-decls* ⟦$GDecl^+$⟧

Semantics *initialise-funcs* ⟦_ : program⟧ : (⇒())+
Rule *initialise-funcs* ⟦*GDecl*⟧ = *initialise-func* ⟦*GDecl*⟧
Rule *initialise-funcs* ⟦*GDecl* $GDecl^+$⟧ =
initialise-func ⟦*GDecl*⟧, *initialise-funcs* ⟦$GDecl^+$⟧

Appendix B. Funcon Signatures

This appendix lists the signatures of all funcons and value operations used in the definition of the SIMPLE language (Appendix A), or in any of the funcon definitions presented in Sect. 3.

Appendix B.1. Funcons

Funcon **allocate-initialised-variable**(_ : **types**, _ : **values**) : ⇒**all-variables**
Funcon **allocate-nested-vectors**(_ : **lists**(**naturals**)) : ⇒**vectors**(**variables**(**values**))
Funcon **allocate-variable**(_ : **types**) : ⇒**all-variables**
Funcon **allocate-vector**(_ : **types**, _ : **naturals**) : ⇒**vectors**(**all-variables**)
Funcon **apply**(_ : **thunks**(S⇒T), _ : S) : ⇒T
Funcon **assign**(_ : **variables**(T), _ : T) : ⇒()
Funcon **assigned**(_ : **variables**(T)) : ⇒T
Funcon **bind**(_ : **binders**, _ : **values**) : **environments**
Funcon **bound**(_ : **binders**) : ⇒**values**
Funcon **case**(_ : **patterns**, _ : S⇒T) : S⇒T
Funcon **close**(_ : **thunks**(S⇒T)) : ⇒**thunks**(S⇒T)
Funcon **closure**(_ : ⇒T, _ : **environments**) : ⇒T
Funcon **effect**(_ : **values**) : ()
Funcon **else**(_ :⇒T, _ : (⇒T)$^+$) : ⇒T
Funcon **force**(_ : **thunks**(S⇒T)) : S⇒T
Funcon **give**(_ : S, _ : S⇒T) : ⇒T
Funcon **given** : T⇒T
Funcon **handle-thrown**(_ : S⇒T, _ : X⇒T) : S⇒T
Funcon **if-then-else**(_ : **booleans**, _ : ⇒T, _ : ⇒T) : ⇒T
Funcon **left-to-right**(_ : (⇒**values**)*) : ⇒(**values***)
Funcon **match**(_ : **values**, _ : **patterns**) : ⇒**environments**
Funcon **print**(_ : **values**) : ⇒()
Funcon **print-list**(_ : **lists**(**values**)) : ⇒()
Funcon **read** : ⇒**values**
Funcon **sequential**(_ : (⇒**values**)*) : ⇒(**values***)
Funcon **scope**(_ : **environments**, _ : ⇒T) : ⇒T
Funcon **throw**(_ : **values**) : ⇒T
Funcon **thunk**(_ : S⇒T) : **thunks**(S⇒T)
Funcon **vector-map**(_ : S⇒T, _ : **vectors**(S)) : ⇒**vectors**(T)
Funcon **while**(_ : ⇒**booleans**, _ : ⇒()) : ⇒()

Appendix B.2. Value Operations

Funcon **discard-empty-tuples**(_ : **values***) : (**values***)
Funcon **integer-add**(_ : **integers***) : **integers**
Funcon **integer-divide**(_ : **integers**, _ : **integers**) : **integers**
Funcon **integer-modulo**(_ : **integers**, _ : **integers**) : **integers**
Funcon **integer-multiply**(_ : **integers***) : **integers**

Funcon **integer-negate**(_ : **integers**) : **integers**
Funcon **integer-subtract**(_ : **integers**, _ : **integers**) : **integers**
Funcon **is-less**(_ : **rationals**, _ : **rationals**) : **booleans**
Funcon **is-less-or-equal**(_ : **rationals**, _ : **rationals**) : **booleans**
Funcon **is-greater**(_ : **rationals**, _ : **rationals**) : **booleans**
Funcon **is-greater-or-equal**(_ : **rationals**, _ : **rationals**) : **booleans**
Funcon **is-equal**(_ : **values**, _ : **values**) : **booleans**
Funcon **lookup**(_ : S, _ : **maps**(S, T)) : T
Funcon **map-override**(_ : **maps**$(S, T)^*$) : **maps**(S, T)
Funcon **map-unite**(_ : **maps**$(S, T)^*$) : **maps**(S, T)
Funcon **not**(_ : **booleans**) : **booleans**
Funcon **variant**(_ : **values**, _ : **values**) : **variants**(M)
Funcon **variant-value**(_ : **variants**(M)) : **values**
Funcon **vector-length**(_ : **vectors**(T)) : **naturals**
Funcon **vector-index**(_ : **vectors**(T), _ : **naturals**) : T
Funcon **vector-repeat**(_ : **naturals**, _ : T) : **vectors**(T)