BRICS

Basic Research in Computer Science

BRICS RS-03-53 Doh & Mosses: Composing Programming Languages by Combining Action-Semantics Modules

Composing Programming Languages by Combining Action-Semantics Modules

Kyung-Goo Doh
Peter D. Mosses

BRICS Report Series RS-03-53

ISSN 0909-0878 December 2003

Copyright © 2003, Kyung-Goo Doh & Peter D. Mosses.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone:	+45 8942 3360
Telefax:	+45 8942 3255
Internet:	BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/03/53/`

Composing Programming Languages by Combining Action-Semantics Modules [1]

Kyung-Goo Doh [2,3]

Department of Computer Science and Engineering
Hanyang University, Ansan, South Korea

Peter D. Mosses [4,5]

BRICS & Department of Computer Science
University of Aarhus, Denmark

Abstract

This article demonstrates a method for composing a programming language by combining action-semantics modules. Each module is defined separately, and then a programming-language module is defined by combining existing modules. This method enables the language designer to gradually develop a language by defining, selecting and combining suitable modules. The resulting modular structure is substantially different from that previously employed in action-semantic descriptions.

It also discusses how to resolve the conflicts that may arise when combining modules, and indicates some advantages that action semantics has over other approaches in this respect.

Key words: action semantics, modularity, ASF+SDF

[1] Published in *Science of Computer Programming*, 47(1):3–36, 2003.
[2] This work was supported by grant No.R01-2000-00287 from the Basic Research Program of the Korea Science & Engineering Foundation.
[3] Email: `doh@cse.hanyang.ac.kr`
[4] Supported by BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation.
[5] Email: `pdmosses@brics.dk`

1 Introduction

Hoare noted in his POPL'73 paper [1] that much of programming language design is consolidation, not innovation. In other words, a language designer should largely utilize constructions and principles that have worked well in earlier design projects and experiments [2]. (cf. the evident close relationship between C and C++, C and Java [6], and Java and C# [7]). To this end, language definitions should be modularized. Then, a language designer's job may involve the definition or reuse of many different alternative language modules, to choose the best combination of them based on language-design principles, and to reject any that show mutual inconsistencies. Any remaining minor inconsistencies or overlaps may be reconciled by applying good engineering principles. A semantic framework must provide a high degree of modularity so that the designer can build up a language description smoothly and uniformly. For defining (concrete or abstract) context-free syntax, BNF grammars have excellent modularity: the union of two BNF grammars (without start symbols) is always well-formed, and specifies the language with all the constructs specified by the two grammars separately; for defining semantics, the framework of *action semantics* appears to offer similar advantages.

Action semantics has been developed by Mosses and Watt [3–7]. The motto of action semantics is to allow *useful* semantic description of *realistic* programming languages [3]. The potential uses of action semantic descriptions include documentation of decisions during language design, language standardization, guidance for implementors, compiler and interpreter generation, and formal reasoning about programs; Mosses [4] gives a survey of some actual uses.

In action semantics, a high-level notation called *action notation* is used to describe the meaning of a programming language. Primitive actions exist for the fundamental behaviours of information processing: value passing, arithmetic, binding creation and lookup, storage allocation and manipulation, and so on. Actions are composed into a combined action with combinators controlling the flow of information. Actions have semi-descriptive English names, which gives a novice reader hints of the intuition behind them. Furthermore, action semantics specifications are inherently *modular*. Hence, they can be straightforwardly extended and modified to reflect language design changes, and to reuse parts of an existing language definition for specifying similar, related languages. In fact, some real-world programming languages, such as Pascal [8], and Standard ML [9,10], have been successfully described in action semantics.

[6] Java is a trademark of Sun Microsystems Inc.
[7] C# is a trademark of Microsoft Corporation.

1.1 This work

This article demonstrates a method for composing a programming language by combining action-semantics modules. There are three kinds of modules:

semantic functions modules, which merely declare the names and types of semantic functions (without any semantic equations);
semantic equations modules, each of which declares the syntax of, and defines the semantics of, a single language construct; and
semantic entities modules, which declare the standard notation for actions, and define further auxiliary sorts and operations.

Each module is specified separately, and then a complete programming-language module is defined by properly combining existing modules. This method enables the language designer to gradually develop a language by defining, selecting, and combining suitable modules. The size of each module is extremely small compared to those previously employed in action-semantic descriptions, and their overall organization is significantly different: the original style was to specify abstract syntax in one module, and to have a separate module for each syntactic sort, specifying the semantic functions and equations for all the syntactic constructs of that sort together. The novel style of modularization proposed here greatly facilitates the direct reuse of entire modules in action semantics.

When combining modules, conflicts may occur in the definition of a combined module. This article suggests how to resolve the problem of conflicts, exploiting several key features of the action semantics framework. Modules for constructs stemming from the so-called Landin-Tennent design principles (the principles of abstraction, parameterization, correspondence, and qualification) [2,11] are particularly uniform, and straightforward to combine.

Note that only dynamic semantics is considered in this article, since the main focus here is to illustrate the good modularity and extensibility of action semantics. Static semantics (e.g., type-checking semantics) can be defined in inference-rule format as in Plotkin's lecture notes on SOS [12] and in Schmidt's textbook on typed programming languages [2], or in action notation as in Watt's work [9,10], becoming an essential part of a language specification.

1.2 Road map

The rest of the article is organized as follows: Section 2, briefly introduces how to specify action semantics in a specification framework called ASF+SDF. Section 3 defines modules necessary to compose a simple expression language.

Section 4 presents modules for expression bindings, expression blocks, expression parameters, and expressions with effects. Section 5 shows how a language module can be composed by combining related modules, and discusses how to resolve conflicts between combined modules. Section 6 concludes, also considering some other semantic frameworks that aim to support modularity.

2 Action Semantics

Action semantics uses context-free grammars to define the structure of abstract-syntax trees, and inductively defined semantic functions to give semantics to such trees. Each semantic equation is compositional and maps an abstract-syntax tree to an action representing the meaning of the tree. In this section, we briefly review the notation for actions, and then illustrate how to specify action semantics descriptions in ASF+SDF.

2.1 Action Notation

Actions are semantic entities that represent implementation-independent computational behaviour of programs. The performance of an action, which may be part of an enclosing action, either *terminates normally* (the performance of the enclosing action continues normally); or *terminates exceptionally* (the enclosing action is skipped until the exception is handled); or *fails*, corresponding to abandoning the current alternative of a choice (any remaining alternative is tried); or *diverges*, never terminating (the enclosing action also diverges).

The performance of an action processes transient data; it creates and accesses bindings of token to data; it allocates and manipulates primary storage; and it communicates between distributed agents. Actions have various *facets*. The *basic* facet processes independently of information, focusing on control flow; the *functional* facet processes transient information, including actions operating on data; the *declarative* facet processes scoped information, including actions operating on bindings; the *imperative* facet processes stable information, including actions operating on storage cells; and the *communicative* facet processes permanent information, including actions operating on distributed systems of agents. Actions with different facets can be freely combined to form multi-faceted actions. So-called *yielders* are used to inspect the current information (without changing it).

Some action notation important to understand the underlying concepts of defined languages is explained along with the semantic descriptions below. However, some highly suggestive data notations are not explained, for brevity.

Note that despite its verbose and casual appearance, action notation is completely formal; a list of all the symbols used in the present paper is provided in an appendix. A full, formal description of action notation can be found in Mosses's book on action semantics [3], and in a more recent Modular SOS definition [6].

This paper uses a new version of action notation (AN-2) [13], which has a significantly simpler kernel than the original version, but which is otherwise quite similar in use. The differences between the two versions do not substantially affect the modularity issues addressed in the present paper.

2.2 Action semantics in ASF+SDF

In this paper, we use the ASF+SDF formalism [14,15] to express action semantics descriptions. ASF+SDF differs somewhat from the meta-notation usually employed in action semantics [3], but it has the advantage that the modules can be checked for well-formedness using the current version of the ASF+SDF Meta-Environment.[8] Moreover, ASF+SDF is a well-established formalism that may already be familiar to many readers. In fact it is the result of the marriage of two formalisms: ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). ASF is based on the notion of a module declaring a signature (sorts, function symbols, and variables) and giving a set of conditional equations defining their properties. Modules can be imported into other modules. SDF allows the definition of concrete (lexical and context-free) syntax for operations and variables. The following overview indicates how action semantics descriptions are written in ASF+SDF; for further details of ASF+SDF, the reader is referred to [14,15].

With the novel modular structure for action semantics descriptions proposed here, a description is divided into the following kinds of modules:

Semantic Functions: Each such module declares a single semantic function, say `f`, for a particular sort of syntax, say `s`:

```
"f" "[[" s "]]" -> t
```

where `t` is some sort of semantic entities—usually the sort `Action` of all possible actions. It imports the semantic entities modules that are needed to express entities of sort `t`. The double brackets `[[...]]` are included as part of the (mixfix) notation for each semantic function (they have to be quoted in declarations, but not when they are used).

[8] The ASD Tools [16], previously developed to generate ASF+SDF modules from a notation similar to the original meta-notation of action semantics, are incompatible with the new modular structure proposed here.

The module also declares variables ranging over `s`, for instance:

```
"S"[1-9]? -> s
```

declares `S`, `S1`, ..., `S9` as variables.

Semantic Equations: Such a module declares an abstract syntax constructor operation, and gives a semantic equation defining the result of applying a particular semantic function to abstract syntax trees constructed by this operation—generally applying other semantic functions to components.

Suppose each symbol `w1`, ..., `wn` is either a quoted symbol `"..."` or a syntactic sort symbol. Then if `s` is also a syntactic sort symbol,

```
w1 ... wn -> s
```

declares an abstract syntax constructor with the indicated argument sorts (distinguished from other constructs partly by the quoted symbols). The semantics of all trees thus constructed is specified by a semantic equation written:

```
f [[ v1 ... vn ]] = T
```

where each `vi` is either an (unquoted) symbol `wi` or a variable ranging over a sort `wi`, and `T` is a term of sort `t` specifying how the semantics `fi[[vi]]` of the components are combined.

The module imports the semantic functions modules for all the syntactic sorts and semantic functions involved, and any required semantic entities modules that are not already indirectly imported.

Semantic Entities: These modules may specify new sorts of data with constructors and selectors, include these sorts into previously-declared sorts, and define derived operations. The standard semantic entities module `AN` declares all the symbols provided by (the new version of) action notation for expressing actions, data, and yielders. (In fact `AN` itself has an interesting internal modular structure, but it is irrelevant to our main purpose in the present paper, and not discussed further here.)

To specify a sort `s` with constructor operation `f` and argument sorts `s1`, ..., `sm`, we generally keep to prefix notation and declare the operation thus:

```
f(s1,...,sm) -> s
```

where `s1,...,sm` are the argument sorts and `s` the result sort. For perspicuity we shall also label the component sorts with the names of the corresponding selector operations, e.g.:

```
f(g1:s1,...,gm:sm) -> s
```

However, such labels do *not* (presently) give rise to operation declarations in ASF+SDF, so a separate declaration of each selector (of the form `"gi" s -> si`, for prefix notation `gi(S)`) has to be given. Derived operations are defined straightforwardly by equations.

To compose a language, one simply imports the semantic equations modules for all the desired constructs (the relevant semantic functions and semantic entities modules are imported indirectly). Note that different semantic equations modules for the same abstract syntax construct should not be imported together: their inconsistency has to be removed first, as described in Section 5.

Concrete illustrations of how action semantic descriptions are formulated in ASF+SDF are given in Sections 3 and 4. The full ASF+SDF sources of all the examples are available from `http://www.brics.dk/~pdm/LDTA-01/`.

3 Action-Semantics Modules for an Expression Language

In this section, we define, in ASF+SDF, action-semantics modules that allow us to compose an expression language, as an illustration of the general approach. We start with a semantics entities module named `Values` as follows: [9]

```
module Values
  imports AN
  exports
    sorts Value
    context-free syntax
      Value -> Datum
```

The module `Values` above imports the predefined module `AN`, which specifies the entire action notation (version 2, referred to as AN-2 [13]). `Value`, a new sort of data, is merely declared to be a subsort of `Datum`, which is the sort of all individual items of data in action notation; exactly which values are in `Value` is left to be further specified in other modules.

Next, a semantic functions module `Exp` for expressions is specified as follows:

```
module Exp
  imports Values
  exports
    sorts Exp
    context-free syntax
      "evaluate" "[[" Exp "]]" -> Action %% giving Value
    variables "E"[1-9]? -> Exp
```

In action semantics, a semantic function maps an abstract-syntax tree to an action representing its computational meaning. The module `Exp` declares the semantic function `evaluate` and its functionality, leaving its meaning to be

[9] Some lines of SDF, necessary for the implementation but otherwise of no real interest, have occasionally been elided from the examples.

defined by semantic equations modules. The line declaring `evaluate` indicates that for every abstract-syntax tree `E` in the abstract-syntax domain `Exp`, the semantic entity `evaluate [[E]]` is an action. The comment at the end of the line (starting `%%`) indicates that `evaluate [[E]]` gives a result of sort `Value` (on normal termination), i.e. `Value` is the sort of *expressible* values.[10] The last line of the module declares variables ranging over `Exp`, for use in semantic equations modules that involve expressions.

The following module declares a semantic function for binary operators:

```
module Op2
  imports Values
  exports
    sorts Op2
    context-free syntax
      "operate2" "[[" Op2 "]]" -> Action %% taking (Value,Value)
                                         %% giving Value
    variables "O2" -> Op2
```

The following module defines the syntax and semantics of expressions with binary operators. Note again that the specifics of what kinds of binary operators may be used is left to be specified separately.

```
module Exp/binary
  imports Exp Op2
  exports
    context-free syntax
      Exp Op2 Exp -> Exp
  equations
    [1] evaluate [[ E1 O2 E2 ]] =
        ( evaluate [[E1]] and evaluate [[E2]] )
        then operate2 [[O2]]
```

The notation `Exp Op2 Exp -> Exp` above is equivalent to the usual BNF notation `Exp ::= Exp Op2 Exp`. The variables used in the above module are declared in the modules `Exp` and `Op2`.

The meaning of `evaluate [[ E1 O2 E2 ]]` can be explained informally as follows: both `E1` and `E2` are evaluated to give values, and then the values are passed to be operated upon by `O2`. Notice how close this informal explanation is to Equation `[1]` of the module `Exp/binary` above (which is completely formal, despite its "natural" appearance). As the reader might have noticed, the semantic equations in action semantics are *compositional*, as in Scott-Strachey style denotational semantics [17,18]: the semantics of any compound phrase is

[10] The AN-2 notation for formally specifying such properties of actions has not yet been settled.

determined in terms of the semantics of its subphrases.

The action combinator `and` is a basic combinator that represents implementation-dependent order of performance; when there is no interference between the two sub-actions of `and`, the order of evaluation is not significant. The `and` combinator makes a tuple of transient values given by its sub-actions. The `then` combinator passes transient values from the left sub-action to the right sub-action.

Each specific binary operator can be defined in a separate module as follows:

```
module Op2/plus
  imports Op2
  exports
    context-free syntax
      "+" -> Op2
      Int -> Value
  equations
    [2] operate2 [[ + ]] = give ( the int #1 + the int #2 )
```

The inclusion of `Int` in `Value` is clearly needed when integer addition is allowed in expressions, so it is specified above. (An alternative style would be to defer all specifications of sort inclusions until after the composition of a language has been decided.)

Lowercase constant symbols are used in action notation to indicate sorts of data, e.g. `int` indicates the sort `Int`. When `ds` indicates any sort of data, the data operation `the ds` projects its argument onto that sort, the result being undefined if the argument is not in the sort. For any (positive) numeral `n`, the data operation `#n` selects the `n`'th component from any data tuple having at least that many components. The composition `the ds #n` of these data operations is a yielder that first selects the `n`'th component then checks that it is of the sort indicated by `ds`. Hence above, `the int #1 + the int #2` yields the sum of the first two components of the given data tuple, provided that they are both of sort `Int`. For any yielder Y, the action `give` Y simply gives the value yielded by Y (an item of data is a special case of a yielder).

Expression grouping, can be specified as follows:

```
module Exp/group
  imports Exp
  exports
    context-free syntax
      "(" Exp ")" -> Exp {bracket}
```

By declaring the syntax above as a bracketing construct, no node is generated for it in the abstract syntax tree, so no semantic equation for `evaluate [[(E)]]` is needed.

The module `Lit` below declares `value` to be a semantic function for the syntactic sort `Lit` of literal constants. The semantics of literal constants is assumed to be not only inherently "mathematical" but also independent of the context, and it is taken to be an item of data (rather than an action). Notice that no assumption is made here about whether the sort `Lit` is included in `Exp`.

```
module Lit
  imports AN
  exports
    sorts Lit LitValue
    context-free syntax
      "value" "[[" Lit "]]" -> LitValue
      LitValue -> Datum
    variables "L"[1-9]? -> Lit
```

Shown below is a semantic equations module which specifies the syntax and semantics of literal constants in expressions. Exactly what constitutes the literal constants is left unspecified.

```
module Exp/literal
  imports Exp Lit
  exports
    context-free syntax
      Lit -> Exp
      LitValue -> Value
  equations
    [3] evaluate [[ L ]] = give value [[L]]
```

The notation `Lit -> Exp` above specifies the inclusion of the abstract syntax of `Lit` in `Exp`, and corresponds to the usual BNF notation `Exp ::= Lit`. Equation `[3]` above specifies how the values of literal constants contribute to the actions that evaluate expressions.

Note that what kinds of literals there might be is still left unspecified in the above module. Some specific literals, such as numeric constants, can be specified in separate modules.

The sort `Nat` of natural numbers used in the following module is provided by `AN`, together with ordinary decimal notation. The following semantic equations merely confirm that the semantics of a numeric literal is indeed decimal.[11]

```
module Lit/numeric
  imports Lit
  exports
    sorts Num
    context-free syntax
      Num -> Lit
      Nat -> LitValue
    lexical syntax
      [0-9]+ -> Num
```

[11] In fact it isn't necessary to be so pedantic: an alternative is to include `Nat` in the syntax of literals, and let `value[[N]] = N`.

```
hiddens
  variables "C+" -> CHAR+
            "C"  -> CHAR
equations
  [4-0]  value [[ num("0") ]] = 0
  %% ...
  [4-9]  value [[ num("9") ]] = 9
  [4-10] value [[ num(C+ C) ]] =
         (10 * value [[num(C+)]]) + value [[num(C)]]
```

The variables `C` and `C+` are declared as hidden above since the module is essentially declaring both the syntax and semantics for `Num`.

The next module departs slightly from the strictly minimalist style proposed in this paper, by specifying the semantics of two different constructs together. In general, it seems best to stick to specifying all constructs separately, although here, it indeed seems quite unlikely that a language would include `true` without `false`, or vice versa.

```
module Lit/boolean
  imports Lit
  exports
    context-free syntax
      "true"  -> Lit
      "false" -> Lit
      Bool -> LitValue
  equations
    [5] value [[ true ]] =  true
    [6] value [[ false ]] = false
```

By the way, the `Bool` values `true` and `false` used on the right of the above equations are provided by the standard action notation (which is imported indirectly, via the module `Lit`).

By now, the reader should have become reasonably familiar with our use of ASF+SDF to specify semantic functions and semantic equations module. Some further examples of modules specifying semantic entities will be provided in Section 4. As a further example, relational expressions and conditional expressions can be similarly specified by defining modules as follows:

```
module Op1
  imports Values
  exports
    sorts Op1
    context-free syntax
      "operate1" "[[" Op1 "]]" -> Action %% taking Value
                                         %% giving Value
    variables "O1" -> Op1

module Op1/not
  imports Op1
  exports
    context-free syntax
      "not" -> Op1
      Bool -> Value
  equations
    [7] operate1 [[ not ]] = give ( not the bool )

module Op2/equal
  imports Op2
  exports
    context-free syntax
      "=" -> Op2
      Bool -> Value
  equations
    [8] operate2 [[ = ]] =
        when (the value #1 = the value #2)
        then give true otherwise give false

module Exp/unary
  imports Exp Op1
  exports
    context-free syntax
      Op1 Exp -> Exp
  equations
    [9] evaluate [[ O1 E ]] =
        evaluate [[E]] then operate1 [[O1]]

module Exp/if-then-else
  imports Exp
  exports
    context-free syntax
      "if" Exp "then" Exp "else" Exp -> Exp
      Bool -> Value
  equations
    [10] evaluate [[ if E1 then E2 else E3 ]] =
         evaluate [[E1]] then
         select(
           ( given true then evaluate [[E2]] ) or
           ( given false then evaluate [[E3]] ) )
```

Notice that the action combinator `or` above, generally used for nondeterministic choice, here represents a *deterministic* choice, since at least one of two sub-actions must always fail.

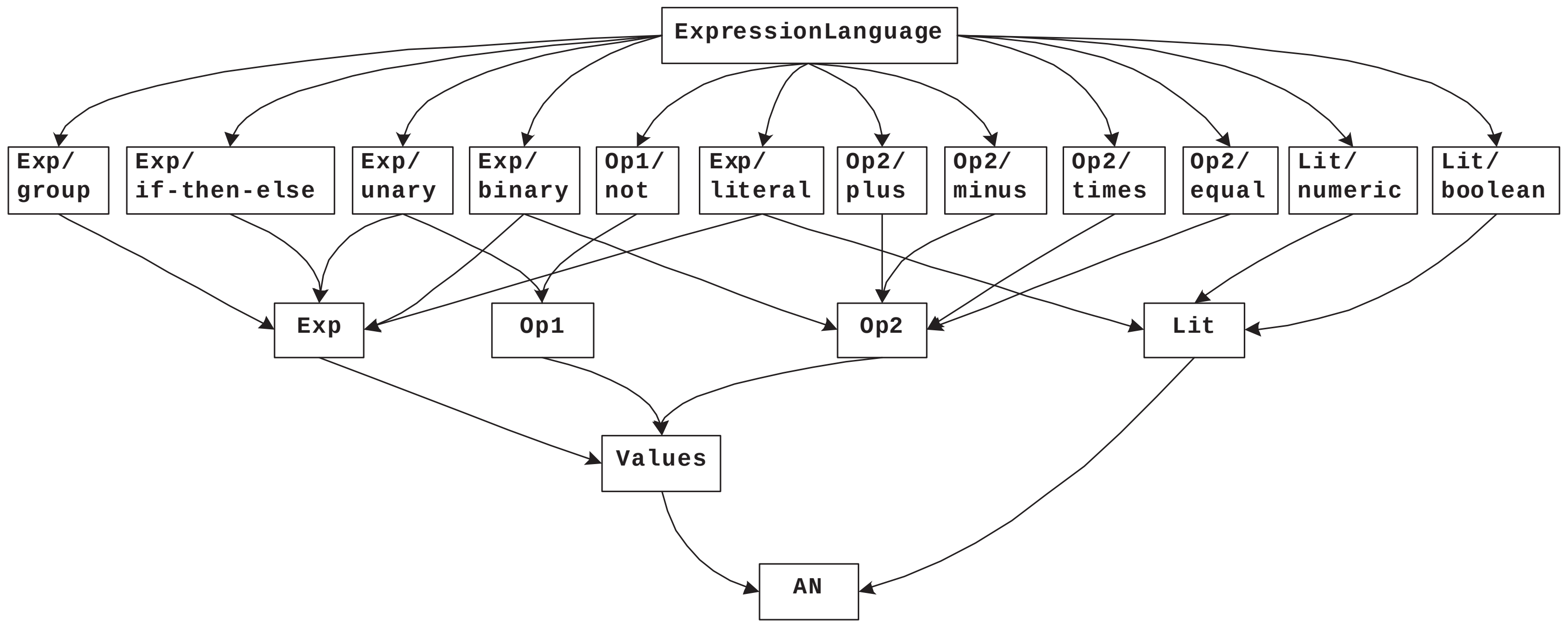

Fig. 1. The module dependency graph for `ExpressionLanguage`

Modules can be combined to specify a programming language. For example, the modules defined so far can be combined to define an expression language as follows:

```
module ExpressionLanguage
  imports
    Exp/literal    Exp/unary        Exp/binary
    Exp/group      Exp/if-then-else
    Lit/numeric    Lit/boolean
    Op1/not        Op2/equal
    Op2/plus       Op2/minus        Op2/times
```

Note that shared notation is the crucial feature when combining modules: their internal structure and import relationship are irrelevant. The dependency relationships among imported modules in `ExpressionLanguage` are depicted as a graph in Figure 1. The graph shown is the exact copy of the one generated by the Meta-Environment with a minor layout modification to fit in a page.

4 Modules as Building Blocks for Programming Languages

In this section, we define modules for expression bindings, expression blocks, expression parameters, and expressions with effects, in turn. On the way, we contemplate the semantics of eager vs. lazy binding, static vs. dynamic scoping, call-by-value vs. call-by-name parameter passing schemes, and expressions with effects.

4.1 Modules for Expression Bindings

According to the Landin-Tennent programming-language design principles, the phrases in any semantically meaningful syntactic class may be named, which is specifically called the *abstraction principle*. The process of giving a name to a program construct is called *binding*. A named expression is called an *expression abstraction*, which can be invoked later by simply mentioning its name.

The semantic functions module `Ide` for names (identifiers) is as follows:

```
module Ide
  imports AN
  exports
    sorts Ide
    context-free syntax
```

```
      "token" "[[" Ide "]]" -> Token
    variables "I"[1-9]? -> Ide
```

The symbol `Ide` in the body of the above module is a syntactic sort for names, whereas `Token`, imported from `AN`, is the sort of all data items that may be bound to values in actions.

The semantic function module `Dec` for declarations (binding constructs) is specified as follows:

```
module Dec
  imports AN
  exports
    sorts Dec
    context-free syntax
      "declare" "[[" Dec "]]" -> Action %% giving Bindings
    variables "D"[1-9]? -> Dec
```

The constructs for declarations are grouped into a separate syntax sort, `Dec`, the details of which are yet to be specified. The functionality of the semantic function `declare` indicates that for every abstract-syntax tree `D` in the syntactic sort `Dec`, the semantic entity `declare [[D]]` is an action which gives bindings.

The meaning of an expression abstraction may be different depending on *when* the expression is evaluated. An expression may be evaluated before it is bound to a name (called eager binding), or after it is invoked (called lazy binding).

4.1.1 Eager Binding

The semantic equations module for expression-binding with eager evaluation, `Dec/val`, is specified as follows:

```
module Dec/val
  imports Dec Ide Exp
  exports
    context-free syntax
      "val" Ide "=" Exp -> Dec
  equations
    [11] declare [[ val I = E ]] =
           evaluate [[E]] then
           give binding (token [[I]], the bindable)
```

The semantic equation `[11]` above shows that the body of an expression binding, `E`, is evaluated before being bound to a token, implying eager evaluation. The sort of all values that can be bound to tokens in actions is called `Bindable`, and is yet to be specified. In the case of eager binding, the Landin-Tennent principle motivates identifying `Bindable` with `Value`, or at least letting it include all values; but by leaving the relationship between `Bindable` and `Value` open, the above module may be useful also for composing languages whose design does not dogmatically adhere to the mentioned principle. Note that when not all values are bindable, the above action may fail.

The semantic equation module `Exp/const-val` for name invocation is defined as follows:

```
module Exp/const-val
  imports Exp Ide
  exports
    context-free syntax
      Ide -> Exp
  equations
    [12] evaluate [[ I ]] =
           give the value bound to token [[I]]
```

The name invocation looks up current bindings and gives the value bound to the name. Here, the action may fail when not all bindables are values — just the opposite of the situation with the module `Dec/val`.

4.1.2 Lazy Binding

In lazy binding, an expression is not evaluated and is bound to a name. The semantic entities module `Functions` introduces a new sort, `Function`, representing a function abstraction. It also specifies a new action `apply` which takes a tuple of an action and a datum and performs the action with the given datum. The primitive action `enact` simply performs the action given to it as data (`Action` is actually a subsort of `Data`).

```
module Functions
  imports AN
  exports
    sorts Function
    context-free syntax
      Function -> Datum
      apply -> Action %% taking (Action,Datum)
  equations
    [13] apply = enact (provide the datum#2 then the action#1)
```

The next semantic entities module is defined specifically for nullary functions. A nullary function is constructed from an action that corresponds to an unevaluated expression, and the action can be retrieved by applying the selector `action` to it:

```
module Functions/nullary
  imports Functions
  exports
    context-free syntax
      nullary-function(action:Action) -> Function
```

The semantic functions module for expression binding with lazy evaluation, `Dec/fun-nullary`, is now defined as follows:

```
module Dec/fun-nullary
  imports Dec Ide Exp Functions/nullary
  exports
    context-free syntax
      "fun" Ide "=" Exp -> Dec
      Function -> Bindable
  equations
    [14] declare [[ fun I = E ]] =
         give binding (token [[I]],
                       nullary-function (closure (evaluate [[E]])))
```

The yielder `nullary-function` A, where A is an action, yields a datum encapsulating A, but keeping no bindings. In order to keep the bindings that are current at the evaluation of `nullary-function` A, `closure` must be used: `closure` lets the encapsulated action keep the current bindings, and then when the encapsulated action is performed, it receives the kept bindings. Thus, `nullary-function (closure (evaluate [[E]]))` yields an encapsulated action of `evaluate E`, but the action incorporates the bindings available at the time, i.e., at declaration-time. Thus, the action `evaluate [[E]]` in Equation `[14]` above is not performed, but encapsulated with the current bindings and bound to a name (i.e., the evaluation is delayed). Since a nullary function is bound to a name, `Function` is specified to be included in `Bindable`.

The semantic equations module `Exp/fun-nullary` for lazy invocation is defined as follows:

```
module Exp/fun-nullary
  imports Exp Ide Functions/nullary
  exports
    context-free syntax
      Ide -> Exp
      Function -> Bindable
  equations
    [15] evaluate [[ I ]] =
         enact action(the function bound to token [[I]])
```

The encapsulated action (function) is enacted and performed, receiving the bindings incorporated at declaration-time, when the name is invoked as shown in Equation `[15]` above. Since the encapsulated action receives the same bindings regardless of where it is enacted and performed, it is called *static binding.*

4.1.3 Dynamic Binding

The semantic equations modules for expression binding with lazy evaluation and dynamic binding are defined as follows:

```
module Dec/fun-nullary-dynamic
  imports Dec Ide Exp Functions/nullary
  exports
    context-free syntax
      "fun" Ide "=" Exp -> Dec
      Function -> Bindable
  equations
    [16] declare [[ fun I = E ]] =
         give binding (token [[I]],
                       nullary-function (evaluate [[E]]))

module Exp/fun-nullary-dynamic
  imports Exp Ide Functions/nullary
  exports
    context-free syntax
      Ide -> Exp
      Function -> Bindable
  equations
    [17] evaluate [[ I ]] =
         enact (closure action(the function bound to token [[I]]))
```

The yielder `closure action(the function bound to token [[I]])` yields an action that incorporates the bindings available at the time, i.e., at invocation-time. Thus the enaction performs the action encapsulated in the action, letting it receive the current bindings. (The primitive action `enact` does not itself add any further bindings to those already incorporated in the action given to it.)

4.1.4 Modules for Multiple Declarations

The semantic equations module for multiple, independent declarations is defined as follows:

```
module Dec/seq-indep
  imports Dec
  exports
    context-free syntax
      Dec "," Dec -> Dec
  equations
    [18] declare [[ D1 , D2 ]] =
         declare [[D1]] and then declare [[D2]]
         then give disjoint union
```

The action combinator `and then` indicates that the declarations are declared sequentially, and then the disjoint union of the produced bindings is formed (with exceptional termination when both declarations produce a binding for the same token).

The module for multiple, sequential declarations is defined as follows:

```
module Dec/seq
  imports Dec
  exports
    context-free syntax
      Dec ";" Dec -> Dec
  equations
    [19] declare [[ D1 ; D2 ]] =
         declare [[D1]] before declare [[D2]]
```

The action combinator `before` indicates that the bindings created by `declare [[D1]]` can be used in `declare [[D2]]`, and the bindings created by `declare [[D2]]` overrides the bindings created by `declare [[D1]]`.

4.2 Modules for Expression Blocks

The Landin-Tennent qualification principle says that any semantically meaningful syntactic class may admit local declarations. This means in particular that any expression belonging to the syntax class `Exp` can have local definitions. A construct admitting local definitions is called a *block*. The semantic equations module for an expression block is defined as follows [12]:

```
module Exp/let
  imports Exp Dec
  exports
    context-free syntax
      "let" Dec "in" Exp -> Exp
  equations
    [20] evaluate [[ let D in E ]] =
         furthermore declare [[D]] hence evaluate [[E]]
```

[12] In future versions of ASF+SDF, it should be possible to specify a single parameterized module for arbitrary blocks, and instantiate it to get the desired expression blocks.

The action `furthermore declare [[D]]` overlays the current bindings with the ones produced by `declare [[D]]`. Then the `hence` combinator passes the overlaid bindings to the action `evaluate [[E]]`. This corresponds exactly to an ordinary block structure. Thus, bindings declared in `D` can only be referred in the body `E`, not outside the block.

The distinction between *static* and *dynamic* binding arises when an abstraction created in the scope of one set of bindings may get applied in the scope of another set. In the examples given above, `let` blocks could involve both static and dynamic scopes for expression abstractions: it is the semantics of the abstractions themselves that determines whether or not the abstraction-time bindings get encapsulated together with the action representing the expression evaluation, for later use; and it is the semantics of name lookup that determines whether or not the lookup-time bindings are made available to the encapsulated action. When an abstraction has determined that it is using the static bindings, it simply ignores any lookup-time bindings supplied by the semantics of name reference. On the other hand, when the semantics of the abstraction ignores the abstraction-time bindings, it depends on the semantics of name lookup as to whether the dynamic bindings will be supplied or not.

4.3 Modules for Expression Parameters

The Landin-Tennent parameterization principle says that phrases from any semantically meaningful syntactic class may be parameters. In particular, any expression belonging to the syntax class `Exp` can be a parameter. In this subsection, we define modules for an expression abstraction with an expression parameter, along with an invocation construct. Modules dealing with declaration abstractions and declaration parameters may be specified analogously.

The meaning of an abstraction invocation with an actual parameter can be varied depending on when its parameter (argument) is evaluated. Here we examine the semantics of two different parameter-passing schemes: call-by-value and call-by-name.

4.3.1 Call-by-Value Parameter-Passing

In the call-by-value parameter-passing scheme, an argument is fully evaluated at the time of invocation of a parameterized abstraction. Then the evaluated value is bound to a formal parameter.

The following semantic entities module is defined specifically for call-by-value unary functions.

```
module Functions/unary-value
  imports Functions
  exports
    sorts Unary-Value-Function
    context-free syntax
      unary-value-function(action:Action) -> Unary-Value-Function
      Unary-Value-Function -> Function
```

The semantic equations modules for call-by-value parameter passing are defined as follows:

```
module Dec/fun-unary-value
  imports Dec Ide Exp Functions/unary-value
  exports
    context-free syntax
      "fun" Ide "(" "val" Ide ")" "=" Exp -> Dec
      Unary-Value-Function -> Bindable
  equations
    [21] declare [[ fun I1 (val I2) = E ]] =
         give binding
           (token [[I1]],
            unary-value-function (
              closure ( furthermore bind (token [[I2]], the value)
                        hence evaluate [[E]] )))

module Exp/fun-unary-value
  imports Exp Ide Functions/unary-value
  exports
    context-free syntax
      Ide "(" Exp ")" -> Exp
      Unary-Value-Function -> Bindable
  equations
    [22] evaluate [[ I ( E ) ]] =
         ( give the unary-value-function bound to token [[I]]
           and then evaluate [[E]] )
         then apply (action(the function#1), the value#2)
```

Equation `[22]` in the module `Exp/fun-unary-value` shows that an argument `E` of function application is evaluated to a value, and then the unary function bound to a token `I` is applied to the value, so that the evaluated argument value is bound to a formal parameter. The binding acts as a local declaration in the body of the function. Equation `[21]` in the module `Dec/fun-unary-value` indicates that the bindings current at the definition are used when the unary function is applied, implying static scoping. Note that the module `Exp/const-val` in Section 4.1.1 (rather than `Exp/fun-nullary` in Section 4.1.2) is combined smoothly together with these modules, which shows that eager binding and call-by-value scheme fit well together.

4.3.2 Call-by-Name Parameter-Passing

In the call-by-name parameter-passing scheme, the evaluation of an argument is delayed until it is used in the body of the called function. That is, the unevaluated argument is bound to a formal parameter, and then evaluated every time the formal parameter is invoked in the body of the called function.

The following semantic entities module is defined specifically for call-by-name unary functions.

```
module Functions/unary-name
  imports Functions
  exports
    sorts Unary-Name-Function
    context-free syntax
      unary-name-function(action:Action) -> Unary-Name-Function
      Unary-Name-Function -> Function
```

The semantic equations modules for call-by-name parameter passing are defined as follows:

```
module Dec/fun-unary-name
  imports Dec Ide Exp Functions/unary-name
  exports
    context-free syntax
      "fun" Ide "(" "name" Ide ")" "=" Exp -> Dec
      Unary-Name-Function -> Bindable
  equations
    [23] declare [[ fun I1 (name I2) = E ]] =
           give binding
             (token [[I1]],
              unary-name-function (
                closure ( furthermore bind (token [[I2]], the action)
                          hence evaluate [[E]] )))

module Exp/fun-unary-name
  imports Exp Ide Functions/unary-name
  exports
    context-free syntax
      Ide "(" Exp ")" -> Exp
      Unary-Name-Function -> Bindable
  equations
    [24] evaluate [[ I ( E ) ]] =
           ( give the unary-name-function bound to token [[I]]
             and give closure (evaluate [[E]]) )
           then apply (action(the function#1), the action#2)
```

Equation `[24]` in the module `Exp/fun-unary-name` shows that an argument `E` of function application is *not* evaluated (i.e., the corresponding action `evaluate E` is encapsulated), and then the unary function is applied to the unevaluated argument, with the result that the encapsulated action representing the unevaluated argument is bound to a formal parameter as shown in the module `Dec/fun-unary-name`. The `closure` in Equation `[24]` in the module `Exp/fun-unary-name` suggests that the bindings available when the parameterized abstraction is invoked should be used when the argument is invoked and evaluated in the body of parameterized abstraction, implying static binding. Since the action representing unevaluated argument is enacted in the body of parameterized function, the module `Exp/name-val` below (instead of `Exp/const-val` in Section 4.1.1) should be used with these modules. As a result, we can see that lazy binding and call-by-name fit well together.

```
module Exp/name-val
  imports Exp Ide
  exports
    context-free syntax
      Ide -> Exp
      Action -> Bindable
  equations
    [25] evaluate [[ I ]] = enact the action bound to token [[I]]
```

4.4 Modules for Expressions with Effects

In this section, we examine modules for expressions with effects. We first show the semantic entities module `Variables` introducing a new sort `Variable` and two new actions, `assign` and `dereference`, necessary to define expression modules with effects.

```
module Variables
  imports AN
  exports
    sorts Variable
    context-free syntax
      Variable -> Datum
      "assign"      -> Action %% taking (Variable,Value)
                              %% giving ()
      "dereference" -> Action %% taking Variable
                              %% giving Value
```

We then define the semantic entities module `Variables/simple` specifying the meanings of the actions, `assign` and `dereference`, for simple variables.

```
module Variables/simple
  imports Variables
  exports
    sorts Simple-Variable
    context-free syntax
      "simple-variable"(cell:Cell) -> Simple-Variable
      Simple-Variable -> Variable
  equations
    [26] assign =
         given (the simple-variable#1, the storable#2)
         then update (cell(the simple-variable#1), the storable#2)
    [27] dereference =
         given the simple-variable
         then inspect (cell(the simple-variable))
```

We next define semantic equations modules for expressions with effects, with an ML-like syntax. Equation `[28]` in the module `Exp/new-var-init` below states the meaning of evaluating `ref E` is: the expression `E` evaluates to a value, a new cell is allocated, the evaluated value is stored in the cell, and then the allocated cell is returned as a variable.

```
module Exp/new-var-init
  imports Exp Variables/simple
  exports
    context-free syntax
      "ref" Exp -> Exp
      Simple-Variable -> Value
  equations
    [28] evaluate [[ ref E ]] =
         evaluate [[E]] then create
         then give simple-variable(the cell)
```

Equation `[29]` in the module `Exp/var-val` below indicates that the meaning of evaluating `! E` is: the expression `E` evaluates to a variable and the value stored in the variable is returned. Notice that this module would remain unchanged if compound variables are introduced, provided that the definition of `dereference` is extended appropriately (which can be done by adding new modules, leaving those given above unchanged).

```
module Exp/var-val
  imports Exp Variables
  exports
    context-free syntax
      "!" Exp -> Exp
      Variable -> Value
  equations
    [29] evaluate [[ ! E ]] =
         evaluate [[E]] then dereference the variable
```

Equation [30] in the module Exp/assign below defines the meaning of evaluating E1 := E2 as: an expression E1 evaluates to a variable and an expression E2 evaluates to a value; then the value is assigned to the variable. As with dereferencing, the use of assign allows the modules to remain unchanged if compound variables are introduced.

```
module Exp/assign
  imports Exp Variables
  exports
    context-free syntax
      Exp ":=" Exp -> Exp
      Variable -> Value
  equations
    [30] evaluate [[ E1 := E2 ]] =
         ( evaluate [[E1]] and evaluate [[E2]] )
         then assign (the variable#1, the value#2)
```

Note that the values are now extended to include variables.

Finally, the semantic equations module for expression sequencing is defined as follows:

```
module Exp/seq
  imports Exp
  exports
    context-free syntax
      Exp ";" Exp -> Exp
  equations
    [31] evaluate [[ E1 ; E2 ]] =
         evaluate [[E1]] then skip then evaluate [[E2]]
```

The action combinator then in Equation [31] above forces its sub-actions to be performed sequentially. The basic action skip discards the value given by evaluate [[E1]].

In this section, we have built modules for various language constructs: expression bindings, expression blocks, expression parameters and expressions with effects. One could also define declaration bindings, declaration blocks, and declaration parameters in a similar fashion, but the presentation would not illustrate any new points of interest, so we do not bother with that here.

5 Combining Modules

The modules developed in the previous sections can be combined to become complete programming-language modules, as indicated by the module

`ExpressionLanguage` in Section 3. Combining modules can be achieved simply by importing them together into another module, assuming that the symbols they share correspond to common features. This is because the actions in the semantic equations remain well-formed (and meaningful), regardless of how rich the denotations of component constructs become (e.g. when purely functional expressions are enriched with side-effects). However, we need to consider what should happen when the modules to be combined specify different sort inclusions involving the same sorts, or different semantic equations for the same syntactic constructs.

The only problem with sort inclusions is when both `S1 -> S2` and `S2 -> S1` occur together in a combined module (directly or indirectly): they imply that the sorts are identical, including exactly the same values. In ASF+SDF, it appears that such mutual inclusions do not affect parsing, but one might alternatively make the sorts into so-called aliases.

When there are two different semantic equations for the same syntactic construct in the modules to be combined, we say that there is a *conflict*. In fact, some combinations of the modules specified in Section 4 do result in conflicts. Such conflicts may be resolved automatically by unifying the actions in conflicting semantic equations, using `(tentatively A1) otherwise A2` to form a choice between them. To avoid patently undesirable combinations, however, we require the resulting action to be equivalent to the combination in the opposite order. [13]

For instance, let us try to combine the two modules `Exp/const-val` and `Exp/fun-nullary`. Then we have a conflict between the following two equations:

```
  evaluate [[ I ]] = give the value bound to token [[I]]
  evaluate [[ I ]] = enact action(the function bound to token [[I]])
```

The two actions can be unified provided that the semantic entities of sort `value` do not include any of sort `function`:

```
  evaluate [[ I ]] = ( tentatively
                         give the value bound to token [[I]] )
                     otherwise
                       enact action(the function bound to token [[I]])
```

where the action combination `(tentatively A1) otherwise A2` performs `A1`, and skips `A2` if `A1` terminates normally, but performs `A2` (with the same data

[13] Action equivalence is undecidable, but we may use any safe decidable approximation to it.

as `A1`) in the case that `A1` terminates exceptionally. [14]

The action in the new equation above means that if the datum bound to `I` is of sort `value`, then the expression evaluation must simply give that value; if the datum is of sort `function`, then the evaluation must enact the action; and if it is neither a value nor a function, the evaluation must terminate exceptionally. Since the given datum cannot be simultaneously of sort `value` and of sort `function`, the order of combination is insignificant, as required.

Such unification of actions permits the modular composition of the languages that, for example, use the same syntax for applying parameterless functions and for referring to ordinary constant values, provided that parameterless functions themselves are not regarded as values. Note, however, that if modules do need to use values of sort `function` in representing expressible values, the function abstractions there should always be embedded into other sorts by use of distinct constructor functions, to allow subsequent combination with modules such as `Exp/fun-nullary`. Safest of all is to follow the usual practice in action-semantic descriptions, and always embed entities such as function abstractions in distinct abstraction sorts when using them as (expressible, bindable, or storable) values. For example, a constructor for parameterized functions could embed the action in an abstraction sort `Function`, which could then be included in `Value`, whereas a sort `Thunk` embedding the values of parameterless expression abstractions would be excluded from `Value`.

It appears that `Dec/fun-nullary` and `Dec/fun-nullary-dynamic` cannot be unified since, as the following equation shows, the order of combination is significant:

```
declare [[ fun I = E ]] =
  ( tentatively
    give binding (token [[I]],
                  nullary-function closure (evaluate [[E]])) )
  otherwise
    give binding (token [[I]],
                  nullary-function (evaluate [[E]]))
```

When two modules cannot be unified, we say that they are *inconsistent*.

Let us look at some modules with no conflict. A first-order lazy functional language supporting static scoping is defined by including modules as follows:

```
module FirstOrderLazyFunctionalLanguage
  imports ExpressionLanguage
          Dec/fun-nullary       Exp/fun-nullary
          Dec/seq-indep         Exp/let
```

[14] `tentatively A` fails when `A` terminates exceptionally with no data.

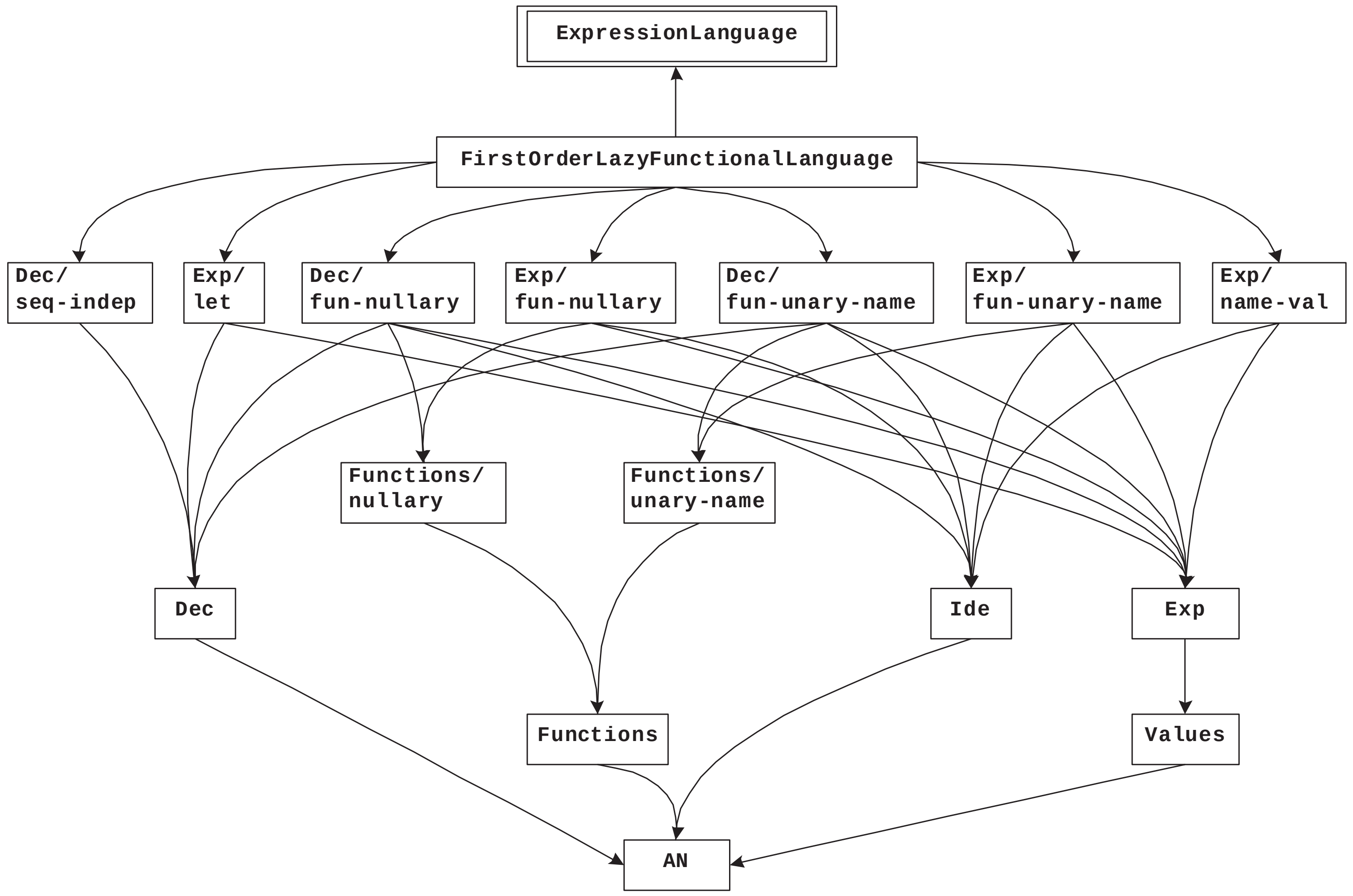

Fig. 2. The module dependency graph for FirstOrderLazyFunctionalLanguage

```
          Dec/fun-unary-name    Exp/fun-unary-name
          Exp/name-val
```

The dependency relationships among imported modules in `FirstOrderLazyFunctionalLanguage` are shown in Figure 2. Since there are no conflicting semantic equations in the combined modules above, we say that the modules are safely combined. Notice that although `Exp/fun-nullary` imports only `Exp`, which does not itself specify the syntax or semantics of any particular expression constructs, the effect of the combination is just as if all the modules for expression constructs had also been imported.

As an another example, let us look at the module for a first-order eager functional language with effects:

```
module FirstOrderEagerFunctionalLanguageWithEffects
  imports ExpressionLanguage
          Dec/val               Exp/const-val
          Dec/seq               Exp/let
          Dec/fun-unary-value   Exp/fun-unary-value
          Exp/new-var-init      Exp/var-val
          Exp/assign            Exp/seq
  exports
    context-free syntax
      Simple-Variable -> Bindable
      Int -> Storable
```

The dependency relationships among imported modules in `FirstOrderEagerFunctionalLanguageWithEffects` are shown in Figure 3. The combination in the above module presents no conflicting semantic equations. However, since the sorts `Bindable` and `Storable` have remained unspecified in the imported modules, they need to be fully specified in the combined module in order for the language to be complete. This means that the decision about which sorts of values should be bindable and/or storable has been left open until we form a complete language. In fact, we might choose them minimally, to be no more than required in the above combined module; at the other extreme, we might choose them as follows:

```
      Simple-Variable | Int | Bool -> Bindable
      Simple-Variable | Int | Bool -> Storable
```

Note that in the meanwhile `Value` in the combined module must include at least the following sorts:

```
      Simple-Variable | Int | Bool -> Value
```

The observant reader may have noticed that the combination of the functional `ExpressionLanguage` module with the modules, `Exp/new-var-init`,

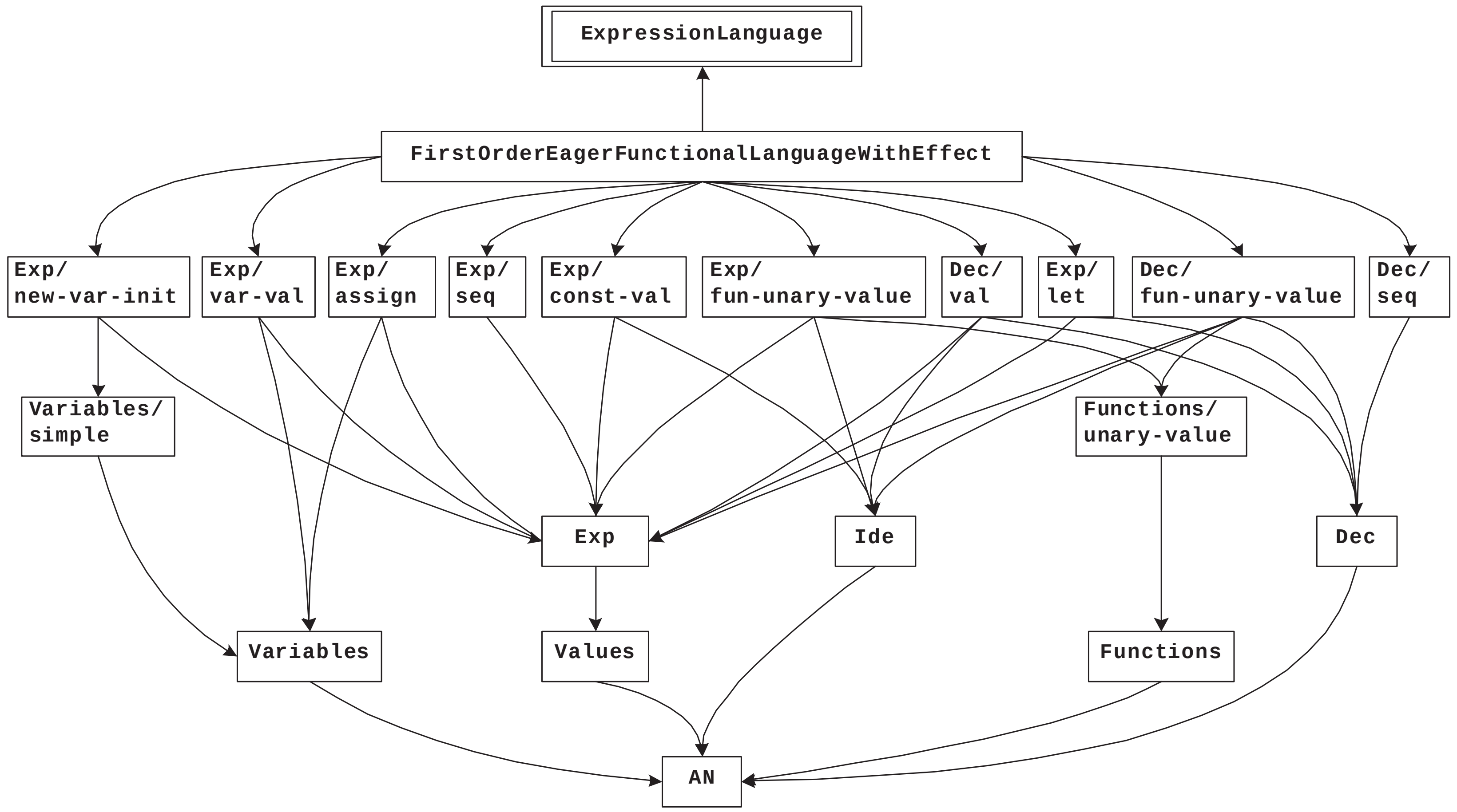

Fig. 3. The module dependency graph for `FirstOrderEagerFunctionalLanguageWithEffects`

`Exp/var-val` and `Exp/assign`, has undermined the determinism of the described language: assignments occurring in sub-expressions of the same arithmetic expression may be interleaved in any order, and this clearly may lead to different possible values of expressions. Such nondeterminism is quite different from that arising when unifying actions, since it does not involve a choice between performing different actions, and merely reflects the possibility of observing an internal nondeterminism that was already present in the computational semantics of actions representing expression evaluation; its appearance should therefore not invalidate module combination. Incidentally, even if one does regard nondeterminism due to interleaving as undesirable, and perhaps wishes to prohibit just those programs whose outcome may depend on which interleaving is chosen, the nondeterministic semantics is still needed, in order to distinguish the prohibited programs from the others.

We have illustrated our approach by combining action-semantics modules based on expression-based constructs taken from functional programming languages. It is, of course, equally possible to develop imperative languages by defining a language module for conventional imperative constructs, with a new syntax domain `Command` including assignment, loop and sequencing constructs, and then defining and combining modules as we have done in this paper.

With the modular structure previously adopted in action semantics, each module typically defines a semantic function on an entire syntactic sort. Thus a conventional action-semantic description of the first-order language with effects would have essentially just two modules defining semantic functions: one for evaluating expressions, the other for declaring definitions. The inherent modularity of action notation allows reuse of individual semantic equations in other descriptions, but it is unlikely that such large modules would ever be reused *in toto*, merely by referring to them.

In contrast, the much finer modular structure proposed in this paper should encourage the direct reuse of modules from one action-semantic description in later ones. Note however that although the direct use of ASF+SDF as shown in this paper is quite convenient and perspicuous, it would probably be advantageous to generate the desired ASF+SDF modules from a more concise meta-notation (e.g. eliminating lengthy keywords such as `context-free syntax`), as in the ASD Tools developed by van Deursen and Mosses [16].

Finally, let us note that we have here focused on the modules defining semantic equations. In large-scale action-semantic descriptions, also the specification of *semantic entities* (providing data types and action abbreviations for higher-level concepts, such as compound variables and communication protocols) has an interesting modular structure. When the modules defining semantic equations are structured so as to group related constructs together, the modules defining semantic entities may be more easily localized, making it apparent that they are only needed in connection with the semantics of particular language constructs. However, an appropriate visualization of the import relationship between modules may be as effective as (and more flexible than) an explicit indication of the intended grouping of modules for semantic entities with those for semantic equations.

6 Conclusion

We have demonstrated how to use action semantics to define and combine language modules. The good modularity and extensibility of action semantics help us systematically develop programming languages. Particularly in the presence of the language modules for similar languages, new language modules can be defined with only a few modifications when using the fine-grained modular structure proposed here.

The ideas developed in this article can be adopted to guide the design of domain-specific languages and/or rapidly prototyped special-purpose languages. When one designs such a language, one can analyze its problem domain, design a core language, and then gradually build up language features to it according to some rational design principles, such as the Landin-Tennent principles.

It should also be possible to reuse modules in the style of those shown in this paper when describing previously-designed languages; in a future paper, we intend to report on a case-study involving the action-semantic description of an existing domain-specific language.

Good *tool support* [19] is clearly essential for checking the well-formedness and consistency of modules (and ultimately, for generating prototype implementations from them). It appears that use of ASF+SDF and the Meta-Environment [15] is a good basis for such tool support. The implementation of tools should be facilitated by our adoption of the much-simplified version of action notation that was proposed by Lassen, Mosses, and Watt at the Action Semantics Workshop [13].

Other semantic frameworks that aim to provide a high degree of modularity have been developed. For example, Moggi [20] has proposed the use of monads and monad transformers in denotational semantics. Cartwright and Felleisen [21] have proposed the use of an operationally-motivated style of denotational semantics, in the interests of modularity; it uses auxiliary notation similar to that of Moggi, but appears to be not so general (nondeterminism and concurrency are excluded). Liang and Hudak have implemented Moggi's monad transformers [20] in their *modular monadic semantics* framework [22].

Wansbrough and Hamer have used the modular monadic framework to give a modular monadic semantics of action notation, and called it *modular monadic action semantics* [23]. In response to such work, Mosses has developed *Modular SOS* [24] which provides significantly greater modularity in SOS, and he has redefined action notation in Modular SOS, improving dramatically the modularity of the definition [6]. The Abstract State Machine (ASM) approach [25] also provides modularity for operational semantics, through the use of a particularly neat notation for updating and accessing components of a global configuration. Although the ASM approach generally lacks the compositionality of SOS-based frameworks, the Montages presentation of ASM [26] provides a separate module for each syntactic construct, specifying how a local ASM for that construct is plugged into the global ASM, together with the firing rules associated with the local ASM.

It appears that none of the above frameworks supports the definition and combination of language modules to the same extent as action semantics does. For instance, the monadic approach to denotational semantics would require a particular composition of monad transformers to be specified for each module; not only would this be repetitive, but also it is unclear how to combine independently-specified compositions. Similarly, Modular SOS requires the specification of compositions of so-called label transformers—although there it would be quite straightforward to combine the transformers (since their order of composition is insignificant, and their symbolic indices allow duplication to be avoided). The Montages approach to ASM inspired our reconsideration of the modular structure of action semantics, and it also supports combination of modules for individual constructs into a complete language. However, it seems that Montages modules from one language description cannot in general be reused without reformulation in other descriptions, since the notation used in the underlying ASM formalism may well vary.

In a recent paper [27] Menezes and Moura propose a novel "component-based" style of action semantics. Although their proposal has some interesting aspects, it requires the semantic equations to be formulated in a style that is significantly different from that previously used in action semantics. We believe that our proposal in the present paper, which is based on a simple rearrangement of the pieces already found in previous action-semantic descriptions without changes to the semantic equations themselves, provides an easier route towards easier composition of programming languages by combing action-semantics modules.

There is no silver bullet in designing a good programming language. However, we hope a design tool such as the one proposed in this article may be used to enhance the quality of language-design activity.

Appendix A: Inclusions Between Sorts of Semantic Entities

`ExpressionLanguage`:

$$\begin{tikzcd}
 & & \text{Datum} \\
 & \text{Value} \arrow[ur] & \\
\text{Int} \arrow[ur] & & \text{LitValue} \arrow[uu] \arrow[ul] \\
\text{Nat} \arrow[u] \arrow[urr] & \text{Bool} \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

`FirstOrderLazyFunctionalLanguage`:

$$\begin{tikzcd}
 & & \text{Datum} & & \\
 & \text{Value} \arrow[ur] & & \text{Bindable} \arrow[ul] & \\
\text{Int} \arrow[ur] & & \text{LitValue} \arrow[uu] \arrow[ul] & \text{Function} \arrow[u] & \text{Action} \arrow[ul] \\
\text{Nat} \arrow[u] \arrow[urr] & \text{Bool} \arrow[uu] \arrow[ur] & & \text{Unary-Name-Function} \arrow[u] &
\end{tikzcd}$$

`FirstOrderEagerFunctionalLanguageWithEffects`:

$$\begin{tikzcd}
 & & & \text{Datum} & & & \\
\text{Storable} \arrow[urrr] & & \text{Value} \arrow[ur] & & & & \text{Bindable} \arrow[ulll] \\
 & \text{Int} \arrow[ul] \arrow[ur] & & \text{LitValue} \arrow[uu] \arrow[ul] & & \text{Variable} \arrow[uull] \arrow[ulll] & \text{Function} \arrow[u] \\
 & \text{Nat} \arrow[u] \arrow[urr] & \text{Bool} \arrow[uu] \arrow[ur] & & & \text{Simple-Variable} \arrow[u] \arrow[uulll] \arrow[uur] & \text{Unary-Value-Function} \arrow[u]
\end{tikzcd}$$

Appendix B: List of Action Notation Symbols

The following list includes only those symbols used in the examples given in this paper (a list of all symbols of AN-2 would be roughly twice as long).

```
sorts

  Action Data Datum DataSort
  Token Bindable Bindings
  Cell Storable Nat Int Bool
  Yielder DataSort DataOp Enquirer

context-free syntax

  Action                        -> Datum  %% actions as data
  Token                         -> Datum  %% used in bindings
  Bindable                      -> Datum  %% used in bindings
  Bindings                      -> Datum  %% binding maps
  Cell                          -> Datum  %% primitive storage
  Storable                      -> Datum  %% in cells
  Int                           -> Datum  %% integers
  Bool                          -> Datum  %% truth-values

  Datum                         -> Data   %% 1-tuple
  "(" {Data ","}* ")"           -> Data   %% tuples

  Nat                           -> Int    %% integers

  Action "and" Action           -> Action %% tupling
  Action "and" "then" Action    -> Action %% sequencing
  Action "then" Action          -> Action %% composition
  "tentatively" Action          -> Action %% may fail
  Action "otherwise" Action     -> Action %% recovery
  "select"
    "(" Action "or" Action ")"  -> Action %% alternatives
  Action "hence" Action         -> Action %% scope
  Action "before" Action        -> Action %% scope sequencing
  "furthermore" Action          -> Action %% scope extension

  "skip"                        -> Action %% null
  "enact"                       -> Action %% action performance
  "create"                      -> Action %% gives storage cell
  "update"                      -> Action %% changes cell contents
  "inspect"                     -> Action %% reads cell contents

  "provide" Data                -> Action %% giving fixed data
  "give" Yielder              -> Action %% giving variable data
  "given" Yielder             -> Action %% matching given data
  "when" Enquirer             -> Action %% testing predicate
  Action Yielder              -> Action %% give yielded data
                                        %% then perform action

  Data                        -> Yielder %% constant
  DataOp                      -> Yielder %% application
  DataOp Yielder              -> Yielder %% composition
  "(" {Yielder ","}* ")"      -> Yielder %% tupling

  "bound" "to" Yielder        -> Yielder %% current binding
  "closure" Yielder           -> Yielder %% freeze yielded action

  "the" DataSort              -> DataOp %% project to subsort
  "#" Nat                     -> DataOp %% select component
  "binding"                   -> DataOp %% construct binding
  "+" | "-" | "*"             -> DataOp %% integer operations

  action | yielder |
  data | datum |
  token | bindable | bindings |
  cell | storable |
  nat | int | bool            -> DataSort %% subsort projectors

  Yielder "=" Yielder         -> Enquirer %% equality test

%% DataOp includes all data operations (and action combinators).
%% Applications of data operations may be written infix.
%% Prefix applications have higher priority than infix applications.
%% Infix applications are left associative.
%% Conventional notation for Nat, Int, and Bool is included.
```

Acknowledgements

David Schmidt provided the idea of defining and combining language-definition modules; Olivier Danvy encouraged us to write this paper and made numerous comments on the contents and structure of previous drafts; Tae-Hyung Choi and Soo-Kang Lee showed their expertise in Visio[15] by drawing figures in the paper; and the anonymous referees supplied useful comments and corrections.

[15] Visio is a trademark of Microsoft Corporation.

References

[1] C. A. R. Hoare, Hints on programming language design, in: POPL'73, Proc. 1st ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1973, also in [28].

[2] D. A. Schmidt, The Structure of Typed Programming Languages, MIT Press, 1994.

[3] P. D. Mosses, Action Semantics, no. 26 in Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1992.

[4] P. D. Mosses, Theory and practice of action semantics, in: MFCS'96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland, Vol. 1113 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 37–61.

[5] P. D. Mosses, A tutorial on action semantics, tutorial notes for FME'94 (Formal Methods Europe, Barcelona, 1994) and FME'96 (Formal Methods Europe, Oxford, 1996), also available from the author at `http://www.brics.dk/Projects/AS/` (Mar. 1996).

[6] P. D. Mosses, A modular SOS for action notation, Tech. Rep. BRICS RS-99-56, Dept. of Computer Science, University of Aarhus (Dec. 1999).

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

[8] P. D. Mosses, D. A. Watt, Pascal: Action semantics, draft, Version 0.6, Available from the authors at `http://www.brics.dk/Projects/AS/` (Mar. 1993).

[9] D. A. Watt, Standard ML action semantics, draft, Version 0.5, Available from the author at `http://www.brics.dk/Projects/AS/` (May 1997).

[10] D. A. Watt, The static and dynamic semantics of Standard ML, in: AS'99, 2nd International Workshop on Action Semantics (ed. Mosses, P. D., and Watt, D. A.), BRICS NS-99-3, Dept. of Computer Science, University of Aarhus, 1999, pp. 155–172.

[11] R. D. Tennent, Language design methods based on semantic principles, Acta Informatica 8 (1977) 97–112.

[12] G. D. Plotkin, A structural approach to operational semantics, Lecture Notes DAIMI FN-19, Dept. of Computer Science, University of Aarhus (1981).

[13] S. B. Lassen, P. D. Mosses, D. A. Watt, An introduction to AN-2, the proposed new version of Action Notation, in: AS 2000, no. NS-00-6 in Notes Series, BRICS, Dept. of Computer Science, Univ. of Aarhus, 2000, pp. 19–36.

[14] 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, J. Visser, The ASF+SDF Meta Environment: a component-based

language development environment, in: R. Wilhelm (Ed.), CC'2001, Compiler Construction, Vol. 2027 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 365–370.

[15] A. van Deursen, J. Heering, P. Klint (Eds.), Language Prototyping, Vol. 5 of AMAST Series in Computing, World Scientific, 1996.

[16] A. van Deursen, P. D. Mosses, ASD: The action semantic description tools, in: AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich, Vol. 1101 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 579–582.

[17] D. A. Schmidt, Denotational Semantics: A Methodology for Language Development, Allyn and Bacon, 1986.

[18] J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1977.

[19] J. Heering, P. Klint, Semantics of programming languages: A tool-oriented approach, ACM SIGPLAN Notices 35 (3) (2000) 39–48.

[20] E. Moggi, An abstract view of programming languages, Tech. Rep. ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh (1990).

[21] R. Cartwright, M. Felleisen, Extensible denotational language specifications, in: M. Hagiya, J. C. Mitchell (Eds.), TACS'94, Symposium on Theoretical Aspects of Computer Software, Sendai, Japan, Vol. 789 of Lecture Notes in Computer Science, Springer-Verlag, 1994, pp. 244–272.

[22] S. Liang, P. Hudak, Modular denotational semantics for compiler construction, in: ESOP'96, Proc. 6th European Symp. on Programming, Linköping, Vol. 1058 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 219–234.

[23] K. Wansbrough, J. Hamer, A modular monadic action semantics, in: Proc. Conf. on Domain-Specific Languages, The USENIX Association, 1997, pp. 157–170.

[24] P. D. Mosses, Foundations of Modular SOS (extended abstract), in: MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska Poreba, Poland, Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 70–80, full version published as BRICS RS-99-54, Dept. of Computer Science, University of Aarhus, 1999.

[25] Y. Gurevich, Evolving algebras 1993: Lipari guide, in: E. Börger (Ed.), Specification and Validation Methods, Oxford University Press, 1995.

[26] P. Kutter, A. Pierantonio, Montages: Specifications of realistic programming languages, Journal of Universal Computer Science 3 (5) (1997) 416–442.

[27] L. Menezes, H. Moura, Component-based action semantics: A new approach for programming language specification, in: SBLP 2001, V Brazilian Symposium on Programming Languages, 2001.

[28] C. A. R. Hoare, C. B. Jones, Essays in Computing Science, Prentice-Hall, 1989.

Recent BRICS Report Series Publications

RS-03-53 Kyung-Goo Doh and Peter D. Mosses. *Composing Programming Languages by Combining Action-Semantics Modules*. December 2003. 39 pp. Appears in *Science of Computer Programming*, 47(1):2–36, 2003.

RS-03-52 Peter D. Mosses. *Pragmatics of Modular SOS*. December 2003. 22 pp. Invited paper, published in Kirchner and Ringeissen, editors, *Algebraic Methodology and Software Technology: 9th International Conference*, AMAST '02 Proceedings, LNCS 2422, 2002, pages 21–40.

RS-03-51 Ulrich Kohlenbach and Branimir Lambov. *Bounds on Iterations of Asymptotically Quasi-Nonexpansive Mappings*. December 2003. 24 pp.

RS-03-50 Branimir Lambov. *A Two-Layer Approach to the Computability and Complexity of Real Numbers*. December 2003. 16 pp.

RS-03-49 Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. *Online On-the-Fly Testing of Real-time Systems*. December 2003. 14 pp.

RS-03-48 Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, and Andrzej Wasowski. *Danfoss EKC Trial Project Deliverables*. December 2003. 53 pp.

RS-03-47 Hans Hüttel and Jiří Srba. *Recursive Ping-Pong Protocols*. December 2003. To appear in the proceedings of 2004 IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues in the Theory of Security (WITS'04).

RS-03-46 Philipp Gerhardy. *The Role of Quantifier Alternations in Cut Elimination*. December 2003. 10 pp. Extends paper appearing in Baaz and Makowsky, editors, *European Association for Computer Science Logic: 17th International Workshop*, CSL '03 Proceedings, LNCS 2803, 2003, pages 212-225.

RS-03-45 Peter Bro Miltersen, Jaikumar Radhakrishnan, and Ingo Wegener. *On converting CNF to DNF*. December 2003. 11 pp. A preliminary version appeared in Rovan and Vojtás, editors, *Mathematical Foundations of Computer Science: 28th International Symposium*, MFCS '03 Proceedings, LNCS 2747, 2003, pages 612–621.