Reusable Components of Semantic Specifications[*]

Martin Churchill[1], Peter D. Mosses[2], Neil Sculthorpe[2], and Paolo Torrini[2]

[1] Google, Inc.
[2] PLanCompS project, Swansea University, Swansea, UK
http://www.plancomps.org

Abstract. Semantic specifications of programming languages typically have poor modularity. This hinders reuse of parts of the semantics of one language when specifying a different language – even when the two languages have many constructs in common – and evolution of a language may require major reformulation of its semantics. Such drawbacks have discouraged language developers from using formal semantics to document their designs.

In the PLanCompS project, we have developed a component-based approach to semantics. Here, we explain its modularity aspects, and present an illustrative case study: a component-based semantics for Caml Light. We have tested the correctness of the semantics by running programs on an interpreter generated from the semantics, comparing the output with that produced on the standard implementation of the language.

Our approach provides good modularity, facilitates reuse, and should support co-evolution of languages and their formal semantics. It could be particularly useful in connection with domain-specific languages and language-driven software development.

Keywords: modularity, reusability, component-based semantics, fundamental constructs, funcons, modular SOS

1 Introduction

Various programming constructs are common to many languages. For instance, assignment statements, sequencing, conditional branching, loops and procedure calls are almost ubiquitous among languages that support imperative programming; expressions usually include references to declared variables and constants, arithmetic and logical operations on values, and function calls; and blocks are provided to restrict the scope of local declarations. The details of such constructs often vary between languages, both regarding their syntax and their intended behaviour, but sometimes they are identical.

Many constructs are also 'independent', in that their contributions to program behaviour are unaffected by the presence of other constructs in the same language. For instance, consider conditional expressions 'E_1 ? E_2 : E_3'. How they are evaluated is unaffected by whether expressions involve variable references, side effects, function calls, process synchronisation, etc. In contrast, the behaviour of a loop may depend on whether the language includes break and continue statements.

[*] This is the author's version of the work. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-662-46734-3_4

1.1 Modularity and Reusability

We consider a semantic specification framework to have *good modularity* when independent constructs can be specified separately, once and for all. Such frameworks support verbatim reuse of the specifications of common independent constructs between different language specifications. They also reduce the amount of reformulation needed when languages evolve.

Poor Modularity. It is well known that various semantic frameworks do not have good modularity. A particularly familiar example of a framework with poor modularity is structural operational semantics (SOS) [56]. As a simple illustration of the lack of modularity, consider specifying the evaluation of conditional expressions in the small-step SOS style developed by Plotkin:

$$\frac{E_1 \to E_1'}{E_1 \mathbin{?} E_2 : E_3 \to E_1' \mathbin{?} E_2 : E_3} \tag{1}$$

$$\texttt{true} \mathbin{?} E_2 : E_3 \to E_2 \tag{2}$$

$$\texttt{false} \mathbin{?} E_2 : E_3 \to E_3 \tag{3}$$

The transition formula $E \to E'$ asserts the possibility of a step of the computation of the value of E such that, after making the step, E' remains to be evaluated. The inference rule (1) specifies that computing the value of '$E_1 \mathbin{?} E_2 : E_3$' may involve computing the value of E_1; the axioms (2, 3) specify how the computation proceeds after E_1 has computed the value `true` or `false`. If the computation of the value of E_1 does not terminate, neither does that of '$E_1 \mathbin{?} E_2 : E_3$'; if it terminates with a value other than `true` or `false`, the computation of '$E_1 \mathbin{?} E_2 : E_3$' is stuck: it cannot make any further steps.

However, suppose we are specifying the semantics of a simple imperative language that includes also expressions of the form '$I = E$', intended to assign the value of E to a simple variable named I and return the value. We might specify the evaluation of such assignment expressions as follows.

$$\frac{\rho \vdash (E, \sigma) \to (E', \sigma')}{\rho \vdash (I = E, \sigma) \to (I = E', \sigma')} \tag{4}$$

$$\rho \vdash (I = V, \sigma) \to (V, \sigma[\rho(I) \mapsto V]) \tag{5}$$

The environment ρ above represents the current bindings of identifiers (e.g., to declared imperative variables) and the store σ represents the values currently assigned to such variables. The formula $\rho \vdash (E, \sigma) \to (E', \sigma')$ asserts that, after making the step, E' remains to be evaluated (or has been fully evaluated) and σ' reflects any side-effects. Axiom (5) specifies that when the value V of E has been computed, it is also the value of the enclosing expression; the resulting store reflects the assignment of that value to the variable bound to I in ρ.

Conventional SOS requires the semantics of all constructs in the same syntactic category to be specified using the same form of transition formulae. This is intrinsically a non-modular requirement. In the above example, it means we have to reformulate rules (1–3) as follows – in effect, *weaving* the extra arguments (here ρ, σ and σ') of the required transition formulae into the original rules.

$$\frac{\rho \vdash (E_1, \sigma) \to (E_1', \sigma')}{\rho \vdash (E_1 \,\texttt{?}\, E_2 \,\texttt{:}\, E_3, \sigma) \to (E_1' \,\texttt{?}\, E_2 \,\texttt{:}\, E_3, \sigma')} \tag{6}$$

$$\rho \vdash (\texttt{true}\,\texttt{?}\, E_2 \,\texttt{:}\, E_3, \sigma) \to (E_2, \sigma) \tag{7}$$

$$\rho \vdash (\texttt{false}\,\texttt{?}\, E_2 \,\texttt{:}\, E_3, \sigma) \to (E_3, \sigma) \tag{8}$$

Different SOS rules would be needed for specifying conditional expressions in other languages. For example, in a pure functional language, the transition formulae could be simply $\rho \vdash E \to E'$; in a process language, they would involve labels on transitions, e.g., $E \xrightarrow{a} E'$. The notation used to specify a language construct depends not only on the features of that particular construct, but also on the features of all the *other* constructs in the language. This flagrant disregard for modularity in conventional SOS implies that it is simply not possible to specify *once and for all* the semantics of conditional expressions (or any other programming constructs).

A Hindrance to Reuse. Several semantic frameworks are just as non-modular as SOS, whereas others have a somewhat higher degree of modularity (as discussed in Sect. 5). However, a further and almost universal feature of semantic descriptions of programming languages affects potential *reuse* of their parts: the common practice of using notation from the *concrete* syntax of a language when defining its semantics. For instance, the SOS rules for conditional expressions above might be based on a grammar including the following productions:

$$E : \mathit{exp} ::= \mathit{exp} \,\texttt{?}\, \mathit{exp} \,\texttt{:}\, \mathit{exp} \mid \texttt{true} \mid \texttt{false} \tag{9}$$

Such grammars provide a concise and suggestive specification of the abstract syntax (i.e., compositional structure) of programs, and are generally preferred to the original style of abstract syntax specification developed by McCarthy [36]. These grammars are typically highly ambiguous, but parsing and disambiguation are usually handled as a preliminary step before the semantics, so ambiguity is not a problem. However, the use of concrete terminal symbols to distinguish language constructs entails that our SOS rules for '$E_1 \,\texttt{?}\, E_2 \,\texttt{:}\, E_3$' cannot be directly reused for a language using different concrete syntax for conditional expressions, e.g., '$\texttt{if}\, E_1 \,\texttt{then}\, E_2 \,\texttt{else}\, E_3$'.

Without support for both modularity and reuse, the development and subsequent revision of a formal semantics for a major programming language is inherently a huge effort, often regarded as disproportionate to its benefits [23].

1.2 Fundamental Constructs (Funcons)

Our component-based approach to semantics addresses both modularity and reusability. Its crucial novel feature is the introduction of an *open-ended* collection of so-called *fundamental constructs*, or *funcons*. Many of the funcons correspond closely to simplified language constructs. But in contrast to language constructs, each funcon has a fixed interpretation, which we specify, *once and for all*,[3] using a modular variant of SOS called MSOS [42]. For example, the collection includes a funcon written '**if-true**(E_1, E_2, E_3)', whose interpretation corresponds directly to that of the language construct 'E_1 **?** E_2 **:** E_3' considered above.

Language Specification. To specify the semantics of a language, we translate all its constructs to funcons. Thanks to the closeness of funcons to language constructs, the translation is generally rather simple to specify. For instance, the translation of 'E_1 **?** E_2 **:** E_3' could be trivial, simply using '**if-true**' to combine the translations of E_1, E_2, E_3; translation of conditional expressions that have a different type of condition (e.g., test for zero) involves inserting operations to test the value of E_1.

Each funcon has both static and dynamic semantics. A *single* translation of a language to funcons therefore defines *both* the static and dynamic semantics of the language. Sometimes it is necessary to adjust the induced static semantics by inserting further funcons, e.g., our '**if-true**' funcon requires its second and third arguments to have a common type, but the intended static semantics of 'E_1 **?** E_2 **:** E_3' might require inclusion between the minimal types of E_2 and E_3. Funcons for making such static checks have vacuous dynamic semantics.

Defining the semantics of a language by translating it to funcons is somewhat analogous to defining the semantics of a full language by translation to a kernel sublanguage whose semantics is defined directly, as for Standard ML [38]. However, the direct definition of a kernel language is language-specific, and does not provide reusable components.

Funcon Specification. The funcon specifications are expected to be highly reusable components of language specifications. Their crucial feature is that when funcons are combined in a language specification, or when a new funcon is added to the open-ended collection, the specifications *never* require any changes. MSOS has particular advantages in that respect, but it should be possible to specify funcons also using other highly modular frameworks, e.g., the K framework [58], as illustrated in [50].

When the syntax or semantics of a language construct changes, however, the specification of its translation to funcons has to change accordingly (since we never change the semantics of funcons) so the translation specification itself is inherently not so reusable. We explain all this further, and provide some simple introductory examples, in Sect. 2.

Case Study. The main contribution of this paper is in Sect. 3, where we illustrate the modularity and practical applicability of our approach by presenting excerpts

[3] The specifications of the current collection of funcons will not be finalised until we have tested their use in two further major case studies, as discussed in Sect. 6.

from a moderate-sized case study: a component-based semantics of Caml Light [32]. This language is used for teaching functional programming, but also has imperative features. For selected language constructs, we give conceptual explanations of the funcons involved in their translations, and present the MSOS specifications of the semantics of the funcons. We have made the complete case study available online [12]. The PLanCompS project [55] is carrying out two further major case studies to demonstrate the extent to which funcons can be reused in specifications of different languages.

Tool Support. We have tested the correspondence between our component-based semantics of Caml Light and the standard implementation of the language [32, version 0.75], by running programs using a (modular!) interpreter generated from the MSOS specifications of the funcons [2,3,44]. Although the focus of this paper is on the features of component-based language specifications, we describe and illustrate our current tool support, which involves Spoofax [28] and Prolog, in Sect. 4. Further tool support is being developed by the PLanCompS project.

Related Work. We discuss related work and alternative approaches in Sect. 5, then conclude and outline further work in Sect. 6. This paper is an extended and improved version of a Modularity '14 conference paper [13].

2 Component-Based Semantics

In this section, we first explain the general concepts underlying *fundamental constructs* (*funcons*), giving some simple examples. We then consider how to specify translations from programming languages to funcons. Finally, we recall MSOS (a modular variant of SOS) and show how we use it to specify, once and for all, the static and dynamic semantics of each funcon as a highly reusable component of language specifications.

2.1 Funcon Notation

As mentioned in Sect. 1.2, many funcons correspond closely to simplified programming language constructs. However, each funcon has fixed syntax and semantics. For example, executing the funcon term written **assign**(E_1, E_2) always has the effect of evaluating the funcon term E_1 to a variable, E_2 to a value (in any order, possibly with interleaving), then assigning the value to the variable; its static semantics requires the type of E_1 to be that of a variable for storing values of the type of E_2. In contrast, a language construct written '$E_1 = E_2$' may be interpreted as an assignment or as an equality test, depending on the language, and the details of the interpretation may differ (e.g., regarding the possibility of coercions, composite variables, or failure). In a logic programming language, '$E_1 = E_2$' is interpreted as unification, which differs more fundamentally.

Sorts and Signatures. We can introduce a notion of well-formedness for funcon terms, based on sorts and signatures. The *signature* of a funcon determines its name, how many arguments it takes (if any), the sort of each argument, and the sort of the result. In our approach signatures provide also a form of *strictness annotation*, relying on a notion of *lifting*, introduced below. We consider a funcon term to be *well-sorted* when each of its argument terms (if any) is well-sorted and is of the sort required by its lifted signature.

We distinguish between *value sorts* and *computation sorts*. The pre-defined value sorts include *booleans* (the values **false** and **true**), *ints* (the unbounded integers), *unit* (the single value **null**), *ids* (identifiers), and *variables* (imperative variables). Generic pre-defined value sorts include *lists*(X) (finite lists of values of sort X) and *maps*(X, Y) (finite maps from values of sort X to values of sort Y). New value sorts (such as *records* and *vectors*) can be defined using algebraic data types, instantiation of generic sorts, and subsort inclusion.

Values for us are intrinsically *independent* of the computational context in which they occur. For any value sort X, *computes*(X) is the computation sort of funcon terms which, whenever their executions terminate normally, compute values of sort X. The following computation sorts reflect fundamental conceptual distinctions commonly found in programming languages.

- The sort of *expressions* (*exprs*) is for funcons that compute arbitrary *values*, possibly with side-effects.
- The sort of *declarations* (*decls*) is for funcons that compute *environments*, which are maps from identifiers to values.
- The sort of *commands* (*comms*) is for funcons that are executed for their effects, computing always the same **null** value.

The computation sorts *exprs*, *decls* and *comms* abbreviate instances of *computes*(X); if needed, further sort abbreviations could be introduced. Importantly, the *effects* of computations of sort *computes*(X) are completely unconstrained: they may include abrupt termination, assignment, spawning concurrent processes, communication, synchronisation, etc. Note that a computation sort *computes*(X) always includes the value sort X as a subsort, since we regard values as terminated computations.

Table 1 shows the signatures of some funcons. The funcons **if-true** (conditional choice), **scope** (local binding), **seq** (sequencing) and **supply** (value-passing) are polymorphic: the sort variable X in a signature may be instantiated (uniformly) with any value sort.

Lifting. Value sorts in signatures can always be *lifted* to computation sorts. For example, consider the value operation **not**(*booleans*) : *booleans*. By lifting the signature to **not**(*exprs*) : *exprs* we can use **not** as a funcon, applying it to any expression E. The value of **not**(E) is computed by first computing the value of E, then (provided that this is a value of sort *booleans*) applying the unlifted **not** operation. The same principle applies to funcons with a single value sort argument, such as **assigned-value**: its lifted signature is **assigned-value**(*exprs*) : *exprs*, *Funcon sorts*

$$\begin{aligned}
\mathit{comms} &= \mathit{computes}(\mathit{unit})\\
\mathit{decls} &= \mathit{computes}(\mathit{environments})\\
\mathit{exprs} &= \mathit{computes}(\mathit{values})
\end{aligned}$$

Funcon signatures

$$\begin{aligned}
\mathbf{assign}(\mathit{variables}, \mathit{values}) &: \mathit{comms}\\
\mathbf{assigned\text{-}value}(\mathit{variables}) &: \mathit{exprs}\\
\mathbf{bind\text{-}value}(\mathit{ids}, \mathit{values}) &: \mathit{decls}\\
\mathbf{bound\text{-}value}(\mathit{ids}) &: \mathit{exprs}\\
\mathbf{effect}(\mathit{values}) &: \mathit{comms}\\
\mathbf{given} &: \mathit{exprs}\\
\mathbf{if\text{-}true}(\mathit{booleans}, \mathit{computes}(X), \mathit{computes}(X)) &: \mathit{computes}(X)\\
\mathbf{scope}(\mathit{environments}, \mathit{computes}(X)) &: \mathit{computes}(X)\\
\mathbf{seq}(\mathit{unit}, \mathit{computes}(X)) &: \mathit{computes}(X)\\
\mathbf{supply}(\mathit{values}, \mathit{computes}(X)) &: \mathit{computes}(X)\\
\mathbf{while\text{-}true}(\mathit{exprs}, \mathit{comms}) &: \mathit{comms}
\end{aligned}$$

Table 1. Some funcon sorts and signatures

and the computation of the argument value is followed by applying the original funcon to it. For funcons such as **if-true** and **scope**, which have one or more further arguments with explicit computation sorts, the computation of those argument(s) depends on the funcon itself. An extreme case of this is **while-true**(E, C), where the computations E and C generally need to be repeated, depending on the values computed by E.

When we lift value operations and funcons (such as **assign**) with two or more value sort arguments, those argument values may be computed in any order, allowing also interleaving of side-effects. We can use the funcons **supply** and **given** to insist on a particular order of funcon argument evaluation. For example, **supply**$(E_2,$ **assign**$(E_1,$ **given**$))$ always evaluates E_2 before E_1. The funcon **given** refers to the value computed by the first argument of the closest-enclosing **supply**.

Although lifting of value operations to funcons is reminiscent of functional programming, the argument computations themselves need not be purely functional: they may throw exceptions, assign to variables, spawn concurrent processes, or even diverge, and their interleaving may give rise to nondeterminism. In Sect. 2.3, we shall see how MSOS allows us to specify the interleaving of computations without making any assumptions at all about their possible effects.

Well-Typedness. The lifted signatures of funcons and value operations determine a set of well-sorted funcon terms for each sort. However, the well-sortedness of a funcon term is independent of its context, and it does not exclude terms whose computation leads to the *value* of an argument not being of the required sort. For example, consider **if-true**(E, C_1, C_2), which is well-sorted whenever E is of sort *exprs* and C_1, C_2 are both of sort *comms*: after evaluating E, the computation gets stuck unless the value of E is true or false. In contrast, **if-true**(E, C_1, C_2) is *well-typed* in a particular context only if E is guaranteed to compute a Boolean value whenever it terminates normally. The well-typedness of funcon terms is required by their static semantics, which is considered in Sect. 2.4.

2.2 Language Semantics

We next consider how to specify a translation from a programming language to funcons. Each funcon has not only dynamic semantics (as illustrated in Sect. 2.3) but also static semantics (see Sect. 2.4), so a single translation of complete programs to funcon terms determines both the static and dynamic semantics of the programs.

The starting point for specifying a translation to funcons is a context-free grammar for the abstract syntax of the source language. We define functions mapping abstract syntax trees generated by the grammar to terms of the appropriate computation or value sorts. The functions are compositional: the translation of a composite language construct is a combination of the translations of its components. We specify the translation functions inductively, by equations (much as in denotational semantics).

The following examples illustrate how to specify the translation of some simple language constructs to funcons. Their main purpose is to show the form of the equations used to define the translation functions. Section 3 provides excerpts from a component-based semantics for a complete language, demonstrating how our approach scales up, and how to translate some less straightforward language constructs to funcons.

Expressions. Let *exp* be the nonterminal symbol for expressions in some programming language. We specify that the function *expr*⟦ _ ⟧ translates abstract syntax trees generated by *exp* to funcon terms of sort *exprs* thus:

$$\mathit{expr}[\![\ \_ : \mathit{exp}\ ]\!] : \mathit{exprs}$$

Note that language constructs are always inside ⟦ ⋯ ⟧, and funcons outside, so clashes of notation between them are insignificant. Let the meta-variable E, optionally subscripted and/or primed, range over abstract syntax trees generated by *exp*.

Recall the conditional expressions specified in SOS in Sect. 1. When their conditions are Boolean-valued, the intended semantics of these expressions correspond exactly to the semantics of the funcon **if-true** (lifted from *booleans* to *exprs* in its first argument), so we can specify their translation very simply indeed:

$$\mathit{expr}[\![\, E_1 \;\texttt{?}\; E_2 \;\texttt{:}\; E_3 \,]\!] = \textbf{if-true}(\mathit{expr}[\![\, E_1 \,]\!], \mathit{expr}[\![\, E_2 \,]\!], \mathit{expr}[\![\, E_3 \,]\!]) \tag{10}$$

The variant where E_1 is a numerical expression can be specified by inserting the appropriate value operations to compute **true** when the value of E_1 is non-zero, and **false** otherwise:

$$\begin{aligned}\mathit{expr}[\![\, E_1 \;\texttt{?}\; E_2 \;\texttt{:}\; E_3 \,]\!] = \textbf{if-true}(&\textbf{not}(\textbf{equal}(\mathit{expr}[\![\, E_1 \,]\!], 0)),\\ &\mathit{expr}[\![\, E_2 \,]\!], \mathit{expr}[\![\, E_3 \,]\!])\end{aligned} \tag{11}$$

Notice that the well-sortedness of the terms in the above equation comes from lifting the value operations **not** and **equal** to the computation sort *exprs*. Lifting also allows the following straightforward translation of equality test expressions.

$$\mathit{expr}[\![\, E_1 \texttt{==} E_2 \,]\!] = \textbf{equal}(\mathit{expr}[\![\, E_1 \,]\!], \mathit{expr}[\![\, E_2 \,]\!]) \tag{12}$$

To specify left-to-right evaluation of '$E_1 \texttt{==} E_2$', we can use the funcons **supply** and **given**, as follows.

$$\mathit{expr}[\![\, E_1 \texttt{==} E_2 \,]\!] = \textbf{supply}(\mathit{expr}[\![\, E_1 \,]\!], \textbf{equal}(\textbf{given}, \mathit{expr}[\![\, E_2 \,]\!])) \tag{13}$$

When identifiers I in expressions can refer only to (imperative) variables, we can translate them as follows:

$$\mathit{expr}[\![\, I \,]\!] = \textbf{assigned-value}(\textbf{bound-value}(\mathit{id}[\![\, I \,]\!])) \tag{14}$$

Here $\mathit{id}[\![\, \_ \,]\!]$ translates identifiers in a language to elements of our pre-defined value sort *ids*. The funcon **assigned-value** requires its argument to compute a variable, and gives the value currently assigned to that variable. When identifiers might also refer to other sorts of values, we use a funcon (not illustrated here) that gives the same result as **assigned-value** when the value of its argument is a variable, and otherwise simply returns the value.

Statements. Let *stm* be the nonterminal symbol for statements S in some programming language. The corresponding sort of funcons is *comms* (commands), so we use the following translation function.

$$\mathit{comm}[\![\, \_ : \mathit{stm} \,]\!] : \mathit{comms}$$

An assignment statement '$I \texttt{=} E\ \texttt{;}$' corresponds to a straightforward combination of the **assign** and **bound-value** funcons:

$$\mathit{comm}[\![\, I \texttt{=} E\ \texttt{;} \,]\!] = \textbf{assign}(\textbf{bound-value}(\mathit{id}[\![\, I \,]\!]), \mathit{expr}[\![\, E \,]\!]) \tag{15}$$

The following translation of assignment *expressions* illustrates repeated use of a previously computed value, which is first assigned, then returned as the result:

$$\begin{aligned}\mathit{expr}[\![\, I \texttt{=} E \,]\!] = \textbf{supply}(\mathit{expr}[\![\, E \,]\!], \textbf{seq}(&\textbf{assign}(\textbf{bound-value}(\mathit{id}[\![\, I \,]\!]), \textbf{given}),\\ &\textbf{given}))\end{aligned} \tag{16}$$

The combination of assignment expressions and the following expression statements (which discard the value of E) allows the specification of assignment *statements* in (15) to be derived.

$$\mathit{comm}\,[\![\, E \texttt{ ; }]\!] = \textbf{effect}(\mathit{expr}\,[\![\, E \,]\!]) \tag{17}$$

Our translation of if-else statements uses the same polymorphic **if-true** funcon as that of conditional expressions above:

$$\mathit{comm}\,[\![\, \texttt{if(}\, E \,\texttt{)}\; S_1 \;\texttt{else}\; S_2 \,]\!] = \textbf{if-true}(\mathit{expr}\,[\![\, E \,]\!], \mathit{comm}\,[\![\, S_1 \,]\!], \mathit{comm}\,[\![\, S_2 \,]\!]) \tag{18}$$

For if-then statements, we can exploit the usual ‘desugaring’, which we specify by the following equation.

$$\mathit{comm}\,[\![\, \texttt{if(}\, E \,\texttt{)}\; S \,]\!] = \mathit{comm}\,[\![\, \texttt{if(}\, E \,\texttt{)}\; S \;\texttt{else \{ \}} \,]\!] \tag{19}$$

Provided that we do not introduce circularity between such equations, they give the effect of translating a language to a kernel sublanguage, followed by translation of the kernel constructs to funcons. When the grammar of the kernel is of particular interest, we could exhibit it, and separate the specification of desugaring from the specification of the translation of the kernel to funcons.

The translation of the empty statement ‘`{ }`’ used above is just as simple as one might expect:

$$\mathit{comm}\,[\![\, \texttt{\{ \}} \,]\!] = \textbf{null} \tag{20}$$

While-statements with Boolean conditions correspond exactly to our **while-true** funcon (without any lifting, since the computations of both the expression E and the statement$S$ may need to be repeated):

$$\mathit{comm}\,[\![\, \texttt{while(}\, E \,\texttt{)}\; S \,]\!] = \textbf{while-true}(\mathit{expr}\,[\![\, E \,]\!], \mathit{comm}\,[\![\, S \,]\!]) \tag{21}$$

Our final illustrative example of specifying translations demonstrates a technique used frequently in our Caml Light case study in Sect. 3. Statement sequences may consist of more than two statements, but our **seq** funcon for sequencing commands takes only two arguments. In the following equation, we use ‘$\cdots$’ *formally* as a meta-variable ranging over stm^* (possibly-empty sequences of statements).

$$\mathit{comm}\,[\![\, S_1 \;\, S_2 \;\, \cdots \,]\!] = \textbf{seq}(\mathit{comm}\,[\![\, S_1 \,]\!], \mathit{comm}\,[\![\, S_2 \;\, \cdots \,]\!]) \tag{22}$$

To translate a sequence of just two statements, ‘$S_1\ S_2\ \cdots$’ matches ‘$\cdots$’ with the empty sequence, and we can then regard ‘$S_2\ \cdots$’ as a single statement, whose translation is specified by our other equations. To translate a sequence of three or more statements, ‘$S_1\ S_2\ \cdots$’ matches ‘$\cdots$’ with a non-empty sequence, and we can use the above equation recursively to translate ‘$S_2\ \cdots$’. For instance, the above equations translate a sequence of the form ‘$S_1\ S_2\ S_3$’ to a funcon term $\textbf{seq}(C_1, \textbf{seq}(C_2, C_3))$, where each C_i is the translation of the single statement S_i.

We give many further examples of specifying translations from language constructs to funcons in Sect. 3.

2.3 Dynamic Semantics of Funcon Notation

The preceding subsections illustrate how we use sorts and signatures to specify the syntax of funcon notation, and how we specify translation functions that map programs to funcon terms. We now explain and illustrate how to specify the dynamic semantics of funcons, once and for all, using a modular form of operational semantics; Section 2.4 does the same for their static semantics.

Modular SOS (MSOS). Modular SOS [42] is a simple variant of structural operational semantics (SOS) [56]. It allows a particularly high degree of reuse without any need for reformulation. The specification of each language construct in MSOS is independent of the features of the other constructs included in the language. This is achieved by incorporating all auxiliary entities used in transition formulae (environments, stores, etc.) in *labels* (L) on transitions. Thus transition formulae for expressions are always of the form $E \xrightarrow{L} E'$ (and similarly for other sorts of language constructs).

The MSOS notation for labels ensures automatic propagation of all *unmentioned* auxiliary entities between the premise(s) and conclusion of each rule. For this to work, the labels on adjacent steps of a computation are required to be *composable*, and a set of *unobservable* labels is distinguished.[4] This allows the following MSOS rules for the dynamic semantics of conditional expressions '$E_1 \,?\, E_2 : E_3$' to be used both for imperative and for purely functional languages, without any reformulation:

$$\frac{E_1 \xrightarrow{L} E_1'}{(E_1 \,?\, E_2 : E_3) \xrightarrow{L} (E_1' \,?\, E_2 : E_3)} \tag{23}$$

$$(\texttt{true} \,?\, E_2 : E_3) \xrightarrow{\tau} E_2 \tag{24}$$

$$(\texttt{false} \,?\, E_2 : E_3) \xrightarrow{\tau} E_3 \tag{25}$$

The variable τ varies over all *unobservable* labels, whereas L ranges over arbitrary labels. By not mentioning specific auxiliary entities, the rules assume neither their presence nor their absence, ensuring reusability – even when expression evaluation can throw exceptions or spawn concurrent processes. This also makes the rules significantly simpler, and easier to read (some conventional rules in the literature almost hurt ones eyes to look at!) and reduces the likelihood of making clerical mistakes when formulating them.

[4] In fact labels in MSOS are the morphisms of a *category*, and the unobservable labels are identity morphisms, as explained in [42]. However, models of MSOS specifications correspond to ordinary labelled transition systems.

The MSOS rules for assignment expressions are as follows.

$$\frac{E \xrightarrow{L} E'}{(I = E) \xrightarrow{L} (I = E')} \tag{26}$$

$$(I = V) \xrightarrow{\rho,\sigma,\sigma'=\sigma[\rho(I)\mapsto V],\tau} V \tag{27}$$

The notation used on the transition arrow in (27) above indicates that when assignment expressions are included in a language, the labels on transitions are to have at least an environment ρ and a pair of stores σ, σ'. The inclusion of τ in the label specifies that any further components must be unobservable, which is necessary to ensure that executing this simple assignment expression cannot have further effects (e.g., assigning to other variables, printing, or throwing an exception).[5]

If we include the above conditional expressions and assignment expressions in the same language, no changes at all are needed to their MSOS rules – in marked contrast to the weaving required in SOS, as illustrated in Sect. 1.

Implicitly-Modular SOS (I-MSOS) [49] This combines the benefits of MSOS regarding modularity and reusability with the familiar notational style of ordinary SOS: auxiliary entities not actually mentioned in a rule are implicitly propagated between its premise(s) and conclusion, just as in MSOS, but *without* the notational burden of putting an explicit label on every transition relation.

All that is needed is to declare the notation used for the transition formulae being specified (which is in any case normal practice in SOS descriptions of programming languages, e.g. [54]), distinguishing any required auxiliary arguments from the syntactic source and target of transitions. Here, we do this by insisting on some notational conventions commonly followed in SOS:

- Environments ρ (and any other entities that are *preserved* by successive transitions) are written before a turnstile, e.g., $\mathsf{env}\,\rho \vdash \_ \rightarrow \_$.
- Stores σ (and any other entities that can be *updated* by transitions) are written after the syntactic source and target, e.g., $(\_\,, \mathsf{store}\,\sigma) \rightarrow (\_\,, \mathsf{store}\,\sigma')$.
- Signals ε (and any other entities *emitted* by transitions) are written as labels above transition symbols, e.g., $\_ \xrightarrow{\mathsf{exception}\,\varepsilon} \_$.

The entities are tagged with distinct markers (such as env, store and exception) to ensure that they cannot be confused with other entities needed in the same position.

[5] For an assignment that might throw an exception, the corresponding MSOS rule would make explicit the conditions under which that occurs, and incorporate the exception flag in the label.

The I-MSOS rules for conditional expressions can be formulated *exactly* as the SOS rules (1–3) given in Sect. 1. The I-MSOS rules for assignment expressions are as follows.

$$\frac{E \to E'}{(I = E) \to (I = E')} \tag{28}$$

$$\mathsf{env}\,\rho \vdash (I = V, \mathsf{store}\,\sigma) \to (V, \mathsf{store}\,\sigma[\rho(I) \mapsto V]) \tag{29}$$

Notice that entities such as environments ρ and stores σ can (and should!) be omitted whenever a rule does not involve inspecting or updating them.

It is straightforward to generate MSOS rules directly from I-MSOS rules (and label categories from transition formulae declarations). The label patterns in a generated rule involve only those auxiliary entities explicitly mentioned in the original I-MSOS rule. The foundations of MSOS [42], together with its recently developed modular bisimulation theory and congruence format [11], provide correspondingly modular foundations for I-MSOS specifications.

I-MSOS Specifications of Funcons The I-MSOS rules given below specify the dynamic semantics of all the funcons whose signatures are listed in Table 1 (page 7). In these rules, the (optionally subscripted or primed) meta-variables C range over *comms*, D over *decls*, E over *exprs*, V over *values*, and X over arbitrary computations (including their computed values).

When specifying the dynamic semantics of a funcon using small-step I-MSOS rules, the so-called ‘congruence’ rules for evaluation of any lifted arguments can be generated from the signature and left implicit, which dramatically improves the conciseness of our specifications. The elimination of the many tedious congruence rules that would be needed in small-step SOS specifications of funcons is a major advantage of our approach, and the resulting conciseness of small-step I-MSOS specifications is competitive with that of specifications using the popular framework of reduction semantics, based on evaluation contexts [19]. This feature of I-MSOS is closely related to (and was inspired by) the use of strictness annotations in the K framework [58].

The funcon **if-true**(V, X_1, X_2) is generic: X_1, X_2 can be of the same arbitrary computation sort (usually expressions or commands). Its first argument is generally lifted from *booleans* to *exprs*. Since the rule specifying the evaluation of the lifted argument is implied by the signature, only the following two rules need to be explicitly specified.

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \to X_1 \tag{30}$$

$$\textbf{if-true}(\textbf{false}, X_1, X_2) \to X_2 \tag{31}$$

$\textsf{\textbf{seq}}(C, X)$ is generic in its second argument, whereas its first argument is always a command (lifted from the value sort *unit* in the signature). The funcon first executes C. The value **null** is computed by all commands on normal termination, so all we need to specify is that when that has happened, the computation continues with X:

$$\textsf{\textbf{seq}}(\textsf{\textbf{null}}, X) \to X \tag{32}$$

$\textsf{\textbf{effect}}(E)$ is the command funcon which evaluates the expression E and then discards the value. The argument is lifted to expressions from a value sort, so here again the rule specifying the evaluation of the argument is left implicit.

$$\textsf{\textbf{effect}}(V) \to \textsf{\textbf{null}} \tag{33}$$

$\textsf{\textbf{while-true}}(E, C)$ involves repeated evaluation of the expression E, and repeated execution of the command C, so the signature cannot involve lifting from value sorts. The execution of this funcon is specified simply by the obvious unfolding rule, exploiting the existence of the funcons **if-true** and **seq**.[6]

$$\textsf{\textbf{while-true}}(E, C) \to \textsf{\textbf{if-true}}(E, \textsf{\textbf{seq}}(C, \textsf{\textbf{while-true}}(E, C)), \textsf{\textbf{null}}) \tag{34}$$

$\textsf{\textbf{assign}}(E_1, E_2)$ is a command funcon that simply updates the imperative variable V_1 computed by E_1 to the value V_2 computed by E_2.

$$\frac{V_1 \in \textsf{\textbf{dom}}(\sigma)}{(\textsf{\textbf{assign}}(V_1, V_2), \textsf{store}\,\sigma) \to (\textsf{\textbf{null}}, \textsf{store}\,\sigma[V_1 \mapsto V_2])} \tag{35}$$

Notice that the rules above for assignment mention stores σ but not environments ρ. It is characteristic that, in contrast to many language constructs, each funcon generally involves only one kind of auxiliary entity.

The assignment funcon is compatible with shared-memory access by concurrent threads: the steps specified above are atomic updates, and can be serialised.

The expression funcon $\textsf{\textbf{assigned-value}}(E)$ inspects the value currently stored in the variable computed by E, without changing it.

$$(\textsf{\textbf{assigned-value}}(V), \textsf{store}\,\sigma) \to (\sigma(V), \textsf{store}\,\sigma) \tag{36}$$

$\textsf{\textbf{bind-value}}(I, E)$ is a declaration funcon used to compute the single-point environment that maps I to the value of E.

$$\textsf{\textbf{bind-value}}(I, V) \to \{I \mapsto V\} \tag{37}$$

$\textsf{\textbf{bound-value}}(I)$ is an expression funcon that inspects the value currently bound to the identifier I; the result is undefined (and the rule inapplicable, which would lead to a stuck computation) if I is not in the domain of the current environment ρ.

$$\textsf{env}\,\rho \vdash \textsf{\textbf{bound-value}}(I) \to \rho(I) \tag{38}$$

[6] In small-step semantics, the use of auxiliary funcons for specifying **while-true** appears to be unavoidable.

$\textsf{\textbf{scope}}(D, X)$ executes the declaration D to compute an environment ρ_1, then binds the identifiers in the domain of ρ_1 locally in the computation X, letting these bindings override the bindings represented by the current environment ρ. This funcon is lifted in its first argument, whereas the rule for the computation of its second argument has to be explicitly specified, since the environment is not merely propagated. Rule (40) applies only when V is a value, which is independent of the current context.

$$\frac{\textsf{env}\,(\rho_1/\rho) \vdash X \to X'}{\textsf{env}\,\rho \vdash \textsf{\textbf{scope}}(\rho_1, X) \to \textsf{\textbf{scope}}(\rho_1, X')} \tag{39}$$

$$\textsf{\textbf{scope}}(\rho_1, V) \to V \tag{40}$$

given is an expression which gives the value computed by the closest-enclosing **supply**. The rules specifying these funcons below are essentially simplified versions of the above rules for **bound-value** and **scope**, with **given** corresponding to the value currently bound to a fixed pseudo-identifier, propagated by a corresponding auxiliary entity.

$$\textsf{given}\,V \vdash \textsf{\textbf{given}} \to V \tag{41}$$

$$\frac{\textsf{given}\,V \vdash X \to X'}{\textsf{given}\,\_ \vdash \textsf{\textbf{supply}}(V, X) \to \textsf{\textbf{supply}}(V, X')} \tag{42}$$

$$\textsf{\textbf{supply}}(V_1, V_2) \to V_2 \tag{43}$$

This concludes the specification of the dynamic semantics of all the funcons whose signatures are shown in Table 1. The rules have been validated indirectly: by generating Prolog clauses from them, then using those clauses to execute programs in various languages according to their translations to funcons, as described in Sect. 4.

Most of the funcons specified above are (re)used in the Caml Light case study presented in Sect. 3, together with some more advanced funcons involving abstractions, patterns and exceptions. Before that, let us see how to specify the static semantics of funcons.

2.4 Static Semantics of Funcon Notation

For a program in some programming language, its static semantics represents analysis that is supposed to be done on the (parsed) program text before running the program. In many languages, the scopes of identifier bindings are determined by the structure of the program, and the required analysis checks that there are no unbound occurrences. Such languages may also be statically typed, i.e., the type of values potentially computed by each expression in the program can be determined, and checked to be consistent with the type of values required by the context of the expression. When the types of identifiers are not given explicitly in the program, the analysis needs to infer them. The outcome of the analysis of an entire program is either its type, or an indication that some part of it is not well-typed.

Running a program usually involves executing only certain parts of it, in a particular order, possibly with iteration, procedure activation, abrupt termination, etc. Small-step (I-M)SOS is particularly well-suited to specifying the dynamic semantics of such programs. In contrast, static analysis of a program generally involves the analysis of each of its parts just once, in no particular order, to carry out all the required checks; this motivates the use of the big-step style of (I-M)SOS for specifying static semantics [27].

The static semantics of a language is specified by a *typing relation* between expressions E, types T, and *typing contexts* Γ (mapping identifiers to their types), conventionally written '$\Gamma \vdash E : T$'; further arguments of the typing relation (e.g., store types, type variable assignments) may be introduced, if needed. We can informally read the relation as saying that expression E has type T in context Γ. When the typing relation is *sound* in relation to evaluation, this means that E can only compute a value of type T when the environment can be typed by Γ. An environment ρ is typed by Γ whenever Γ maps each identifier I in the domain of ρ to the type of the value $\rho(I)$. The typing relation can be inductively specified using *typing rules* [54], which are similar in form to big-step SOS rules. Using I-MSOS, we can formulate our typing rules omitting auxiliary entities whenever they merely need to be propagated.

I-MSOS Specifications of Funcons The I-MSOS rules given below specify the static semantics of all the funcons whose signatures are listed in Table 1 (page 7). The meta-variable T ranges over the value sort *types*, which provides notation for type constants and constructors independently of language syntax.

The funcon **if-true**(E, X_1, X_2) requires E to have type *booleans*, and X_1 and X_2 to have the same arbitrary type T, all in the same typing context Γ (which I-MSOS lets us leave implicit):

$$\frac{E : \mathit{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true}(E, X_1, X_2) : T} \tag{44}$$

seq(C, X) requires C to be a well-typed command, which corresponds to it having the singleton type *unit*. The funcon then has the same type as X:

$$\frac{C : \mathit{unit} \qquad X : T}{\textbf{seq}(C, X) : T} \tag{45}$$

effect(E) merely requires E to be a well-typed expression:

$$\frac{E : T}{\textbf{effect}(E) : \mathit{unit}} \tag{46}$$

while-true(E, C) requires E to have type *booleans*:

$$\frac{E : \mathit{booleans} \qquad C : \mathit{unit}}{\textbf{while-true}(E, C) : \mathit{unit}} \tag{47}$$

An imperative variable for storing values of a specific type T has the type $\mathit{variables}(T)$. The funcon command $\mathbf{assign}(E_1, E_2)$ requires the types of E_1 and E_2 to match accordingly:

$$\frac{E_1 : \mathit{variables}(T) \qquad E_2 : T}{\mathbf{assign}(E_1, E_2) : \mathit{unit}} \tag{48}$$

$\mathbf{assigned\text{-}value}(E)$ allows E to have any variable type:

$$\frac{E : \mathit{variables}(T)}{\mathbf{assigned\text{-}value}(E) : T} \tag{49}$$

For languages where identifiers might be bound to constant values as well as to variables, whether to use **assigned-value** or not depends on the type of the identifier. Assuming that the types of identifiers are statically determined, the static semantics of funcons could subsequently eliminate irrelevant alternatives.[7]

$\mathbf{bind\text{-}value}(I, E)$ is a declaration, and its type is the single-point typing context that maps I to the type of E:

$$\frac{E : T}{\mathbf{bind\text{-}value}(I, E) : \{I \mapsto T\}} \tag{50}$$

$\mathbf{bound\text{-}value}(I)$ has the type determined by the current typing context Γ (which I-MSOS allowed us to leave implicit in all the above rules). If I is not in the domain of Γ, $\Gamma(I)$ is undefined, and the rule cannot be applied.

$$\mathsf{env}\,\Gamma \vdash \mathbf{bound\text{-}value}(I) : \Gamma(I) \tag{51}$$

$\mathbf{scope}(D, X)$ adjusts the typing context used for X to account for the local bindings computed by D:

$$\frac{\mathsf{env}\,\Gamma \vdash D : \Gamma_1 \qquad \mathsf{env}(\Gamma_1/\Gamma) \vdash X : T}{\mathsf{env}\,\Gamma \vdash \mathbf{scope}(D, X) : T} \tag{52}$$

It is important here that we require environments to map identifiers to *values* (not computations), and for our values to be context-free. These requirements suffice to ensure that the typeability of a dynamic environment is preserved by overriding. Consequently, given this typing rule for **scope**, the corresponding dynamic rules (39, 40) are safe with respect to type preservation.

The static semantics of **given** and **supply** is related to that of **bound-value** and **scope** in the same way as their dynamic semantics. It involves the introduction of a read-only entity that shows the type of value provided by **supply**.

$$\mathsf{given}\,T \vdash \mathbf{given} : T \tag{53}$$

$$\frac{\mathsf{given}\,T_1 \vdash E : T \qquad \mathsf{given}\,T \vdash X : T'}{\mathsf{given}\,T_1 \vdash \mathbf{supply}(E, X) : T'} \tag{54}$$

[7] Static semantics sometimes requires such so-called partial evaluation.

This concludes the specification of the static semantics of all the funcons whose signatures are shown in Table 1. Under these static semantics, each dynamic rule is safe from the point of view of type preservation. Therefore, the typing rules for the funcons that we have considered in this section are sound in relation to their dynamic semantics: when a funcon term has type T in a typing context Γ, the values it computes on normal termination in an environment typed by Γ are always of type T.

3 An Illustrative Case Study: Caml Light

Caml Light descends from Caml, a predecessor of the language OCaml, and is similar to the core of Standard ML [38]. It has first-class functions, assignable state, exception handling mechanisms, and pattern matching. It is statically typed, and supports algebraic data types and polymorphism.

The syntax and semantics of Caml Light are specified in its reference manual [32]. It contains a formal context-free grammar of 'concrete abstract syntax': this generates Caml Light programs, but disambiguation details are abstracted away. However, the explanation it gives of the intended semantics of Caml Light programs is completely informal.

In this section, after introducing the syntax of Caml Light, we illustrate our approach by presenting excerpts from a component-based semantics of the language. Section 3.1 gives an overview of the required values and funcons; Sect. 3.2 gives examples of specifying the translation of Caml Light into combinations of funcons; Sect. 3.3 specifies the dynamic semantics of the funcons; Sect. 3.4 specifies their static semantics; and Sect. 3.5 specifies the translation of Caml Light constructs that involve funcons which only have static significance. The complete specification of the translation of Caml Light to funcons is provided in the Appendix, and the sources of the full specifications can be found online [12].

Caml Light is a language built around *expressions* which compute values, including numbers, strings, function abstractions, tuples and lists. Commands (or statements) are not a separate syntactic category, but rather expressions that compute a particular null value, written `()`. Expressions are given a type, which includes ground types (e.g. `int`), tuple types (e.g. `int*int`) and function types (e.g. `int->int`). Commands and `()` have type `unit`.

Some example Caml Light programs can be found in Table 2. First, we see a recursively defined Fibonacci function `fib`. The function is defined using the `function` constructor, introducing a closed function abstraction. Identifiers may be bound to particular values within an expression using `let` bindings, and recursive functions using the `let rec` construct. Formal arguments can also appear as parameters before the '`=`', as in the definitions of `append` and `insertion_sort`.

As well as expressions, values and types, Caml Light supports matching values against *patterns* which bind identifiers. This is demonstrated in the `append` example, where the first argument `zs` is matched against two patterns: the empty

```
let rec fib = function n ->
  if n < 2 then n else fib(n-1) + fib(n-2) ;;

let rec append zs ys =
  match zs with
  | []    -> ys
  | x::xs -> x :: (append xs ys) ;;

let insertion_sort a =
  for i = 1 to vect_length a - 1 do
    let val_i = a.(i) in
    let j = ref i in
    while !j > 0 & val_i < a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      j := !j - 1
    done;
    a.(!j) <- val_i
  done;;
```

Table 2. Some example Caml Light programs

list `[]`, and the list-constructor pattern `x::xs`, which binds `x` to the head and `xs` to the tail of a nonempty list.

Caml Light also supports imperative behaviour, as can be seen in the `insertion_sort` example, acting on an array. Arrays are mutable: their content may be updated. A single assignable reference cell is constructed using `ref`, and it may be accessed using explicit dereferencing '`!`' and updated using '`:=`'. In this example we also see two different looping constructs.

An extract of the Caml Light reference grammar [32] is given in Table 3.

3.1 Further Funcon Notation

In Sect. 2, we introduced some basic funcons for commands, declarations and expressions. We next consider the further funcons used in our Caml Light case study, involving abstractions, patterns and exception handling. They are listed in Table 4, with their signatures. We discuss their semantics informally, focusing on dynamic semantics; see Sects. 3.3 and 3.4 for their formal specifications.

Abstractions. A value of sort *funcs* is an abstraction encapsulating an expression that computes a value which may depend on the value of an argument supplied by application. Dependence of the expression on the current environment may occur only when its execution is forced (by applying the abstraction)

Lexical syntax

$$\begin{array}{rl}
I : & \textit{ident} \\
\textit{Int} : & \textit{integer-literal} \\
\textit{Float} : & \textit{float-literal} \\
\textit{Char} : & \textit{char-literal} \\
\textit{String} : & \textit{string-literal}
\end{array}$$

Context-free syntax

$$\begin{array}{rl}
T : \textit{typexpr} ::= & \texttt{'}\ \textit{ident} \mid \textit{typexpr}\ \texttt{->}\ \textit{typexpr} \mid \textit{typexpr}\ (\texttt{*}\ \textit{typexpr})^{+} \\
& \mid \textit{typeconstr} \mid \textit{typexpr}\ \textit{typeconstr} \\
& \mid \texttt{(}\ \textit{typexpr}\,(\texttt{,}\ \textit{typexpr})^{*}\ \texttt{)}\ \ \textit{typeconstr} \\
C : \textit{constant} ::= & \textit{integer-literal} \mid \textit{float-literal} \mid \textit{char-literal} \mid \textit{string-literal} \\
& \mid \texttt{false} \mid \texttt{true} \mid \texttt{[]} \mid \texttt{()} \\
P : \textit{pattern} ::= & \textit{ident} \mid \texttt{\_} \mid \textit{pattern}\ \texttt{as}\ \textit{ident} \\
& \mid \texttt{(}\ \textit{pattern}\ \texttt{)} \mid \texttt{(}\ \textit{pattern} : \textit{typexpr}\ \texttt{)} \\
& \mid \textit{pattern}\,\texttt{|}\,\textit{pattern} \mid \textit{constant} \mid \textit{pattern}\ (\texttt{,}\ \textit{pattern})^{+} \\
& \mid \texttt{[]} \mid \texttt{[}\,\textit{pattern}\ (\texttt{;}\ \textit{pattern})^{*}\ \texttt{]} \mid \textit{pattern}\ \texttt{::}\ \textit{pattern} \\
E : \textit{exp} ::= & \textit{ident} \mid \textit{constant} \mid \texttt{(}\ \textit{exp}\ \texttt{)} \mid \texttt{begin}\ \textit{exp}\ \texttt{end} \\
& \mid \texttt{(}\ \textit{exp} : \textit{typexpr}\ \texttt{)} \mid \textit{exp}\ (\texttt{,}\ \textit{exp})^{+} \\
& \mid \textit{exp}\ \texttt{::}\ \textit{exp} \mid \texttt{[}\ \textit{exp}\ (\texttt{;}\ \textit{exp})^{*}\ \texttt{]} \mid \texttt{[|}\ \textit{exp}\ (\texttt{;}\ \textit{exp})^{*}\ \texttt{|]} \\
& \mid \textit{exp}\ \textit{exp} \mid \textit{prefix-op}\ \textit{exp} \mid \textit{exp}\ \textit{infix-op}\ \textit{exp} \\
& \mid \texttt{not}\ \textit{exp} \mid \textit{exp}\ \texttt{\&}\ \textit{exp} \mid \textit{exp}\ \texttt{or}\ \textit{exp} \\
& \mid \textit{exp}\ \texttt{.(}\ \textit{exp}\ \texttt{)} \mid \textit{exp}\ \texttt{.(}\ \textit{exp}\ \texttt{)}\ \texttt{<-}\ \textit{exp} \\
& \mid \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{exp}\ (\texttt{else}\ \textit{exp})^{?} \\
& \mid \texttt{while}\ \textit{exp}\ \texttt{do}\ \textit{exp}\ \texttt{done} \\
& \mid \texttt{for}\ \textit{ident}\ \texttt{=}\ \textit{exp}\ (\texttt{to} \mid \texttt{downto})\ \textit{exp}\ \texttt{do}\ \textit{exp}\ \texttt{done} \\
& \mid \textit{exp}\ \texttt{;}\ \textit{exp} \\
& \mid \texttt{match}\ \textit{exp}\ \texttt{with}\ \textit{simple-matching} \\
& \mid \texttt{function}\ \textit{simple-matching} \\
& \mid \texttt{try}\ \textit{exp}\ \texttt{with}\ \textit{simple-matching} \\
& \mid \texttt{let}\ (\texttt{rec})^{?}\ \textit{let-binding}\ (\texttt{and}\ \textit{let-binding})^{*}\ \texttt{in}\ \textit{exp} \\
\textit{SM} : \textit{simple-matching} ::= & \textit{pattern}\ \texttt{->}\ \textit{exp}\ (\texttt{|}\ \textit{pattern}\ \texttt{->}\ \textit{exp})^{*} \\
\textit{LB} : \textit{let-binding} ::= & \textit{pattern}\ \texttt{=}\ \textit{exp}
\end{array}$$

Table 3. An extract of the Caml Light reference grammar [32], with EBNF replaced by $(\cdot)^{*}$, $(\cdot)^{+}$, $(\cdot)^{?}$ and the nonterminal *expr* renamed to *exp*

Abstraction sorts

$$\begin{aligned}
&\mathit{funcs} = \mathit{abs}(\mathit{values}, \mathit{values})\\
&\mathit{patts} = \mathit{abs}(\mathit{values}, \mathit{environments})
\end{aligned}$$

Funcon and abstraction signatures

$$\begin{aligned}
\textsf{\textbf{abs}}(\mathit{exprs}) &: \mathit{funcs}\\
\textsf{\textbf{any}} &: \mathit{patts}\\
\textsf{\textbf{apply}}(\mathit{funcs}, \mathit{values}) &: \mathit{exprs}\\
\textsf{\textbf{bind}}(\mathit{ids}) &: \mathit{patts}\\
\textsf{\textbf{catch}}(\mathit{exprs}, \mathit{funcs}) &: \mathit{exprs}\\
\textsf{\textbf{catch-else-rethrow}}(\mathit{exprs}, \mathit{funcs}) &: \mathit{exprs}\\
\textsf{\textbf{close}}(\mathit{funcs}) &: \mathit{exprs}\\
\textsf{\textbf{closure}}(\mathit{computes}(X), \mathit{environments}) &: \mathit{computes}(X)\\
\textsf{\textbf{else}}(\mathit{computes}(X), \mathit{computes}(X)) &: \mathit{computes}(X)\\
\textsf{\textbf{fail}} &: \mathit{computes}(X)\\
\textsf{\textbf{match}}(\mathit{values}, \mathit{patts}) &: \mathit{decls}\\
\textsf{\textbf{only}}(\mathit{values}) &: \mathit{patts}\\
\textsf{\textbf{patt-abs}}(\mathit{patts}, \mathit{exprs}) &: \mathit{funcs}\\
\textsf{\textbf{patt-union}}(\mathit{patts}, \mathit{patts}) &: \mathit{patts}\\
\textsf{\textbf{prefer-over}}(\mathit{abs}(X, Y), \mathit{abs}(X, Y)) &: \mathit{abs}(X, Y)\\
\textsf{\textbf{throw}}(\mathit{values}) &: \mathit{computes}(X)
\end{aligned}$$

Table 4. Funcon signatures (see also Table 1)

or when a closure is formed (by copying the current environment into the expression). Static scoping is obtained by computing the closure of each abstraction when it is created; application of abstractions otherwise gives dynamic scoping.

Abstractions A can be constructed using **patt-abs**(P, E), which abstracts an expression E over a pattern P. When no matching of the given value is needed, **abs**(E) allows E to refer to it using the funcon **given**. Abstractions can be turned into self-contained function closures using the **close** funcon, to ensure static scoping. Abstractions may be applied to argument values using the **apply** funcon. An application of the abstraction **prefer-over**(A_1, A_2) applies A_1 to the given argument value, applying A_2 only if that *fails*. (Failure is a kind of abrupt termination, considered in more detail in Sect. 3.3.)

Patterns. A value of sort *patts* is an abstraction encapsulating a declaration that computes an environment from a given value. An example pattern is **any**, which matches any value and produces no bindings, modelling the '_' wildcard in Caml Light. The funcon **only** takes a value and matches just that value, again producing no bindings. The pattern **bind**(I) matches any value, and binds I to it. Compound patterns may be constructed out of more primitive patterns.

Exceptions. The computation **throw**(V) terminates abruptly, and so can be seen to compute a value of any sort, vacuously. The **catch** funcon handles abrupt termination of its first argument by applying a function to the thrown value. The **catch-else-rethrow** funcon abbreviates a variant on this: it rethrows the exception should it fail to be in the domain of the handler.

3.2 Caml Light Semantics

We translate Caml Light (abstract syntax trees) into funcon terms. The signatures of the translation functions are listed in Table 5. For Caml Light, computed values include ground constants (integers, Booleans, strings, floats, chars) as well as records (maps, wrapped in a data constructor), variants for disjoint unions (a single value tagged with a constructor), tuples, and function abstractions.

Semantics

$$\begin{aligned}
\mathsf{\textit{id}}[\![\_ : \textit{ident}]\!] &: \mathsf{\textit{ids}} \\
\mathsf{\textit{type}}[\![\_ : \textit{typexpr}]\!] &: \mathsf{\textit{types}} \\
\mathsf{\textit{value}}[\![\_ : \textit{constant}]\!] &: \mathsf{\textit{values}} \\
\mathsf{\textit{patt}}[\![\_ : \textit{pattern}]\!] &: \mathsf{\textit{patts}} \\
\mathsf{\textit{expr}}[\![\_ : \textit{exp}]\!] &: \mathsf{\textit{exprs}} \\
\mathsf{\textit{func}}[\![\_ : \textit{simple-matching}]\!] &: \mathsf{\textit{funcs}} \\
\mathsf{\textit{decl}}[\![\_ : \textit{let-binding}]\!] &: \mathsf{\textit{decls}}
\end{aligned}$$

Table 5. Translation function signatures

We next show some of the equations specifying the translation of Caml Light programs to funcon terms. We will first consider dynamic semantics, specifying a translation which captures the intended runtime behaviour. Often, this translation will also capture the static semantics correctly (since each funcon by design has a natural combination of dynamic and static semantics). When it does not, we need to add funcons to the translation to reflect the intended compile-time behaviour, as we illustrate in Sect. 3.5.

Conditional. Caml Light's conditional construct on Booleans is translated straightforwardly into the **if-true** funcon we have already seen:

$$\mathit{expr}[\![\, \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \,]\!] = \\ \quad \textbf{if-true}(\mathit{expr}[\![\, E_1 \,]\!], \mathit{expr}[\![\, E_2 \,]\!], \mathit{expr}[\![\, E_3 \,]\!]) \tag{55}$$

Here we are lifting the funcon **if-true** in the first argument to computations that *might* compute a Boolean, and similarly when lifting is applied to pure data operations, such as **not**. The static semantics of the translation of a complete program to funcons checks that the arguments are in fact of type *booleans*.

$$\mathit{expr}[\![\, \texttt{not}\ E \,]\!] = \textbf{not}(\mathit{expr}[\![\, E \,]\!]) \tag{56}$$

We also use the **if-true** funcon in the translation of other Caml Light constructs, such as the conditional conjunction operator:

$$\mathit{expr}[\![\, E_1\ \texttt{\&}\ E_2 \,]\!] = \textbf{if-true}(\mathit{expr}[\![\, E_1 \,]\!], \mathit{expr}[\![\, E_2 \,]\!], \textbf{false}) \tag{57}$$

Sequencing. The sequencing construct of Caml Light is translated as follows:

$$\mathit{expr}[\![\, E_1\ \texttt{;}\ E_2 \,]\!] = \textbf{seq}(\textbf{effect}(\mathit{expr}[\![\, E_1 \,]\!]), \mathit{expr}[\![\, E_2 \,]\!]) \tag{58}$$

Here, we explicitly discard the computed value of the first expression.

Pattern Matching. We translate Caml Light's simple matching construct *SM* to a function abstraction using *func*⟦ _ ⟧. Our analysis of a match expression is as an application of such an abstraction to the matched expression, inserting **prefer-over** to take into account what happens when the pattern fails to match the given value:

$$\mathit{expr}[\![\, \texttt{match}\ E\ \texttt{with}\ \mathit{SM} \,]\!] = \\ \quad \textbf{apply}(\textbf{prefer-over}(\mathit{func}[\![\, \mathit{SM} \,]\!], \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure}))), \\ \qquad \mathit{expr}[\![\, E \,]\!]) \tag{59}$$

The funcon **cl-match-failure** is Caml Light-specific, and is defined simply as a convenient abbreviation for the (translated) `Match_failure` constructor of Caml Light's built in `exn` type.

Function Application. The funcon **apply** corresponds directly to Caml Light's call-by-value function application:

$$\mathit{expr}[\![\, E_1\ E_2 \,]\!] = \textbf{apply}(\mathit{expr}[\![\, E_1 \,]\!], \mathit{expr}[\![\, E_2 \,]\!]) \tag{60}$$

The signature of **apply** indicates that it should be applied to a function abstraction and an argument *value*; it is lifted here to computations. We would specify call-by-name semantics by forming a (parameterless closed) abstraction from the argument expression, to prevent its premature evaluation.

Function Abstraction. Caml Light is a functional language, and we represent functions as abstraction values.

$$\mathit{expr}[\![\, \texttt{function}\, SM \,]\!] = \qquad (61)$$
$$\quad \textbf{prefer-over}(\mathit{func}[\![\, SM \,]\!], \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure})))$$

Simple Matchings. We will next see how *func*⟦ _ ⟧ translates simple matchings to abstractions. For a single body, the **patt-abs** funcon captures matchings accurately; sequences of simple matchings are combined using **prefer-over**. We use the **close** funcon to specify static bindings.

$$\mathit{func}[\![\, P \;\texttt{->}\; E \,]\!] = \textbf{close}(\textbf{patt-abs}(\mathit{patt}[\![\, P \,]\!], \mathit{expr}[\![\, E \,]\!])) \qquad (62)$$

$$\mathit{func}[\![\, P \;\texttt{->}\; E \;\texttt{|}\; SM \,]\!] = \textbf{prefer-over}(\mathit{func}[\![\, P \;\texttt{->}\; E \,]\!], \mathit{func}[\![\, SM \,]\!]) \qquad (63)$$

Declarations. Local declarations are provided in Caml Light by the ‘`let-in`’ construct, corresponding to the **scope** funcon:

$$\mathit{expr}[\![\, \texttt{let}\, LB \;\; \texttt{in}\, E \,]\!] = \textbf{scope}(\mathit{decl}[\![\, LB \,]\!], \mathit{expr}[\![\, E \,]\!]) \qquad (64)$$

Let-bindings are translated to declarations.

$$\mathit{decl}[\![\, P \,\texttt{=}\, E \,]\!] = \qquad (65)$$
$$\quad \textbf{match}(\mathit{expr}[\![\, E \,]\!],$$
$$\qquad \textbf{prefer-over}(\mathit{patt}[\![\, P \,]\!], \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure}))))$$

An identifier expression refers to its bound value.

$$\mathit{expr}[\![\, I \,]\!] = \textbf{bound-value}(\mathit{id}[\![\, I \,]\!]) \qquad (66)$$

The preceding two equations account for dynamic semantics. To accurately model Caml Light’s let-polymorphism, further details are required, which we outline in Sect. 3.5 (113).

Catching Exceptions. Caml Light’s try construct corresponds directly to the **catch-else-rethrow** funcon:

$$\mathit{expr}[\![\, \texttt{try}\, E \,\texttt{with}\, SM \,]\!] = \textbf{catch-else-rethrow}(\mathit{expr}[\![\, E \,]\!], \mathit{func}[\![\, SM \,]\!]) \qquad (67)$$

Also here, further details are required to capture Caml Light’s static semantics, see Sect. 3.5 (112).

Basic Patterns. We have notation corresponding directly to basic patterns.

$$\mathit{patt}[\![\, I \,]\!] = \textbf{bind}(\mathit{id}[\![\, I \,]\!]) \qquad (68)$$
$$\mathit{patt}[\![\, \texttt{\_} \,]\!] = \textbf{any} \qquad (69)$$
$$\mathit{patt}[\![\, C \,]\!] = \textbf{only}(\mathit{value}[\![\, C \,]\!]) \qquad (70)$$

Compound Data. Caml Light expressions include tupling. We represent tuple values using the **tuple-empty** and binary **tuple-prefix** data constructors. These are lifted to computations in the usual way. We use a small auxiliary translation function *expr-tuple*⟦ _ ⟧:

$$\mathit{expr}\llbracket\, E_1 \;\mathbf{,}\; E_2 \;\cdots \rrbracket = \mathit{expr\text{-}tuple}\llbracket\, E_1 \;\mathbf{,}\; E_2 \;\cdots \rrbracket \tag{71}$$

$$\mathit{expr\text{-}tuple}\llbracket\, E \,\rrbracket = \textbf{tuple-prefix}(\mathit{expr}\llbracket\, E \,\rrbracket, \textbf{tuple-empty}) \tag{72}$$

$$\mathit{expr\text{-}tuple}\llbracket\, E_1 \;\mathbf{,}\; E_2 \;\cdots \rrbracket = \textbf{tuple-prefix}(\mathit{expr}\llbracket\, E_1 \,\rrbracket, \mathit{expr\text{-}tuple}\llbracket\, E_2 \;\cdots \rrbracket) \tag{73}$$

Compound Patterns. Patterns may also be combined using sequential choice, reusing the **prefer-over** funcon.

$$\mathit{patt}\llbracket\, P_1 \mid P_2 \,\rrbracket = \textbf{prefer-over}(\mathit{patt}\llbracket\, P_1 \,\rrbracket, \mathit{patt}\llbracket\, P_2 \,\rrbracket) \tag{74}$$

One may also bind an identifier to the value matched by a pattern:

$$\mathit{patt}\llbracket\, P \;\mathsf{as}\; I \,\rrbracket = \textbf{patt-union}(\mathit{patt}\llbracket\, P \,\rrbracket, \textbf{bind}(\mathit{id}\llbracket\, I \,\rrbracket)) \tag{75}$$

Built-In Operators. In Caml Light, many built-in operators (e.g., assignment, dereferencing, allocation, and raising exceptions) are provided in the initial library as identifiers bound to functions (and may be rebound in programs). We reflect this by using the funcon **scope** to provide an initial environment to the translations of entire Caml Light programs.

3.3 Dynamic Semantics of Further Funcon Notation

In Sect. 2.3 we explained and illustrated how to define the dynamic semantics of the simple funcons introduced in Sect. 2.1. We now define the dynamic semantics of the further funcons introduced in Sect. 3.1, which involve abstractions, patterns and exceptions. See Table 4 (page 21) for the signatures of these funcons.

Abstractions. An abstraction **abs**(X) is a value constructed from a computation X that may depend on a given argument value. The funcon **apply** takes a computed abstraction value **abs**(X) and an argument value V:

$$\textbf{apply}(\textbf{abs}(X), V) \to \textbf{supply}(V, X) \tag{76}$$

(The funcon **supply** was introduced in Sect. 2.) The **apply** funcon is lifted in both arguments.

When an abstraction **abs**(X) is applied, evaluation of **bound-value**(I) in X gives the value *currently* bound to I, which corresponds to dynamic scopes for non-local bindings. To specify static scoping, we use the **close** funcon, which takes an abstraction and returns a closure formed from it and the current environment.

$$\mathsf{env}\ \rho \vdash \textbf{close}(\textbf{abs}(X)) \rightarrow \textbf{abs}(\textbf{closure}(X, \rho)) \tag{77}$$

The auxiliary funcon **closure** (not used when specifying translations) sets the current environment for any computation X:

$$\frac{\mathsf{env}\ \rho \vdash X \rightarrow X'}{\mathsf{env}\ \_ \vdash \textbf{closure}(X, \rho) \rightarrow \textbf{closure}(X', \rho)} \tag{78}$$

$$\textbf{closure}(V, \rho) \rightarrow V \tag{79}$$

Thus, whether a language is statically or dynamically scoped may be specified in its translation to funcons simply by the presence or absence of the **close** funcon when forming abstractions.

Patterns. A pattern can be seen as a form of abstraction: while a function computes a value depending on a given value, a pattern computes an *environment* depending on a given value. Matching the value of an expression E to a pattern P computes an environment. It corresponds to the application of P to E:

$$\textbf{match}(E, P) \rightarrow \textbf{apply}(P, E) \tag{80}$$

The funcon **patt-abs**(P, X) is similar to **abs**(X), except that it takes also a pattern P that is matched against the given value to compute an environment. This allows nested abstractions to refer to arguments at different levels, using the identifiers bound by the respective patterns. The following rule defines the dynamic semantics of **patt-abs** using the **abs** constructor:

$$\textbf{patt-abs}(P, X) \rightarrow \textbf{abs}(\textbf{scope}(\textbf{match}(\textbf{given}, P), X)) \tag{81}$$

Patterns may be constructed in various ways. For example, the pattern **bind**(I) matches any value and binds the identifier I to it:

$$\textbf{bind}(I) \rightarrow \textbf{abs}(\textbf{bind-value}(I, \textbf{given})) \tag{82}$$

The wildcard pattern **any** also matches any value, but computes the empty environment:

$$\textbf{any} \rightarrow \textbf{abs}(\emptyset) \tag{83}$$

Other patterns do not match all values. An extreme example is the pattern **only**(V), matching just the single value V or executing the funcon **fail**, which is defined below (87).

$$\textbf{only}(V) \rightarrow \textbf{abs}(\textbf{if-true}(\textbf{equal}(\textbf{given}, V), \emptyset, \textbf{fail})) \tag{84}$$

The definition of the operation **prefer-over** on abstractions and (as a special case) on patterns involves the funcon **else**, which is defined below (88–90).

$$\textsf{\textbf{prefer-over}}(\textsf{\textbf{abs}}(X), \textsf{\textbf{abs}}(Y)) \to \textsf{\textbf{abs}}(\textsf{\textbf{else}}(X, Y)) \tag{85}$$

For patterns, **prefer-over** corresponds to ordered *alternatives*, as found in Caml Light.

Another way to combine two patterns, also found in Caml Light, is *conjunctively*, requiring them both to match, and uniting their bindings. This corresponds to the funcon **patt-union**:

$$\textsf{\textbf{patt-union}}(\textsf{\textbf{abs}}(X), \textsf{\textbf{abs}}(Y)) \to \textsf{\textbf{abs}}(\textsf{\textbf{map-union}}(X, Y)) \tag{86}$$

Here, the data operation **map-union** is lifted to computations.

Failure and Back-Tracking. The funcon **fail** emits the signal ‘failure **true**’ and then makes a transition to the funcon **stuck** (which has no further transitions).

$$\textsf{\textbf{fail}} \xrightarrow{\textsf{failure}\,\textsf{\textbf{true}}} \textsf{\textbf{stuck}} \tag{87}$$

The funcon **else** allows recovery from failure. The signal ‘failure **false**’ indicates that the computation is proceeding normally, and is treated as unobservable.

$$\frac{X \xrightarrow{\textsf{failure}\,\textsf{\textbf{false}}} X'}{\textsf{\textbf{else}}(X, Y) \xrightarrow{\textsf{failure}\,\textsf{\textbf{false}}} \textsf{\textbf{else}}(X', Y)} \tag{88}$$

$$\frac{X \xrightarrow{\textsf{failure}\,\textsf{\textbf{true}}} X'}{\textsf{\textbf{else}}(X, Y) \xrightarrow{\textsf{failure}\,\textsf{\textbf{false}}} Y} \tag{89}$$

$$\textsf{\textbf{else}}(V, Y) \xrightarrow{\textsf{failure}\,\textsf{\textbf{false}}} V \tag{90}$$

Exceptions. We specify exception throwing and handling in a modular way using the emitted signals ‘exception **some**(V)’ and ‘exception **none**’ (the latter is unobservable).

$$\textsf{\textbf{throw}}(V) \xrightarrow{\textsf{exception}\,\textsf{\textbf{some}}(V)} \textsf{\textbf{stuck}} \tag{91}$$

If the first argument of the funcon **catch** signals an exception **some**(V), it applies its second argument (an abstraction) to V.

$$\frac{X \xrightarrow{\textsf{exception}\,\textsf{\textbf{none}}} X'}{\textsf{\textbf{catch}}(X, Y) \xrightarrow{\textsf{exception}\,\textsf{\textbf{none}}} \textsf{\textbf{catch}}(X', Y)} \tag{92}$$

$$\frac{X \xrightarrow{\textsf{exception}\,\textsf{\textbf{some}}(V)} X'}{\textsf{\textbf{catch}}(X, Y) \xrightarrow{\textsf{exception}\,\textsf{\textbf{none}}} \textsf{\textbf{apply}}(Y, V)} \tag{93}$$

$$\textsf{\textbf{catch}}(V, Y) \xrightarrow{\textsf{exception}\,\textsf{\textbf{none}}} V \tag{94}$$

The following funcon abbreviates a useful variant of **catch**: exceptions are propagated when the application of the abstraction to them fails.

$$\begin{aligned}&\textsf{\textbf{catch-else-rethrow}}(E, A) \rightarrow\\&\quad\textsf{\textbf{catch}}(E, \textsf{\textbf{prefer-over}}(A, \textsf{\textbf{abs}}(\textsf{\textbf{throw}}(\textsf{\textbf{given}}))))\end{aligned} \tag{95}$$

For funcons whose I-MSOS rules do not mention the exception entity, exceptions are implicitly propagated to the closest enclosing funcon that can handle them. When the translation of a program to funcons involves **throw**, it needs to be enclosed in **catch**, to ensure that (otherwise-)unhandled exceptions cause abrupt termination.

3.4 Static Semantics of Further Funcon Notation

In Sect. 2.4 we explained and illustrated how to define the static semantics of the simple funcons introduced in Sect. 2.1. We now define the static semantics of the further funcons introduced in Sect. 3.1, complementing the dynamic semantics defined in Sect. 3.3. See Table 4 (page 21) for the signatures of these funcons.

Abstractions. As mentioned in Sect. 3.3, an abstraction **abs**(X) has dynamic scopes for its non-local bindings. When the abstraction is applied, the computation X is forced, and these bindings have to be provided by the context of the application.[8] The type of **abs**(X) is of the form *abs*(Γ, T_1, T_2), and reflects the potential dependence of X not only on the argument value supplied by **apply**, but also on the bindings available at application time. Abstractions are themselves values, so their types are specified independently of the current context in the following rule:

$$\frac{\mathsf{env}\ \Gamma, \mathsf{given}\ T_1 \vdash X : T_2}{\mathsf{env}\ \_, \mathsf{given}\ \_ \vdash \textsf{\textbf{abs}}(X) : \textsf{\textit{abs}}(\Gamma, T_1, T_2)} \tag{96}$$

Notice that an abstraction can have many types; in particular, when X does not refer to the given value at all, the argument type T_1 is arbitrary, and when X does not refer to non-local bindings, Γ is arbitrary, so it can be the empty context $\emptyset$.

The abstraction computed by the expression **close**(**abs**(X)) is closed, having static scopes for non-local bindings. More generally, a well-typed expression **close**(E) has a type of the form *abs*$(\emptyset, T_1, T_2)$ in a context that provides all required non-local bindings for the abstraction computed by E.[9]

$$\frac{\mathsf{env}\ \Gamma \vdash E : \textsf{\textit{abs}}(\Gamma_1, T_1, T_2) \qquad \Gamma_1 \subseteq \Gamma}{\mathsf{env}\ \Gamma \vdash \textsf{\textbf{close}}(E) : \textsf{\textit{abs}}(\emptyset, T_1, T_2)} \tag{97}$$

[8] In effect, non-local bindings correspond to implicit parameters.

[9] The notation *abs*(T_1, T_2) for abstraction types in the conference version of this paper [13] abbreviates *abs*$(\emptyset, T_1, T_2)$.

Combining rules (96) and (97) we obtain a derived rule corresponding to the usual typing rule for abstractions with static scopes:

$$\frac{\mathsf{env}\,\Gamma, \mathsf{given}\,T_1 \vdash X : T_2}{\mathsf{env}\,\Gamma, \mathsf{given}\,\_ \vdash \mathbf{close}(\mathbf{abs}(X)) : \mathit{abs}(\emptyset, T_1, T_2)} \tag{98}$$

The typing rule for application specifies that the abstraction must be closed, but otherwise is the same as the usual rule for static bindings:

$$\frac{E_1 : \mathit{abs}(\emptyset, T_2, T) \qquad E_2 : T_2}{\mathbf{apply}(E_1, E_2) : T} \tag{99}$$

This approach to assigning static types to dynamically scoped abstractions is similar to the handling of implicit parameters proposed in [34]. However, while they extend statically scoped lambda calculus with implicit variables, we introduce types of dynamically scoped abstractions that can be specialised to statically scoped ones.

Patterns. A pattern is a *closed* abstraction that (when it matches a given value) computes an environment. The type of a pattern P is of the form $\mathit{abs}(\emptyset, T, \Gamma)$, where Γ determines the types of the identifiers bound when P is matched to a value of type T.

The typing rule for **match** is similar to that for **apply** (99):

$$\frac{E : T \qquad P : \mathit{abs}(\emptyset, T, \Gamma)}{\mathbf{match}(E, P) : \Gamma} \tag{100}$$

The typing rule for **patt-abs**(P, X) is similar to that for **abs**(X) (96), except that the environment in which X is typed is updated with the type of the environment computed by P:

$$\frac{\mathsf{env}\,\Gamma, \mathsf{given}\,T \vdash P : \mathit{abs}(\emptyset, T_1, \Gamma_2) \qquad \mathsf{env}(\Gamma_2/\Gamma_1), \mathsf{given}\,T_1 \vdash X : T_2}{\mathsf{env}\,\Gamma, \mathsf{given}\,T \vdash \mathbf{patt\text{-}abs}(P, X) : \mathit{abs}(\Gamma_1, T_1, T_2)} \tag{101}$$

The typing rules for patterns are as follows:

$$\mathbf{bind}(I) : \mathit{abs}(\emptyset, T, \{I \mapsto T\}) \tag{102}$$

$$\mathbf{any} : \mathit{abs}(\emptyset, T, \emptyset) \tag{103}$$

$$\frac{V : T}{\mathbf{only}(V) : \mathit{abs}(\emptyset, T, \emptyset)} \tag{104}$$

$$\frac{P_1 : \mathit{abs}(\emptyset, T, \Gamma_1) \qquad P_2 : \mathit{abs}(\emptyset, T, \Gamma_2)}{\mathbf{patt\text{-}union}(P_1, P_2) : \mathit{abs}(\emptyset, T, \mathbf{map\text{-}union}(\Gamma_1, \Gamma_2))} \tag{105}$$

The funcon **prefer-over** is applicable to arbitrary abstractions, not just patterns, and so has a more general typing rule:

$$\frac{P_1 : \mathit{abs}(\Gamma_1, T_1, T_2) \qquad P_2 : \mathit{abs}(\Gamma_2, T_1, T_2) \qquad \Gamma_1 \subseteq \Gamma \qquad \Gamma_2 \subseteq \Gamma}{\mathbf{prefer\text{-}over}(P_1, P_2) : \mathit{abs}(\Gamma, T_1, T_2)} \tag{106}$$

Failure and Back-Tracking. The funcon **fail** may have any type. The funcon **else** requires both its arguments to have the same type.

$$\textsf{\textbf{fail}} : T \tag{107}$$

$$\frac{X_1 : T \qquad X_2 : T}{\textsf{\textbf{else}}(X_1, X_2) : T} \tag{108}$$

Exceptions. The static semantics of the funcon **throw** allows it and its argument to have any type. The funcons **catch** and **catch-else-rethrow** check that the abstraction used to handle thrown exceptions computes values of the same type as normal termination.

$$\frac{E : T'}{\textsf{\textbf{throw}}(E) : T} \tag{109}$$

$$\frac{X : T \qquad E : \textsf{\textit{abs}}(\emptyset, T', T)}{\textsf{\textbf{catch}}(X, E) : T} \tag{110}$$

$$\frac{X : T \qquad E : \textsf{\textit{abs}}(\emptyset, T', T)}{\textsf{\textbf{catch-else-rethrow}}(X, E) : T} \tag{111}$$

3.5 Caml Light Static Semantics

The translation specified in Sect. 3.2 appears to accurately reflect the dynamic semantics of Caml Light programs. The funcons used in the translation also have static semantics, which provides a ‘default’ static semantics for the programs. In most cases, this agrees with the intended static semantics of Caml Light – but not always. In the latter cases, we modify the translation by inserting additional funcons which affect the static semantics, but which leave the dynamic semantics unchanged. We consider some examples. The signatures of the extra funcons involved are shown in Table 6.

Funcon signatures

$$\begin{aligned}
\textsf{\textbf{generalise-decl}}(\textsf{\textit{decls}}) &: \textsf{\textit{decls}} \\
\textsf{\textbf{instantiate-if-poly}}(\textsf{\textit{exprs}}) &: \textsf{\textit{exprs}} \\
\textsf{\textbf{restrict-domain}}(\textsf{\textit{abs}}(X, Y), \textsf{\textit{types}}) &: \textsf{\textit{abs}}(X, Y)
\end{aligned}$$

Table 6. Signatures of some funcons for adjusting static semantics

Catching Exceptions. The translation of $\mathtt{try}\,E\,\mathtt{with}\,SM$ in Sect. 3.2 (67) allows any value to be thrown as an exception and caught by the handler. In Caml Light, however, the values that can be thrown and caught are restricted to those included in the type `exn`, so static semantics needs to check that $\mathit{func}[\![\,SM\,]\!]$ has type $\mathtt{exn\text{-}\!>}X$ for some X. This can be achieved using $\textsf{\textbf{restrict-domain}}(E, T)$, which checks that the type of E is that of an abstraction with argument type T, but otherwise (statically and dynamically) behaves just like E. The modified translation equation is:[10]

$$\begin{aligned}&\textsf{\textit{expr}}[\![\,\mathtt{try}\,E\,\mathtt{with}\,SM\,]\!] = \\ &\quad\textsf{\textbf{catch-else-rethrow}}(\textsf{\textit{expr}}[\![\,E\,]\!], \\ &\qquad\textsf{\textbf{restrict-domain}}(\textsf{\textit{func}}[\![\,SM\,]\!], \textsf{\textbf{bound-type}}(\textsf{\textit{id}}[\![\,\mathtt{exn}\,]\!])))\end{aligned} \tag{112}$$

Using Polymorphism. Caml Light has polymorphism, where a type may be a type schema including universally quantified variables. The interpretation of identifier binding inspection using just the **bound-value** funcon (66) does not account for instantiation of polymorphic variables. We can rectify this as follows:

$$\textsf{\textit{expr}}[\![\,I\,]\!] = \textsf{\textbf{instantiate-if-poly}}(\textsf{\textbf{bound-value}}(\textsf{\textit{id}}[\![\,I\,]\!])) \tag{113}$$

The funcon **instantiate-if-poly** takes all universally quantified type variables in the type of its argument, and allows them to be instantiated arbitrarily; it does not affect the dynamic semantics.

Generating Polymorphism. Expressions with polymorphic types in Caml Light arise from let definitions, where types are generalised as much as possible, up to a constraint regarding imperative behaviour known as *value-restriction* [60]. The appropriate funcon is **generalise-decl**, which finds all generalisable types in its argument environment and explicitly quantifies them, universally. Whether this generalisation should be applied is determined entirely by the outermost production of the right-hand side (E) of the let definition.

$$\begin{aligned}&\textsf{\textit{decl}}[\![\,P\,\mathtt{=}\,E\,]\!] = \textsf{\textbf{generalise-decl}}(\textsf{\textit{decl-mono}}[\![\,P = E\,]\!]) \\ &\qquad\text{if } E \text{ is generalisable}\end{aligned} \tag{114}$$

$$\begin{aligned}&\textsf{\textit{decl}}[\![\,P\,\mathtt{=}\,E\,]\!] = \textsf{\textit{decl-mono}}[\![\,P = E\,]\!] \\ &\qquad\text{if } E \text{ is not generalisable}\end{aligned} \tag{115}$$

The translation funcon $\textsf{\textit{decl-mono}}[\![\,\_\,]\!]$ is the same as the version of $\textsf{\textit{decl}}[\![\,\_\,]\!]$ specified in Sect. 3.2 (65) for dynamic semantics.

$$\begin{aligned}&\textsf{\textit{decl-mono}}[\![\,P\,\mathtt{=}\,E\,]\!] = \\ &\quad\textsf{\textbf{match}}(\textsf{\textit{expr}}[\![\,E\,]\!], \\ &\qquad\textsf{\textbf{prefer-over}}(\textsf{\textit{patt}}[\![\,P\,]\!], \textsf{\textbf{abs}}(\textsf{\textbf{throw}}(\textsf{\textbf{cl-match-failure}}))))\end{aligned} \tag{116}$$

[10] When I is bound to a type, $\textsf{\textbf{bound-type}}(I)$ corresponds to $\textsf{\textbf{bound-value}}(I)$, but it is evaluated as part of static semantics.

3.6 The Full Caml Light Case Study

There is not sufficient space here to present all of our component-based semantics of the Caml Light language. The complete translation for the subset of Caml Light presented in Table 3 is given in the Appendix. Our translation of the following further constructs, together with the specifications of all the required funcons, is available online [12].

Values: records, with and without mutable fields; variant values.
Patterns: variant patterns; record patterns.
Expressions: operations on variants and records; function abstractions with multiple-matching arguments.
Global definitions: type abbreviations; record types; variant types; exception definitions.
Top level: module implementations; the core library.

We have not yet given semantics for modules (interfaces, directives, and references to declarations qualified by module names).

This concludes the presentation of illustrative excerpts from our Caml Light case study. Our confidence in the accuracy of the specifications of the translation and of the funcons used in it is based partly on the simplicity and perspicuity of the specifications, as illustrated above, partly on our tool support for validating them, which is described in the next section.

4 Tool Support

This section gives an overview of the tools we have used in connection with the case study presented in Sect. 3. These tools support parsing programs, translating programs to funcon terms, generating an interpreter from funcon specifications, and running programs on generated interpreters. They also support developing and browsing the specifications of funcons and languages.

The main technical requirement for such tools is to be consistent with the foundations of our specifications. Using the tools to run programs in some specified language (and comparing the results with running the same programs on some reference implementation) then tests the correctness of the language specification. With our component-based approach, the language specification consists of the equations for translating programs to funcons, together with the static and dynamic rules of the funcons used in the translation.

The PLanCompS project is currently developing integrated tools to support component-based language specification, but these are not yet ready for use in case studies. We have therefore used a combination of several existing tools to develop and test our specification of Caml Light: SDF for parsing programs; ASF+SDF and Stratego for translating programs to funcons; Prolog for parsing I-MSOS specifications and generating Prolog code from them, and for executing funcon terms; and Spoofax for generating editors for our specification languages.

The rest of this section summarises what the various tools do, and illustrates the support they have provided for our specification of Caml Light (CL). All our source code is available for download along with the CL specification.

4.1 Parsing Programs

The syntax of CL is defined in its reference manual [32] by a highly ambiguous context-free grammar in EBNF (see Table 3, page 20) together with some tables and informal comments regarding the intended disambiguation. We originally extracted the text of the grammar from the HTML version of the reference manual and converted it (semi-automatically) to SDF [61], which supports the specification of arbitrary context-free grammars.

We used the existing tool support for SDF to generate a scannerless GLR parser for CL, which was able to parse various test programs. To obtain a unique parse-tree for a program, however, expressions generally needed additional grouping parentheses. SDF supports several ways of specifying disambiguation, including relative priorities, left/right associativity, prefer/avoid annotations, and follow-restrictions. These allowed us to express most of the intended disambiguation without introducing auxiliary nonterminal symbols, albeit with some difficulty (e.g., we ended up using position-specific non-transitive priorities). A closer investigation by colleagues working on disambiguation techniques led to the quite surprising result that SDF's disambiguation mechanisms are actually inadequate to specify one particular feature of expression grouping that is required by CL [1]. Fortunately, it appears that CL programmers tend to insert grouping parentheses to avoid potential misinterpretation in such cases, so although we know that ambiguity could arise when using our parser, we have not found practical programs for which that happens.

One of the initial advantages of using SDF was its support by the ASF+SDF Meta-Environment [8], which provided an IDE with some pleasant features. However, ASF+SDF is no longer maintained or developed, so we recently switched to Spoofax [28] for generating a CL parser from our SDF grammar. Our Spoofax editor project for CL supports parsing of CL programs while editing them in Eclipse, and we use the Spoofax command-line interface when running test suites.

4.2 Translating Programs to Funcons

After parsing a CL program, we need to be able to translate it to funcons. ASF+SDF [8] allowed such translation rules to be specified as term rewriting equations, based on the CL syntax together with notation for translation functions, meta-variables, and funcons, all specified in SDF. When we switched from ASF+SDF to Spoofax, we started to use Stratego [62] for specifying term rewriting. Fortunately, it was possible to re-express our ASF+SDF equations quite naturally as Stratego rules, by exploiting its support for concrete syntax; see Fig. 1 for some examples.

Figure 2 shows a funcon term resulting from pressing the 'Generation' button after parsing a CL program in the Spoofax editor. To obtain funcon terms in

```
to-funcons:
  |[ expr[: ~E1 or ~E2 :] ]| ->
  |[ if_true(expr[: ~E1 :], true, expr[: ~E2 :]) ]|
to-funcons:
  |[ expr[: if ~E1 then ~E2 :] ]| ->
  |[ expr[: if ~E1 then ~E2 else ( ) :] ]|
to-funcons:
  |[ expr[: if ~E1 then ~E2 else ~E3 :] ]| ->
  |[ if_true(expr[: ~E1 :], expr[: ~E2 :], expr[: ~E3 :] ) ]|
to-funcons:
  |[ expr[: while ~E1 do ~E2 done :] ]| ->
  |[ while_true(expr[: ~E1 :], effect(expr[: ~E2 :])) ]|
```

Fig. 1. Some Stratego rules for transforming CL to funcons

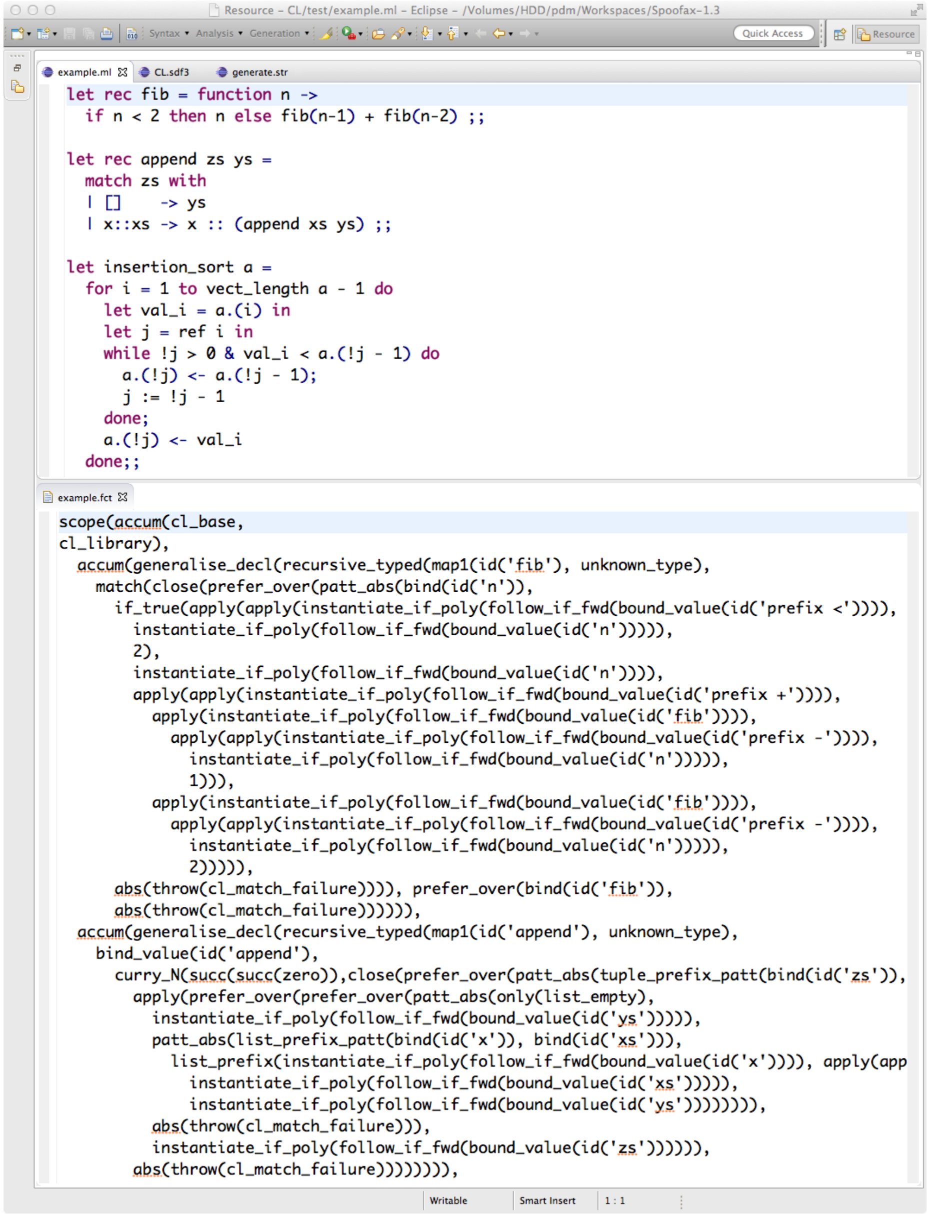
the format used by our Prolog-based tools, we invoke a pretty-printer generated from SDF3 templates for funcon signatures.

4.3 Translating I-MSOS Rules to Prolog

A notation called MSDF had previously been developed for specifying transition rules for funcons in connection with teaching operational semantics using MSOS [44], along with a definite clause grammar (DCG) for parsing MSDF, and Prolog code for transforming each MSDF rule to a Prolog clause (MSDF is used also in the Maude MSOS Tool [10]). The PLanCompS project has developed a variant of MSDF called CSF for specifying I-MSOS rules for funcons. CSF is parsed using a DCG when translating rules to Prolog; we also have a Spoofax editor for CSF, based on an SDF grammar. Figure 3 shows an example of a CSF specification.

As with the original version of MSDF, we use a Prolog program to transform parsed CSF rules to Prolog clauses (supporting not only transitions but also typing assertions and equations) and to run funcon terms. A shell script invokes the Prolog program to generate Prolog code from our current collection of CSF specifications of funcons and values in a few seconds; further scripts run entire test suites. When all the generated clauses are loaded together with a small amount of fixed runtime code (mainly for MSOS label composition), funcon terms can be executed.

Directly interpreting small-step transition rules for funcons is inherently inefficient [3]: each step of running a program searches for a transition from the top of the entire program, and the term representing the program gets repeatedly unfolded in connection with recursive function calls. The number of Prolog inference steps is sometimes alarmingly high, but we have managed to execute a wide range of CL test programs. We intend to apply semantics-preserving transformation of small-step rules to so-called ‘pretty-big-step’ rules following [2] to remove this source of inefficiency.



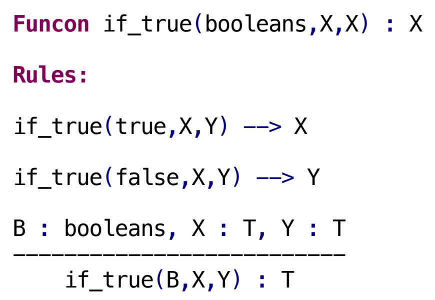
```
let rec fib = function n ->
  if n < 2 then n else fib(n-1) + fib(n-2) ;;

let rec append zs ys =
  match zs with
  | []    -> ys
  | x::xs -> x :: (append xs ys) ;;

let insertion_sort a =
  for i = 1 to vect_length a - 1 do
    let val_i = a.(i) in
    let j = ref i in
    while !j > 0 & val_i < a.(!j - 1) do
      a.(!j) <- a.(!j - 1);
      j := !j - 1
    done;
    a.(!j) <- val_i
  done;;
```

```
scope(accum(cl_base,
cl_library),
  accum(generalise_decl(recursive_typed(map1(id('fib'), unknown_type),
    match(close(prefer_over(patt_abs(bind(id('n')),
      if_true(apply(apply(instantiate_if_poly(follow_if_fwd(bound_value(id('prefix <')))),
        instantiate_if_poly(follow_if_fwd(bound_value(id('n'))))),
        2),
        instantiate_if_poly(follow_if_fwd(bound_value(id('n')))),
        apply(apply(instantiate_if_poly(follow_if_fwd(bound_value(id('prefix +')))),
          apply(instantiate_if_poly(follow_if_fwd(bound_value(id('fib')))),
            apply(apply(instantiate_if_poly(follow_if_fwd(bound_value(id('prefix -')))),
              instantiate_if_poly(follow_if_fwd(bound_value(id('n'))))),
              1))),
          apply(instantiate_if_poly(follow_if_fwd(bound_value(id('fib')))),
            apply(apply(instantiate_if_poly(follow_if_fwd(bound_value(id('prefix -')))),
              instantiate_if_poly(follow_if_fwd(bound_value(id('n'))))),
              2))))),
      abs(throw(cl_match_failure)))), prefer_over(bind(id('fib')),
      abs(throw(cl_match_failure)))))),
  accum(generalise_decl(recursive_typed(map1(id('append'), unknown_type),
    bind_value(id('append'),
      curry_N(succ(succ(zero)),close(prefer_over(patt_abs(tuple_prefix_patt(bind(id('zs')),
        apply(prefer_over(prefer_over(patt_abs(only(list_empty),
          instantiate_if_poly(follow_if_fwd(bound_value(id('ys'))))),
          patt_abs(list_prefix_patt(bind(id('x')), bind(id('xs'))),
            list_prefix(instantiate_if_poly(follow_if_fwd(bound_value(id('x')))), apply(app
              instantiate_if_poly(follow_if_fwd(bound_value(id('xs'))))),
              instantiate_if_poly(follow_if_fwd(bound_value(id('ys'))))))))),
          abs(throw(cl_match_failure))),
          instantiate_if_poly(follow_if_fwd(bound_value(id('zs')))))),
        abs(throw(cl_match_failure))))))),
```

Fig. 2. A CL program and the generated funcon term



```
Funcon if_true(booleans,X,X) : X

Rules:

if_true(true,X,Y) --> X

if_true(false,X,Y) --> Y

B : booleans, X : T, Y : T
--------------------------
    if_true(B,X,Y) : T
```

Fig. 3. A CSF specification

4.4 A Component-Based Semantics Specification Language

We are developing CBS, a unified specification language designed for use in component-based semantics. CBS allows specification of abstract syntax grammars (essentially BNF with regular expressions), the signatures and equations for translation functions, and the signatures and rules for values and funcons, so it can replace our current combination of SDF, Stratego and CSF. Use of CBS should provide considerably greater notational consistency, and improve the readability of our specifications.

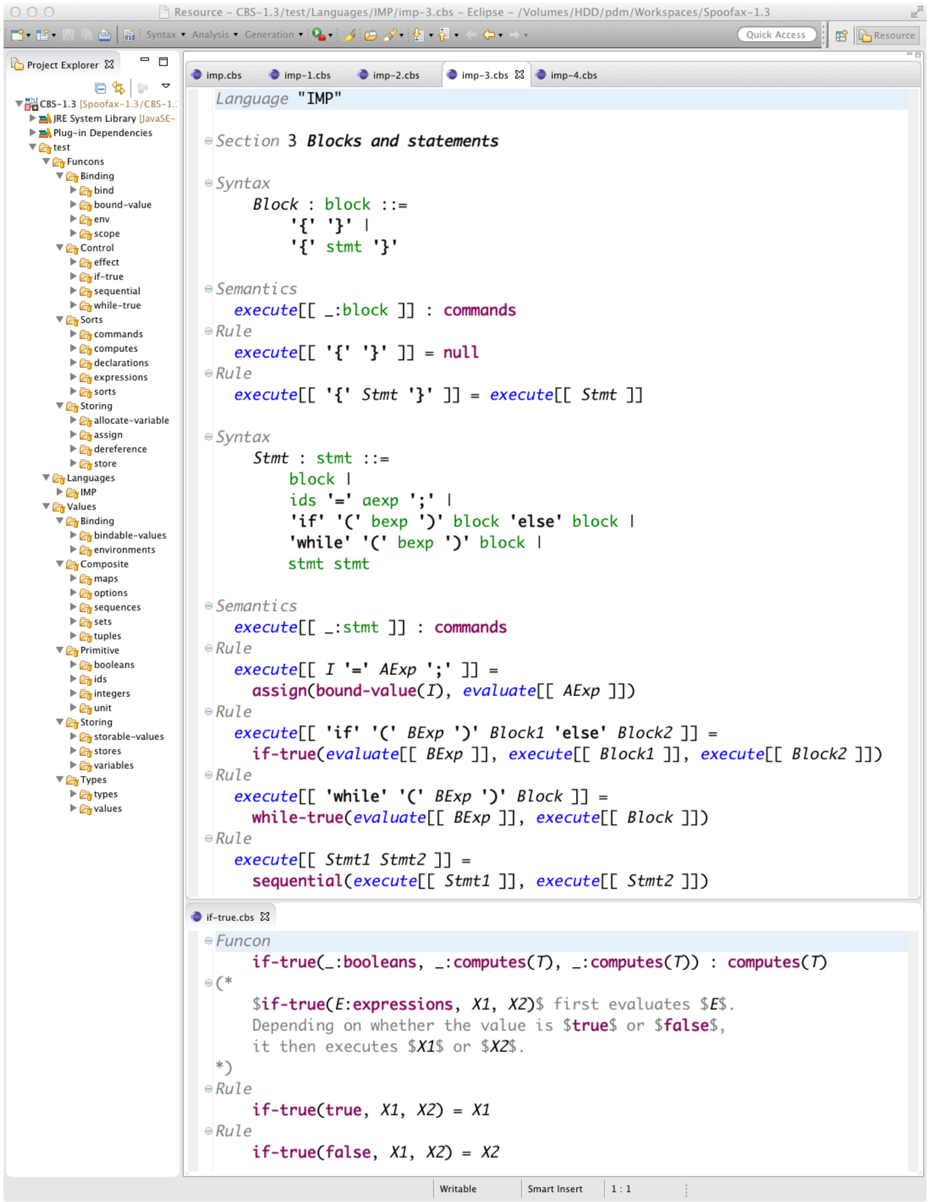
We have used Spoofax to create an editor for CBS, exploiting name resolution to check that all the notation used in a CBS project has been uniquely defined (possibly in a different file) and to hyperlink uses of funcons to their specifications. Figure 4 illustrates the use of the CBS editor to check the specification of a small imperative language for notational consistency in the presence of CBS specifications of the required funcons and values.

We are currently re-specifying CL in CBS. We intend to generate SDF and Stratego code from the CBS specification of the translation of CL to funcons, and Prolog rules from the CBS specifications of the individual funcons. We would also like to generate LaTeX source code from CBS, to ensure consistency between examples provided in articles such as this, and the specifications that we have tested.

We expect our current case study of component-based semantics (C#) to be developed entirely in CBS, supported by tools running in Spoofax. Further tools currently being developed in the PLanCompS project are to integrate support for CBS with recent advances in GLL parser generation and disambiguation [26], aiming to provide a complete workbench for language specification.

5 Related Work

The component-based framework presented and illustrated in the previous sections was inspired by features of many previous frameworks. In this section, we



```
Language "IMP"

Section 3 Blocks and statements

Syntax
    Block : block ::=
        '{' '}' |
        '{' stmt '}'

Semantics
  execute[[ _:block ]] : commands
Rule
  execute[[ '{' '}' ]] = null
Rule
  execute[[ '{' Stmt '}' ]] = execute[[ Stmt ]]

Syntax
    Stmt : stmt ::=
        block |
        ids '=' aexp ';' |
        'if' '(' bexp ')' block 'else' block |
        'while' '(' bexp ')' block |
        stmt stmt

Semantics
  execute[[ _:stmt ]] : commands
Rule
  execute[[ I '=' AExp ';' ]] =
    assign(bound-value(I), evaluate[[ AExp ]])
Rule
  execute[[ 'if' '(' BExp ')' Block1 'else' Block2 ]] =
    if-true(evaluate[[ BExp ]], execute[[ Block1 ]], execute[[ Block2 ]])
Rule
  execute[[ 'while' '(' BExp ')' Block ]] =
    while-true(evaluate[[ BExp ]], execute[[ Block ]])
Rule
  execute[[ Stmt1 Stmt2 ]] =
    sequential(execute[[ Stmt1 ]], execute[[ Stmt2 ]])
```

```
Funcon
    if-true(_:booleans, _:computes(T), _:computes(T)) : computes(T)
(*
    $if-true(E:expressions, X1, X2)$ first evaluates $E$.
    Depending on whether the value is $true$ or $false$,
    it then executes $X1$ or $X2$.
*)
Rule
    if-true(true, X1, X2) = X1
Rule
    if-true(false, X1, X2) = X2
```

Fig. 4. CBS in use on IMP, a small imperative language

mainly consider its relationship to semantic frameworks that have a high degree of modularity.

Algebraic Specification. Heering and Klint proposed in the early 1980s to structure complete definitions of programming languages as libraries of reusable components [22]. This motivated the development of ASF+SDF [4], which provided strong support for modular structure in algebraic specifications. However, an ASF+SDF definition of a programming language did not, in general, permit the reuse of the individual language constructs in the definitions of other languages. The main hindrances to reuse in ASF+SDF were coarse modular structure (e.g., specifying all expression constructs in a single module), explicit propagation of auxiliary entities, and direct specification of language constructs. Other algebraic specification frameworks (e.g., OBJ [20]) emphasised finer modular structure, but still did not provide reusable components of language specifications. These issues are illustrated and discussed further in [48].

Monads. At the end of the 1980s, Moggi [39] introduced the use of monads and monad transformers in denotational semantics. (In fact Scott and Strachey had themselves used monadic notation for composition of store transformations in the early 1970s, and an example of a monad transformer can also be found in the VDM definition of PL/I, but the monadic structure was not explicit [47].) Monads avoid explicit propagation of auxiliary entities, and monad transformers are highly reusable components. Various monad transformers have been defined (e.g., see [35]) with operations that in many cases correspond to our funcons; monads can also make a clear distinction between sets T of values and sets of computations $M(T)$ of values in T.

One drawback of monad transformers with respect to modularity is that different orders of composition can lead to different semantics. For example, one order of composition of the state and exception monad transformers preserves the state when an exception is thrown, whereas the other restores it. In contrast, the semantics of our funcons is independent of the order in which they are added. The concept of monad transformers inspired the development of MSOS [42], the modular variant of SOS that we use to define funcons.

An alternative way of defining monads has been developed by Plotkin and Power [57] using Lawvere theories instead of monad transformers. Recently, Delaware et al. [14] presented modular monadic meta-theory, combining modular datatypes with monad transformers, focusing on modularisation of theorems and proofs. Both these frameworks assume some familiarity with Category Theory. In contrast, the foundations of our component-based framework involve MSOS, where labels happen to be morphisms of categories, but label composition can easily be explained without reference to Category Theory.

Abstract State Machines. Kutter and Pierantonio [30] proposed the Montages variant of abstract state machines (ASMs) with a separate module for each language construct. Reusability was limited partly by the tight coupling of components to concrete syntax.

Börger et al. [6,7] gave modular ASM semantics for JAVA and C#, identifying features shared by the two languages, but did not define components intended for wider reuse.

ASM specifications generally make widespread use of ad-hoc abbreviations for patterns of rules, and sometimes redefine these abbreviations when extending a described language. In our component-based approach, in contrast, the specifications of the funcons remain fixed, and it is only the specification of the translation to funcons that may need to change when extending the language.

Action Semantics. This framework combined features of denotational, operational and algebraic semantics. It was developed initially by Mosses and Watt [40,41,51]. The notation for actions used in action semantics can be regarded as a collection of funcons. Action notation supported specification of sequential and interleaved control flow, abrupt termination and its handling, scopes of bindings, imperative variables, asynchronous concurrent processes, and procedural abstractions, but the collection of actions was not extensible. Actions were relatively primitive, being less closely related to familiar programming constructs than funcons (e.g., conditional choice was specified using guards, and iteration by an 'unfolding'). Various algebraic laws allowed reasoning about action equivalence. Although action semantics was intended for specifying dynamic semantics, Doh and Schmidt [16] explored the possibility of using it also for static semantics.

The modular structure of specifications in action semantics was conventional, with separate sections for abstract syntax, auxiliary entities, and semantic equations. Doh and Mosses [15] proposed replacing it by a component-based structure, defining the abstract syntax and action semantics of each language construct in a separate module, foreshadowing the modular structure of funcon specifications (except that static semantics was not addressed).

Iversen and Mosses [24] introduced so-called Basic Abstract Syntax (BAS), which is a direct precursor of our current collection of funcons. They specified a translation from the Core of Standard ML to BAS, and gave action semantics for each BAS construct, with tool support using ASF+SDF [9]. However, having to deal with both BAS and action notation was a drawback. Mosses et al. [25,43,45,46] reported on subsequent work that led to the present paper.

TinkerType. Levin and Pierce developed the TINKERTYPE system [33] to support reuse of conventional SOS specifications of individual language constructs. The idea was to have a variant of the specification of each construct for each combination of language features. To define a new language with reuse of a collection of previously specified constructs, TINKERTYPE could determine the union of the auxiliary entities needed for their individual specifications, and assemble the language definition from the corresponding variants. This approach alleviated some of the symptoms of poor reusability in SOS.

Ott. Another system supporting practical use of conventional SOS is OTT [59], which allows for specifications to be compiled to the languages of various theorem provers, including HOL (based on classical higher-order logic). OTT facilitates use of SOS, providing a metalanguage that supports variable binding and substitution; however, it does not provide support for reusable components.

Owens et al. [52,53] used Ott to specify a sublanguage of OCaml corresponding closely to Caml Light. Owens [53] used the HOL code automatically generated from the language specification to prove a type soundness theorem. The dynamic semantics is formulated in terms of small-step rules, relying on congruence rules to specify order of evaluation. The approach departs from traditional SOS [56] in using substitution rather than environments; Ott requires binding occurrences of variables to be annotated as such in the abstract syntax. The need for renaming of bound value variables is avoided by not reducing under value variable binders, and by relying on the assumption that well-typed programs have no free value variables (i.e., they are context-independent). The static semantics uses De Brujin indices to represent type variables, and relies on substitution to deal with type variables in explicit type annotations. The use of labels to avoid explicit mention of the store is similar to MSOS. Some of the choices of techniques used in the specification are motivated by the HOL proofs – notably, their use of congruence rules instead of evaluation contexts, and of De Brujin indices.

The OCaml Light specification is comparatively large: 173 rules for the static semantics and 137 rules for the dynamic semantics. It is interesting to observe that out of the 61 rules that are given for expression evaluation [52, Sect. 4.9], 18 are congruence rules, and 17 are exception propagation rules. Ultimately, little more than a third of the rules are reductions; these are the only ones which would need to be explicitly stated using an approach that takes full advantage of strictness annotations and of MSOS labels. For example, the Ott rules for evaluating if-else expressions are the following:

$$\frac{\vdash e_1 \xrightarrow{L} e_1'}{\vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \xrightarrow{L} \textbf{if}\ e_1'\ \textbf{then}\ e_2\ \textbf{else}\ e_3}\ \textsf{ifthenelse\_ctx}$$

$$\frac{}{\vdash \textbf{if}\ (\%\textbf{prim raise})\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \longrightarrow (\%\textbf{prim raise})\ v}\ \textsf{if\_raise}$$

$$\frac{}{\vdash \textbf{if true then}\ e_2\ \textbf{else}\ e_3 \longrightarrow e_2}\ \textsf{ifthenelse\_true}$$

$$\frac{}{\vdash \textbf{if false then}\ e_2\ \textbf{else}\ e_3 \longrightarrow e_3}\ \textsf{ifthenelse\_false}$$

The above specification can be compared to that for the **if-true** funcon (Fig. 3). In the Ott specification, the first rule (ifthenelse_ctx) is a congruence rule, and the second one (if_raise) is an exception propagation rule. Only the last two rules above are reduction rules, corresponding to our (30, 31). Note the use of the label L to thread the state in the first rule (as in MSOS), and the absence of environments (due to their use of substitution).

In the Ott typing rule below, E is a typing context, and σ^T is an assignment of types to type variables (needed in connection with polymorphism, to deal with explicit type annotations).

$$\frac{\sigma^T \& E \vdash e_1 : \mathbf{bool} \qquad \sigma^T \& E \vdash e_2 : t \qquad \sigma^T \& E \vdash e_3 : t}{\sigma^T \& E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t}\ \mathsf{ifthenelse}$$

This is similar to the corresponding static rule in our semantics (44). As a purely notational difference, we leave the typing context implicit, following the I-MSOS presentation style. The assignment to type variables is also left implicit in our treatment of polymorphism, see Sect. 3.5.

Evaluation Contexts. Ott supports also reduction semantics based on evaluation contexts. This framework is widely used for proving meta-theoretic results (e.g., type soundness).[11] The semantics of Standard ML presented in [21,31] uses an elaborative approach based on the translation of the source language to a type system (the *internal* language) and on a reduction semantics (relying on evaluation contexts), formalised and proved to be type sound in Twelf. Conciseness is achieved by defining the semantics on the internal language, rather than on the source one. However, the internal language is designed for the translation from a particular source (Standard ML in this case), and it is not particularly oriented toward extensibility and reuse.

The PLT Redex tool [18] runs programs by interpreting their reduction semantics, and has been used to validate language specifications [29]. However, it is unclear whether reduction semantics could be used to define reusable components whose specifications never need changing when combined – in particular, adding new features may require modification of the grammar for evaluation contexts.

Compared to a conventional small-step SOS, the specification of the same language by evaluation rules and the accompanying evaluation-context grammar is usually relatively concise. This is primarily because each congruence rule in the SOS corresponds to a single production of the evaluation context grammar; moreover, exception propagation is usually specified by inference rules in SOS, but by a succinct auxiliary evaluation context grammar in reduction semantics. However, our I-MSOS specifications of funcons avoid the need for many congruence rules, and exception propagation is implicit, which may well make our specifications even more concise than a corresponding reduction semantics.

Rewriting Logic and K. Competing approaches with a high degree of inherent modularity include Rewriting Logic Semantics [37] and the K framework [58]. Both frameworks have well-developed tool support, which allows not only execution of programs according to their semantics, but also model checking. K has been used to specify major programming languages such as C [17] and Java [5].

The lifting of funcon arguments from value sorts to computation sorts is closely related to (and was inspired by) strictness annotations in K. It appears

[11] The lack of HOL support for evaluation contexts discouraged Owens from using them for his OCaml Light case study [53].

possible to specify individual funcons independently in K, and to use the K Tools to translate programming languages to funcons [50], thereby incorporating our component-based approach directly in that framework.

6 Conclusions and Further Work

We regard our Caml Light case study as significant evidence of the applicability and modularity of our component-based approach to semantics. The *key novel feature* is the introduction of an open-ended collection of fundamental constructs (funcons). The abstraction level of the funcons we have used to specify the semantics of Caml Light appears to be optimal: if the funcons were closer to the language constructs, the translation of the language to funcons would have been a bit simpler, but the I-MSOS rules for the funcons would have been considerably more complicated; lower-level funcons (e.g., comparable to the combinators used in action semantics [40,41]) would have increased the size and decreased the perspicuity of the funcon terms used in the translation. Some of the funcons presented here do in fact correspond very closely to Caml Light language constructs (e.g., eager function application and pattern-matching) but we regard that as a natural consequence of the clean design of this particular language, and unlikely to occur when specifying a language whose design is less principled.

Caml Light is a real language, and we have successfully tested our semantics for it by generating funcon terms from programs, running them using Prolog code generated from the I-MSOS rules that define the funcons, then comparing the results with those given by running the same programs on the latest release of the Caml Light system (which is the *de facto* definition of the language). The test programs and funcon terms are available online [12] together with the generated Prolog code for each funcon. We have checked that our test programs exercise every translation equation, and that running them uses every applicable rule of every funcon, so we are reasonably confident in the accuracy of our specifications.

The work reported here is part of the PLanCompS project [55]. Apart from developing and refining the component-based approach to language specification, PLanCompS is developing a chain of tools specially engineered to support its practical use.

Ongoing and future case studies carried out by the PLanCompS project will test the reusability of our funcons. We are already reusing many of those introduced for specifying Caml Light in a component-based semantics for C#. The main test will be to specify the corresponding Java constructs using essentially the same collection of funcons as for C#. We expect the approach to be equally applicable to domain-specific languages, where the benefits of reuse in connection with co-evolution of languages and their specifications could be especially significant.

We are quite happy with the perspicuity of our specifications. Lifting value arguments to computation sorts has eliminated the need to specify tedious 'congruence' rules in the small-step I-MSOS of funcons. The funcon names are reasonably suggestive, while not being too verbose, although there is surely room for improvement. When the PLanCompS project has completed its case studies, it intends to finalise the definitions of the funcons it has developed, and establish an open-access digital library of funcons and language specifications. Until then, the names and details of the funcons presented here should be regarded as tentative.

In conclusion, we consider our component-based approach to be a good example of modularity in the context of programming language semantics. We do not claim that any of the techniques we employ are directly applicable in software engineering, although component-based specifications might well provide a suitable basis for generating implementations of domain-specific languages.

Acknowledgments. Thanks to Erik Ernst and the anonymous referees for helpful comments and suggestions for improvement. The reported work was supported by EPSRC grant (EP/I032495/1) to Swansea University for the PLanCompS project.

Appendix

This appendix contains the translation equations for the subset of Caml Light presented in Table 3, from which the illustrative examples in Sect. 3 are drawn. Our translation of the full Caml Light language is available online [12].

Markup for formatting the equations given below was inserted manually in the Stratego rules used to translate Caml Light programs to funcons. A few equations overlap; in Stratego we apply the more specific ones when possible.

Global names

$$\mathit{id}\llbracket\ I\ \rrbracket = \textbf{id}('I')$$

Type expressions

$$\mathit{type}\llbracket\ \texttt{(}\ T\ \texttt{)}\ \rrbracket = \mathit{type}\llbracket\ T\ \rrbracket$$

$$\mathit{type}\llbracket\ I\ \rrbracket = \textbf{bound-type}(\mathit{id}\llbracket\ I\ \rrbracket)$$

$$\mathit{type}\llbracket\ T_1\ \texttt{->}\ T_2\ \rrbracket = \textbf{abs}(\emptyset, \mathit{type}\llbracket\ T_1\ \rrbracket, \mathit{type}\llbracket\ T_2\ \rrbracket)$$

$$\mathit{type}\llbracket\ T\ I\ \rrbracket = \textbf{instantiate-type}(\mathit{type}\llbracket\ I\ \rrbracket, \mathit{type\text{-}list}\llbracket\ T\ \rrbracket)$$

$$\mathit{type}\llbracket\ \texttt{'}\ I\ \rrbracket = \textbf{typevar}('I')$$

$$\mathit{type}\llbracket\ T_1\ \texttt{*}\ T_2\ \rrbracket = \textbf{tuple-type2}(\mathit{type}\llbracket\ T_1\ \rrbracket, \mathit{type}\llbracket\ T_2\ \rrbracket)$$

$$\mathit{type}\llbracket\ T_1\ \texttt{*}\ T_2\ \texttt{*}\ T_3\ \cdots\ \rrbracket = \textbf{tuple-type-prefix}(\mathit{type}\llbracket\ T_1\ \rrbracket, \mathit{type}\llbracket\ T_2\ \texttt{*}\ T_3\ \cdots\ \rrbracket)$$

$$\mathit{type}\llbracket\ \texttt{(}T_1\ \texttt{,}\ T_2\ \cdots\texttt{)}\ I\ \rrbracket = \textbf{instantiate-type}(\mathit{type}\llbracket\ I\ \rrbracket, \mathit{type\text{-}list}\llbracket\ T_1\ \texttt{,}\ T_2\ \cdots\ \rrbracket)$$

$$\mathit{type\text{-}list}\llbracket\ T\ \rrbracket = \textbf{list1}(\mathit{type}\llbracket\ T\ \rrbracket)$$

$$\mathit{type\text{-}list}\llbracket\ T_1\ \texttt{,}\ T_2\ \cdots\ \rrbracket = \textbf{list-prefix}(\mathit{type}\llbracket\ T_1\ \rrbracket, \mathit{type\text{-}list}\llbracket\ T_2\ \cdots\ \rrbracket)$$

Constants

$$\textit{value}\llbracket\ Int\ \rrbracket = Int$$
$$\textit{value}\llbracket\ Float\ \rrbracket = Float$$
$$\textit{value}\llbracket\ Char\ \rrbracket = \textbf{char}(Char)$$
$$\textit{value}\llbracket\ String\ \rrbracket = String$$
$$\textit{value}\llbracket\ \texttt{false}\ \rrbracket = \textbf{false}$$
$$\textit{value}\llbracket\ \texttt{true}\ \rrbracket = \textbf{true}$$
$$\textit{value}\llbracket\ \texttt{[ ]}\ \rrbracket = \textbf{list-empty}$$
$$\textit{value}\llbracket\ \texttt{( )}\ \rrbracket = \textbf{null}$$

Patterns

$$\textit{patt}\llbracket\ \texttt{(}\ P\ \texttt{)}\ \rrbracket = \textit{patt}\llbracket\ P\ \rrbracket$$
$$\textit{patt}\llbracket\ I\ \rrbracket = \textbf{bind}(\textit{id}\llbracket\ I\ \rrbracket)$$
$$\textit{patt}\llbracket\ \texttt{\_}\ \rrbracket = \textbf{any}$$
$$\textit{patt}\llbracket\ P\ \texttt{as}\ I\ \rrbracket = \textbf{patt-union}(\textit{patt}\llbracket\ P\ \rrbracket, \textbf{bind}(\textit{id}\llbracket\ I\ \rrbracket))$$
$$\textit{patt}\llbracket\ \texttt{(}\ P\ \texttt{:}\ T\ \texttt{)}\ \rrbracket = \textbf{patt-at-type}(\textit{patt}\llbracket\ P\ \rrbracket, \textit{type}\llbracket\ T\ \rrbracket)$$
$$\textit{patt}\llbracket\ P_1\ \texttt{|}\ P_2\ \rrbracket = \textbf{patt-non-binding}(\textbf{prefer-over}(\textit{patt}\llbracket\ P_1\ \rrbracket, \textit{patt}\llbracket\ P_2\ \rrbracket))$$
$$\textit{patt}\llbracket\ P_1\ \texttt{::}\ P_2\ \rrbracket = \textbf{list-prefix-patt}(\textit{patt}\llbracket\ P_1\ \rrbracket, \textit{patt}\llbracket\ P_2\ \rrbracket)$$
$$\textit{patt}\llbracket\ \texttt{[}\ P\ \texttt{]}\ \rrbracket = \textit{patt}\llbracket\ P\ \texttt{::}\ \texttt{[ ]}\ \rrbracket$$
$$\textit{patt}\llbracket\ \texttt{[}\ P_1\ \texttt{;}\ P_2\ \cdots\ \texttt{]}\ \rrbracket = \textit{patt}\llbracket\ P_1\ \texttt{::}\ \texttt{[}\ P_2\ \cdots\ \texttt{]}\ \rrbracket$$
$$\textit{patt}\llbracket\ C\ \rrbracket = \textbf{only}(\textit{value}\llbracket\ C\ \rrbracket)$$
$$\textit{patt}\llbracket\ P_1\ \texttt{,}\ P_2\ \cdots\ \rrbracket = \textit{patt-tuple}\llbracket\ P_1\ \texttt{,}\ P_2\ \cdots\ \rrbracket$$
$$\textit{patt-tuple}\llbracket\ P\ \rrbracket = \textbf{tuple-prefix-patt}(\textit{patt}\llbracket\ P\ \rrbracket, \textbf{only}(\textbf{tuple-empty}))$$
$$\textit{patt-tuple}\llbracket\ P_1\ \texttt{,}\ P_2\ \cdots\ \rrbracket = \textbf{tuple-prefix-patt}(\textit{patt}\llbracket\ P_1\ \rrbracket, \textit{patt-tuple}\llbracket\ P_2\ \cdots\ \rrbracket)$$

Expressions

$$\textit{expr}\llbracket\ I\ \rrbracket = \textbf{instantiate-if-poly}(\textbf{follow-if-fwd}(\textbf{bound-value}(\textit{id}\llbracket\ I\ \rrbracket)))$$
$$\textit{expr}\llbracket\ C\ \rrbracket = \textit{value}\llbracket\ C\ \rrbracket$$
$$\textit{expr}\llbracket\ \texttt{(}\ E\ \texttt{)}\ \rrbracket = \textit{expr}\llbracket\ E\ \rrbracket$$
$$\textit{expr}\llbracket\ \texttt{begin}\ E\ \texttt{end}\ \rrbracket = \textit{expr}\llbracket\ E\ \rrbracket$$
$$\textit{expr}\llbracket\ \texttt{(}\ E\ \texttt{:}\ T\ \texttt{)}\ \rrbracket = \textbf{typed}(\textit{expr}\llbracket\ E\ \rrbracket, \textit{type}\llbracket\ T\ \rrbracket)$$
$$\textit{expr}\llbracket\ E_1\ \texttt{,}\ E_2\ \cdots\ \rrbracket = \textit{expr-tuple}\llbracket\ E_1\ \texttt{,}\ E_2\ \cdots\ \rrbracket$$
$$\textit{expr-tuple}\llbracket\ E\ \rrbracket = \textbf{tuple-prefix}(\textit{expr}\llbracket\ E\ \rrbracket, \textbf{tuple-empty})$$
$$\textit{expr-tuple}\llbracket\ E_1\ \texttt{,}\ E_2\ \cdots\ \rrbracket = \textbf{tuple-prefix}(\textit{expr}\llbracket\ E_1\ \rrbracket, \textit{expr-tuple}\llbracket\ E_2\ \cdots\ \rrbracket)$$
$$\textit{expr}\llbracket\ E_1\ \texttt{::}\ E_2\ \rrbracket = \textbf{list-prefix}(\textit{expr}\llbracket\ E_1\ \rrbracket, \textit{expr}\llbracket\ E_2\ \rrbracket)$$
$$\textit{expr}\llbracket\ \texttt{[}\ E\ \texttt{]}\ \rrbracket = \textit{expr}\llbracket\ E\ \texttt{::}\ \texttt{[ ]}\ \rrbracket$$
$$\textit{expr}\llbracket\ \texttt{[}\ E_1\ \texttt{;}\ E_2\ \cdots\ \texttt{]}\ \rrbracket = \textit{expr}\llbracket\ E_1\ \texttt{::}\ \texttt{[}\ E_2\ \cdots\texttt{]}\ \rrbracket$$

$$\mathit{expr}\llbracket\ \texttt{[|\ \ |]}\ \rrbracket = \textbf{vector-empty}$$

$$\mathit{expr}\llbracket\ \texttt{[|}\ E\ \texttt{|]}\ \rrbracket = \textbf{vector1}(\textbf{alloc}(\mathit{expr}\llbracket\ E\ \rrbracket))$$

$$\begin{aligned}&\mathit{expr}\llbracket\ \texttt{[|}\ E_1\ \texttt{;}\ E_2\ \cdots\ \texttt{|]}\ \rrbracket =\\&\quad\textbf{vector-append}(\mathit{expr}\llbracket\ \texttt{[|}\ E_1\ \texttt{|]}\ \rrbracket, \mathit{expr}\llbracket\ \texttt{[|}\ E_2\ \cdots\ \texttt{|]}\ \rrbracket)\end{aligned}$$

$$\mathit{expr}\llbracket\ E_1\ E_2\ \rrbracket = \textbf{apply}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \mathit{expr}\llbracket\ E_2\ \rrbracket)$$

$$\mathit{expr}\llbracket\ \texttt{-}\ E\ \rrbracket = \textbf{int-negate}(\mathit{expr}\llbracket\ E\ \rrbracket)$$

$$\mathit{expr}\llbracket\ \texttt{-.}\ E\ \rrbracket = \textbf{float-negate}(\mathit{expr}\llbracket\ E\ \rrbracket)$$

$$\mathit{expr}\llbracket\ \texttt{!}\ E\ \rrbracket = \mathit{expr}\llbracket\ \texttt{prefix !}\ E\ \rrbracket$$

$$\mathit{expr}\llbracket\ E_1\ \mathit{IO}\ E_2\ \rrbracket = \mathit{expr}\llbracket\ \texttt{prefix}\ \mathit{IO}\ E_1\ E_2\ \rrbracket$$

$$\mathit{expr}\llbracket\ E_1\ \texttt{.(}\ E_2\ \texttt{)}\ \rrbracket = \mathit{expr}\llbracket\ \texttt{vect\_item}\ E_1\ E_2\ \rrbracket$$

$$\mathit{expr}\llbracket\ E_1\ \texttt{.(}\ E_2\ \texttt{)}\ \texttt{<-}\ E_3\ \rrbracket = \mathit{expr}\llbracket\ \texttt{vect\_assign}\ E_1\ E_2\ E_3\ \rrbracket$$

$$\mathit{expr}\llbracket\ \texttt{not}\ E\ \rrbracket = \textbf{not}(\mathit{expr}\llbracket\ E\ \rrbracket)$$

$$\mathit{expr}\llbracket\ E_1\ \texttt{\&}\ E_2\ \rrbracket = \textbf{if-true}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \mathit{expr}\llbracket\ E_2\ \rrbracket, \textbf{false})$$

$$\mathit{expr}\llbracket\ E_1\ \texttt{or}\ E_2\ \rrbracket = \textbf{if-true}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \textbf{true}, \mathit{expr}\llbracket\ E_2\ \rrbracket)$$

$$\mathit{expr}\llbracket\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \rrbracket = \mathit{expr}\llbracket\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else ( )}\ \rrbracket$$

$$\mathit{expr}\llbracket\ \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \rrbracket = \textbf{if-true}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \mathit{expr}\llbracket\ E_2\ \rrbracket, \mathit{expr}\llbracket\ E_3\ \rrbracket)$$

$$\mathit{expr}\llbracket\ \texttt{while}\ E_1\ \texttt{do}\ E_2\ \texttt{done}\ \rrbracket = \textbf{while-true}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \textbf{effect}(\mathit{expr}\llbracket\ E_2\ \rrbracket))$$

$$\begin{aligned}&\mathit{expr}\llbracket\ \texttt{for}\ I\ \texttt{=}\ E_1\ \texttt{to}\ E_2\ \texttt{do}\ E_3\ \texttt{done}\ \rrbracket =\\&\quad\textbf{apply-to-each}(\textbf{patt-abs}(\textbf{bind}(\mathit{id}\llbracket\ I\ \rrbracket), \textbf{effect}(\mathit{expr}\llbracket\ E_3\ \rrbracket)),\\&\qquad\textbf{int-closed-interval}(\mathit{expr}\llbracket\ E_1\ \rrbracket, \mathit{expr}\llbracket\ E_2\ \rrbracket))\end{aligned}$$

$$\begin{aligned}&\mathit{expr}\llbracket\ \texttt{for}\ I\ \texttt{=}\ E_1\ \texttt{downto}\ E_2\ \texttt{do}\ E_3\ \texttt{done}\ \rrbracket =\\&\quad\textbf{apply-to-each}(\textbf{patt-abs}(\textbf{bind}(\mathit{id}\llbracket\ I\ \rrbracket), \textbf{effect}(\mathit{expr}\llbracket\ E_3\ \rrbracket)),\\&\qquad\textbf{list-reverse}(\textbf{int-closed-interval}(\mathit{expr}\llbracket\ E_2\ \rrbracket, \mathit{expr}\llbracket\ E_1\ \rrbracket)))\end{aligned}$$

$$\mathit{expr}\llbracket\ E_1\ \texttt{;}\ E_2\ \rrbracket = \textbf{seq}(\textbf{effect}(\mathit{expr}\llbracket\ E_1\ \rrbracket), \mathit{expr}\llbracket\ E_2\ \rrbracket)$$

$$\begin{aligned}&\mathit{expr}\llbracket\ \texttt{try}\ E\ \texttt{with}\ \mathit{SM}\ \rrbracket =\\&\quad\textbf{catch-else-rethrow}(\mathit{expr}\llbracket\ E\ \rrbracket,\\&\qquad\textbf{restrict-domain}(\mathit{func}\llbracket\ \mathit{SM}\ \rrbracket, \textbf{bound-type}(\mathit{id}\llbracket\ \texttt{exn}\ \rrbracket)))\end{aligned}$$

$$\mathit{expr}\llbracket\ \texttt{let}\ \mathit{VD}\ \texttt{in}\ E\ \rrbracket = \textbf{scope}(\mathit{decl}\llbracket\ \mathit{VD}\ \rrbracket, \mathit{expr}\llbracket\ E\ \rrbracket)$$

$$\begin{aligned}&\mathit{expr}\llbracket\ \texttt{match}\ E\ \texttt{with}\ \mathit{SM}\ \rrbracket =\\&\quad\textbf{apply}(\textbf{prefer-over}(\mathit{func}\llbracket\ \mathit{SM}\ \rrbracket, \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure}))), \mathit{expr}\llbracket\ E\ \rrbracket)\end{aligned}$$

$$\mathit{expr}\llbracket\ \texttt{function}\ \mathit{SM}\ \rrbracket = \textbf{prefer-over}(\mathit{func}\llbracket\ \mathit{SM}\ \rrbracket, \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure})))$$

Pattern Matching

$$\mathit{func}\llbracket\ P\ \texttt{->}\ E\ \texttt{|}\ \mathit{SM}\ \rrbracket = \textbf{prefer-over}(\mathit{func}\llbracket\ P\ \texttt{->}\ E\ \rrbracket, \mathit{func}\llbracket\ \mathit{SM}\ \rrbracket)$$

$$\mathit{func}\llbracket\ P\ \texttt{->}\ E\ \rrbracket = \textbf{close}(\textbf{patt-abs}(\mathit{patt}\llbracket\ P\ \rrbracket, \mathit{expr}\llbracket\ E\ \rrbracket))$$

Let Bindings

$\textit{decl}\llbracket\ \texttt{rec}\ LB\ \cdots\ \rrbracket =$

$\quad \textbf{generalise-decl}(\textbf{recursive-typed}(\textit{bound-ids}\llbracket\ LB\ \cdots\ \rrbracket, \textit{decl-mono}\llbracket\ LB\ \cdots\ \rrbracket))$

$\textit{bound-ids}\llbracket\ LB_1\ \texttt{and}\ LB_2\ \cdots\ \rrbracket =$

$\quad \textbf{map-union}(\textit{bound-ids}\llbracket\ LB_1\ \rrbracket, \textit{bound-ids}\llbracket\ LB_2\ \cdots\ \rrbracket)$

$\textit{bound-ids}\llbracket\ I\ \texttt{=}\ E\ \rrbracket = \textbf{map1}(\textit{id}\llbracket\ I\ \rrbracket, \textbf{unknown-type})$

$\textit{bound-ids}\llbracket\ \texttt{(}\ I\ \texttt{:}\ T\ \texttt{)}\ \texttt{=}\ E\ \rrbracket = \textbf{map1}(\textit{id}\llbracket\ I\ \rrbracket, \textit{type}\llbracket\ T\ \rrbracket)$

$\textit{decl}\llbracket\ LB_1\ \texttt{and}\ LB_2\ \cdots\ \rrbracket = \textbf{map-union}(\textit{decl}\llbracket\ LB_1\ \rrbracket, \textit{decl}\llbracket\ LB_2\ \cdots\ \rrbracket)$

$\textit{decl}\llbracket\ P\ \texttt{=}\ E\ \rrbracket = \textbf{generalise-decl-if-true}(\textit{val-res}\llbracket\ E\ \rrbracket, \textit{decl-mono}\llbracket\ P\ \texttt{=}\ E\ \rrbracket)$

$\textit{decl-mono}\llbracket\ LB_1\ \texttt{and}\ LB_2\ \cdots\ \rrbracket =$

$\quad \textbf{map-union}(\textit{decl-mono}\llbracket\ LB_1\ \rrbracket, \textit{decl-mono}\llbracket\ LB_2\ \cdots\ \rrbracket)$

$\textit{decl-mono}\llbracket\ P\ \texttt{=}\ E\ \rrbracket =$

$\quad \textbf{match}(\textit{expr}\llbracket\ E\ \rrbracket, \textbf{prefer-over}(\textit{patt}\llbracket\ P\ \rrbracket, \textbf{abs}(\textbf{throw}(\textbf{cl-match-failure}))))$

$\textit{val-res}\llbracket\ \texttt{function}\ SM\ \rrbracket = \textbf{true}$

$\textit{val-res}\llbracket\ C\ \rrbracket = \textbf{true}$

$\textit{val-res}\llbracket\ I\ \rrbracket = \textbf{true}$

$\textit{val-res}\llbracket\ \texttt{[|}\ \texttt{|]}\ \rrbracket = \textbf{true}$

$\textit{val-res}\llbracket\ \texttt{(}E\ \texttt{:}\ T\texttt{)}\ \rrbracket = \textit{val-res}\llbracket\ E\ \rrbracket$

$\textit{val-res}\llbracket\ E_1\ \texttt{,}\ E_2\ \rrbracket = \textbf{and}(\textit{val-res}\llbracket\ E_1\ \rrbracket, \textit{val-res}\llbracket\ E_2\ \rrbracket)$

$\textit{val-res}\llbracket\ E_1\ \texttt{,}\ E_2\ \texttt{,}\ E_3\ \cdots\ \rrbracket = \textbf{and}(\textit{val-res}\llbracket\ E_1\ \rrbracket, \textit{val-res}\llbracket\ E_2\ \texttt{,}\ E_3\ \cdots\ \rrbracket)$

$\textit{val-res}\llbracket\ E_1\ \texttt{::}\ E_2\ \rrbracket = \textbf{and}(\textit{val-res}\llbracket\ E_1\ \rrbracket, \textit{val-res}\llbracket\ E_2\ \rrbracket)$

$\textit{val-res}\llbracket\ \texttt{[}\ E\ \texttt{]}\ \rrbracket = \textit{val-res}\llbracket\ E\ \rrbracket$

$\textit{val-res}\llbracket\ \texttt{[}\ E_1\ \texttt{;}\ E_2\ \cdots\ \texttt{]}\ \rrbracket = \textbf{and}(\textit{val-res}\llbracket\ E_1\ \rrbracket, \textit{val-res}\llbracket\ \texttt{[}\ E_2\ \cdots\ \texttt{]}\ \rrbracket)$

$\textit{val-res}\llbracket\ E\ \rrbracket = \textbf{false}$

References

1. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.J.: Safe specification of operator precedence rules. In: SLE 2013. LNCS, vol. 8225, pp. 137–156. Springer, Heidelberg (2013)
2. Bach Poulsen, C., Mosses, P.D.: Deriving pretty-big-step semantics from small-step semantics. In: ESOP 2014. LNCS, vol. 8410, pp. 270–289. Springer, Heidelberg (2014)
3. Bach Poulsen, C., Mosses, P.D.: Generating specialized interpreters for modular structural operational semantics. In: LOPSTR'13. LNCS, vol. 8901, pp. 220–236. Springer, Heidelberg (2015)
4. Bergstra, J.A., Heering, J., Klint, P. (eds.): Algebraic Specification. ACM Press/Addison-Wesley (1989)
5. Bogdănaş, D., Roşu, G.: K-Java: A Complete Semantics of Java. In: POPL'15. ACM (2015)
6. Börger, E., Fruja, N.G., Gervasi, V., Stärk, R.F.: A high-level modular definition of the semantics of C#. Theor. Comput. Sci. 336(2-3), 235–284 (2005)
7. Börger, E., Stärk, R.F.: Exploiting abstraction for specification reuse: The Java/C# case study. In: FMCO 2003. LNCS, vol. 3188, pp. 42–76. Springer, Heidelberg (2003)
8. van den Brand, M.G.J., van Deursen, A., Heering, J., et al.: The ASF+SDF Meta-Environment: a component-based language development environment. In: CC'01. LNCS, vol. 2027, pp. 365–370. Springer, Heidelberg (2001)
9. van den Brand, M.G.J., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. 61(3), 245–264 (2006)
10. Chalub, F., Braga, C.: Maude MSOS tool (accessed January 2015), `https://github.com/fcbr/mmt`
11. Churchill, M., Mosses, P.D.: Modular bisimulation theory for computations and values. In: FoSSaCS 2013. LNCS, vol. 7794, pp. 97–112. Springer, Heidelberg (2013)
12. Churchill, M., Mosses, P.D., Sculthorpe, N., Torrini, P.: Reusable components of semantic specifications: Additional material (2015), `http://www.plancomps.org/taosd2015`
13. Churchill, M., Mosses, P.D., Torrini, P.: Reusable components of semantic specifications. In: Modularity'14. pp. 145–156. ACM (2014)
14. Delaware, B., Keuchel, S., Schrijvers, T., Oliveira, B.C.: Modular monadic meta-theory. In: ICFP'13. pp. 319–330. ACM (2013)
15. Doh, K.G., Mosses, P.D.: Composing programming languages by combining action-semantics modules. Sci. Comput. Program. 47(1), 3–36 (2003)
16. Doh, K.G., Schmidt, D.A.: Action semantics-directed prototyping. Comput. Lang. 19, 213–233 (1993)
17. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL'12. pp. 533–544. ACM (2012)
18. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press, Cambridge, MA, USA (2009)
19. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. 103(2), 235–271 (1992)
20. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press, Cambridge, MA, USA (1996)
21. Harper, R., Stone, C.: A type-theoretic interpretation of Standard ML. In: Proof, Language and Interaction: Essays in Honour of Robin Milner. MIT Press, Cambridge, MA, USA (2000)

22. Heering, J., Klint, P.: Prehistory of the ASF+SDF system (1980–1984). In: ASF+SDF95. pp. 1–4. Programming Research Group, University of Amsterdam (1995), tech. rep. 9504
23. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A history of Haskell: Being lazy with class. In: HOPL-III. pp. 1–55. ACM (2007)
24. Iversen, J., Mosses, P.D.: Constructive action semantics for Core ML. Software, IEE Proceedings 152, 79–98 (2005), special issue on Language Definitions and Tool Generation
25. Johnstone, A., Mosses, P.D., Scott, E.: An agile approach to language modelling and development. Innov. Syst. Softw. Eng. 6(1-2), 145–153 (2010), special issue for ICFEM workshop FM+AM'09
26. Johnstone, A., Scott, E.: Translator generation using ART. In: SLE 2010. LNCS, vol. 6563, pp. 306–315. Springer, Heidelberg (2011)
27. Kahn, G.: Natural semantics. In: STACS'87. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
28. Kats, L.C.L., Visser, E.: The Spoofax language workbench. In: SPLASH/OOPSLA Companion. pp. 237–238. ACM (2010)
29. Klein, C., et al.: Run your research: On the effectiveness of lightweight mechanization. In: POPL'12. pp. 285–296. ACM (2012)
30. Kutter, P.W., Pierantonio, A.: Montages specifications of realistic programming languages. J. Univ. Comput. Sci. 3(5), 416–442 (1997)
31. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of Standard ML. In: POPL'07. pp. 173–184. ACM (2007)
32. Leroy, X.: Caml Light manual (Dec 1997), `http://caml.inria.fr/pub/docs/manual-caml-light`
33. Levin, M.Y., Pierce, B.C.: TinkerType: A language for playing with formal systems. J. Funct. Program. 13(2), 295–316 (Mar 2003)
34. Lewis, J.R., Launchbury, J., Meijer, E., Shields, M.B.: Implicit parameters: Dynamic scoping with static types. In: POPL 2000. pp. 108–118. ACM (2000)
35. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL'95. pp. 333–343 (1995)
36. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing 1962. pp. 21–28. North-Holland (1962)
37. Meseguer, J., Roşu, G.: The rewriting logic semantics project: A progress report. In: FCT 2011. LNCS, vol. 6914, pp. 1–37. Springer, Heidelberg (2011)
38. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
39. Moggi, E.: An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Edinburgh Univ. (1989)
40. Mosses, P.D.: Action Semantics, Cambridge Tracts in Theoretical Computer Science, vol. 26. Cambridge University Press (1992)
41. Mosses, P.D.: Theory and practice of action semantics. In: MFCS'96. LNCS, vol. 1113, pp. 37–61. Springer, Heidelberg (1996)
42. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228 (2004)
43. Mosses, P.D.: A constructive approach to language definition. J. Univ. Comput. Sci. 11(7), 1117–1134 (2005)
44. Mosses, P.D.: Teaching semantics of programming languages with Modular SOS. In: Teaching Formal Methods: Practice and Experience. Electr. Workshops in Comput., BCS (2006)

45. Mosses, P.D.: Component-based description of programming languages. In: Visions of Computer Science. pp. 275–286. Electr. Proc., BCS (2008)
46. Mosses, P.D.: Component-based semantics. In: SAVCBS'09. pp. 3–10. ACM (2009)
47. Mosses, P.D.: VDM semantics of programming languages: Combinators and monads. Formal Aspects Comput. 23, 221–238 (2011)
48. Mosses, P.D.: Semantics of programming languages: Using ASF+SDF. Sci. Comput. Program. 97(1), 2–10 (2013)
49. Mosses, P.D., New, M.J.: Implicit propagation in structural operational semantics. In: SOS 2008. Electr. Notes Theor. Comput. Sci., vol. 229(4), pp. 49–66. Elsevier (2009)
50. Mosses, P.D., Vesely, F.: Funkons: Component-based semantics in K. In: WRLA 2014. LNCS, vol. 8663, pp. 213–229. Springer, Heidelberg (2014)
51. Mosses, P.D., Watt, D.A.: The use of action semantics. In: Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986. pp. 135–166. Elsevier (1987)
52. Owens, S., Peskine, G., Sewell, P.: A formal specification for OCaml: the core language. Tech. rep., University of Cambridge (2008)
53. Owens, S.: A sound semantics for OCaml light. In: ESOP 2008. LNCS, vol. 4960, pp. 1–15. Springer, Heidelberg (2008)
54. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge, MA, USA (2002)
55. PLanCompS: Programming language components and specifications (2011), `http://www.plancomps.org`
56. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
57. Plotkin, G.D., Power, A.J.: Computational effects and operations: An overview. In: Proc. Workshop on Domains VI. Electr. Notes Theor. Comput. Sci., vol. 73, pp. 149–163. Elsevier (2004)
58. Roşu, G., Şerbănuţă, T.F.: K overview and SIMPLE case study. Electr. Notes Theor. Comput. Sci. 304, 3–56 (2014)
59. Sewell, P., Nardelli, F.Z., Owens, S., et al.: Ott: Effective tool support for the working semanticist. J. Funct. Program. 20, 71–122 (2010)
60. Tofte, M.: Type inference for polymorphic references. Inf. Comput. 89(1), 1–34 (1990)
61. Visser, E.: Syntax Definition for Language Prototyping. Ph.D. thesis, University of Amsterdam (1997)
62. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: RTA 2001. LNCS, vol. 2051, pp. 357–362. Springer, Heidelberg (2001)