ISSE manuscript No.
(will be inserted by the editor)

An Agile Approach to Language Modelling and Development

Adrian Johnstone · Peter D. Mosses · Elizabeth Scott

Received: ??? / Accepted: ???

Abstract We have developed novel techniques for component-based specification of programming languages. In our approach, the semantics of each fundamental programming construct is specified independently, using an inherently modular framework such that no reformulation is needed when constructs are combined. A language specification consists of an unrestricted context-free grammar for the syntax of programs, together with an analysis of each language construct in terms of fundamental constructs. An open-ended collection of fundamental constructs is currently being developed.

When supported by appropriate tools, our techniques allow a more agile approach to the design, modelling, and implementation of programming and domain-specific languages. In particular, our approach encourages language designers to proceed incrementally, using prototype implementations generated from specifications to test tentative designs. The components of our specifications are independent and highly reusable, so initial language specifications can be rapidly produced, and can easily evolve in response to changing design decisions.

In this paper, we outline our approach, and relate it to the practices and principles of agile modelling.

Keywords Programming language models · Syntax · Semantics · Agile methods

A. Johnstone · E. Scott
Dept. of Computer Science, Royal Holloway,
University of London, Egham, TW20 0EX, United Kingdom

P. D. Mosses
Dept. of Computer Science, Swansea University,
Singleton Park, Swansea, SA2 8PP, United Kingdom
Tel.: +44 1792 602249, Fax: +44 1792 295708
E-mail: p.d.mosses@swan.ac.uk

1 Introduction

We are developing a component-based approach to language modelling. Our models are formal specifications of syntax and semantics.

Language designers already use formal specifications for syntax; we claim that formal semantics can be useful and worthwhile for them too, provided that the following features are combined in a single, coherent framework:

- a substantial collection of specification components that can be used without change;
- accessible formalisms and notation;
- tool support for developing and checking specifications;
- tools for generating (at least prototype) implementations from tentative language specifications;
- successful case studies involving major languages; and
- an online repository providing access to all specifications.

Our aim is to provide such a framework.

Our component-based approach supports agile modelling, so language designers may start from a rough model of their initial design – possibly making extensive reuse of components from previously-developed models – experiment with a generated prototype implementation, then adjust and extend the model easily as their design evolves.

In contrast, conventional frameworks for semantic specification are inherently non-agile, because adding a new feature to the specified language may require reformulation of the entire specification. For example, suppose we have specified a conventional structural operational semantics [35] for a purely functional language; extending the language with assignable variables or concurrent processes requires adding further arguments to the specified transition relation, and those arguments have to be added in the original specification. Conventional denotational semantics

suffers from the same problem. Agile language modelling is not possible when the specification of each construct depends on which other constructs it is to be combined with.

With few exceptions, language designers have not themselves produced formal specifications of their languages' intended semantics. For instance, the Haskell designers were familiar with the conventional modelling frameworks, and "resorted to denotational semantics to discuss design options", but they also reported [15]:

> "One of our explicit goals was to produce a language that had a formally defined type system and semantics. [...] We were inspired by our brothers and sisters in the ML community, who had shown that it was possible to give a complete formal definition of a language, and the *Definition of Standard ML* [24] had a place of honour on our shelves.
>
> Nevertheless, we never achieved this goal. [...] we always found it a little hard to admit that a language as principled as Haskell aspires to be has no formal definition. But that is the fact of the matter, and it is not without its advantages. In particular, the absence of a formal language definition does allow the language to evolve more easily, because the costs of producing fully formal specifications of any proposed change are heavy, and by themselves discourage changes."

We claim that using our component-based approach, the cost of specifying – and experimenting with – proposed changes would be much lighter than with conventional frameworks.

In the rest of this paper, we outline our approach and analyse its agility. Section 2 is concerned with the specification of language syntax. It points out some drawbacks of the conventional approach, and proposes the use of general context-free grammars, for which efficient parsing algorithms have recently been developed. Section 3 introduces the concept of language-independent fundamental constructs, and illustrates how language syntax can be translated to such constructs. Section 4 recalls two frameworks which allow the semantics of individual constructs to be specified independently; using these frameworks, the semantic specifications of fundamental constructs become highly reusable components of complete language specifications. Section 5 briefly outlines a proposed online repository for component-based specifications of programming and domain-specific languages, and Section 6 describes the kinds of tool support that are needed for practical use of our approach. Section 7 considers the extent to which our approach adheres to the practices and principles of agile modelling. Section 8 reviews related approaches. Section 9 concludes with a summary of our claims.

2 Syntax

Language syntax is often specified in several parts, separating the lexical components or words from the phrases or sentences.

Context-free grammars have been the preferred means to specify the phrase-level syntax since the time of the Algol 60 report. Reference manuals and standards documents use grammars which are comfortable for the explication of the semantics, but typically these are unsuitable for direct implementation using near-deterministic parser generators such as Bison and JavaCC. Most language standards ignore this problem, although the original Java Language Specification [11] provides two grammars – one for explication and one intended for use with parser generators – and its authors went to some trouble to establish their equivalence.

Neither the separation between lexical and phrase-level syntax, nor having two separate grammars for phrase-level syntax, is consistent with an agile approach to language design. They make it difficult to move rapidly to prototyping and to adopt incremental design because several inter-related data structures have to be maintained, and changing one requires corresponding changes to the others. The quality of the language design may be compromised because the design of the lexical components cannot exploit the phrase-level context. Moreover, one must ensure that the grammar used to define the syntax and the grammar used to specify the semantics generate the same language. Maintaining this language equivalence is not trivial: by the time of the third edition of the Java Language Specification [12] significant errors had crept into the implementation grammar. For implementation it would be far better to use the explication grammar, which is the base grammar used by the language's designers, than a derived grammar of doubtful accuracy.

The ASF+SDF Meta-Environment [3] and Stratego/XT [42] are environments that provide some of the features discussed in this section, and indeed the ASF+SDF Meta-Environment was used to implement the Action Environment [5]. The SDF parser generator which underpins both tools uses Farshi-style GLR parsing with SLR(1) tables, and the user is provided with tools that allow some parse-time disambiguation [4].

In practice, SDF tables are monolithic and must be regenerated when specifications change. This can be very time consuming. In addition, the parse-time features are not formally well-founded, and parsers that employ them can fail to correctly match the context free language suggested by the context free rules in their specifications: the language recognised by the parser can change as disambiguation specifications are added, and in fact the recognised language may become context sensitive. This makes it very hard to prove completeness and correctness of the resulting parsers.

Our approach is based on the use of GLL parsers [38] which are fully general, efficient at parse time and, most importantly from a software developer's perspective, require no more processing time to generate than conventional recusive descent parsers. As a result, regenerating the source code for a parser typically takes less time than compiling that code. The disambiguation techniques used in our parsers are also well-founded in the sense that they are guaranteed not to change the language recognised by the parser.

Efficient parsing algorithms such as RNGLR [37, 39] and GLL [38] provide full generality without significant performance penalties. We can use unrestricted context-free grammars to specify the lexical components as well, and we can then combine grammar fragments in an unrestricted way. This supports reuse, and significantly eases the development of language hybrids that 'borrow' features from different existing designs. This is important for agile design, removing the need to maintain two specifications in parallel, and easing the unification of the sometimes subtle variations in lexical syntax adopted by different languages (e.g. some allow '`-`' in identifiers, whereas others treat it as an operator).

General context-free grammars may be ambiguous, and indeed even grammars intended for older near-deterministic tools often include some ambiguities (such as the if-then-else construct), which are typically resolved using longest match. We provide more general disambiguation facilities that allow the undesired derivation trees to be eliminated.

3 Fundamental Constructs

We translate each concrete language phrase to a combination of *language-independent fundamental constructs (funcos)*.[1] The semantics of concrete language constructs are determined by their translation to funcos, which is specified inductively.

Usually, a funco corresponds to a simplified version of some familiar language construct. For instance, there is a funco corresponding to a plain while-do loop: the condition has to be a boolean-valued expression, and abrupt termination of the body always causes abrupt termination of the loop – `break` and `continue` are *not* handled by this funco.

We use abbreviated words to name funcos, such as 'cond-loop'. Symbols and punctuation marks are avoided, as they would inevitably lead to confusion (e.g. '$x=y$' would suggest an equality test to some, but an assignment to others). We are experimenting with different styles of abbreviations, such as 'while-do' instead of 'cond-loop', aiming to maximise their mnemonic value without bias towards any particular language family.

Funcos are classified according to the kind of value that they normally compute when executed: Cmd classifies commands, which compute nothing (or some fixed null value); Dcl classifies declarations, which compute environments (mapping identifiers to constants, variables, procedures, ...); Exp classifies expressions, which compute arbitrary entities; etc. For instance, the classification of the funco named 'cond-loop' is specified as follows:

$$\mathsf{Cmd} ::= \mathsf{cond\text{-}loop}(\mathsf{Exp}, \mathsf{Cmd}) \tag{1}$$

showing also the classification of its arguments.

Our collection of funcos is *open-ended:* new funcos can be added when needed. However, we generally avoid introducing funcos which correspond to simple combinations of existing funcos. For instance, we would not introduce a funco that differs from 'cond-loop' only by having a condition that tests whether a number is non-zero, since that can be specified by a combination of 'cond-loop' with a funco for applying a boolean-valued operation to a numerical operand Exp:

$$\mathsf{cond\text{-}loop}(\mathsf{app}(\mathsf{is\text{-}non\text{-}zero}, Exp), Cmd).$$

Note that the values, operations and predicates of abstract data types can be used as arguments of funcos. Basic mathematical data types include booleans, integers, lists, sets and maps; further data types model floating point numbers, characters, arrays, structures, methods, objects, classes, etc. How to specify such data types is irrelevant here; we merely assume that the specifications are highly reusable components.

The translation of a concrete language phrase may involve more than one funco to combine the translations of its sub-phrases. For example, suppose $Cmd[\![C]\!]$ is the translation of commands C of some language to funco combinations of sort Cmd, and $Exp[\![E]\!]$ is the translation of expressions E to funco combinations of sort Exp. We specify the translations of while-commands and breaks to funcos thus:

$$Cmd[\![\texttt{while (}E\texttt{) }C]\!] = \tag{2}$$
$$\mathsf{catch}(\mathsf{cond\text{-}loop}(Exp[\![E]\!], Cmd[\![C]\!]), \mathsf{abs}(\mathsf{eq}(\mathsf{breaking}), \mathsf{skip}))$$
$$Cmd[\![\texttt{break}]\!] = \mathsf{throw}(\mathsf{breaking}) \tag{3}$$

See [30, 31] for a brief explanation of the funcos catch, throw, abs, eq and skip; the details are not relevant here.

The translations of different constructs are not always completely independent, as illustrated by the need to use the same notation 'breaking' in both the above equations. Moreover, if we subsequently changed the language design so that while-commands also handle `continue`, we would need to change (2) accordingly. Mostly, however, translations from language syntax to funcos can be specified incrementally.

[1] Fundamental constructs were called 'basic abstract syntax' or 'basic abstract constructs' in previous papers.

Initial collections of funcos were tentatively proposed in the original paper on component-based language development and specification [9] and in a case study that provided a specification of the core of Standard ML [17]. A larger collection of about 100 funcos has subsequently been developed in connection with a course on concepts of programming languages at Swansea, and used in the analysis of the illustrative examples of program fragments from Ada, C, C++, Haskell and Java given in [44, Part II].

The development of a collection of funcos is itself an agile process. When a new funco is to be added, all that is needed initially is to specify its name and classification, as illustrated in (1) above; formal specification of its behaviour can be deferred until it is needed as a component when generating an implementation of some language.

A new funco may sometimes require the introduction of a new classifier, but that should not affect the classification of existing funcos. However, before making a substantial collection of funcos available for general use (see Sect. 5), we need to test the generality of the proposed classifications in some further case studies, and fix the names of our existing funcos.

4 Semantics

An essential feature of our approach is the use of frameworks that allow the semantics of different constructs to be specified *independently*. The semantic specification of each funco is a highly reusable component, which does not need to be changed at all when funcos are combined. We cannot use the conventional denotational and operational semantics frameworks, because when using them, the specification of each construct depends on which other (kinds of) constructs it is combined with [30, 31].

4.1 Implicitly-Modular SOS

The recently-developed I-MSOS framework [32] combines the benefits of MSOS [29] regarding reusability with the familiar notational style of ordinary SOS [35]: unmentioned auxiliary entities (such as environments and stores) are propagated implicitly. By not mentioning inessential entities the rules assume neither their presence nor their absence, ensuring reusability – simultaneously eliminating the notational clutter that usually arises in conventional SOS rules for programming constructs unless informal conventions are adopted.

Table 1 specifies the dynamic semantics of a funco for blocks where the scope of the declarations is limited to a command. The specification starts by giving the sort and arity of the funco named ‘block’. Following SOS, I-MSOS models steps of computations by transition relations;

Table 1 Dynamic semantics of a funco in I-MSOS

$Cmd ::= \mathsf{block}(Dcl, Cmd)$ $\quad \boxed{Dcl \to Dcl'}$ $\quad \boxed{Env \vdash Cmd \to Cmd'}$

$$\frac{Dcl \to Dcl'}{\mathsf{block}(Dcl, Cmd) \to \mathsf{block}(Dcl', Cmd)} \tag{4}$$

$$\frac{(Env_1 / Env_0) \vdash Cmd \to Cmd'}{Env_0 \vdash \mathsf{block}(Env_1, Cmd) \to \mathsf{block}(Env_1, Cmd')} \tag{5}$$

$$\mathsf{block}(Env_1, \mathsf{skip}) \to \mathsf{skip} \tag{6}$$

the difference is that here, each auxiliary argument of the relations is formally treated as optional, and can be omitted in inference rules for transitions when it is simply propagated [32]. Above, the grey font used for Env specifies that it is an auxiliary argument of the transition relation $Cmd \to Cmd'$.

Rule (4) specifies that execution of a block can start with the gradual computation of an environment by the declarations Dcl; rule (5) specifies that the resulting Env_1 is combined with the current Env_0 to provide the environment for each step of executing the block body Cmd. Finally, rule (6) specifies that when the body of a block terminates normally (represented by the funco skip), so does the block.

The above specification is a component which could be reused *verbatim* in the semantics of any language whose translation to funcos involves blocks, regardless of whether commands might raise exceptions, or involve interaction with other threads or processes. Notice that the specification did not mention stores: if combined with a funco for assignment, the stores required by the latter would be implicitly propagated.

4.2 Action Semantics

Action semantics is a hybrid of denotational and operational semantics. It supports reuse [26, 27, 33, 43, 45] and compiler generation [6, 16, 34]. The denotations of computational constructs are so-called actions, expressed in action notation; the semantics of action notation itself was defined operationally.

The primitives and combinators provided by action notation include operators expressing sequencing, abrupt termination, nondeterminism, scopes of bindings, effects on storage, message-passing between (asynchronous) processes, etc. Each action combinator implicitly propagates all auxiliary entities (environments, stores, etc.) in a particular way. Action notation satisfies a large collection of algebraic laws.

The action semantics of each language construct remains well-formed and meaningful when further constructs are added to the described language. For example, the action

Table 2 Dynamic semantics of a funco in action semantics

Cmd ::= block(Dcl,Cmd) | *elaborate* Dcl : *Action* giving *Env*
| *execute* Cmd : *Action* giving *Nothing*

execute block(Dcl,Cmd) = (furthermore *elaborate* Dcl) hence *execute* Cmd (7)

semantics of the block funco specified in Table 2 is independent of whether commands could assign to variables, raise exceptions, or involve interaction with other threads or processes. The action 'furthermore A' combines the environment given by performing A with the current environment; 'A_1 hence A_2' uses the environment given by A_1 as the current environment for A_2. The fixed collection of primitives and combinators provided by action semantics give an adequate basis for expressing the dynamic semantics of a wide range of programming constructs.

4.3 Static and Dynamic Semantics

The full specification of a funco gives its static as well as its dynamic semantics. Static semantics generally involves type-checking; it may also involve replacement of some funcos, e.g. due to compile-time evaluation of expressions or static resolution of overloading. The static semantics of a funco determines the relationship between its type and the types of its components, and produces the (possibly changed) funcos that are to be executed according to its dynamic semantics. We aim to prove that the static semantics is sound with respect to the dynamic semantics: the type(s) of each funco should be consistent with the set of values that it can compute.

We specify the static semantics of funcos using I-MSOS, and their dynamic semantics using both I-MSOS and action semantics (the latter is needed to support generation of compiler back-ends). Since we are giving two different specifications for the same semantics, we should really prove that they are indeed equivalent; but for greater agility when developing new funcos, such proofs can be deferred.

5 Online Repository

We are building an online repository of components for use in giving complete specifications of programming and domain-specific languages. The main components correspond to individual funcos, e.g., 'cond-loop', 'block'. There is also a component for each funco classifier: Cmd, Dcl, Exp, etc. (corresponding to an abstract class, not any particular collection of funcos) and for each type of data. Dependency between the components is generally very minor: for instance, the 'block' component depends of the Cmd and Dcl components, and on the component that provides environments *Env* (as an instance of a generic component for maps), but it does not depend on any other funco.

Our semantic specifications of funcos are components that can be used simply by referring to their names, since no changes to them are ever needed. For example, when the funco 'block' has been provided in the repository, any language specification may refer to it, as in the following translation of C-style block statements to funcos:

$$Cmd[\![\{D\ S\}]\!] = \mathsf{block}(Dcl[\![D]\!], Cmd[\![S]\!]) \tag{8}$$

The above translation *completely* specifies the semantics of that language construct; it might itself be reused in the translations of other C-like languages to funcos, although we do not consider it as reusable as our funco components. Algebraic laws for funcos are preserved by combination, and will also be included in the online repository.

Parts of our context-free grammars for concrete programming languages may also be reused. Here, however, specifications can be copied and pasted, since the exact combination of the specified language constructs may not be exactly what is wanted; similarly for the translations of language constructs to funcos.

We shall provide efficient (and free) online access via the web to a digital repository of all our reusable components and language specifications. Our main objective is a browser-based digital library interface that supports searching for, reading and uploading specifications and associated proofs. The repository will export the meta-data of specifications, for use by search engines.

6 Tools

We are planning to implement tools to support agile development of language and funco specifications, comparable to the IDEs commonly used by professional programmers; the Action Environment [5] was an early prototype. These tools will allow browsing of previously-developed specifications, and ensure that new specifications are always well-formed.

We shall also implement new tools for rapid prototyping, generating implementations of languages directly from their component-based specifications. From general context-free grammars for concrete languages we can already generate efficient parsers [19]; from equational specifications of translations from concrete constructs to funcos it is straightforward to generate translators; and from the semantic specifications of funcos we shall generate both interpreters and compiler back-ends, building on previous efforts [6, 7, 16, 28, 34].

7 Agility

We maintain that our component-based approach to specifying languages is much more in keeping with the practices and principles of agile modelling than previous approaches. In this section we consider the features of our approach in the light of these practices and principles (following the description given by Scott W. Ambler [1, version 2]).

7.1 Core Practices

Active Stakeholder Participation Our specifications of language syntax, its translation to funcos, and informal specifications of funcos should all be easily accessible to all stakeholders, enabling them to participate in language design discussions. Rapid prototyping allows stakeholders to experiment with programming in an evolving language from an early stage.

Model With Others Our approach supports discussion of language design and specification issues with others by virtue of the incremental nature of language and funco specification, together with the simplicity and independence of the semantics of each funco. Individuals developing different parts of a language specification should be able to discuss particular issues arising in one part without considering the specifications of the other parts.

Apply The Right Artefact(s) Here, our artefacts are formal specifications. The only case where there appears to be some choice over which specification style to use is when introducing a new funco: the dynamic semantics can be specified in I-MSOS or in action semantics. Ultimately, however, both are needed (as explained at the end of Sect. 4).

Iterate To Another Artefact If a developer gets stuck with specifying the action semantics of a particular funco, it might help to start specifying its I-MSOS, or vice versa. An alternative strategy is to switch to specifying a different (preferably related) funco. Similarly, when a syntactic construct appears to be difficult to translate to the available funcos, the developer may tentatively introduce some new funcos before reverting to specifying the translation.

Prove It With Code Proving (i.e., testing) that programs have the intended behaviour according to a language specification is fundamental in our approach, and we shall provide appropriate tool support. The code which comprises our prototype implementations is generated directly from the specifications, and the designer can write simple test harnesses to get immediate feedback.

Use The Simplest Tools We have chosen formal specification frameworks that are especially simple and practical: our general context-free grammars do not require awareness of technical conditions such as LALR(1); translation to funcos is specified inductively by simple equations; I-MSOS allows simpler-looking rules than conventional SOS; and the action notation used in action semantics has a suggestive notation and simple operation semantics.

Model In Small Increments Our approach supports single-construct increments, both for languages and funcos. Of course, components can constructed in groups as well, and this will probably be the most likely situation in practice. These groups can themselves be constructed iteratively, adding things as new requirements are perceived and removing things that prove to be redundant.

Single Source Information The complete specification of each funco will be stored at one place in the repository. The current specification of an evolving language will also be stored at one place. For common language constructs, however, their syntax and its translation to funcos may be duplicated in many different languages. Such duplication actually encourages agility in language design, as it facilitates editing the details of common constructs.

Collective Ownership Collective ownership is fundamental in our approach, there is no discipline of access rights for creating and editing specifications, except that approved funcos in the online repository must have fixed operational semantics and algebraic properties: once published, any changes to their specifications must preserve their semantics.

Create Several Models in Parallel The specification of the various funcos are independent, so they can be created in parallel. Moreover the static semantics and the two forms of dynamic semantics of each funco can all be created in parallel. The grammar of each language construct can be specified and translated to funcos in parallel, although each funco needs to be introduced before it is used in the translation. Furthermore, it is possible to develop more than one design, trying out different semantic possibilities in parallel, before finalising the language specification.

Create Simple Content Each funco is intended to be as simple as possible. The translation of language syntax to funcos should also be rather simple; if that is not possible, new funcos may be needed. The independent nature of funcos allows the designer to choose the simplest first, and then substitute a richer version if it proves necessary.

Depict Models Simply Although we do not employ diagrammatic notation for specifying programming languages, a translation of language constructs to funcos could be regarded as a "*simple model that shows the key features that you are trying to understand*" [1, Practices of AM], since many funcos correspond to familiar programming concepts and may be understood without studying their detailed semantic specifications.

Display Models Publicly The specifications of approved funcos will be available for public online browsing. Language developers may choose whether to make their specifications publicly available or not. Meta-data about each published funco and language will be exported for use by search engines.

7.2 Core Principles

Model with a Purpose The purposes of giving a formal specification of a language are to document (tentative) language design decisions, to support construction of implementations, and to provide a solid basis for sound reasoning about properties of the language and its programs. The intended readership of the specification includes language designers, implementers, programmers, and verifiers of program correctness. The work described in this paper is precisely aimed at achieving this purpose by making formal semantics cost-effective.

Maximise Stakeholder ROI Our core focus on making components reusable ensures that effort is not wasted re-specifying constructs that already appear in existing languages.

Travel Light Only the syntax of the language and its translation to funcos need to be *maintained* over time. The semantic specification of each funco is fixed, and does not require any maintenance.

Multiple Models Our character-level general grammars allow designers to choose whether to model the lexical and phrase-level aspects of a language separately or in a single structure. For dynamic semantics of funcos, the designer can give either the MSOS or action semantics model; when both are given, proving that they are equivalent can be deferred: the models fulfil their immediate purpose, and they will in any case both be validated indirectly when they are used for generating prototype implementations.

Rapid Feedback The IDE should provide immediate parsing and well-formedness checking of new or updated specifications, immediate generation of prototype implementations from specifications, immediate regeneration after changes, and support for re-running test programs.

Assume Simplicity Each funco generally corresponds to a *simplified* version of a concrete construct. For instance, the 'cond-loop' funco introduced in Sect. 3 has a boolean-valued condition, and does not involve any handling of abrupt termination due to 'break' or 'continue'. The translation of each language construct to funcos can initially be over-simplified, and subsequently enhanced (independently of the specifications of other constructs). The repository allows a designer to begin by using existing components and only requires construction of a new one when it becomes clear that the existing ones are inadequate.

Embrace Change When a language design changes, the syntax and funco translation of the affected constructs can be changed accordingly, and new funcos added if needed. The specifications of the existing funcos never require any changes. Design change is not inhibited by high cost because funcos are independent. Changing one funco for another, or adding a new one, does not effect the other funcos that have been used.

When a new version L' of an evolving language is not backwards-compatible with the previous version L, it would be useful to be able to translate all programs in L to equivalent programs in L'. Separate translation of each language construct might be significantly simpler than API migration (e.g. [2]) and equivalence could be established at the funco level, using algebraic laws.

Incremental Change Our component-based approach allows languages to be designed incrementally, starting with a syntax which is already familiar from previous languages, and whose translation to funcos has already been specified. The syntax can then be gradually extended with further constructs, specifying their translation to funcos incrementally, and tentatively adding new funcos when needed.

Quality Work All the formal specifications (language grammars, translation to funcos, and funco semantics) are to be parsed and checked for well-formedness. The quality of their content is assured by validation, based on a sufficiently diverse suite of test programs written in the specified languages, possibly augmented by proofs of expected properties.

Working Software Is Your Primary Goal Here, the working software consists of prototype language implementations generated directly from formal specifications, allowing the developers of an evolving language to experiment with writing and running programs after each increment. Our repository and tool support are the features which make this prototyping available very early in the design process.

Enabling The Next Effort Is Your Secondary Goal New funcos introduced during the course of a language development can be submitted for approval and subsequent inclusion in the online repository. This makes them available for reuse in the development of other languages, as well as in later stages of the evolution of a particular language.

7.3 Supplementary Principles

Content Is More Important Than Representation Formal specifications of languages and funcos may be entered in draft form before being parsed and checked for well-formedness. Informal specifications of new funcos may be provided independently of entering their formal specifications. Thus a designer can demonstrate parts of the specification without needing to build a complete prototype for the whole system.

Open and Honest Communication We intend to make all approved funcos available on the web, and provide forums for open discussion of their details. As changes are achieved by introducing new funcos and will not effect existing funcos, a designer is free to make new proposals without compromising the needs of others.

8 Related Work

Regarding syntax, the closest related approach is SDF2 [41], implemented in the Meta-Environment [3] and Stratego/XT [42]. We discussed the relationship to SDF in Sect. 2.

Some approaches based on attribute grammars (e.g. Eli [20] and JastAdd [10]) provide reusable components for syntax and static semantics, but do not support specification of dynamic semantics.

Monadic denotational semantics [22, 23, 25] provides various monad transformers as reusable components. When a denotational semantics specifies the denotations of language constructs using the operations of a monad constructed by monad transformers, adding new kinds of constructs does not entail reformulation of the specifications of the previous constructs: a further monad transformer can be used to produce a monad with the required new features. However, each monad transformer needs to redefine all the operations provided by other monad transformers, which can be problematic [18, 36].

The tool-oriented approach to semantics of [14] differs from ours by regarding language definitions primarily as "tool generator input": our specifications are theoretically well-founded. In many other respects, however, our component-based approach can be seen as a fulfilment of the goals outlined in [14], especially regarding modularity and libraries of language constructs. The ASF+SDF framework [8] supports modular specifications of syntax, type checkers, compilers and interpreters, but not *independent* specifications of individual constructs (although it has been used to implement support for independent specifications in action semantics [5, 17]).

The ASM framework [13] has good modularity, and it has been used to specify several major programming languages. A language specification can be presented as a series of extensions, although the macros used sometimes need to be redefined [40]. Our approach was partly inspired by the separate specification of individual language constructs in the Montages variant [21]. However, the grammars specified in Montages are restricted to LALR(1), and the degree of reusability of its specification components is unclear.

9 Conclusion

We have outlined a component-based approach to specification of syntax and semantics. The aim is to support *agile formal modelling* during the design and implementation of programming and domain-specific languages, and thereby combine soundness with rapid development. The achievement of all aspects of the coherent framework outlined in the introduction will require significant further effort, especially regarding major case studies, tool support, and the development of the online repository. However, we are building upon a substantial body of previous work, and this provides confidence in the likely success of our approach.

Acknowledgements The authors gratefully acknowledge the useful comments and suggestions of the anonymous referees.

References

1. Ambler SW (accessed Sep. 2009) Agile Modeling (AM) home page. URL `www.agilemodeling.com`
2. Balaban I, Tip F, Fuhrer RM (2005) Refactoring support for class library migration. In: OOPSLA 2005, ACM, pp 265–279
3. van den Brand M, van Deursen A, Heering J, de Jong HA, de Jonge M, Kuipers T, Klint P, Moonen L, Olivier PA, Scheerder J, Vinju JJ, Visser E, Visser J (2001) The ASF+SDF Meta-environment: A component-based language development environment. In: CC 2001, Springer, LNCS, vol 2027, pp 365–370
4. van den Brand M, Scheerder J, Vinju JJ, Visser E (2002) Disambiguation filters for scannerless generalized LR parsers. In: CC 2002, Springer, LNCS, vol 2304, pp 143–158
5. van den Brand M, Iversen J, Mosses PD (2006) An action environment. Sci Comput Program 61(3):245–264

6. Brown DF, Moura H, Watt DA (1992) Actress: An action semantics directed compiler generator. In: CC '92, Springer, LNCS, vol 641, pp 95–109
7. Chalub F, Braga C (2007) Maude MSOS tool. In: WRLA 2006, Elsevier, ENTCS, vol 176(4), pp 133–146
8. van Deursen A, Heering J, Klint P (eds) (1996) Language Prototyping: An Algebraic Specification Approach, AMAST Series in Computing, vol 5. World Scientific
9. Doh KG, Mosses PD (2003) Composing programming languages by combining action-semantics modules. Sci Comput Program 47(1):3–36
10. Ekman T, Hedin G (2007) The JastAdd system – modular extensible compiler construction. Sci Comput Program 69(1-3):14–26
11. Gosling J, Joy B, Steele G (1996) The Java Language Specification, 1st edn. Addison-Wesley
12. Gosling J, Joy B, Steele G, Bracha G (2005) The Java Language Specification, 3rd edn. Addison-Wesley
13. Gurevich Y (1995) Evolving algebras 1993: Lipari guide. In: Börger E (ed) Specification and Validation Methods, Oxford University Press, pp 9–36
14. Heering J, Klint P (2000) Semantics of programming languages: A tool-oriented approach. SIGPLAN Notices 35(3):39–48
15. Hudak P, Hughes J, Jones SP, Wadler P (2007) A history of Haskell: Being lazy with class. In: HOPL III, ACM, pp 12.1–55
16. Iversen J (2007) An action compiler targeting Standard ML. Sci Comput Program 68(2):79–94
17. Iversen J, Mosses PD (2005) Constructive action semantics for Core ML. IEE Proc-Softw 152:79–98
18. Jaskelioff M (2009) Modular monad transformers. In: Castagna G (ed) ESOP 2009, Springer, LNCS, vol 5502, pp 64–79
19. Johnstone A, Scott E (2007) Proofs and pedagogy; science and systems: The Grammar Tool Box. Sci Comput Program 69:76–85
20. Kastens U, Waite WM (1994) Modularity and reusability in attribute grammars. Acta Inf 31(7):601–627
21. Kutter P, Pierantonio A (1997) Montages: Specifications of realistic programming languages. J Univers Comput Sci 3(5):416–442
22. Labra Gayo JE, Cueva Lovelle JM, Luengo Díez MC, Cernuda del Río A (2002) Reusable monadic semantics of object oriented programming languages. In: SBLP 2002, PUC-Rio, Brazil, pp 86–100
23. Liang S, Hudak P (1996) Modular denotational semantics for compiler construction. In: ESOP '96, Springer, LNCS, vol 1058, pp 219–234
24. Milner R, Tofte M, Harper R, MacQueen D (1997) The Definition of Standard ML – Revised. MIT Press
25. Moggi E (1989) An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Univ. of Edinburgh
26. Mosses PD (1992) Action Semantics, Cambridge Tracts in Theoretical Computer Science, vol 26. Cambridge University Press
27. Mosses PD (1996) Theory and practice of action semantics. In: MFCS '96, Springer, LNCS, vol 1113, pp 37–61
28. Mosses PD (2002) Pragmatics of Modular SOS. In: AMAST'02, Springer, LNCS, vol 2422, pp 21–40
29. Mosses PD (2004) Modular structural operational semantics. J Log Algebraic Program 60–61:195–228
30. Mosses PD (2008) Component-based description of programming languages. In: Visions of Computer Science, BCS, Electronic Proceedings, pp 275–286
31. Mosses PD (2009) Component-based semantics. In: SAVCBS '09, ACM, pp 3–10
32. Mosses PD, New MJ (2009) Implicit propagation in structural operational semantics. In: SOS 2008, Elsevier, ENTCS, vol 224, pp 9.49–66
33. Mosses PD, Watt DA (1987) The use of action semantics. In: Formal Description of Programming Concepts III, North-Holland, pp 135–166
34. de Moura HP, Watt DA (1994) Action transformations in the ACTRESS compiler generator. In: CC '94, Springer, LNCS, vol 786, pp 16–60
35. Plotkin GD (2004) A structural approach to operational semantics. J Log Algebraic Program 60–61:17–139
36. Plotkin GD, Power AJ (2004) Computational effects and operations: An overview. In: Domains VI, Elsevier, ENTCS, vol 73, pp 149–163
37. Scott E, Johnstone A (2006) Right nulled GLR parsers. ACM Trans Program Lang Syst 28(4):577–618
38. Scott E, Johnstone A (2010) GLL parsing. In: LDTA 2009, Elsevier, ENTCS, to appear
39. Scott E, Johnstone A, Economopoulos G (2007) A cubic Tomita-style GLR parsing algorithm. Acta Informatica 44(6):427–461
40. Stärk R, Schmid J, Börger E (2001) Java and the Java Virtual Machine. Springer
41. Visser E (1997) Syntax definition for language prototyping. PhD thesis, University of Amsterdam
42. Visser E (2004) Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In: Domain-Specific Program Generation, Springer, LNCS, vol 3016, pp 216–238
43. Watt DA (1988) An action semantics of Standard ML. In: MFPS III, Springer, LNCS, vol 298, pp 572–598
44. Watt DA (2004) Programming Language Design Concepts. John Wiley & Sons
45. Watt DA, Thomas M (1991) Programming Language Syntax and Semantics. Prentice-Hall