Component-Based Description of Programming Languages

Peter D. Mosses
Department of Computer Science, Swansea University
Singleton Park, Swansea, SA2 8PP, United Kingdom
p.d.mosses@swan.ac.uk

Abstract

Research in formal description of programming languages over the past four decades has led to some significant achievements. These include formal syntax and semantics for complete major programming languages, and theoretical foundations for novel features that might be included in future languages. Nevertheless, to give a completely formal, validated description of any significant programming language using the conventional frameworks remains an immense effort, disproportionate to its perceived benefits. Our diagnosis of the causes of this disappointing situation highlights two major deficiencies in the pragmatic aspects of formal language descriptions in conventional frameworks: lack of reusable components, and poor tool support.

Part of the proposed remedy is a radical shift to a novel component-based paradigm for the development of complete language descriptions, based on simple interfaces between descriptions of syntactic and semantic aspects, and employing frameworks that allow independent description of individual programming constructs. The introduction of a language-independent notation for common programming constructs maximises the reusability of components. Tool support for component-based language description is being developed using the ASF+SDF Meta-Environment; the aim is to provide an efficient component-based workbench for use in design and implementation of future programming languages, accompanied by an online repository for validated formal descriptions of programming constructs and languages.

Keywords: Programming languages, Formal methods, Syntax, Semantics, Modularity, Reuse

1. INTRODUCTION

Research in formal description of programming languages over the past four decades has led to some significant achievements. These include formal syntax and semantics for major programming languages, including ALGOL60, PASCAL, ADA83, STANDARD ML, and JAVA. Theoretical studies have addressed advanced features that might be included in future mainstream languages; they have also led to a deeper understanding of programming languages in terms of abstract mathematical structures, and to significant theorems about substantial fragments of real languages, such as soundness of type systems.

Nevertheless, to give a completely formal, validated description of any significant programming language using the conventional frameworks remains an immense effort, disproportionate to its perceived benefits. For instance, the HASKELL designers abandoned their original stated aim of providing a formal semantics for their language [9] – they did not even give a formal definition of the relatively small kernel language referred to in the HASKELL98 Report [12].

Our diagnosis of the causes of this disappointing situation highlights two major deficiencies in the pragmatic aspects of formal language descriptions: lack of reusable components (and of an online repository for them), and poor tool support. To give a formal description of a major programming

language is in many respects comparable to developing a significant piece of software, where reusable components (packages) and tool support (IDEs) are clearly of vital importance.

Part of the proposed remedy involves a radical shift to a novel *component-based paradigm* for the development of complete language descriptions [25]. This paradigm is based on simple interfaces between descriptions of syntactic and semantic aspects, and employs frameworks that allow *independent* description of *individual programming constructs*, such as Modular SOS [24, 27] and action semantics [23]. It originated from experiments with the modular structure of action semantics [5, 11]. The use of a *language-independent* notation for common programming constructs maximises the reusability of components.

The provision of reusable components should make formal description of programming languages considerably more attractive to designers and implementers, since different languages (even those from different families) often have many constructs in common. Following Plotkin [29], our intention is to require "very little in the way of mathematical background", and to allow "concise and comprehensible" descriptions of existing programming languages. In particular, we avoid (explicit) use of category theory, as that might make our language descriptions significantly less accessible to many potential users.

We are developing tool support for component-based language descriptions using the ASF+SDF Meta-Environment [2]; our aims are to provide an efficient component-based workbench for use in design and implementation of future programming languages, and to establish an online repository for validated formal descriptions of programming constructs and languages. As previously suggested by Heering and Klint [8], it should also be possible to generate a variety of tools for programming languages from their formal descriptions.

Outline: Section 2 recalls the different aspects of conventional descriptions of programming languages, illustrating them with simple fragments of descriptions in selected frameworks. Section 3 identifies the main hindrances for reuse in those frameworks, and shows how they can be avoided. Section 4 motivates and illustrates the development of language-independent notation for common programming constructs, and Section 5 explains how that notation can be exploited in component-based descriptions of complete languages. Section 6 concludes by summarising our contributions and indicating plans for future work.

2. BACKGROUND

Complete formal descriptions of programming languages are usually divided into major parts, concerned with concrete syntax, abstract syntax, static semantics and dynamic semantics. The syntax of a language determines the set of well-formed programs and their hierarchical structure; static semantics involves checking that programs are well-typed; and dynamic semantics specifies the observable effect of running a program. Different formalisms are used in each part, in general.

2.1. Concrete Syntax

Programs are highly-structured texts. Concrete syntax is about recognising the intended phrase structure in a sequence of characters. The grouping of characters into nested phrases can usually be specified by a context-free grammar. Conceptually, sequences of adjacent characters are first grouped into lexemes such as numerals, identifiers, and reserved words; the lexemes are then grouped into phrases. For some languages, however, lexical analysis is context-dependent.

Concrete syntax is described formally using BNF or similar notation for context-free grammars, often extended with some form of regular expressions to facilitate the specification of optional and repeated sub-phrases. If the grammar is ambiguous, it needs to be accompanied by rules that determine which of the possible groupings is the intended one, e.g. giving relative priorities between productions. The disambiguated phrase structure of a program text is represented by its parse tree w.r.t. the grammar.

2.2. Abstract Syntax

The abstract syntax of a program is a representation of its *essential* (i.e. semantically relevant) structure. It is typically obtained by simplifying the parse tree, e.g. by unifying nonterminal symbols, and specified in a BNF-like notation, as in the following illustrative fragment:

$$\begin{array}{lrcl} (\textit{Expressions}) & E & ::= & E \,\&\&\, E \mid \ldots \\ (\textit{Commands}) & C & ::= & C\ C \mid \{\ \} \mid \text{while}\ E\ C \mid \{\ D\ C\ \} \mid \ldots \\ (\textit{Declarations}) & D & ::= & T\ I\ ; \mid \ldots \end{array}$$

(T represents types and I identifiers.) A complete abstract syntax grammar for a language is interpreted as a set of trees whose nodes are labelled by productions. Since abstract syntax grammars are not used for parsing texts, ambiguity is not an issue: each abstract syntax tree represents a particular grouping structure.

Abstract syntax can also be specified by algebraic data type definitions (as provided in functional programming languages such as HASKELL and ML, and in modern algebraic specification languages such as CASL). The branches of a node are usually an ordered sequence, but can also be mappings or sets (e.g. as in VDM [1]).

2.3. Static Semantics

Static semantics specifies compile-time checks on programs: well-typedness, declaration before use, etc. Such checks correspond to syntactic restrictions that are inherently context-sensitive, and cannot be specified by context-free grammars. Occasionally, the grouping of characters or lexemes into phrases in concrete syntax depends on contextual information, but otherwise static semantics is specified for abstract syntax trees, independently of the details of concrete syntax.

Attribute grammars [15] are a powerful formalism for specifying static semantics. The declared types of identifiers can be treated as inherited attributes, and the types of expressions as synthesised attributes. However, it is more usual to specify typing relations inductively [28], e.g.:

$$\frac{\Gamma \vdash E : \text{bool}, \quad \Gamma \vdash C : \text{cmd}}{\Gamma \vdash \text{while}\ E\ C : \text{cmd}} \tag{1}$$

Sometimes, static semantics involves not only checking but also transforming abstract syntax trees, e.g. by stripping away types (which may be redundant once they have been checked) or resolving calls of overloaded procedures.

2.4. Dynamic Semantics

The dynamic semantics of a program models its observable behaviour when executed. Implementation-dependent details (e.g. execution time, storage allocation) are not regarded as observable, in general: only the relation between input and output is modelled.

Many frameworks have been developed for specifying dynamic semantics. Let us recall the main conventional formalisms for operational and denotational semantics.

2.4.1. Structural Operational Semantics

In *operational* semantics, the observable behaviour of a program is usually derived from a transition system, where the transition relation represents steps in the execution of the program. A state of the transition system includes the part of the program that remains to be executed, often together with auxiliary entities such as environments ρ (representing bindings of identifiers) and stores σ (representing assignments to variables). The *structural* style of operational semantics (SOS) developed by Milner [19], Plotkin [29], et al., uses (axioms and inference) rules to specify transition relations inductively. For example, the following rules specify the evaluation of JAVA's conditional conjunction expression using a transition relation written $\rho \vdash \langle E, \sigma \rangle \to \langle E', \sigma' \rangle$:

$$\frac{\rho \vdash \langle E_1, \sigma\rangle \to \langle E_1', \sigma'\rangle}{\rho \vdash \langle E_1 \,\&\&\, E_2, \sigma\rangle \to \langle E_1' \,\&\&\, E_2, \sigma'\rangle} \quad (2) \qquad \rho \vdash \langle \mathsf{true} \,\&\&\, E_2, \sigma\rangle \to \langle E_2, \sigma\rangle \quad (3)$$

$$\rho \vdash \langle \mathsf{false} \,\&\&\, E_2, \sigma\rangle \to \langle \mathsf{false}, \sigma\rangle \quad (4)$$

Rule (2) specifies transitions for the gradual evaluation of E_1; the other rules above specify what is to happen when E_1 has been evaluated to a truth-value.

Assuming that declarations D compute environments ρ (representing the bindings that they establish) and that normal termination of commands C is represented by the null command $\{\ \}$, the following rules specify execution of blocks with local declarations:[1]

$$\frac{\rho \vdash \langle D, \sigma\rangle \to \langle D', \sigma'\rangle}{\rho \vdash \langle \{\ D\ C\ \}, \sigma\rangle \to \langle \{\ D'\ C\ \}, \sigma'\rangle} \quad (5) \qquad \frac{\rho[\rho_1] \vdash \langle C, \sigma\rangle \to \langle C', \sigma'\rangle}{\rho \vdash \langle \{\ \rho_1\ C\ \}, \sigma\rangle \to \langle \{\ \rho_1\ C'\ \}, \sigma'\rangle} \quad (6)$$

$$\rho \vdash \langle \{\ \rho_1\ \ \{\ \}\ \}, \sigma\rangle \to \langle \{\ \}, \sigma\rangle \quad (7)$$

Rule (6) specifies that the environment used for executing the block body C is a combination of the current environment ρ and the environment ρ_1 computed by the local declarations D.

In the so-called 'big-step' style of SOS, also known as 'natural semantics' [13], transitions relate initial states directly to final states. Using a transition relation written $\rho \vdash \langle E, \sigma\rangle \to \langle V, \sigma'\rangle$, where V is the value of E, the following rules specify the big-step SOS of conditional conjunction:

$$\frac{\rho \vdash \langle E_1, \sigma\rangle \to \langle \mathsf{true}, \sigma'\rangle, \quad \rho \vdash \langle E_2, \sigma'\rangle \to \langle V, \sigma''\rangle}{\rho \vdash \langle E_1 \,\&\&\, E_2, \sigma\rangle \to \langle V, \sigma''\rangle} \quad (8) \qquad \frac{\rho \vdash \langle E_1, \sigma\rangle \to \langle \mathsf{false}, \sigma'\rangle}{\rho \vdash \langle E_1 \,\&\&\, E_2, \sigma\rangle \to \langle \mathsf{false}, \sigma'\rangle} \quad (9)$$

The big-step SOS of blocks requires only a single rule:

$$\frac{\rho \vdash \langle D, \sigma\rangle \to \langle \rho_1, \sigma'\rangle, \quad \rho[\rho_1] \vdash \langle C, \sigma'\rangle \to \langle \{\ \}, \sigma''\rangle}{\rho \vdash \langle \{\ D\ C\ \}, \sigma\rangle \to \langle \{\ \}, \sigma''\rangle} \quad (10)$$

In both styles of SOS, notice how each rule for a construct explicitly propagates auxiliary entities that are required only in connection with the semantics of *other* constructs. Such explicit propagation is the main reason for the poor reusability of rules in SOS [27]. It also clutters up the rules, motivating some descriptions to adopt informal 'conventions' that allow implicit propagation, e.g. [4, 20]. (Section 3.4.1 introduces a recent variant of SOS which provides formal foundations for implicit propagation.)

2.4.2. Denotational Semantics

A *denotational semantics* maps each phrase of the described language to its denotation: a semantic entity that represents the contribution of the phrase to overall program behaviour in any context. The semantic mapping is required to be *compositional*, i.e. the denotation of each compound phrase depends on the denotations of its sub-phrases, but not on their form. Denotations are said to be *fully abstract* when observationally-equivalent phrases always have the same denotations.

In the original Scott-Strachey style of denotational semantics [22, 32], denotations are elements of Scott-domains (which are typically continuous higher-order function spaces) and specified in λ-notation, using a fixed-point operator to define the denotations of loops and recursive procedures compositionally. Letting the denotations $\mathcal{E}[\![E]\!]$ of expressions be functions of environments ρ and continuations κ, the denotations of conditional conjunction expressions can be defined by:

$$\mathcal{E}[\![E_1 \,\&\&\, E_2]\!] = \lambda\rho.\lambda\kappa.\ \mathcal{E}[\![E_1]\!]\rho\{\lambda b.\ b \to \mathcal{E}[\![E_2]\!]\rho\kappa, \kappa(\mathsf{false})\} \quad (11)$$

[1] Substitution could be used instead of environments, but to have to define it for each construct would be quite tedious.

(The notation '$b \rightarrow e_1, e_2$' used above expresses conditional choice between the semantic entities e_1 and e_2.) Similarly, the denotations of blocks can be defined by:

$$\mathcal{C}[\![\{\ D\ C\ \}]\!] = \lambda\rho.\lambda\theta.\ \mathcal{D}[\![D]\!]\rho\{\lambda\rho_1.\ \mathcal{C}[\![C]\!](\rho[\rho_1])\theta\} \tag{12}$$

How the denotations of the sub-phrases are combined has to take account of auxiliary entities which, as in SOS, are needed only in connection with the semantics of *other constructs* that happen to be included in the described language.

2.5. Complete Language Descriptions

A complete description of a programming language needs to combine and connect the separate parts specifying syntax and semantics. For each program, the parse tree determined by the concrete syntax has to be mapped to the corresponding abstract syntax tree; in practice, this mapping is usually regarded as sufficiently obvious, and not specified formally. The static semantics is applied to the abstract syntax tree of the program, and determines whether the program is statically correct (e.g. well-typed); if so, the dynamic semantics is applied to the same tree (or to a transformed tree, when the static semantics provides one).

3. REUSABILITY

The possibility of a high degree of reuse is of crucial importance for reducing the effort of giving complete descriptions of programming languages. Unfortunately, the conventional frameworks recalled in Section 2 provide little or no support for reuse. Let us now examine the impediments to reuse, and see how they can be removed by adopting variants of the conventional frameworks.

3.1. Concrete Syntax

The BNF-like grammars used to describe the concrete syntax of languages are unstructured sets of productions, with no easy way of referring to particular productions from descriptions of other languages. Moreover, disambiguation of grouping is often achieved by introducing numerous auxiliary nonterminal symbols, which might need to be renamed when reusing part of a grammar.

SDF (Syntax Definition Formalism) [7] eliminates both problems: sets of productions can be named, and subsequently combined by referring to their names; disambiguation can be specified using relative priorities, thereby avoiding the need for auxiliary nonterminal symbols.

3.2. Abstract Syntax

Although abstract syntax is usually specified using the same kind of BNF-like grammars as for concrete syntax, ambiguity is not an issue here, so neither auxiliary nonterminal symbols nor disambiguating priorities are required.

Recall, however, that the grammar for abstract syntax is usually obtained by merely unifying nonterminal symbols of a grammar for concrete syntax: the terminal symbols are retained – partly for their mnemonic value, partly to avoid having to specify the mapping from concrete to abstract syntax explicitly. Thus the abstract syntax of JAVA conditional expressions would usually be specified by a production such as $E ::= E\,?\,E : E$. However, it would be quite confusing to reuse this notation verbatim in a description of a language from a different family where conditional expressions are written using reserved words. We discuss a solution to this issue in Section 4.

3.3. Static Semantics

Several variants of attribute grammars support reuse of parts of descriptions of static semantics, e.g. ELI [14] and JASTADD [6]. For those who prefer to specify typing relations inductively by typing rules, the TINKERTYPE framework [17] allows sets of typing rules to be named and tagged with 'features', and subsequently referenced for explicit reuse. A drawback of TINKERTYPE is that different versions of the typing rules for each construct are needed, according to which other

constructs it might be combined with. The variants of SOS discussed below avoid the need for multiple versions, and could be useful for static semantics as well as dynamic semantics.

3.4. Dynamic Semantics

The conventional operational and denotational frameworks share the same general problem regarding reuse: the formulation of the description of each construct depends on which *other* constructs are included in the described language. For example, the description of a conditional expression depends on whether expressions include constructs that have side-effects, or terminate abruptly. Significant reformulation may be needed to 'reuse' the description of a construct from one language in the description of a different language in which it occurs.

3.4.1. Modular SOS

Modular SOS (MSOS) [24] is a simple variant of structural operational semantics. It has been designed to allow a particularly high degree of reuse without any need for reformulation. The description of each construct in MSOS is completely independent; this is achieved by incorporating all auxiliary entities (environments, stores, etc.) in *labels* on transitions. The 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]

Using MSOS, the rules for each individual construct form a *reusable component*. For example, the following rules for conditional conjunction could be used in the description of any language that uses the notation E_1 && E_2 with the same interpretation as in JAVA:

$$\frac{E_1 \xrightarrow{\{\ldots\}} E_1'}{E_1 \;\&\&\; E_2 \xrightarrow{\{\ldots\}} E_1' \;\&\&\; E_2} \qquad (13)$$

$$\text{true} \;\&\&\; E_2 \xrightarrow{\{-\}} E_2 \qquad (14)$$

$$\text{false} \;\&\&\; E_2 \xrightarrow{\{-\}} \text{false} \qquad (15)$$

In (13) above, the notation '$\{\ldots\}$' specifies propagation of all label components;[3] in (14)–(15), '$\{-\}$' specifies that the label is unobservable. By not mentioning specific auxiliary entities, the rules assume neither their presence nor their absence, ensuring reusability – and eliminating the clutter that usually arises in conventional SOS rules for programming constructs (illustrated in Section 2.4.1).

The MSOS rules for blocks '$\{\, D\ C \,\}$' require labels to include an environment ρ. The specification of this requirement forms part of the reusable component:

$$\mathit{Label} = \{\rho : \mathit{Env}, \ldots\} \qquad \frac{D \xrightarrow{\{\ldots\}} D'}{\{\, D\ C \,\} \xrightarrow{\{\ldots\}} \{\, D'\ C \,\}} \qquad (16)$$

$$\frac{C \xrightarrow{\{\rho=\rho_0[\rho_1],\ldots\}} C'}{\{\, \rho_1\ C \,\} \xrightarrow{\{\rho=\rho_0,\ldots\}} \{\, \rho_1\ C' \,\}} \qquad (17)$$

$$\{\, \rho_1\ \{\,\}\,\} \xrightarrow{\{-\}} \{\,\} \qquad (18)$$

3.4.2. Implicitly-Modular SOS

The recently-developed I-MSOS framework [27] combines the benefits of MSOS regarding reusability with the familiar notational style of ordinary SOS: auxiliary entities that are not mentioned in a rule are propagated implicitly between its premise(s) and conclusion, without requiring the introduction of explicit labels on transitions. All that is needed is to declare the notation used for the transition relation being defined (which is in any case normal practice in SOS descriptions of programming languages, e.g. [28]), distinguishing any required auxiliary arguments by highlighting. The above MSOS rules are formulated in I-MSOS as follows:

[2] In fact labels in MSOS are the morphisms of a *category*, and the unobservable labels are identity morphisms.
[3] As in STANDARD ML record patterns, '...' is here a symbol of the formal notation, rather than an informal ellipsis.

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

$$\frac{E_1 \to E_1'}{E_1 \,\&\&\, E_2 \to E_1' \,\&\&\, E_2} \quad (19)$$

$$\text{true} \,\&\&\, E_2 \to E_2 \quad (20)$$

$$\text{false} \,\&\&\, E_2 \to \text{false} \quad (21)$$

$$\boxed{\rho \vdash C \to C'}$$

$$\frac{D \to D'}{\{\, D\ C \,\} \to \{\, D'\ C \,\}} \quad (22)$$

$$\frac{\rho[\rho_1] \vdash C \to C'}{\rho \vdash \{\, \rho_1\ C \,\} \to \{\, \rho_1\ C' \,\}} \quad (23)$$

$$\{\, \rho_1\ \{\,\}\,\} \to \{\,\} \quad (24)$$

3.4.3. Monadic Semantics

Regarding denotational semantics, it is somewhat ironic that the main hindrance to reuse is actually the direct use of λ-notation itself for expressing denotations. The way that denotations are expressed depends on their structure; when that structure changes, the definitions of denotations have to be reformulated.

The monadic variant of denotational semantics was introduced by Moggi at the end of the 1980s [21]. Although reuse was not Moggi's original motivation, the pragmatic benefits of monadic semantics in that respect were soon realised. A *monad* consists of a domain constructor T, mapping value domains D to domains $T(D)$ whose elements represent *computations* of values in D, together with two polymorphic operators, return : $D \to T(D)$; and $\gg\!=$: $T(A) \times (A \to T(B)) \to T(B)$. The trivial computation return(d) simply has the value d as its result. When the computation e computes a value d and f is a function mapping values to computations, $e \gg\!= f$ follows the computation e with the computation $f(d)$. For example:

$$\mathcal{E}[\![E_1 \,\&\&\, E_2]\!] = \mathcal{E}[\![E_1]\!] \gg\!= \lambda b.\ b \to \mathcal{E}[\![E_2]\!], \text{return}(\text{false}) \quad (25)$$

Particular kinds of monads provide further operations. For example, the following monadic semantics for blocks requires a monad that propagates environments, and is equipped with an operation local : $(\mathit{Env} \to \mathit{Env}) \to T(A) \to T(A)$ such that local$(f)(e)$ obtains the environment for the computation e by applying f to the current environment:

$$\mathcal{C}[\![\{\, D\ C \,\}]\!] = \mathcal{D}[\![D]\!] \gg\!= \lambda\rho_1.\ \text{local}(\lambda\rho.\ \rho[\rho_1])(C[\![C]\!]) \quad (26)$$

Such a monad can be constructed by applying a standard *monad transformer* [18, 21] to an existing monad. The more recent work on computational effects and operations by Plotkin and Power [30] obtains monads as models of combinations of Lawvere theories, giving further benefits regarding modularity and reuse.

A crucial difference between the original Scott-Strachey style of denotational semantics and monadic semantics is that in the former, the definitions of operators are fixed, whereas in monadic semantics, the definitions of return(d) and $e \gg\!= f$ (and other operators provided by the monad) change when the domains change. (In fact the idea of redefining operators in this way was exploited since the 1970s in the VDM variant of denotational semantics, although the close relationship between VDM and monadic semantics has only recently been noticed [26].)

3.4.4. Action Semantics

A further variant of denotational semantics that supports reuse is action semantics [23]. There, denotations are so-called actions, expressed in action notation; the semantics of action notation itself is defined operationally (originally using an unorthodox variant of SOS, later in MSOS).

The primitives and combinators provided by action notation include not only the monadic operators, but also operators expressing abrupt termination, nondeterminism, scopes of bindings, effects on storage, message-passing between (asynchronous) processes, etc. Each action combinator implicitly propagates auxiliary entities (environments, stores, etc.) in a particular way.

Action notation satisfies a large collection of algebraic laws, including monoid laws for all the binary combinators.

The action semantics of each language construct remains well-formed and meaningful when further constructs are added to the described language. For example, the following action semantics description of JAVA's conditional conjunction is independent of whether expressions could have side-effects, spawn processes, synchronise with other threads, etc.:

$$\textit{evaluate}\ [\![E_1\ \&\&\ E_2]\!] = \textit{evaluate}\ E_1\ \text{then} \\ (\text{check the boolean then}\ \textit{evaluate}\ E_2\ \text{else give false}) \tag{27}$$

The action semantics description for blocks is as follows:

$$\textit{execute}\ [\![\{\ D\ C\ \}]\!] = \text{furthermore}\ \textit{elaborate}\ D\ \text{hence}\ \textit{execute}\ C \tag{28}$$

The action primitives and combinators provide an adequate basis for expressing the dynamic semantics of a wide range of programming constructs. However, action notation itself is a rather novel and unfamiliar notation, with some unusual features; despite the introduction of a significantly simpler revised version in 2000, and the development of various supporting tools [3], analysis of programming constructs in terms of actions has not so far become established as a mainstream approach to semantics.

The next section motivates and introduces a language-independent notation for common programming constructs. These constructs should be significantly more familiar than action primitives and combinators. Section 5 explains how they can be exploited in component-based descriptions of complete languages.

4. ABSTRACT CONSTRUCTS

A programming language consists of a collection of *concrete constructs*, each of which has a particular concrete syntax and semantics. However, the relationship between concrete syntax and semantics depends on the language. Moreover, constructs with the same concrete syntax in different languages may have different semantics (e.g. 'x=y' is an equality test is some languages, but an assignment command in others), and constructs with different concrete syntax may have equivalent semantics (e.g. 'c ? x : y' and 'if c then x else y' are equivalent forms of conditional expression in different languages).

An *abstract construct*, in contrast, has a *fixed, language-independent abstract syntax and semantics*. For instance, an abstract conditional expression E has a particular constructor notation, such as 'cond(E_1, E_2, E_3)' (avoiding bias towards any particular language family) and all possible questions that one might ask about its semantics have definite answers: E_1 is evaluated first to compute a boolean value; if the value is true, the value of E is the same as that of E_2, and E_3 is skipped; if the value is false, the value of E is the same as that of E_3, and E_2 is skipped; any side-effects of evaluating the subexpressions are propagated, as are abrupt termination and nontermination; the bindings visible to E_1, E_2 and E_3 are the same as those visible to E; when E_1 has boolean type and both E_2 and E_3 have type T, E has type T; and so on.

The relationship between the syntax and semantics of individual abstract constructs is not only language-independent but also 1-1: if two abstract constructs have different semantics, they have different syntax (to avoid confusion); and no two abstract constructs with different syntax have exactly the same semantics (to avoid pointless duplication).[4]

Both the syntax and semantics of an abstract construct should be specified formally. For syntax, all we need is the name of the constructor function used to form instances of the construct, and its arity (the sorts of its components, and the sort of the construct itself). For instance, we may specify the syntax of the above conditional expression using BNF-like notation:

$$\textit{Exp} \quad ::= \quad \text{cond}(\textit{Exp}, \textit{Exp}, \textit{Exp})$$

[4] Different *combinations* of abstract constructs (and special cases of different individual abstract constructs) may however have the same semantics, so abstract constructs do not provide *canonical* representations of semantics, in general.

Given the syntax of a construct, its semantics is to be specified independently of other constructs, using a framework such as (I-)MSOS, monadic semantics or action semantics.

4.1. Sorts

As usual in abstract syntax, constructs are naturally classified by sorts that reflect the kind of entity that they normally compute when executed, e.g.:

- Cmd: commands, computing nothing (or some fixed value);
- Exp: expressions, computing arbitrary entities;
- Dcl: declarations, computing environments that map identifiers in Id to arbitrary entities.

The following sorts all classify abstractions, which have to be supplied with arguments before the encapsulated constructs can be executed:

- Pcd: procedure abstractions, encapsulating commands;
- Fnc: function abstractions, encapsulating expressions;
- Prm: (formal) parameter patterns, encapsulating declarations.

The sorts of entity that can be computed include (primitive and composite) values, (simple and composite) assignable variables, symbolic identifiers (Id), and abstractions.

The abbreviated names for sorts above should be reasonably mnemonic (experience indicates that completely unabbreviated words can be tiresome both to write and read in formal language descriptions) and would in any case be explained in a glossary.

4.2. Basic Abstract Constructs

The following selected examples of abstract constructs are given to illustrate the kinds of constructs that may be considered as basic. In each case, some brief hints are given regarding the details of their intended semantics.

- Cmd ::= seq(Cmd, . . . , Cmd) – normal left-to-right command sequencing;
- Cmd ::= skip – normal termination;
- Cmd ::= cond-loop(Exp, Cmd) – corresponds to a simple while-loop (not handling break);
- Cmd ::= catch(Cmd, Pcd) – if Cmd terminates abruptly, and the parameter of the abstraction Pcd matches the accompanying entity, the abrupt termination is handled;
- Cmd ::= throw(Exp) – if Exp terminates normally, the command terminates abruptly with the computed entity;
- Dcl ::= bind(Id, Exp) – binds Id to the entity computed by Exp;
- Exp ::= lookup(Id) – computes the entity currently bound to Id;
- Pcd ::= abs(Prm, Cmd) – parameterised procedure abstraction (with static scoping);
- Pcd ::= alt(Pcd, Pcd) – alternative procedures;
- Prm ::= bind(Id) – parameter that matches any argument entity, and binds Id to it;
- Prm ::= eq(Exp) – parameter that matches only the entity computed by Exp.

An analysis of some concrete constructs in terms of the above basic abstract constructs is illustrated in the next section. Many further basic abstract constructs have already been identified, but further experimentation and case studies are needed before finalising the details.

Note that the collection of abstract constructs is intended to be *open-ended*: new constructs may be added whenever the previous constructs are found to be insufficiently expressive. This is in marked contrast to the fixed semantic metalanguages used in some previous approaches, e.g. λ-notation in denotational semantics, action notation in action semantics, and the intermediate language intended for use in a new semantics of STANDARD ML [16].

Thanks to the independence of our definitions of abstract constructs, adding new constructs should never require reformulation or extension of the definitions of the existing constructs.

Another difference from previous metalanguages is that each abstract construct usually corresponds directly to (a simplified or idealised version of) some concrete construct found in major programming languages.

5. LANGUAGE DESCRIPTIONS

In this section we outline and illustrate our novel paradigm for the development of (partial or complete) language descriptions based on reusable components that define the semantics of individual abstract constructs.[5]

A component-based description of a programming language starts from concrete syntax. For each concrete construct, we specify the notation used for constructing its parse tree (or corresponding abstract syntax tree). This notation will typically be significantly simpler than an unambiguous grammar for the concrete syntax. For example, suppose the concrete constructs of a language to be described include:

$$\begin{array}{lcl} E & ::= & E \,\&\&\, E \mid \ldots \\ C & ::= & \text{while } E\ C \mid \text{break} \mid \text{continue} \mid \ldots \end{array}$$

Next, for each sort of concrete construct, we specify a function mapping each construct of that sort to its analysis as a combination of language-independent abstract constructs. These functions are in general (directly and mutually) recursive; they are specified by a set of inductive equations of the same kind used in denotational semantics. Let $\mathit{Exp}[\![\_]\!]$ be the analysis function mapping concrete expressions to abstract expressions of sort Exp; the following equation defines the analysis of the conditional conjunction in terms of abstract constructs:

$$\mathit{Exp}[\![E_1 \,\&\&\, E_2]\!] = \text{cond}(\mathit{Exp}[\![E_1]\!], \mathit{Exp}[\![E_2]\!], \text{false}) \tag{29}$$

Assuming that the reader already understands the abstract construct 'cond', the above analysis should be significantly easier to understand than the semantic descriptions given in Sections 2 and 3.

Many commonly-occurring programming constructs (command sequencing, conditional commands and expressions, simple while-loops, blocks with local declarations, etc.) would be mapped directly to corresponding abstract constructs. A more interesting illustration is the analysis of concrete while-loops where executing a break command terminates the iteration abruptly:

$$\begin{array}{rcll} \mathit{Cmd}[\![\text{while } E\ C]\!] & = & \text{catch}(\text{cond-loop}(\mathit{Exp}[\![E]\!], \mathit{Cmd}[\![C]\!]), & (30) \\ & & \qquad \text{abs}(\text{eq}(\text{breaking}), \text{skip})) & \\ \mathit{Cmd}[\![\text{break}]\!] & = & \text{throw}(\text{breaking}) & (31) \end{array}$$

For a concrete construct that also allows a 'continue' command to terminate the current execution of the body abruptly, the analysis would be changed to the following:

$$\begin{array}{rcll} \mathit{Cmd}[\![\text{while } E\ C]\!] & = & \text{catch}(\text{cond-loop}(\mathit{Exp}[\![E]\!], & (32) \\ & & \qquad \text{catch}(\mathit{Cmd}[\![C]\!], & \\ & & \qquad\quad \text{abs}(\text{eq}(\text{continuing}), \text{skip})), & \\ & & \qquad \text{abs}(\text{eq}(\text{breaking}), \text{skip})) & \\ \mathit{Cmd}[\![\text{continue}]\!] & = & \text{throw}(\text{continuing}) & (33) \end{array}$$

After analysing each concrete construct in the same manner as illustrated above, all that is needed to complete the description of the language are the independent semantic descriptions of all the abstract constructs used in the analysis functions. However, using a framework such as (I-)MSOS, action semantics, or monadic semantics to define the semantics of the abstract constructs, their descriptions are reusable components, and many of them could be already available – properly validated – in an envisaged online repository. Assuming that readers of the

[5] We no longer call our approach 'constructive' [11, 25], as that could cause confusion in relation to the semantics of Esterel [31], which are constructive in quite a different sense.

complete language description will be able to access the descriptions of the abstract constructs directly in the repository, these do not need to be copied, and they can be referenced simply by giving their abstract syntax.

Thus compared to the conventional paradigm for development of language descriptions, the main effort with the component-based approach lies not in defining the semantics of concrete language constructs, but rather in specifying their analysis in terms of abstract constructs. New abstract constructs (for describing concrete constructs that cannot easily be analysed in terms of the existing ones) may also need to be developed. Provided that abstract constructs remain simple, without the added features that complicate concrete constructs (such as breaks in while-loops), their independent description should generally be quite straightforward – and in any case needs to be done only once for each abstract construct, thanks to (I-)MSOS and other frameworks that support independent description of individual constructs.

6. CONCLUSION

Our main contribution is the principle that complete language descriptions should be based on reusable components consisting of *independent descriptions of individual abstract programming constructs*. Adoption of this principle will require a significant paradigm shift regarding the development of semantic descriptions, involving the study of the essential features of each individual construct in isolation, instead of considering its relationship with other constructs. The conventional semantic frameworks do not support independent description of individual constructs, but some variants which do are now available. An online repository for such reusable components is being developed.

Another contribution is our proposal to establish *language-independent nomenclature for common abstract programming constructs*, to maximise the reusability of their formal descriptions, and provide a stable yet extensible basis for analysing the more complicated concrete constructs found in practical programming languages. Such a nomenclature would also form a restricted vocabulary that should be appropriate for accurately indexing a comprehensive digital library of complete language descriptions.

Tool support is crucial for development of component-based language descriptions. A prototype interactive environment for action semantics has been implemented, based on (a previous version of) the ASD+SDF Meta-Environment [3], but needs further development. We aim to generate useful prototype compilers and interpreters from complete language descriptions (a promising initial experiment has already been reported [10]) enabling rapid experimentation with language design decisions as well as empirical validation of the language description itself.

Finally, we suggest that reliable generation of useful compilers from validated semantic descriptions of programming languages is highly relevant to the overall aims of UKCRC Grand Challenge 6: Dependable Systems Evolution. Such generated compilers would be highly unlikely to replace hand-written optimising compilers for general-purpose languages, but could be advantageous for implementation of domain-specific languages and safety-critical systems.

REFERENCES

[1] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Computer Science Series. Prentice-Hall Int., 1982.

[2] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In *CC 2001*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

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

[4] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP 2007*, volume 4421 of *LNCS*, pages 331–346.

Springer, 2007.
[5] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Program.*, 47(1):3–36, 2003.
[6] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.
[7] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.
[8] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM SIGPLAN Notices*, 35(3):39–48, 2000.
[9] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 12:1–12:55. ACM, 2007.
[10] J. Iversen. An action compiler targeting Standard ML. *Sci. Comput. Program.*, 68(2):79–94, 2007.
[11] J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *Software, IEE Proceedings*, 152:79–98, 2005. Special issue on Language Definitions and Tool Generation.
[12] S. L. P. Jones. Haskell 98: Introduction. *J. Funct. Program.*, 13(1):0–6, 2003.
[13] G. Kahn. Natural semantics. In *STACS'87*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.
[14] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Inf.*, 31(7):601–627, 1994.
[15] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
[16] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL 2007*, pages 173–184. ACM, 2007.
[17] M. Y. Levin and B. C. Pierce. TinkerType: a language for playing with formal systems. *J. Funct. Program.*, 13(2):295–316, 2003.
[18] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *ESOP'96*, volume 1058 of *LNCS*, pages 219–234. Springer, 1996.
[19] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
[20] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
[21] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1990.
[22] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.
[23] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.
[24] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.
[25] P. D. Mosses. A constructive approach to language definition. *J. UCS*, 11(7), 2005.
[26] P. D. Mosses. VDM semantics of programming languages: Combinators and monads. In *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *LNCS*, pages 483–503. Springer, 2007.
[27] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. In *SOS 2008*, volume to appear of *ENTCS*. Elsevier, 2008.
[28] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[29] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
[30] G. D. Plotkin and A. J. Power. Computational effects and operations: An overview. In *Domains VI*, volume 73 of *ENTCS*, pages 149–163. Elsevier, 2004.
[31] D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.
[32] D. S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Inst. Symposia*, pages 19–46. Polytechnic Institute of Brooklyn, 1971.