Journal of Universal Computer Science, vol. 11, no. 7 (2005), 1117-1134
submitted: 28/4/05, accepted: 9/6/05, appeared: 28/7/05 © J.UCS

A Constructive Approach to Language Definition

Peter D. Mosses
(Swansea University, Wales, United Kingdom
p.d.mosses@swansea.ac.uk)

Abstract: Most approaches to formal semantics are based on the assumption that all the constructs of a language are defined together. The details of the definition of each construct can (and usually do) depend on which other constructs are included in the given language. This limits reuse of definitions of common constructs.

With the more constructive approach proposed here, the semantics of each basic abstract programming construct is defined separately and independently. The semantics of a full language is obtained by translating its constructs into the basic abstract constructs, whose definitions are thus reused verbatim.

The frameworks of Modular SOS and Action Semantics can both be used in conjunction with the proposed approach. Some illustrations are given.

Key Words: semantics of programming languages, action semantics, structural operational semantics, modularity
Category: D.3.1, D.3.3, F.3.2, F.3.3

1 Introduction

Haskell 98 is a popular functional programming language. Its design, based largely on ideas enjoying a wide consensus in the functional programming community, was a major collaborative effort, involving many experts over many years. As stated in the Revised Report on Haskell 98 [Peyton Jones, 2003], the Haskell designers originally intended that the language "should be completely described by the publication of a formal syntax and semantics." They published a satisfactory formal syntax, but as yet no formal semantics for Haskell 98 has appeared. Although the large number of syntactic identities listed in the Revised Report are supposed to define a translation that maps arbitrary Haskell programs into what the authors call "the Haskell kernel" [Peyton Jones, 2003, Section 1.2], neither the syntax nor the semantics of the kernel is described, and it has not been verified that the given identities do indeed define a translation function.

One might wonder why the Haskell community appears to have abandoned its stated goal of providing a formal semantics for Haskell 98 (although several useful studies toward that goal have been published, see e.g. [Harrison et al., 2002]). The main reason seems to be that the effort required was perceived as large, and out of proportion to the benefits of complete formalization. In the case of Standard ML, *The Definition* [Milner et al., 1997] formally defines both the syntax and the semantics of its kernel (referred to as "the bare language"), and it gives some straightforward rules which are easily seen to define a translation from the full language to the kernel; on the other hand, the rules defining the evaluation of the bare language constructs are quite intricate, and indeed rather daunting.

[Watt, 1996] analyzes the potential role for formal definitions of syntax and semantics in the language design process. Some pragmatic weaknesses of the 'natural semantics' framework used in *The Definition* of Standard ML are indicated, and it is suggested that adoption of the action semantics framework [Mosses, 1992, Mosses and Watt, 1987, Watt, 1991] would encourage designers to use formal semantics not only for defining the finished languages, but also for documenting interim design choices along the way. Nevertheless, it remained a huge effort to produce an action semantics definition of a larger language such as Java: using action semantics instead of natural semantics may have eliminated some problems, but it is doubtful that the scale of the improvement was sufficient to persuade many of the targeted language designers to use formal semantics.

So, if optimizing the style of language definitions is inadequate to make use of formal semantics attractive to language designers, what else can be changed? Our proposal here is, somewhat paradoxically, *to stop focussing so much on defining complete languages.* Instead, the semantics of each basic programming *construct* should be defined separately, and independently of all the other constructs that might be included together with it in various languages. The semantics of a whole language can then be derived from the semantics of a collection of basic constructs by giving a (context-free) translation to combinations of those constructs. Let us call a semantics presented in this way *constructive*, regardless of which framework is used for defining the semantics of the individual constructs.

The requirement of defining the basic constructs separately and independently is a very strong one, and not met by many frameworks. We shall later see two semantic frameworks which do support constructive semantics: variants of Modular SOS [Mosses, 2004b] and action semantics. Although the Montages variant [Kutter and Pierantonio, 1997] of the ASM framework inspired the development of constructive semantics, it remains unclear whether it too provides full support for independent definitions of basic constructs, since the auxiliary notation used in ASM definitions of different languages varies widely.

The way that the semantics of Standard ML is obtained from the the semantics of its kernel constructs by defining a translation seems to be quite similar in spirit to constructive semantics. However, note the following differences in the case of constructive semantics:

- all the target constructs have to be defined separately and independently of each other;
- the set of target constructs does not have to be minimal; and

– the target constructs are not themselves required to be included in the full language.

These differences give a higher degree of flexibility when defining translations in constructive semantics.

The rest of the paper proceeds as follows: Section 2 discusses the choice of which constructs to regard as basic, giving examples and indicating how the semantics of some concrete constructs of Haskell and Standard ML can be defined by translation to combinations of basic constructs. Section 3 introduces Modular Structural Operational Semantics and illustrates how it can be used to define the semantics of the individual basic constructs; Section 4 does the same for Action Semantics. Section 5 concludes, indicating how to test whether a particular framework might support constructive semantics, and inviting language designers to see whether constructive semantics does indeed encourage them to use formal methods during the development process.

2 Constructs

Constructive semantics involves mapping concrete language constructs to combinations of basic abstract constructs. We shall start by recalling the distinction between abstract and concrete constructs, and discuss how to distinguish between basic and derived constructs. We shall then introduce a language-independent classification of constructs, and provide some illustrative examples of individual basic abstract constructs. Finally, we shall see how familiar concrete constructs from Haskell and Standard ML can be translated to combinations of these basic constructs.

2.1 Abstract and Concrete Constructs

Consider the familiar while-loop. The concrete syntax of while-loops is usually something like “while exp do cmd” or “while(exp)cmd”, where the condition exp is an expression and the body cmd is a command (although in Standard ML, both the body and the while-loop itself are classified as expressions). The concrete syntax also needs to determine an unambiguous parsing of the text of the various constructs; this may involve the introduction of distinctions between different subsets of expressions. The corresponding abstract syntax merely has to distinguish while-loops from all other constructs which also have an expression and a command as components, but a different intended semantics, such as if-then and repeat-until commands.

When giving an abstract syntax for a particular language, it is common practice to use notation which is strongly suggestive of the concrete syntax of that language. With constructive semantics, however, we shall be considering abstract constructs *in vacuo*, so it is appropriate to adopt a language-independent prefix notation, writing for instance "cond-loop exp cmd" or "cond-loop(exp,cmd)" to suggest a particular notion of conditional loop. (Some might prefer to use full words instead of abbreviations.)

2.2 Basic and Derived Constructs

Hundreds of programming languages have been developed, and they include a huge variety of constructs. Some of the constructs are quite common, and have a simple intended interpretation; these are promising candidates for use as basic constructs in constructive semantics, with their own semantics being defined directly. Other constructs are included in only very few languages, or have a complicated interpretation, and it is preferable to define their semantics constructively, by translating them to basic constructs.

On closer examination, the while-loop itself has a few variants, determined by whether:

1. the possible values of the condition are either boolean (with false terminating the loop) or values of another type (e.g., zero terminating the loop, and any other value letting the loop continue);
2. the execution of a break construct in the body causes abrupt termination of the smallest enclosing loop; and
3. the execution of a continue construct in the body causes abrupt termination of current iteration of the smallest enclosing loop, after which the loop proceeds normally.

Should we take all the above variants (and their combinations) as basic constructs?

Consider first the issue of whether the condition of a while-loop is boolean-valued or not. Assuming the existence of an operation that maps zero to false and all other values to true, and of another that maps false back to zero and true to some other integer, we could easily define either of these two variants in terms of the other. Since neither is significantly less basic than the other, it seems best to treat them both as basic abstract constructs.

The situation is quite different with the variants involving abrupt termination, which definitely should *not* be considered as basic. Here, the break and continue constructs can be translated directly to basic constructs which throw special exceptions; the corresponding variants of the while-loop are correspondingly translated to combinations of exception handlers with the basic while-loop. (We assume that exception handlers are the only basic constructs which do not simply propagate the exceptions thrown by their components.)

In a similar way, a while-loop where the body and loop are both expressions can easily be defined by translation, using basic constructs for converting expressions to commands and vice versa: the body expression is translated to a body command (enclosing it in a basic construct that discards the value), and the while-command is translated to an expression by returning the null value.

2.3 Basic Sorts of Constructs

Let us now examine how abstract constructs are to be classified. We have mentioned while-loops and their components as being either expressions or commands — but how did such classifications arise, and what others might be needed? Can we find classifications that are entirely language-independent?

Following common practice in the design of abstract syntax, we shall classify basic abstract programming constructs primarily according to what sort of data they compute. Sorts of data include numbers, booleans, characters, lists, arrays, records, references, objects, function and procedure abstractions, and types. For any sort T of data, let T-Cmp be the sort of constructs which compute data of sort T (on normal termination). We shall regard elements of data as abstract constructs themselves (with trivial computations), thus including each data sort T in T-Cmp. (In the monadic style of denotational semantics, the denotation of a construct of sort T-Cmp would be an element of M(T) for some monad M.)

The following instances of the above sorts provide appropriate classifications of many basic constructs:

Exp = Val-Cmp: Expressions are constructs which normally compute values of various sorts. Let Val be the sort of all expression values. For any subsort V of Val, let V-Exp be the subsort of Exp consisting of V-valued expressions. For instance, when the sort Bool of boolean values is a subsort of Val, Bool-Exp is the sort of boolean-valued expressions.

Cmd = ()-Cmp: Commands are constructs which normally have effects, but do not compute any data. We take Cmd as the sort of commands.

Dec = Env-Cmp: Declarations are constructs which normally compute environments. An environment represents a set of bindings of identifiers (of sort Id) to bindable values (of sort BVal). Let Env be the sort of environments, then Dec is the sort of declarations.

Var = Loc-Cmp: Variables (in imperative languages) are constructs which compute locations for storing values. The value stored at a location can be both updated and inspected. Let Loc be the sort of locations for values, so Var is the sort of variables.

For any data sorts S, T, let S-to-T-Cmp be the sort of constructs that compute *abstractions* which, when applied to data of sort S, normally compute data of sort T. (In the monadic style of denotational semantics, the denotation of a construct of sort S-to-T-Cmp would be an element of M(S→M(T)) for some monad M.)

The following instances of S-to-T-Cmp provide appropriate classifications of constructs concerned with abstraction and application:

Func = Arg-to-Val-Cmp: function expressions.

Proc = Arg-to-()-Cmp: procedure expressions.

Param = Arg-to-Env-Cmp: formal parameters.

Arg-to-Loc-Cmp: variable abstractions.

Another essential feature of a construct is whether it can be used as an *alternative* whose computation may *fail*. We shall assume that the sorts T-Cmp and S-to-T-Cmp classify only infallible constructs. Let T-Alt be the supersort of T-Cmp obtained by allowing constructs whose computation may terminate with failure instead of with an entity of sort T. (In the monadic style of denotational semantics, the denotation of a construct of sort T-Alt would be an element of a so-called error or exception monad.) Similarly for the supersort S-to-T-Alt of S-to-T-Cmp, which classifies constructs where failure might arise either before or after the computation of an abstraction and its application to data.

Some examples of alternatives are the following:

Cmd-Alt = ()-Alt: Constructs of sort Cmd-Alt include guarded commands.

Func-Alt = Arg-to-Val-Alt: Constructs of sort Fun-Alt include components of functions defined by sets of equations, as well as components of the bodies of case expressions.

Patt = Arg-to-Env-Alt: Patterns are similar to (formal) parameters, in that they may compute environments. However, a pattern can also fail to match the argument value, in which case no environment is computed.

The sorts T-Cmp, S-to-T-Cmp, T-Alt, and S-to-T-Alt all reflect *essential language-independent features* of constructs. For instance, the essential feature of an expression is that it normally computes a value; whether the computation refers to bindings or stored values, has side-effects, throws exceptions, or might not terminate at all, depends on just which constructs can occur in expressions.

The repertoire of sorts introduced above appear to be adequate for classifying most of the abstract constructs needed for the constructive semantics of general-purpose programming languages. A few further sorts will be required in connection with constructs used for concurrent and reactive programming.

2.4 Basic Individual Constructs

We have established some basic sorts for classifying constructs. Let us now consider some illustrative examples of basic individual constructs. Later in this section, we shall see how some concrete constructs of Haskell and Standard ML can be translated to simple combinations of these basic constructs; and subsequent sections will show that the formal semantics of such basic constructs is reasonably simple to define.

Meta-Notation

$$s ::= c(s_1, \ldots, s_n)$$

specifies that c names a construct of sort s and has components of sorts $s_1, \ldots, s_n$. (A collection of such specifications corresponds to a definition of a generalized algebraic data structure [Sheard, 2004].) Similarly,

$$s ::= s'$$

specifies that s includes s' as a subsort.

Note that the same name c may be used for constructs of different sorts and/or with different sorts of components, but then any ambiguity that can arise when writing terms has to be semantically irrelevant: if the results of different ways of combining constructs look exactly the same, they must have exactly the same semantics. (A similar treatment of overloading is adopted in CASL, the Common Algebraic Specification Language [Astesiano et al., 2002, Bidoit and Mosses, 2004, Mosses, 2004c].)

Names of constructs are written in lowercase, whereas names of sorts are Capitalized, and (sort) variables are written entirely in uppercase. N.B. The names used here for basic constructs are probably non-optimal, and should not be regarded as stable. The informal descriptions of the intended interpretation of the various constructs given below are not intended to be either complete or unambiguous.

Loops

Cmd ::= cond-loop(Bool-Exp,Cmd)

The above construct corresponds to a while-command with a boolean condition, and we shall regard it as basic. The following construct differs regarding the order of the loop body and the condition, and corresponds to a repeat-until command, which may be regarded as equally basic:

Cmd ::= cond-loop(Cmd,Bool-Exp)

Further basic loop constructs include one with no condition:

Cmd ::= cond-loop(Cmd)

and a construct with separate commands for execution before and after testing the termination condition:

Cmd ::= cond-loop(Cmd,Bool-Exp,Cmd)

Clearly, all the above loop constructs are closely related, and in fact the first three of them could easily be defined in terms of the last one (using also the null command). We prefer nevertheless to regard the first three as basic, so as to avoid having to derive such simple and familiar constructs from a somewhat more complicated and much less familiar construct.

In contrast, we prefer to regard for-loops as derived constructs, despite their familiarity, since they can easily be defined in terms of the much simpler while-loop (using the command sequencing construct introduced below).

Conditionals

T-Cmp ::= cond(Bool-Exp,T-Cmp,T-Cmp)

for all sorts T of data. With T = Val we obtain conditional expressions, and with T empty we obtain conditional commands.

Taking binary conditional choice (here with a boolean condition) as a basic construct is in contrast to both Haskell and Standard ML, where conditional expressions are regarded as 'syntactic sugar' (i.e., abbreviations) for case-expressions. It is also possible to define a conditional command as a symmetric choice between guarded commands [Sampaio, 1997] (provided that evaluating the condition has no side-effects). Here, however, we prefer to treat binary conditionals, case-expressions, guarded commands, and symmetric choice all as basic. The motivation is again to avoid defining simple constructs in terms of ones which may seem less simple. On the other hand, the conditional command with a single branch is easily defined in terms of the simple and familiar binary conditional command, so there is no need to treat it as basic too.

Data

T-Cmp ::= T

It is natural to regard elements of any data sort T as basic constructs themselves, with trivial computations. For instance, the familiar boolean values true and false in the data sort Bool are also basic abstract constructs of sort Bool-Exp. Similarly for T empty, indicating that no data is computed:

()-Cmp ::= ()

Typed Expressions

T-Exp ::= typed(Exp,T-Type)

Assume that for each subsort T of Val, there is a single element of sort T-Type. For instance, for the sort Bool we have a single entity bool of sort Bool-Type. When the component expression computes a value not in the subsort of Val determined by the component type, the computation of the above construct terminates abnormally (without computing any value); otherwise, it computes the same value as the component expression.

Sequencing

Cmd ::= seq(Cmd,Cmd)

Command sequencing is clearly a particularly basic construct. The following constructs for sequencing expressions with commands are however just as basic:

Exp ::= seq(Cmd,Exp)

Exp ::= seq(Exp,Cmd)

The order of computation is from left to right in all these sequencing constructs.

Side-Effects

Cmd ::= effect(Exp)

This basic construct forms a command from an expression, simply by discarding the value; the sequencing constructs above allow expressions to be formed from commands by supplying some value.

Binding

Dec ::= bind-val(Id,BVal-Cmp)

After computing a bindable value, the above construct computes the environment which represents the binding of the identifier to that value. Note that identifiers in Id, like values in Val, are constructs which do not require any computation.

Lazy binding might either be regarded as a basic variant of value-binding, or derived from the above construct using abstractions.

Simultaneous Declarations

Dec ::= simult(Dec,Dec)

The above basic construct computes the union of the environments computed by the component declarations. Their computations are interleaved; when the component declarations have side-effects, the interleaving may affect the resulting environments. The following basic construct is a sequential variant of simultaneous declaration:

Dec ::= simult-seq(Dec,Dec)

It insists on the (left-to-right) sequencing of the computations of its components. Neither of these two constructs for simultaneous declaration can easily be derived from the other, so it is necessary to regard them both as basic.

Accumulating Declarations

Dec ::= accum(Dec,Dec)

In contrast to simultaneous declarations, the above construct includes the second component in the scope of the bindings given by the first component; this precludes the possibility of interleaving. Here, the bindings given by the second component override those given by the first component when there is a clash of identifiers, but other variants are possible (and equally basic).

Local Declarations

T-Cmp ::= local(Dec,T-Cmp)

Here, the scope of the declaration is restricted to the second component, and the bindings that it gives can make holes in the scope of any previous bindings for the same identifiers.

A significant number of further basic abstract constructs have been developed, defined using Action Semantics, and used to give a constructive action semantics of the Standard ML Core Language [Iversen and Mosses, 2005]. However, the above selection of basic constructs should be enough to illustrate the kinds of constructs that are regarded as basic and available for use in constructive semantic definitions of programming languages.

2.5 Translating Concrete Constructs to Basic Abstract Constructs

The characteristic feature of constructive semantics is that concrete language constructs are translated to combinations of basic abstract constructs. When the translation of each concrete construct is properly defined, the formal semantics of the basic constructs determines the formal semantics of the concrete constructs. The translation of each construct is independent, and can be much simpler to understand than a direct definition of its formal semantics.

We shall now illustrate how straightforwardly various constructs from Haskell and Standard ML can be translated to combinations of the basic abstract constructs introduced above. Constructive semantics does not insist on the use of any particular formalism for defining translations; here we shall merely give a schematic indication of the intended translation.

Conditional Expressions

Clearly, the concrete if-then-else expressions of Haskell and Standard ML can be translated to instances of the basic abstract conditional construct. The condition may however need to be explicitly restricted to have only boolean values, using a basic abstract typed expression. Thus the translation of an if-then-else expression has the general form: cond(typed(–,bool),–,–).

Loop Expressions

Standard ML has a while-expression whose body is also an expression, the value of which is simply discarded. The value computed on termination of the loop is the null value. The translation of such an expression has the form seq(cond-loop(typed(–,bool),effect(–)),null). The use of seq and effect show how while-expressions can be understood in terms of the familiar while-commands, and exhibit the essentially imperative nature of Standard ML's while-expressions (which would useless in a pure functional language).

Bindings

A value-binding for a single identifier in Standard ML can be translated directly to the basic abstract construct bind-val(–,–). A value-binding that has a compound pattern instead of an identifier requires a more complicated translation, and it is preferable to introduce a basic construct that generalizes bind-val to patterns, defining its semantics directly. The basic abstract pattern constructs needed for the translation of Standard ML's patterns appear to be much the same as those needed for Haskell, except for the so-called irrefutable patterns in the latter language.

Compound Declarations

Standard ML's 'and' for combining value bindings translates directly to the basic abstract construct simult-seq. Standard ML's declaration sequences (written with an optional semicolon as separator) translate to the basic accum construct; and recursive definitions of functions translate to combinations of value-bindings, abstractions, and a basic abstract construct for making arbitrary declarations recursive. Lists of (top-level) declarations in Haskell will translate to combinations that make their recursion explicit.

The above illustrations should have give an impression of how concrete language constructs can be translated to combinations of basic abstract constructs. A full translation from the Core of Standard ML to a somewhat more comprehensive set of basic constructs than those presented here has been specified in ASF [Iversen and Mosses, 2005] and validated. A corresponding translation for Haskell has not yet been worked out, but preliminary investigations suggest that it will be possible to reuse many of the same basic abstract constructs that were involved in the translation of Standard ML.

3 Modular SOS

Structural Operational Semantics (SOS) [Plotkin, 1981] is a well-known framework that can be used for specifying the semantics of concurrent systems [Aceto et al., 2001, Milner, 1990] and programming languages [Milner et al., 1997]. It has been widely taught [Hennessy, 1990, Nielson and Nielson, 1992, Plotkin, 1981, Slonneger and Kurtz, 1995, Winskel, 1993]. In general, labels on transitions in SOS represent interaction possibilities, such as communication and/or synchronization between concurrent processes. When using SOS to define the semantics of sequential programming languages, however, labels are typically not used at all: all auxiliary information (such as environments and stores) is incorporated in the configurations of the transition system.

Modular SOS (MSOS) [Mosses, 1999a, Mosses, 2002, Mosses, 2004b] goes to the opposite extreme: labels are exploited as much as possible [Mosses, 2004a]. Configurations in MSOS are simply abstract syntax trees (together with computed values), and auxiliary entities such as environments and stores are incorporated in the labels on transitions.

Taking environments and stores out of configurations and putting them in labels gives a clear separation between syntactic entities and those representing semantic information: configurations in MSOS always represent what remains to be computed (as usual in the SOS of concurrent systems), and the label on a transition represents all the "information processing" associated with it: the information available for inspection, any updates to that information, and any new information produced by the transition itself.

The information processing of transitions in a computation is subject to the obvious constraint that the part of it available for inspection remains stable, except when updated by the transitions themselves. This constraint is represented in MSOS by taking the labels to be the morphisms of a category, and requiring labels on adjacent transitions to be composable. The objects of the label category correspond to states of the processed information. Identity morphisms are naturally used to label unobservable (silent) transitions.

In general, we may assume that the label category is a product of several simple categories: a discrete category for environments, a pre-order category for stores, a monoid category for synchronization signals, etc. (Perhaps surprisingly, these are the only kinds of component categories that are needed in MSOS.) Exploiting symbolic indices rather than positional notation, we may regard labels as records, and use familiar notation for record patterns to extract components.

Rules for constructs concerned only with normal flow of control (sequencing, conditionals, loops, etc.) are naturally formulated without concern for components of labels. The pattern `{...}` indicates a completely arbitrary label; a pattern such as `{env=Env,...}` allows reference to a required component of a label without mentioning other components. Unobservable (identity) labels can be simply omitted when there is no need to refer to their components.

We shall now see how easily MSOS supports constructive semantics. Rules similar to those shown below were presented in previous papers [Mosses, 2002, Mosses, 2004b], and rules for many of the constructs of Standard ML Core Language are given in the author's lecture notes on *Fundamental Concepts and Formal Semantics of Programming Languages* (available from the author).

The plain-text format used for MSOS in the illustrative examples given in Tables 1–6 is supported by a Prolog-based tool which allows such rules to be parsed, translated to Prolog, and used to run programs. Sorts, optionally suffixed with digits and/or primes, are used as meta-variables.

```
State ::= T-Cmp

Final ::= T
```

Table 1: `MSOS/Cmp`: *Computational Constructs*

```
Cmd ::= cond-loop(Bool-Exp,Cmd)

see Cmp/Cond, Cmd/Seq

cond-loop(Bool-Exp, Cmd) -->
  cond(Bool-Exp, seq(Cmd,cond-loop(Bool-Exp,Cmd)), ())
```

Table 2: `MSOS/Cmd/Cond-Loop`: *Loops*

```
T-Cmp ::= cond(Bool-Exp,T-Cmp,T-Cmp)

Val ::= Bool

                    Bool-Exp -{...}-> Bool-Exp'
-------------------------------------------------------------------
cond(Bool-Exp,T-Cmp1,T-Cmp2) -{...}-> cond(Bool-Exp',T-Cmp1,T-Cmp2)

cond(true,T-Cmp1,T-Cmp2)  --> T-Cmp1

cond(false,T-Cmp1,T-Cmp2) --> T-Cmp2
```

Table 3: `MSOS/Cmp/Cond`: *Conditionals*

```
Cmd ::= seq(Cmd,Cmd)

          Cmd1 -{...}-> Cmd1'
---------------------------------------
seq(Cmd1,Cmd2) -{...}-> seq(Cmd1',Cmd2)

seq((),Cmd2) --> Cmd2
```

Table 4: `MSOS/Cmd/Seq`: *Sequencing*

```
Dec ::= simult(Dec,Dec)

Label = {env:Env,...}

             Dec1 -{...}-> Dec1'
---------------------------------------------
simult(Dec1,Dec2) -{...}-> simult(Dec1',Dec2)

             Dec2 -{...}-> Dec2'
---------------------------------------------
simult(Dec1,Dec2) -{...}-> simult(Dec1,Dec2')

simult(Env1,Env2) --> Env1+Env2
```

Table 5: `MSOS/Dec/Simult`: *Simultaneous Declarations*

```
T-Cmp ::= local(Dec,T-Cmp)

Label = {env:Env,...}

             Dec -{...}-> Dec'
-------------------------------------------
local(Dec,T-Cmp) -{...}-> local(Dec',T-Cmp)

            T-Cmp -{env=(Env1/Env0),...}-> T-Cmp'
------------------------------------------------------
local(Env1,T-Cmp) -{env=Env0,...}-> local(Env1,T-Cmp')

local(Env1,T) --> T
```

Table 6: `MSOS/Cmp/Local`: *Local Declarations*

4 Action Semantics

Action Semantics is a hybrid of denotational and operational semantics, originally developed by the present author in collaboration with David Watt in the second half of the 1980's [Mosses, 1992, Mosses and Watt, 1987, Watt, 1991]. As in denotational semantics, inductively-defined semantic functions map program phrases to their denotations; but here, the denotations are so-called *actions*, and the notation used for expressing actions, called *action notation*, is defined operationally.

Action notation was originally defined using (a notational variant of) SOS [Mosses, 1992]. However, the definition had poor modularity [Wansbrough and Hamer, 1997]; this dissatisfaction prompted the development of MSOS, and its use to give a new definition of action notation [Mosses, 1999b]. The high degree of modularity of MSOS greatly facilitated the subsequent development of a new version of action notation [Lassen et al., 2000, Mosses, 2000].

Actions are provided to represent control and data flow, scopes of bindings, effects on storage, and interactive processes. This allows a simple and direct representation of many programming concepts, avoiding the need for the kind of indirect encoding that is needed when using λ-notation. Moreover, the design of action notation is such that no reformulation of an action term is needed when sub-actions become richer, as happens in action semantics when the described language is extended with new features.

Originally, the author was quite satisfied with the modularity of action semantic descriptions, which does indeed ensure a high degree of extensibility and modifiability. The idea of providing reusable modules that define the action semantics of individual commonly-occurring constructs didn't emerge until joint work with Kyung-Goo Doh [Doh and Mosses, 2003] on the use of action semantics in connection with composing programming languages.

The illustrations of constructive action semantics given in Tables 7–13 describe the same constructs as the illustrations of MSOS in Sect. 3.

5 Conclusion

We have proposed an approach to language definition where each basic abstract programming construct is defined separately and independently, and the semantics of a full language is obtained by translating its constructs into the basic abstract constructs. The frameworks of MSOS and action semantics can both be used in conjunction with the proposed approach.

How about other frameworks that are regarding as providing a high degree of modularity, such as the monadic style of denotational semantics, or Abstract State Machines? Could they also be used for defining the semantics of basic abstract constructs? If so, they too would support the constructive approach.

```
action : T-Cmp -> Action & using () & giving T
```

Table 7: `AS/Cmp`: *Computations*

```
T-Cmp ::= cond(Bool-Exp,T-Cmp,T-Cmp)

Val ::= Bool

action cond(Bool-Exp,T-Cmp1,T-Cmp2) =
    action Bool-Exp then
    maybe check the bool and-then
    action T-Cmp1
    else action T-Cmp2
```

Table 8: `AS/Cmp/Cond`: *Conditionals*

```
T-Cmp ::= T

action T = give T
```

Table 9: `AS/Cmp/Data`: *Data Constructs*

```
Cmd ::= cond-loop(Bool-Exp,Cmd)

Val ::= Bool

action cond-loop(Bool-Exp, Cmd) =
    unfolding
    ( action Bool-Exp then
      maybe check the bool and-then
      action Cmd and-then unfold
      else skip )
```

Table 10: `AS/Cmd/Cond-Loop`: *Loops*

```
Cmd ::= seq(Cmd,Cmd)

action seq(Cmd1, Cmd2) =
    action Cmd1 and-then action Cmd2
```

Table 11: `AS/Cmd/Seq`: *Sequencing*

```
Dec ::= simult(Dec,Dec)

action simult(Dec1,Dec2) =
    action Dec1 and action Dec2
    then give disj-union
```

Table 12: `AS/Dec/Simult`: *Simultaneous Declarations*

```
T-Cmp ::= local(Dec,T-Cmp)

action local(Dec,T-Cmp) =
    furthermore action Dec
    scope action T-Cmp
```

Table 13: `AS/Cmp/Local`: *Local Declarations*

Proponents of such frameworks are invited to provide independent semantic definitions of some collection of basic abstract constructs. Those who are still sceptical about the whole idea of defining the semantics of constructs independently are encouraged to identify particular constructs which could present difficulties for MSOS or action semantics (other than constructs for dealing with continuations, for which a modular treatment in these frameworks is known to be lacking).

Acknowledgements

The author is most grateful to the organizers of SBLP 2005 for the invitation to give a keynote talk on this work at the symposium, and for encouraging the preparation of this paper. Thanks also to Martín Musicante for useful comments on a previous version.

References

[Aceto et al., 2001] Aceto, L., Fokkink, W., and Verhoef, C. (2001). Structural operational semantics. In Bergstra, J. A., Ponse, A., and Smolka, S. A., editors, *Handbook of Process Algebra*, chapter 3, pages 197–292. Elsevier Science.

[Astesiano et al., 2002] Astesiano, E., Bidoit, M., Krieg-Brückner, B., Mosses, P. D., Sannella, D., and Tarlecki, A. (2002). CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196.

[Bidoit and Mosses, 2004] Bidoit, M. and Mosses, P. D. (2004). CASL *User Manual.* LNCS 2900 (IFIP Series). Springer.

[Doh and Mosses, 2003] Doh, K.-G. and Mosses, P. D. (2003). Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36.

[Harrison et al., 2002] Harrison, W., Sheard, T., and Hook, J. (2002). Fine control of demand in Haskell. In *MPC '02: Proceedings of the 6th International Conference on Mathematics of Program Construction*, LNCS 2386, pages 68–93. Springer.

[Hennessy, 1990] Hennessy, M. (1990). *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics.* Wiley, New York.

[Iversen and Mosses, 2005] Iversen, J. and Mosses, P. D. (2005). Constructive action semantics for Core ML. *IEE Proceeding on Software*, 152(2):79–98.

[Kutter and Pierantonio, 1997] Kutter, P. and Pierantonio, A. (1997). Montages: Specifications of realistic programming languages. *JUCS*, 3(5):416–442.

[Lassen et al., 2000] Lassen, S. B., Mosses, P. D., and Watt, D. A. (2000). An introduction to AN-2, the proposed new version of Action Notation. In *AS 2000, Third Intl. Workshop on Action Semantics, Recife, Brazil, Proceedings*, BRICS NS-00-6, pages 19–36. Dept. of Comput. Sci., Univ. of Aarhus.

[Milner, 1990] Milner, R. (1990). Operational and algebraic semantics of concurrent processes. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press.

[Milner et al., 1997] Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML (Revised).* The MIT Press.

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

[Mosses, 1999a] Mosses, P. D. (1999a). Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, LNCS 1672, pages 70–80. Springer.

[Mosses, 1999b] Mosses, P. D. (1999b). A modular SOS for Action Notation (extended abstract). In Mosses, P. D. and Watt, D. A., editors, *AS'99, Proc. 2nd International Workshop on Action Semantics*, BRICS NS-99-3, pages 131–142, Dept. of Computer Science, Univ. of Aarhus.

[Mosses, 2000] Mosses, P. D. (2000). AN-2: Revised action notation—syntax and semantics. Available at `http://www.brics.dk/~pdm/papers/Mosses-AN-2-Semantics/`.

[Mosses, 2002] Mosses, P. D. (2002). Pragmatics of modular SOS. In Haeberer, A. M., editor, *AMAST'02, Proc. 9th International Conference on Algebraic Methods and Software Technology*, LNCS 2422, pages 21–40. Springer.

[Mosses, 2004a] Mosses, P. D. (2004a). Exploiting labels in structural operational semantics. *Fundamenata Informaticae*, 60:17–31.

[Mosses, 2004b] Mosses, P. D. (2004b). Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228. Special issue on SOS.

[Mosses, 2004c] Mosses, P. D., editor (2004c). CASL *Reference Manual.* LNCS 2960 (IFIP Series). Springer.

[Mosses and Watt, 1987] Mosses, P. D. and Watt, D. A. (1987). The use of action semantics. In *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*, pages 135–166. North-Holland.

[Nielson and Nielson, 1992] Nielson, H. R. and Nielson, F. (1992). *Semantics with Applications: A Formal Introduction.* Wiley, Chichester, UK.

[Peyton Jones, 2003] Peyton Jones, S. (2003). *Haskell 98 Language and Libraries.* Cambridge University Press.

[Plotkin, 1981] Plotkin, G. D. (1981). A structural approach to operational semantics. Technical report, Dept. of Computer Science, Univ. of Aarhus. Reprinted in JLAP, special issue on SOS, 2004.

[Sampaio, 1997] Sampaio, A. (1997). *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing.* World Scientific.

[Sheard, 2004] Sheard, T. (2004). Languages of the future. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press.

[Slonneger and Kurtz, 1995] Slonneger, K. and Kurtz, B. L. (1995). *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley.

[Wansbrough and Hamer, 1997] Wansbrough, K. and Hamer, J. (1997). A modular monadic action semantics. In *Conference on Domain-Specific Languages*, pages 157–170. The USENIX Association.

[Watt, 1991] Watt, D. A. (1991). *Programming Language Syntax and Semantics.* Prentice-Hall.

[Watt, 1996] Watt, D. A. (1996). Why don't programming language designers use formal methods? In *Anais XXIII Seminário Integrado de Software e Hardware*, pages 1–16. Universidade Federál de Pernambuco, Recife.

[Winskel, 1993] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction.* MIT Press.