Reusable Components of Semantic Specifications

Martin Churchill Peter D. Mosses Paolo Torrini

Swansea University, Swansea, UK

martin.churchill@keble.oxon.org, {p.d.mosses, p.torrini}@swansea.ac.uk

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

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.13 [*Software Engineering*]: Reusable Software

Keywords modularity; reusability; co-evolution; 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.

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.

It is well known that various semantic frameworks do not have good modularity. For example, using structural operational semantics (SOS) [39] we might start by specifying the evaluation of conditional expressions as follows.

$$\boxed{E \rightarrow E'}$$

$$\frac{E_1 \rightarrow E_1'}{E_1 \,?\, E_2 : E_3, \rightarrow E_1' \,?\, E_2 : E_3} \tag{1}$$

$$\texttt{true} \,?\, E_2 : E_3 \rightarrow E_2 \tag{2}$$

$$\texttt{false} \,?\, E_2 : E_3 \rightarrow E_3 \tag{3}$$

The transition formula $E \rightarrow E'$ 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 '$E_1 \,?\, E_2 : E_3$' involves computing the value of E_1; the axioms (2) and (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 '$E_1 \,?\, E_2 : E_3$'; if it terminates with a value other than `true` or `false`, the computation of '$E_1 \,?\, E_2 : E_3$' is stuck: it cannot make any further steps.

If we are specifying the semantics of a simple imperative language, we would specify the evaluation of an assignment expression '$I = E$', assigning the value of E to a simple variable named I, as follows.

$$\boxed{\rho \vdash (E, \sigma) \rightarrow (E', \sigma')}$$

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

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

The environment ρ represents the current bindings of identifiers (e.g., to declared variables) and the store σ represents the values currently assigned to variables. The formula $\rho \vdash (E, \sigma) \rightarrow (E', \sigma')$ asserts that, after making the step, E' remains to be 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 σ' reflects the assignment of that value to the variable bound to I in ρ.

However, if conditional expressions are included in the same language as the above assignment expressions, conventional SOS requires their semantics to be specified using the same form of

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2772-5/14/04…$15.00.
http://dx.doi.org/10.1145/2577080.2577099

transition formulae, $\rho \vdash (E, \sigma) \rightarrow (E', \sigma')$, so we need to reformulate rules (1–3) as follows.

$$\frac{\rho \vdash (E_1, \sigma) \rightarrow (E_1', \sigma')}{\rho \vdash (E_1 \mathbin{?} E_2 : E_3, \sigma) \rightarrow (E_1' \mathbin{?} E_2 : E_3, \sigma')} \quad (6)$$

$$\rho \vdash (\texttt{true} \mathbin{?} E_2 : E_3, \sigma) \rightarrow (E_2, \sigma) \quad (7)$$

$$\rho \vdash (\texttt{false} \mathbin{?} E_2 : E_3, \sigma) \rightarrow (E_3, \sigma) \quad (8)$$

In effect, we have to *weave* the extra arguments of the required transition formulae (here ρ, σ and σ') into the original rules.

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 $\rho \vdash E \rightarrow E'$; in a process language, they would involve labels on transitions, e.g., $E \xrightarrow{\alpha} E'$. 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 means that in conventional SOS, it is simply not possible to specify the semantics of conditional expressions (or any other programming constructs) once and for all.

A further issue affecting potential reuse of parts of language specifications is the common practice of using notation from the concrete syntax of a language when defining its semantics. For instance, the SOS rules illustrated above are based on the following fragment of a grammar for expressions:

$$E : \mathit{exp} ::= \mathit{exp} \mathbin{?} \mathit{exp} : \mathit{exp} \quad (9)$$

Such grammars provide a concise and suggestive specification of the compositional structure of programs, and are generally preferred to the original style of abstract syntax specification developed by McCarthy [22]. They are typically highly ambiguous, but semantics is defined on abstract syntax trees, making it independent of parsing and disambiguation issues. Regarding modularity, however, the use of concrete terminal symbols entails that our SOS rules for '$E \mathbin{?} E : E$' cannot be directly reused for a language using different concrete syntax for conditional expressions, e.g., '$\texttt{if}\, E\, \texttt{then}\, E\, \texttt{else}\, E$'.

Without support for reuse and co-evolution, the development and subsequent revision of a formal semantics for a major programming language is inherently a huge effort, often regarded as disproportionate to the benefits [13].

Our component-based approach to semantics addresses both the above modularity issues. 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,* using a modular variant of SOS called MSOS [28]. 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 '$E_1 \mathbin{?} E_2 : E_3$' considered above.

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 '$E_1 \mathbin{?} E_2 : E_3$' is 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 involves inserting operations to test the value of E_1.

Each funcon has both static and dynamic semantics. 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. For example, our '**if-true**' funcon requires its second and third arguments to have a common supertype, but the intended static semantics of '$E_1 \mathbin{?} E_2 : E_3$' might require checking for inclusion between the minimal types of E_2 and E_3. Funcons for making such static checks have vacuous dynamic semantics.

The funcon specifications are expected to be highly reusable components of language specifications. 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 widely reusable. We explain all this further, and provide some simple introductory examples, in Sect. 2.

The main contribution of this paper is in Sect. 3, where we illustrate the modularity and practical applicability of our approach by presenting excerpts from a case study: a component-based semantics of Caml Light [18]. 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.[1] We have also tested the correspondence between our component-based semantics of Caml Light and the standard implementation of the language, by running programs using a (modular!) interpreter generated from the MSOS specifications of the funcons [1, 2]. Preliminary tool support for our language specifications is based on SDF [11] and Prolog; the PLanCompS project [38] is developing further tool support. We are also carrying out major case studies, to demonstrate the extent to which funcons can be reused in specifications of different languages.

Finally, we discuss related work and alternative approaches in Sect. 4, before concluding and outlining further work in Sect. 5.

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 funcons.

2.1 Funcon Syntax

As mentioned in the Introduction, many funcons correspond closely to simplified programming language constructs. However, each funcon has fixed syntax and semantics. For example, the funcon written **assign**(E_1, E_2) always has the effect of evaluating E_1 to a variable, E_2 to a value (in any order), then assigning the value to the variable. 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.

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. For any sort X of values, let $\mathit{comp}(X)$ be the sort of computations which, whenever they terminate normally, compute values of sort X. The following computation sorts reflect fundamental conceptual distinctions in programming languages.

- The sort of *commands* $\mathit{comm} = \mathit{comp}(\mathit{skip})$ is for funcons that are executed for their effects: the sort skip has only one value.
- The sort of *expressions* $\mathit{expr} = \mathit{comp}(\mathit{value})$ is for funcons that compute values of the language-dependent sort value (they might also have effects).
- The sort of *declarations* $\mathit{decl} = \mathit{comp}(\mathit{env})$ is for funcons that compute environments, mapping identifiers to values.

[1] www.plancomps.org/churchill2014

assign(*var*, *value*)	: *comm*
assigned-value(*var*)	: *expr*
bound-value(*id*)	: *expr*
effect(*value*)	: *comm*
given	: *expr*
if-true(*boolean*, *comp*(X), *comp*(X))	: *comp*(X)
seq(*skip*, *comp*(X))	: *comp*(X)
supply(*value*, *comp*(X))	: *comp*(X)
while-true(*expr*, *comm*)	: *comm*

Table 1. Some funcon signatures

Note that *comp*(X) includes X as a subsort: we regard values as terminated computations.

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

The following sorts used in Table 1 are all value sorts: *boolean* (the values **false** and **true**), *id* (identifiers, denoting values given by **bound-value**), *skip* (the value **skip**), *value* (all values) and *var* (imperative variables). Further value sorts include familiar data types such as *int* (the unbounded integers) and instances of generic data types such as *list*(X) and *map*(X,Y). Abstraction values of sort *abs*(X,Y) are formed from computations. New value sorts can be defined using algebraic data types and subsort inclusions.

Lifting We can lift operations from value sorts to computation sorts. For example, consider the negation operation **not**(*boolean*) : *boolean*. By lifting the signature to **not**(*expr*) : *expr* we can use **not** as a funcon. The value of **not**(E) is computed by first computing the value of E, then (provided that this is a *boolean* value) applying the negation operation. The same principle applies to funcons with a value sort argument, such as **assigned-value**. Its lifted signature is **assigned-value**(*expr*) : *expr*, and the computation of the argument value is followed by applying the original funcon to it. When we lift value operations and funcons with two or more value sort arguments, the argument values may be computed in any order; the funcons **given** and **supply** can be used to restrict to sequential evaluation of lifted arguments, when required, as illustrated below.

The lifted signatures determine a set of *well-sorted terms* for each sort. The well-sortedness of a funcon term is independent of its context.

2.2 Language Semantics

We next consider how to specify translations from programming languages to funcons. The translation of complete programs to funcon terms determines 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 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 *expr* thus:

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

Using the name '*expr*' for both the function and its target sort makes it easy to see that our funcon terms below are well-sorted. 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 *boolean* to *expr* in its first argument), so we can specify their translation very simply indeed:

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

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:

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

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 *expr*. Lifting also allows the following straightforward translation of equality test expressions.

$$\mathit{expr}[\![E_1 == E_2]\!] = \quad \boxed{\mathit{exp} ::= \mathit{exp} == \mathit{exp}}$$
$$\textbf{equal}(\mathit{expr}[\![E_1]\!], \mathit{expr}[\![E_2]\!]) \tag{12}$$

To specify left-to-right evaluation of E_1, E_2, we can use the funcons **supply** and **given**, as follows.

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

When identifiers can be bound only to (imperative) variables, we translate an identifier I occurring in an expression so that it gives the value currently assigned to the variable:

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

If identifiers can also be bound to other sorts of values, we use a funcon (not illustrated here) that inspects the assigned value when its argument is a variable, and otherwise returns its argument.

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

$$\mathit{comm}[\![\mathit{stm}]\!] : \mathit{comm}$$

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

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

The following translation of assignment *expressions* illustrates repeated use of a previously computed value.

$$\mathit{expr}\,[\![I = E]\!] = \qquad \boxed{\mathit{exp} ::= \mathit{id} = \mathit{exp}}$$

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

The combination of assignment expressions and the following expression-statements (which discard the value of E) makes the separate specification of assignment *statements* in (15) redundant.

$$\mathit{comm}\,[\![E\ ;]\!] = \qquad \boxed{\mathit{stm} ::= \mathit{exp}\ ;}$$

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

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

$$\boxed{\mathit{stm} ::= \texttt{if}\ (\ \mathit{exp}\ )\ \mathit{stm}\ \texttt{else}\ \mathit{stm}}$$

$$\mathit{comm}\,[\![\texttt{if}\,(\ E\ )\ S_1\ \texttt{else}\ S_2]\!] =$$

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

For if-then statements without an else-part, we can exploit the usual 'desugaring', which we specify by the following equation.

$$\mathit{comm}\,[\![\texttt{if}\,(\ E\ )\ S]\!] = \qquad \boxed{\mathit{stm} ::= \texttt{if}\ (\ \mathit{exp}\ )\ \mathit{stm}}$$

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

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 as simple as one might expect:

$$\mathit{comm}\,[\![\{\ \ \}]\!] = \qquad \boxed{\mathit{stm} ::= \{\ \ \}}$$

$$\textbf{skip} \tag{20}$$

While-statements correspond exactly to our **while-true** funcon (without any lifting):

$$\mathit{comm}\,[\![\texttt{while}\,(\ E\ )\ S]\!] = \qquad \boxed{\mathit{stm} ::= \texttt{while}\ (\ \mathit{exp}\ )\ \mathit{stm}}$$

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

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).

$$\mathit{comm}\,[\![S_1\ S_2\ \cdots]\!] = \qquad \boxed{\mathit{stm} ::= \mathit{stm}\ \mathit{stm}^+}$$

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

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 Funcon Semantics

The preceding subsections illustrate how we use sorts and signatures to specify the syntax of funcons, and how we specify translation functions that map programs to funcon terms. We now explain how to specify the semantics of each funcon, once and for all.

MSOS Modular SOS [28] is a simple variant of structural operational semantics (SOS). 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* on transitions. Thus transition formulae for expressions are always of the form $E \xrightarrow{L} E'$ (and similarly for other sorts of 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.[2]

For example, the following MSOS rules for conditional expressions '$E_1\ ?\ E_2\ :\ E_3$' could be used for both imperative and for purely functional languages:

$$\boxed{E \xrightarrow{L} E'}$$

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

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

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

The variable τ varies over all *unobservable* labels. By not mentioning specific auxiliary entities, the rules assume neither their presence nor their absence, ensuring reusability. This also makes the rules significantly simpler to read.

The MSOS rules for assignment expressions are as follows.

$$\boxed{E \xrightarrow{L} E'}$$

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

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

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 an environment ρ and a pair of stores σ, σ'. The inclusion of τ in a label specifies that any further components must be unobservable.

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

I-MSOS Although MSOS successfully addresses the modularity issues of SOS, the requirement to label all transitions is an unwelcome notational burden. The *Implicitly-Modular SOS (I-MSOS)* framework [34] combines the benefits of MSOS regarding 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, without requiring the introduction of explicit labels on transitions.

[2] In fact labels in MSOS are the morphisms of a *category*, and the unobservable labels are identity morphisms.

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. [37]), 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., $\mathsf{env}\,\rho \vdash E \to E'$.
- Stores σ (and any other entities that can be *updated* by transitions) are written after the syntactic source and target, e.g., $(E, \mathsf{store}\,\sigma) \to (E', \mathsf{store}\,\sigma')$.
- Signals ε (and any other entities *emitted* by transitions) are written as labels on transition symbols, e.g., $E \xrightarrow{\mathsf{exception}\,\varepsilon} E'$.

The markers such as env, store and exception are used in case further entities are needed in the same position.

The I-MSOS rules for conditional expressions are formulated *exactly* as (1–3) in Sect. 1; those for assignment expressions need to be augmented with 'store' markers, but entities propagated between the premise and conclusion of a rule can be left implicit:

$$\boxed{\mathsf{env}\,\rho \vdash (E, \mathsf{store}\,\sigma) \to (E', \mathsf{store}\,\sigma')}$$

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

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

When specifying funcons, the so-called 'patience' rules for evaluation of lifted arguments are left implicit, which significantly improves the conciseness of our specifications. For instance, the second argument V_2 of **assign** below (30) can be lifted from *value* to *expr*, but the rule for its patient evaluation, corresponding to (28), does not need to be given.

It is straightforward to generate MSOS rules directly from I-MSOS rules (and label categories from transition formulae declarations). The foundations of MSOS [28], together with its recently developed modular bisimulation theory and congruence format [7], provide correspondingly modular foundations for I-MSOS specifications.

Typing Rules MSOS and I-MSOS can also be used to specify typing rules, allowing auxiliary entities such as typing contexts to be left implicit in most rules. Typing formulae such as $E : T$ are similar to big-step evaluation formulae, where an expression (statically) computes a type. Following convention, we denote the current typing context by Γ. When specifying typing rules for a funcon with arguments of value sorts, the arguments are lifted to expressions.

I-MSOS Specifications of Funcons The following I-MSOS rules define the semantics of the funcons whose signatures are listed in Table 1. In these rules the meta-variable C ranges over *comm*, D over *decl*, E over *expr*, T over *type*, V over *value*, and X over arbitrary computations (including their computed values).

Assignment commands: $\boxed{(C, \mathsf{store}\,\sigma) \to (C', \mathsf{store}\,\sigma')}$

$$(\textbf{assign}(V_1, V_2), \mathsf{store}\,\sigma) \to (\textbf{skip}, \mathsf{store}\,\sigma[V_1 \mapsto V_2]) \tag{30}$$

$$\boxed{C : \textbf{comm}}$$

$$\frac{E_1 : \textbf{var}(T),\ E_2 : T}{\textbf{assign}(E_1, E_2) : \textbf{comm}} \tag{31}$$

A well-typed command has a unique type, written **comm**.

Variable references: $\boxed{(E, \mathsf{store}\,\sigma) \to (E', \mathsf{store}\,\sigma')}$

$$(\textbf{assigned-value}(V), \mathsf{store}\,\sigma) \to (\sigma(V), \mathsf{store}\,\sigma) \tag{32}$$

$$\boxed{E : T}$$

$$\frac{E : \textbf{var}(T)}{\textbf{assigned-value}(E) : T} \tag{33}$$

Identifier references: $\boxed{\mathsf{env}\,\rho \vdash E \to E'}$

$$\mathsf{env}\,\rho \vdash \textbf{bound-value}(I) \to \rho(I) \tag{34}$$

$$\boxed{\mathsf{env}\,\Gamma \vdash E : T}$$

$$\mathsf{env}\,\Gamma \vdash \textbf{bound-value}(I) : \Gamma(I) \tag{35}$$

Side-effects: $\boxed{C \to C'}$

$$\textbf{effect}(V) \to \textbf{skip} \tag{36}$$

$$\boxed{C : \textbf{comm}}$$

$$\frac{E : T}{\textbf{effect}(E) : \textbf{comm}} \tag{37}$$

Given value: $\boxed{\mathsf{given}\,V \vdash E \to E'}$

$$\mathsf{given}\,V \vdash \textbf{given} \to V \tag{38}$$

$$\boxed{\mathsf{given}\,T \vdash E : T'}$$

$$\mathsf{given}\,T \vdash \textbf{given} : T \tag{39}$$

Conditional choice: $\boxed{E \to E'}$

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \to X_1 \tag{40}$$

$$\textbf{if-true}(\textbf{false}, X_1, X_2) \to X_2 \tag{41}$$

$$\boxed{E : T}$$

$$\frac{E : \textbf{boolean},\ X_1 : T,\ X_2 : T}{\textbf{if-true}(E, X_1, X_2) : T} \tag{42}$$

Sequencing: $\boxed{C \to C'}$

$$\textbf{seq}(\textbf{skip}, X) \to X \tag{43}$$

$$\boxed{C : \textbf{comm}}$$

$$\frac{C : \textbf{comm},\ X : T}{\textbf{seq}(C, X) : T} \tag{44}$$

Supplying a value: $\boxed{\mathsf{given}\,V \vdash X \to X'}$

$$\frac{\mathsf{given}\,V \vdash X \to X'}{\mathsf{given}\,\_ \vdash \textbf{supply}(V, X) \to \textbf{supply}(V, X')} \tag{45}$$

$$\mathsf{given}\,\_ \vdash \textbf{supply}(V_1, V_2) \to V_2 \tag{46}$$

$$\boxed{\mathsf{given}\,T \vdash X : T'}$$

$$\frac{\mathsf{given}\,T_1 \vdash E : T \qquad \mathsf{given}\,T \vdash X : T'}{\mathsf{given}\,T_1 \vdash \textbf{supply}(E, X) : T'} \tag{47}$$

While-loops: $\boxed{C \to C'}$

$$\textbf{while-true}(E, C) \to \textbf{if-true}(E, \textbf{seq}(C, \textbf{while-true}(E, C)), \textbf{skip}) \tag{48}$$

$\boxed{C : \textbf{comm}}$

$$\frac{E : \textbf{boolean},\ C : \textbf{comm}}{\textbf{while-true}(E, C) : \textbf{comm}} \tag{49}$$

Soundness Funcons have signatures specifying the maximal sorts in which each argument and the resulting terms are contained, with respect to a natural subtyping relation. Value sorts (as an open-ended set), together with types of the form $\mathit{abs}(T_1, T_2)$ (corresponding to abstractions at the typing level) define syntactic types that can be used to type funcon terms, thus making it possible to specify inductively minimal sorts of computed values. Each funcon is associated to a typing rule which allows us to derive typing judgements for all the related terms, given a typing assignment for the identifiers in the environment (i.e., a typing context). Well-formed terms, constructed by application of funcons to arguments according to their signatures, are meant to be those that can be typed.

We have polymorphism, needed for languages like Caml Light and Java, and deal with it by simply allowing for identifiers that represent type variables, mapped to types by the environment. The type of a funcon term thus depends on the typing context, as well as on type parameters. For example, the funcon **bound-value** has a typing rule such that when the argument I (an identifier) is assigned type T in the typing context Γ, **bound-value**(I) also has type T – this holds when T is a value sort such as *bool*, as well as when it is a type expression depending on type variables.

All the dynamic rules for the funcons used in our component-based semantics of Caml Light are type preserving. This guarantees type soundness, in the sense that if the translation of a Caml Light expression to a funcon term has type T and it computes a value V, then V is included in the set of values determined by T.

3. An Illustrative Case Study

Caml Light descends from Caml, a predecessor of the language *OCaml*, and is similar to the core of Standard ML [24]. 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 [18]. 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 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 3.2 gives an overview of the required values and funcons; Sect. 3.3 specifies the translation of Caml Light abstract syntax (trees) into combinations of funcons; and Sect. 3.4 specifies the static and dynamic semantics of the funcons using I-MSOS. The full specifications can be found online.[3]

3.1 Caml Light

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 type (e.g. `int->int`). Commands and `()` have type `unit`.

[3] www.plancomps.org/churchill2014

```
let rec (fib : int -> int) = fun 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 array_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. Example Caml Light programs

Some example Caml Light programs can be found in Table 2. First, we see a recursively defined Fibonacci function `fib`, with the explicit type `int->int`. The function is defined using the `fun` 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 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. An 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 is given in Table 3 (using meta-variables as nonterminal symbols, for brevity).

3.2 Values and Funcons

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

Declarations We bind an identifier to a particular value in a declaration using the **bind-value** funcon. To limit the visibility of a declaration to an arbitrary computation, we use the **scope** funcon, which is lifted to act on declarations in its first argument.

Abstractions Values of sort *func* are function abstractions which compute a value from a given value: $\mathit{func} = \mathit{abs}(\mathit{value}, \mathit{value})$. Such abstractions can be constructed using the binary **abs** constructor, which abstracts an expression over a given pattern. They 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. The abstraction **prefer-over**(A_1, A_2) applies A_1, but then applies A_2 if A_1 fails.

Constants

$$C ::= \texttt{()} \mid \texttt{[]} \mid \textit{literals for numbers, characters, strings}$$

Expressions

$$\begin{aligned} E ::=\ & I \mid C \mid \texttt{(}\,E\,\texttt{)} \mid \texttt{begin}\,E\,\texttt{end} \mid \texttt{(}\,E : T\,\texttt{)}\\ \mid\ & E\ (\texttt{,}\,E)^+ \mid K\ E \mid E\,\texttt{::}\,E \mid \texttt{[}\,E\ (\texttt{;}\,E)^*\,\texttt{]}\\ \mid\ & \texttt{[|}\,E\ (\texttt{;}\,E)^*\,\texttt{|]} \mid \texttt{\{}\,L\,\texttt{=}\,E\ (\texttt{;}\,L\,\texttt{=}\,E)^*\,\texttt{\}}\\ \mid\ & E\ E \mid Op\ E \mid E\ Op\ E \mid E\ \texttt{\&}\ E \mid E\ \texttt{or}\ E\\ \mid\ & E\,\texttt{.}\,L \mid E\,\texttt{.}\,L\,\texttt{<-}\,E \mid E\,\texttt{.(}\,E\,\texttt{)} \mid E\,\texttt{.(}\,E\,\texttt{)}\,\texttt{<-}\,E\\ \mid\ & \texttt{if}\,E\,\texttt{then}\,E\ (\texttt{else}\,E)^? \mid \texttt{while}\,E\,\texttt{do}\,E\,\texttt{done}\\ \mid\ & \texttt{for}\,I\,\texttt{=}\,E\ (\texttt{to} \mid \texttt{downto})\ E\,\texttt{do}\,E\,\texttt{done}\\ \mid\ & E\,\texttt{;}\,E \mid \texttt{match}\,E\,\texttt{with}\,SM \mid \texttt{fun}\,MM\\ \mid\ & \texttt{function}\,SM \mid \texttt{try}\,E\,\texttt{with}\,SM\\ \mid\ & \texttt{let}\ (\texttt{rec})^?\ LB\ (\texttt{and}\,LB)^*\ \texttt{in}\,E \end{aligned}$$

Simple Matchings

$$SM ::= P\,\texttt{->}\,E\ (\texttt{|}\ P\,\texttt{->}\,E)^*$$

Multiple Matchings

$$MM ::= P^+\,\texttt{->}\,E\ (\texttt{|}\ P^+\,\texttt{->}\,E)^*$$

Let Bindings

$$LB ::= P\,\texttt{=}\,E \mid I\ P^+\,\texttt{=}\,E$$

Patterns

$$\begin{aligned} P ::=\ & I \mid \_ \mid P\,\texttt{as}\,I \mid \texttt{(}\,P\,\texttt{)} \mid \texttt{(}\,P : T\,\texttt{)} \mid P\texttt{|}P\\ \mid\ & C \mid K\ P \mid P\ (\texttt{,}\,P)^+ \mid \texttt{[]} \mid P\,\texttt{::}\,P\\ \mid\ & \texttt{[}\,P\ (\texttt{;}\,P)^*\ \texttt{]} \mid \texttt{\{}\,L\,\texttt{=}\,P\ (\texttt{;}\,L\,\texttt{=}\,P)^*\,\texttt{\}} \end{aligned}$$

Type Expressions

$$T ::= \texttt{'}\,I \mid \texttt{(}\,T\,\texttt{)} \mid T\,\texttt{->}\,T \mid T\ (\texttt{*}\,T)^+$$

Table 3. An extract of the Caml Light reference grammar, with EBNF replaced by $\cdot^*$, $\cdot^+$, $\cdot^?$, and nonterminals by meta-variables (I ranges over identifiers, K over constructors, and L over labels)

Patterns A pattern is another sort of abstraction, computing an environment from a given value: *patt* = *abs*(*value*, *env*). An example pattern is **any**, which matches any value and produces no bindings, accurately 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. For example, if F is a binary data constructor, the pattern **invert** F (P_1, P_2) will match values of the form $F(X,Y)$ provided X matches P_1 and Y matches P_2, combining the generated bindings.

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 is a variant on this: it rethrows the exception should it fail to be in the domain of the handler.

abs(*expr*)	: *func*
abs(*patt*, *expr*)	: *func*
accum(*env*, *decl*)	: *decl*
any	: *patt*
apply(*func*, *value*)	: *expr*
bind(*id*)	: *patt*
bind-value(*id*, *value*)	: *decl*
catch(*expr*, *func*)	: *expr*
catch-else-rethrow(*expr*, *func*)	: *expr*
close(*func*)	: *expr*
closure(*comp*(X), *env*)	: *comp*(X)
else(*comp*(X), *comp*(X))	: *comp*(X)
generalise-all(*decl*)	: *decl*
instantiate-if-poly(*expr*)	: *expr*
invert F (*patt*, . . . , *patt*)	: *patt*
match(*value*, *patt*)	: *decl*
only(*value*)	: *patt*
patt-union(*patt*, *patt*)	: *patt*
prefer-over(*abs*(X,Y), *abs*(X,Y))	: *abs*(X,Y)
restrict-domain(*abs*(X,Y), *type*)	: *abs*(X,Y)
scope(*env*, *comp*(X))	: *comp*(X)
throw(*exception*)	: *comp*(X)
when-true(*boolean*, *comp*(X))	: *comp*(X)

Table 4. Funcon signatures (see also Table 1)

id $[\![I]\!]$	: *id*	Identifiers
value $[\![C]\!]$	: *value*	Constants
expr $[\![E]\!]$	: *expr*	Expressions
abs $[\![SM]\!]$	: *abs*	Simple Matchings
decl $[\![LB]\!]$	: *decl*	Let Bindings
patt $[\![P]\!]$	: *patt*	Patterns
type $[\![T]\!]$	: *type*	Type Expressions

Table 5. Translation function signatures

3.3 Language Semantics

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

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). If it does not, we may need to add funcons to the translation to reflect the intended compile-time behaviour.

3.3.1 Dynamic Semantics

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

$$\textit{expr}\llbracket \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \rrbracket = \textbf{if-true}(\textit{expr}\llbracket E_1\rrbracket, \textit{expr}\llbracket E_2\rrbracket, \textit{expr}\llbracket E_3\rrbracket) \tag{50}$$

Note that here we are lifting **if-true** to be applied to computations that *might* compute a Boolean in the first argument, from the base signature $\textbf{if-true}(\textit{boolean}, \textit{comp}(X), \textit{comp}(X)) : \textit{comp}(X)$.

Lifting can also be applied to pure data operations, such as $\textbf{not}(\textit{boolean}) : \textit{boolean}$.

$$\textit{expr}\llbracket \texttt{not}\ E_1 \rrbracket = \textbf{not}(\textit{expr}\llbracket E_1\rrbracket) \tag{51}$$

We also use the **if-true** funcon to provide meaning to other productions, e.g., Caml Light's Boolean 'and' operator:

$$\textit{expr}\llbracket E_1\ \texttt{\&}\ E_2 \rrbracket = \textbf{if-true}(\textit{expr}\llbracket E_1\rrbracket, \textit{expr}\llbracket E_2\rrbracket, \textbf{false}) \tag{52}$$

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

$$\textit{expr}\llbracket E_1\ \texttt{;}\ E_2 \rrbracket = \textbf{seq}(\textbf{effect}(\textit{expr}\llbracket E_1\rrbracket), \textit{expr}\llbracket E_2\rrbracket) \tag{53}$$

Here, we explicitly discard the computed value of the first expression, using the **effect** funcon.

Pattern matching We translate Caml Light's simple matching construct SM to a function abstraction using $\textit{abs}\llbracket \_ \rrbracket$. 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:

$$\textit{expr}\llbracket \texttt{match}\ E\ \texttt{with}\ SM \rrbracket = \textbf{apply}(\textbf{prefer-over}(\textit{abs}\llbracket SM\rrbracket, \textbf{abs}(\textbf{any}, \textbf{throw}(\texttt{'Match\_failure'}))), \textit{expr}\llbracket E\rrbracket) \tag{54}$$

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

$$\textit{expr}\llbracket E_1\ E_2 \rrbracket = \textbf{apply}(\textit{expr}\llbracket E_1\rrbracket, \textit{expr}\llbracket E_2\rrbracket) \tag{55}$$

The signature of **apply** indicates that it should be applied to an abstraction and an argument *value*, which is then lifted to take a computation argument. We would specify call-by-name semantics by forming a (parameterless) 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. We use the **close** funcon to specify static bindings, and also specify what should happen if the simple matching SM fails to match the given argument.

$$\textit{expr}\llbracket \texttt{function}\ SM \rrbracket = \textbf{close}(\textbf{prefer-over}(\textit{abs}\llbracket SM\rrbracket, \textbf{abs}(\textbf{any}, \textbf{throw}(\texttt{'Match\_failure'})))) \tag{56}$$

Simple matchings We will next see how $\textit{abs}\llbracket \_ \rrbracket$ translates simple matchings SM to abstractions. For a single body, the binary **abs** funcon captures matchings accurately; sequences of simple matchings are combined using **prefer-over**.

$$\textit{abs}\llbracket P_1\ \texttt{->}\ E_1 \rrbracket = \textbf{abs}(\textit{patt}\llbracket P_1\rrbracket, \textit{expr}\llbracket E_1\rrbracket) \tag{57}$$

$$\textit{abs}\llbracket P_1\ \texttt{->}\ E_1\ \texttt{|}\ P_2\ \texttt{->}\ E_2\ \cdots \rrbracket = \textbf{prefer-over}(\textit{abs}\llbracket P_1\ \texttt{->}\ E_1\rrbracket, \textit{abs}\llbracket P_2\ \texttt{->}\ E_2\ \cdots\rrbracket) \tag{58}$$

Declarations Local declarations are provided in Caml Light by the '$\texttt{let}\ LB\ \texttt{in}\ E$' construct, corresponding to the **scope** funcon:

$$\textit{expr}\llbracket \texttt{let}\ LB\ \texttt{in}\ E \rrbracket = \textbf{scope}(\textit{decl}\llbracket LB\rrbracket, \textit{expr}\llbracket E\rrbracket) \tag{59}$$

Value-definitions are translated to declarations:

$$\textit{decl}\llbracket P\ \texttt{=}\ E \rrbracket = \textbf{match}(\textit{expr}\llbracket E\rrbracket, \textbf{prefer-over}(\textit{patt}\llbracket P\rrbracket, \textbf{abs}(\textbf{any}, \textbf{throw}(\texttt{'Match\_failure'})))) \tag{60}$$

An identifier expression refers to its bound value.

$$\textit{expr}\llbracket I \rrbracket = \textbf{bound-value}(\textit{id}\llbracket I\rrbracket) \tag{61}$$

The preceding two equations account for dynamic semantics. To accurately model Caml Light's let-polymorphism, further details are required, which we will outline in Sect. 3.3.2 below.

Catching exceptions Caml Light's try construct corresponds directly to the **catch-else-rethrow** funcon:

$$\textit{expr}\llbracket \texttt{try}\ E\ \texttt{with}\ SM \rrbracket = \textbf{catch-else-rethrow}(\textit{expr}\llbracket E\rrbracket, \textit{abs}\llbracket SM\rrbracket) \tag{62}$$

Also here, a more refined analysis will be required to accurately capture Caml Light's static semantics.

Basic Patterns We have funcons corresponding directly to Caml Light's basic patterns.

$$\textit{patt}\llbracket I \rrbracket = \textbf{bind}(\textit{id}\llbracket I\rrbracket) \tag{63}$$

$$\textit{patt}\llbracket \_ \rrbracket = \textbf{any} \tag{64}$$

$$\textit{patt}\llbracket C \rrbracket = \textbf{only}(\textit{value}\llbracket C\rrbracket) \tag{65}$$

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 $\textit{expr-tuple}\llbracket \_ \rrbracket$:

$$\textit{expr}\llbracket E_1\ \texttt{,}\ E_2\ \cdots \rrbracket = \textit{expr-tuple}\llbracket E_1\ \texttt{,}\ E_2\ \cdots\rrbracket \tag{66}$$

$$\textit{expr-tuple}\llbracket E_1 \rrbracket = \textbf{tuple-prefix}(\textit{expr}\llbracket E_1\rrbracket, \textbf{tuple-empty}) \tag{67}$$

$$\textit{expr-tuple}\llbracket E_1\ \texttt{,}\ E_2\ \cdots \rrbracket = \textbf{tuple-prefix}(\textit{expr}\llbracket E_1\rrbracket, \textit{expr-tuple}\llbracket E_2\ \cdots\rrbracket) \tag{68}$$

The translation of the corresponding pattern constructors involves **invert** F (where F can be an arbitrary data constructor).

$$\textit{patt}\llbracket P_1\ \texttt{,}\ P_2\ \cdots \rrbracket = \textit{patt-tuple}\llbracket P_1\ \texttt{,}\ P_2\ \cdots\rrbracket \tag{69}$$

$$\textit{patt-tuple}\llbracket P_1 \rrbracket = \textbf{invert tuple-prefix}(\textit{patt}\llbracket P_1\rrbracket, \textbf{only}(\textbf{tuple-empty})) \tag{70}$$

$$\textit{patt-tuple}\llbracket P_1\ \texttt{,}\ P_2\ \cdots \rrbracket = \textbf{invert tuple-prefix}(\textit{patt}\llbracket P_1\rrbracket, \textit{patt-tuple}\llbracket P_2\ \cdots\rrbracket) \tag{71}$$

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

$$\textit{patt}\llbracket P_1\ \texttt{|}\ P_2 \rrbracket = \textbf{prefer-over}(\textit{patt}\llbracket P_1\rrbracket, \textit{patt}\llbracket P_2\rrbracket) \tag{72}$$

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

$$\textit{patt}\llbracket P\ \texttt{as}\ I \rrbracket = \textbf{patt-union}(\textit{patt}\llbracket P\rrbracket, \textbf{bind}(\textit{id}\llbracket I\rrbracket)) \tag{73}$$

3.3.2 Accounting for Static Semantics

The translation specified above accurately reflects 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 such 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.

Catching exceptions The translation of `try` E `with` SM above (62) 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 $\mathit{abs}[\![SM]\!]$ has type `exn->`X for some X. This can be achieved using **restrict-domain**(A, T), which has a type only if the argument type of the abstraction A is T, and which dynamically behaves just like A.

$$\begin{aligned}&\mathit{expr}[\![\texttt{try}\,E\,\texttt{with}\,SM]\!] = \\ &\quad\textbf{catch-else-rethrow}(\mathit{expr}[\![E]\!], \\ &\qquad\textbf{restrict-domain}(\mathit{abs}[\![SM]\!], \\ &\qquad\quad\textbf{bound-type}(\textbf{typeid}(\texttt{'exn'}))))\end{aligned} \tag{74}$$

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

$$\mathit{expr}[\![I]\!] = \textbf{instantiate-if-poly}(\textbf{bound-value}(\mathit{id}[\![I]\!])) \tag{75}$$

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* [42]. The appropriate funcon is **generalise-all**, 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.

$$\begin{aligned}&\mathit{decl}[\![P = E]\!] = \textbf{generalise-all}(\mathit{decl\text{-}mono}[\![P = E]\!]) \\ &\qquad\text{if } E \text{ is generalisable}\end{aligned} \tag{76}$$

$$\begin{aligned}&\mathit{decl}[\![P = E]\!] = \mathit{decl\text{-}mono}[\![P = E]\!] \\ &\qquad\text{if } E \text{ is not generalisable}\end{aligned} \tag{77}$$

The translation funcon $\mathit{decl\text{-}mono}[\![\_]\!]$ is the same as the version of $\mathit{decl}[\![\_]\!]$ specified in Sect. 3.3.1 for dynamic semantics.

$$\begin{aligned}&\mathit{decl\text{-}mono}[\![P = E]\!] = \\ &\quad\textbf{match}(\mathit{expr}[\![E]\!], \textbf{prefer-over}(\mathit{patt}[\![P]\!], \\ &\qquad\textbf{abs}(\textbf{any}, \textbf{throw}(\texttt{'Match\_failure'}))))\end{aligned} \tag{78}$$

Assignment and dereferencing 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.

3.4 Funcon Semantics

In Sect. 2.3, we explained and illustrated how to define the static and dynamic semantics of some simple funcons using *Implicitly-Modular SOS* [34]. We now define some further funcons used in the semantics of Caml Light, involving abstractions, environments, patterns, etc. See Table 4 for the signatures of the funcons.

3.4.1 Scoping

We represent bindings of identifiers to values by environments ρ. The environment $\{I \mapsto V\}$ maps I to V; ρ_1/ρ_2 is the environment where bindings in ρ_1 override bindings for the same identifiers in ρ_2. The current environment is preserved by successive transitions, so in I-MSOS notation it appears before the turnstile.

A declaration computes an environment, and can be made local to a computation X using the **scope** funcon. The following I-MSOS rules define its dynamic semantics.

$$\boxed{\mathsf{env}\,\rho \vdash X \to X'}$$

$$\frac{\mathsf{env}\,(\rho_1/\rho) \vdash X \to X'}{\mathsf{env}\,\rho \vdash \textbf{scope}(\rho_1, X) \to \textbf{scope}(\rho_1, X')} \tag{79}$$

$$\mathsf{env}\,\rho \vdash \textbf{scope}(\rho_1, V) \to V \tag{80}$$

Rule (80) applies only when V is a value, which is always independent of the current bindings. The lifted **scope** funcon, which takes a declaration (computing an environment) as its first argument, is defined by an implicit patience rule determined by the signature.

The following I-MSOS rule defines the static semantics of the lifted **scope** funcon. Notice that the type of a declaration D is a typing context Γ_1.

$$\boxed{\mathsf{env}\,\Gamma \vdash X : T}$$

$$\frac{\mathsf{env}\,\Gamma \vdash D : \Gamma_1 \qquad \mathsf{env}\,(\Gamma_1/\Gamma) \vdash X : T}{\mathsf{env}\,\Gamma \vdash \textbf{scope}(D, X) : T} \tag{81}$$

3.4.2 Abstractions

An abstraction $\textbf{abs}(X)$ is a value constructed from a computation X that may depend on a given argument value. Abstractions have types $\mathit{abs}(T_1, T_2)$, where T_1 is the type of the argument and T_2 is the type of the computation when given that type of argument.

$$\boxed{\mathsf{given}\,T \vdash X : T'}$$

$$\frac{\mathsf{given}\,T_1 \vdash X : T_2}{\mathsf{given}\,\_ \vdash \textbf{abs}(X) : \mathit{abs}(T_1, T_2)} \tag{82}$$

The funcon **apply** takes an abstraction $\textbf{abs}(X)$ and an argument value V, and supplies V to X.

$$\boxed{X \to X'}$$

$$\textbf{apply}(\textbf{abs}(X), V) \to \textbf{supply}(V, X) \tag{83}$$

(The funcon **supply** was introduced in Sect. 2.) The **apply** funcon is lifted in both arguments. Its typing rule is standard:

$$\boxed{E : T}$$

$$\frac{E_1 : \mathit{abs}(T_2, T) \qquad E_2 : T_2}{\textbf{apply}(E_1, E_2) : T} \tag{84}$$

The unary abstraction constructor $\textbf{abs}(X)$ allows X to depend on a single given argument value. The *binary* abstraction funcon $\textbf{abs}(P, X)$ takes also a pattern P, which 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 the binary **abs** funcon.

$$\boxed{E \to E'}$$

$$\textsf{abs}(P, X) \to \textsf{abs}(\textsf{scope}(\textsf{match}(\textsf{given}, P), X)) \tag{85}$$

Here **match** is a pattern-matching funcon, defined in Sect. 3.4.4. Patterns are themselves abstractions, and have types of the form $\textit{abs}(T, \Gamma)$ where Γ is a typing context. The static semantics of binary **abs** is as follows.

$$\boxed{\textsf{env}\,\Gamma \vdash E : T}$$

$$\frac{\textsf{env}\,\Gamma \vdash P : \textit{abs}(T_1, \Gamma_1) \qquad \textsf{env}\,(\Gamma_1/\Gamma) \vdash X : T_2}{\textsf{env}\,\Gamma \vdash \textsf{abs}(P, X) : \textit{abs}(T_1, T_2)} \tag{86}$$

We will omit the typing rules in the rest of this section, for brevity.

3.4.3 Static Scoping

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.

$$\boxed{\textsf{env}\,\rho \vdash E \to E'}$$

$$\textsf{env}\,\rho \vdash \textsf{close}(\textsf{abs}(X)) \to \textsf{abs}(\textsf{closure}(X, \rho)) \tag{87}$$

The funcon **closure** can be used to set the current environment for any computation X:

$$\boxed{\textsf{env}\,\rho \vdash X \to X'}$$

$$\frac{\textsf{env}\,\rho \vdash X \to X'}{\textsf{env}\,\_ \vdash \textsf{closure}(X, \rho) \to \textsf{closure}(X', \rho)} \tag{88}$$

$$\textsf{env}\,\_ \vdash \textsf{closure}(V, \rho) \to V \tag{89}$$

3.4.4 Basic Patterns

Matching the value of an expression E to a pattern P computes an environment. It corresponds to the application of P to E:

$$\boxed{D \to D'}$$

$$\textsf{match}(E, P) \to \textsf{apply}(P, E) \tag{90}$$

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

$$\boxed{P \to P'}$$

$$\textsf{bind}(I) \to \textsf{abs}(\textsf{bind}(I, \textsf{given})) \tag{91}$$

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

$$\textsf{any} \to \textsf{abs}(\emptyset) \tag{92}$$

Other patterns do not match all values. An extreme example is the pattern **only**(V), matching just the single value V:

$$\textsf{only}(V) \to \textsf{abs}(\textsf{when-true}(\textsf{equal}(\textsf{given}, V), \emptyset)) \tag{93}$$

3.4.5 Failure and Back-Tracking

The funcon **when-true**(E, X) guards a computation X with a Boolean-valued condition E. When the value of E is **false**, the funcon emits the signal 'failed **true**' while its computation makes a transition to the funcon **stuck** (which has no further transitions). The signal 'failed **false**' indicates that the computation is proceeding normally, and is treated as unobservable.

$$\boxed{X \xrightarrow{\textsf{failure}\,B} X'}$$

$$\textsf{when-true}(\textsf{true}, X) \xrightarrow{\textsf{failure}\,\textsf{false}} X \tag{94}$$

$$\textsf{when-true}(\textsf{false}, X) \xrightarrow{\textsf{failure}\,\textsf{true}} \textsf{stuck} \tag{95}$$

The funcon **else** allows recovery from failure.

$$\frac{X \xrightarrow{\textsf{failure}\,\textsf{false}} X'}{\textsf{else}(X, Y) \xrightarrow{\textsf{failure}\,\textsf{false}} \textsf{else}(X', Y)} \tag{96}$$

$$\frac{X \xrightarrow{\textsf{failure}\,\textsf{true}} X'}{\textsf{else}(X, Y) \xrightarrow{\textsf{failure}\,\textsf{false}} Y} \tag{97}$$

$$\textsf{else}(V, Y) \xrightarrow{\textsf{failure}\,\textsf{false}} V \tag{98}$$

3.4.6 Compound Patterns

The funcon **else** is used in the definition of the operation **prefer-over** on abstractions and (as a special case) on patterns:

$$\boxed{P \to P'}$$

$$\textsf{prefer-over}(\textsf{abs}(X), \textsf{abs}(Y)) \to \textsf{abs}(\textsf{else}(X, Y)) \tag{99}$$

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**:

$$\textsf{patt-union}(\textsf{abs}(X), \textsf{abs}(Y)) \to \textsf{abs}(\textsf{map-union}(X, Y)) \tag{100}$$

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

3.4.7 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).

$$\boxed{X \xrightarrow{\textsf{exception}\,V} X'}$$

$$\textsf{throw}(V) \xrightarrow{\textsf{exception}\,\textsf{some}(V)} \textsf{stuck} \tag{101}$$

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

$$\boxed{E \xrightarrow{\textsf{exception}\,V} E'}$$

$$\frac{X \xrightarrow{\textsf{exception}\,\textsf{none}} X'}{\textsf{catch}(X, Y) \xrightarrow{\textsf{exception}\,\textsf{none}} \textsf{catch}(X', Y)} \tag{102}$$

$$\frac{X \xrightarrow{\textsf{exception}\,\textsf{some}(V)} X'}{\textsf{catch}(X, Y) \xrightarrow{\textsf{exception}\,\textsf{none}} \textsf{apply}(Y, V)} \tag{103}$$

$$\textsf{catch}(V, Y) \xrightarrow{\textsf{exception}\,\textsf{none}} V \tag{104}$$

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

$$\boxed{E \to E'}$$

$$\textsf{catch-else-rethrow}(E, A) \to \textsf{catch}(E, \textsf{prefer-over}(A, \textsf{abs}(\textsf{throw}(\textsf{given})))) \tag{105}$$

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.

This concludes the presentation of our Caml Light case study.

4. Related Work

Heering and Klint proposed in the early 1980s to structure complete definitions of programming languages as libraries of reusable components [12]. This motivated the development of ASF+SDF [3], which provides strong support for modular structure in algebraic specifications. However, an ASF+SDF definition of a programming language does not, in general, permit the reuse of the individual language constructs in the definitions of other languages. As discussed in [33], the main hindrances to reuse in ASF+SDF are coarse modular structure (e.g., specifying all expression constructs in a single module), explicit propagation of auxiliary entities, and direct specification of language constructs.

At the end of the 1980s, Moggi [25] 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 [32].) Monads avoid explicit propagation of auxiliary entities, and monad transformers are highly reusable components. Various monad transformers have been defined (e.g., see [20]) with operations that in many cases correspond to our funcons; monads also make a clear distinction between values and computations. One drawback of monad transformers is that different orders of composition can lead to different semantics; in contrast, our funcons are independent of the order in which they are added. The concept of monad transformers inspired the development of MSOS, our modular variant of SOS.

An alternative way of defining monads has been developed by Plotkin and Power [40] using Lawvere theories instead of monad transformers. Recently, Delaware et al. [8] presented modular monadic meta-theory, combining modular datatypes with monad transformers, focusing on modularisation of theorems and proofs.

Kutter and Pierantonio [17] 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 and others [4, 5] gave modular ASM semantics for JAVA and C#, identifying features shared by the two languages, but did not define components intended for wider reuse.

Doh and Mosses [9] first proposed replacing the conventional modular structure of specifications in action semantics [26, 27] by a component-based structure, defining the abstract syntax and action semantics of each language construct in a separate module. Iversen and Mosses [14] 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 [6]. The action notation used in action semantics can itself be regarded as a primitive collection of funcons; having to deal with both BAS and action notation was a drawback. Mosses and others [15, 29–31] have reported on subsequent work that led to the present paper.

Levin and Pierce developed the TinkerType system [19] 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.

Another system supporting practical use of conventional SOS is Ott [41], which was used by Owens [36] to specify a sublanguage of OCaml corresponding closely to Caml Light. A type soundness theorem was proved, based on HOL code generated by Ott from the language specification. Ott facilitates use of SOS, but any reuse of previous specifications requires manual copying, pasting and editing, which is not evident in the resulting specification.

Ott supports also reduction semantics based on evaluation contexts. This framework is widely used for proving meta-theoretic results (e.g., type soundness). The PLT-Redex tool [16] runs programs by interpreting their reduction semantics, and has been used to validate language specifications. However, evaluation context grammars appear to be inherently non-modular, which seems to preclude use of reduction semantics to define reusable components.

Competing approaches with a high degree of inherent modularity include Rewriting Logic Semantics [23] and the K Framework [21]. The lifting of funcon arguments from value sorts to computation sorts is closely related to strictness annotations in K. It appears possible to specify individual funcons independently in K, and to use the K Tools to translate programming languages to funcons [35], thereby incorporating our component-based approach directly in that framework.

Haeri and Schupp [10] are developing a novel framework that focuses on reusable components of language implementations. It will be interesting to see how well it scales up to larger languages.

5. 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 [26, 27]) would have increased the size and decreased the perspicuity of the funcon terms used in the translation.

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[4] together with the generated Prolog code for each funcon. At the time of writing, we have not yet checked whether our test programs exercise every translation equation, nor whether running them uses every rule of every funcon. Nevertheless, we are reasonably confident in the accuracy of our specifications.

The work reported here is part of the PLANCOMPS project [38]. 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

[4] `www.plancomps.org/churchill2014`

be to specify the corresponding JAVA constructs using essentially the same collection of funcons as for C#. The project is also aiming to test whether the approach is 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 'patience' 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 the anonymous reviewers and Neil Sculthorpe for their helpful comments on the submitted version of this paper. The reported work was supported by an EPSRC grant (EP/I032495/1) to Swansea University for the PLANCOMPS project.

References

[1] C. Bach Poulsen and P. D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *ESOP'14*, LNCS. Springer, 2014. To appear.

[2] C. Bach Poulsen and P. D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *LOPSTR'13*, LNCS. Springer, 2014. To appear.

[3] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[4] E. Börger and R. F. Stärk. Exploiting abstraction for specification reuse. the Java/C# case study. In *FMCO 2003*, volume 3188 of *LNCS*, pages 42–76. Springer, 2003.

[5] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theor. Comput. Sci.*, 336 (2-3):235–284, 2005.

[6] M. G. J. v. d. Brand, J. Iversen, and P. D. Mosses. An action environment. *Sci. Comput. Program.*, 61(3):245–264, 2006.

[7] M. Churchill and P. D. Mosses. Modular bisimulation theory for computations and values. In *FoSSaCS 2013*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.

[8] B. Delaware, S. Keuchel, T. Schrijvers, and B. C. Oliveira. Modular monadic meta-theory. In *ICFP'13*, pages 319–330. ACM, 2013.

[9] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1): 3–36, 2003.

[10] S. H. Haeri and S. Schupp. Reusable components for lightweight mechanisation of programming languages. In *Software Composition 2013*, volume 8088 of *LNCS*, pages 1–16. Springer, 2013.

[11] J. Heering and P. Klint. *The Syntax Definition Formalism SDF*, chapter 6. In Bergstra et al. [3], 1989.

[12] J. Heering and P. Klint. Prehistory of the ASF+SDF system (1980–1984). In *ASF+SDF95*, pages 1–4. Programming Research Group, University of Amsterdam, 1995. Tech. rep. 9504.

[13] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL-III*, pages 1–55. ACM, 2007.

[14] J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *Software, IEE Proceedings*, 152:79–98, 2005.

[15] A. Johnstone, P. D. Mosses, and E. Scott. An agile approach to language modelling and development. *Innov. Syst. Softw. Eng.*, 6(1-2): 145–153, 2010.

[16] C. Klein et al. Run your research: On the effectiveness of lightweight mechanization. In *POPL'12*, pages 285–296. ACM, 2012.

[17] P. W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *J. Univ. Comput. Sci.*, 3(5):416–442, 1997.

[18] X. Leroy. Caml light manual, Release 0.74, December 1997. URL `http://caml.inria.fr/pub/docs/manual-caml-light`.

[19] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *J. Funct. Program.*, 13(2):295–316, Mar. 2003.

[20] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, pages 333–343, 1995.

[21] D. Lucanu, T.-F. Serbanuta, and G. Rosu. K Framework distilled. In *WRLA 2012*, volume 7571 of *LNCS*, pages 31–53. Springer, 2012.

[22] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 1962*, pages 21–28. North-Holland, 1962.

[23] J. Meseguer and G. Rosu. The rewriting logic semantics project: A progress report. In *FCT 2011*, volume 6914 of *LNCS*, pages 1–37. Springer, 2011.

[24] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[25] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., 1989.

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

[27] P. D. Mosses. Theory and practice of action semantics. In *MFCS'96*, volume 1113 of *LNCS*, pages 37–61. Springer, 1996.

[28] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[29] P. D. Mosses. A constructive approach to language definition. *J. Univ. Comput. Sci.*, 11(7):1117–1134, 2005.

[30] P. D. Mosses. Component-based description of programming languages. In *Visions of Computer Science*, Electr. Proc., pages 275–286. BCS, 2008.

[31] P. D. Mosses. Component-based semantics. In *SAVCBS'09*, pages 3–10. ACM, 2009.

[32] P. D. Mosses. VDM semantics of programming languages: Combinators and monads. *Formal Aspects Comput.*, 23:221–238, 2011.

[33] P. D. Mosses. Semantics of programming languages: Using ASF+SDF. *Sci. Comput. Program.*, 2013.

[34] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *SOS 2008*, volume 229(4) of *Electr. Notes Theor. Comput. Sci.*, pages 49–66. Elsevier, 2009.

[35] P. D. Mosses and F. Vesely. Funkons: Component-based semantics in K. In *WRLA 2014*, LNCS. Springer, 2014. To appear.

[36] S. Owens. A sound semantics for OCaml light. In *ESOP 2008*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008.

[37] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[38] PLanCompS. PLANCOMPS: Programming language components and specifications, 2011. URL `http://www.plancomps.org`.

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

[40] G. D. Plotkin and A. J. Power. Computational effects and operations: An overview. In *Proc. Workshop on Domains VI*, volume 73 of *Electr. Notes Theor. Comput. Sci.*, pages 149–163. Elsevier, 2004.

[41] P. Sewell, F. Z. Nardelli, S. Owens, et al. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20:71–122, 2010.

[42] M. Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.