Submitted to the *Proceedings of the Eighth Workshop on Mathematical Foundations of Progmmming Semantics (MFPS VIII), Oxford, England, April 1992,* which is to be published as a special issue of *Theoretical Computer Science,* 1993.

If the paper gets accepted, it may appear in the *Proceedings* with the same title but in a revised form, and subsequent citations should then refer only to the *Proceedings*.

The Operational Semantics of Action Notation

Peter D. Mosses*

September 1992
Submitted for the Proceedings of MFPS VIII

Abstract

Action notating is used in the action semantics framwork, for specifying actions representing program behaviour. It is defies by a structural operational semantics together with a bisimulation-bsed equivalence that satisfies some simply algebraic laws.

1 Introduction

Action notation is used in *action semantics,* a recently-developed framework for formal semantics [11, 15]. The primary aim of action semantics is to allow *useful* semantic decriptions of *realistic* programming languages.

Action semantics combines formality with many good pragmatic features. Regarding comprehensibility and accessibility, for instance, action semantic descriptions compete with informal language descriptions. Action semantic descriptions scale up smoothly from small example languages to full-blown practical languages. The addition of new constructs to a described language does not require reformulation of the already-given description. An action semantic description of one language can make widespread reuse of that of aother, related language. All these pragmatic features are highly desirable. Action sematics is, however, so far the *only* semantic framework that enjoys them!

*Computer Science Department, Aarhus University, Ny Munkegade Bldg. 540, DK-8000 Aarhus C, Denmark; E-mail: pdmosses@daimi.au.dk

Action semantics is *compositional*, like denotations semantics [9]. The main difference between action semantics and denotations semantics concerns the universe of semantic entities: action semantics uses entities called *actions,* rather than the higher-order functions used with denotational semantics. Actions are inherently more operational than functions: when *performed,* actions process information *gradually.*

Primitive actions, and the various ways of combining actions, correspond to fundamental concepts of information professing. Action semantis provides a particular notation for expressing actions. The symbols of action notation are suggestive words, rather than cryptic signs, which makes it possible to get a broad impression of an action semantic description from a superficial reading, even without previous experience of action semantics. The action *combinators,* a notable feature of action notation, obey desirable algebraic laws that can be used for reasoning about semantic equivalence.

See [11] for a comprehensive exposition of action semantics, which also illustrates its claimed pragmatic qualities.

Here, we focus our attention on the formal definition of action notation. The definition consists of a structural operational semantics [14, 4, 1], together with a bisimulation equivalence. A novel feature of the definition is the use of *Horn clauses* instead of inference rules. Moreover, we exploit a recently-developed *unified* meta-notation, based on the framework of unified algebras [7, 8] allowing functions that return proper *sorts* when applied to individuals; this lets us represent transition relations as functions, with nonde-terministic choices between conngurations being represented as proper sorts. These new techniques allow us to deal with structural operational semantics within a purely algebraic framework.

It is worth pointing out that the structural operational semantics of action notation induces an operational semantics for all languages described using action semantics. However, the induced semantics is not really structural in the usual sense, since configurations involve action terms rather than program syntax. Note that a structural operational semantics for a programming language usually involves repetitious patterns of rules for transitions, for instance determining a sequential order of execution of the components of various phrases; an action semantics for the language uses a single combinator to express the fundamental concept of sequencing, and the structural operational semantics of the combinator specifies the corresponding pattern of transitions, once and for all. Thus action semantics can be regarded as a technique for factorization of a conventional structural operational semantics.

Why isn't action notation defined denotationally? That would have the advantage of inducing denotational models for all languages with action semantic descriptions, as well as making domain theory available for reasoning about actions. The difficulty is that the full action notation involves concepts, such as concurrency and unbounded nondeterminism, whose available denotational models are not only very intricate but also not fully abstract with respect to the intended operational semantics of actions. Such a denotational 'model' would not satisfy all the desired algebraic laws. (See [10] for an experiment with defining action notation as auxiliary notation in denotational semantics.)

On the other hand, although our combination of structural operational semantics and bisimulation does verify the essential algebraic laws, this does not provide a sufficiently strong action theory for reasoning about nontrivial progam equivalence. It is currently unclear how to develop a stronger action theory, to avoid the need for direct and tedious a reasoning at the operational level.

The plan of this paper is as follows. Section 2 gives an informal explanation of the concept of action the use of the action used in action semantics. It also provides a small example that illustrates subset of action notation considered here. Section 3 defines the structural operational semantics of our action notation. Section 4 defines a bisimulation equivalence on actions. An Appendix summarizes the unified meta-notation used throughout. No previous exposure to action semantics or unified algebraic specifications is assumed, although a general familiarity with semantic descriptions and algebraic specifications may be helpful.

2 Action Notation

Just as the lambda-notation is used in denotational semantics for specifying functions [9], so our action notation is used in action semantics for specifying *actions* [11]. Action notation includes also notation for *data* and for auxiliary entities called *yielders*.

Actions are essentially dynamic, *computational* entities. The *performance* of an action directly represents information processing behavior and reflects the gradual, step-wise nature of computation. Items of data are, in contrast, essentially static, *mathematical* entities, representing pieces of information, e.g., particular numbers. (Of course actions are 'mathematical' too, in the sense that they are abstract, formally-defined entities, analogous to abstract machines defined in automata theory.) A yielder represents an *unevaluated* item of data, whose value depends on the *current information,* i.e., the previously-computed and input values that are available to the performance of the enclosing action. For example, a yielder might sways evaluate to the datum currently stored in a particular cell, which could change during the performance of an action.

2.1 Actions

A performance of an action, which may be part of an enclosing action, either:

- *completes*, corresponding to normal termination (the performance of the enclosing action proceeds normally); or
- *escapes*, corresponding to exceptional termination (parts of the enclosing action are skipped until the escape is trapped); or
- *fails*, corresponding to abandoning the performance of an action (the enclosing action performs an alternative action, if there is one, otherwise it fails too); or
- *diverges*, corresponing to nontermination (the enclosing action also diverges).

Actions can be used to represent the semantics of programs: action performances correspond to possible program behaviors. Furthermore, actions can represent the (perhaps indirect) contribution that *parts* of programs, such as statements and expressions, make to the semantics of entire programs.

An action may be nondeterministic, having different possible performances for the see initial information. Nondeterminism represents implementation-dependence, where the behaviour of a program (or the contribution of a part of it) may vary between different implementations—or even between different instants of time on the same implementation. Note that nondeterminism does not imply actual randomness: each implementation of a nondeterministic behaviour may be absolutely deterministic.

The information processed by action performance may be classified according to how far it tends to be propagated, as follows:

- *transient:* tuples of data, corresponding to intermediate results;
- *scoped:* bindings of tokens to data, corresponding to symbol tables;
- *stable:* data stored in cells, corresponding to the values assigned to variables;
- *permanent:* data communicated between distributed actions.

Transient information is made available to an action for immediate use. Scoped information, in contrast, may generally be referred to throughout an entire action, although it may also be hidden temporarily. Stable information can be changed, but not hidden, in the action, and it persists until explicitly destroyed. Permanent information cannot even be changed, merely augmented.

When an action is performed, transient information is given only on completion or escape, and scoped information is produced only on completion. In contrast, changes to stable information and extensions to permanent information are made *during* action performance, and are unaffected by subsequent divergence, failure, or escape.

The different kinds of information give rise to so-called *facets* of actions, focusing on the processing of at most one kind of information at a time:

- the *basic* facet, processing independently of information (control flows);
- the *functional* facet, processing transient information (actions are *given* and *give* data);
- the *declarative* facet, processing scoped information (actions *receive* and *produce* bindings);
- the *imperative* facet, processing stable information (actions *reserve* and *unreserve* cells of storage, and *change* the data stored in cells); and
- the *communicative* facet, processing permanent information (actions *send* messages, *receive* messages in buffers, and offer *contracts* to *agents*).

These facets of actions are independent. For instance, changing the data stored in a cell—or even unreserving the cell—does not affect any bindings. There are, however, some *directive* actions, which process a mixture of scoped and stable information, so as to provide finite representations of self-referential bindings. There are also some *hybrid* primitive actions and combinators, which involve more than one kind of information at once, such as an action that both reserves a cell of storage and gives it as transient data. In this paper, for simplicity, we ignore the communicative and directive facets of actions; we also ignore escapes (exceptional termination).

The notation for specifying actions consists of action *primitives*, which may involve yielders, and action *combinators*, which operate on one or two *subactions*. Action notation provides also some notation for specifying *sorts* of actions.

2.2 Yielders

Yielders are entities that can be *evaluated* to yield data during action performance. The data yielded may depend on the current information, i.e., the given transients, the received bindings, and the current state of the storage. In fact action notation provides primitive yielders that evaluate to compound data (tuples, maps, lists) representing entire slices of the current information, such as the current state of storage. Evaluation cannot affect the current information.

Compound yielders can be formed by the application of data operations to yielders. The data yielded by evaluating a compound yielder are the result of applying the operation to the data yielded by evaluating the operands, For instance, one can form the sum of two number yielders. Items of data are a special case of data yielders, and always yield themselves when evaluated.

2.3 Data

The information processed by actions consists of items of *data*, organized in stuctures that give access to the individual items. Data can include various familiar mathematical entities, such as truth-values, numbers, characters, stings, lists, sets, and maps. It can also include entities such as tokens and cells, used for accessing other items. Actions themselves are not data, but they can be incorporated in so-called *abstractions*, which are data, and subsequently *enacted* back into actions. (Abstraction and enaction are a specie cue of socalled *reification* and *reflection.*) New kinds of data can be introduced *ad hoc*, for representing special pieces of information.

2.4 Notation

Consider the example of action notation given in Box 2.1. It illustrate the use of the main primitive actions and combinators. The intended operational interpretation of the specified action corresponds to the semantic of the statement

$$\mathsf{FOR\ i\ IN\ [1..10]\ DO\ s\ :=\ s + i}$$

which is supposed to have the effect of adding up the indicated values of i in the variable s, in an unspecified order.

The symbols used in action notation are somewhat more verbose than is usual in Semitic notation. Nevertheless, they are *entirely formal!* We glow infix and 'mixfix' symbols, as well as prefix; infix symbols have weaker precedence than prefix symbols. Vertical lines group the terms on their right, providing a clear indication of overall term structure. (In fact quite a few of the lines in the example could be eliminated without introducing ambiguity, but then the term structure might be less obvious to readers who haven't seen action notation before.)

This is not the place for a full explanation of all the details of action notation; the interested reader is referred to [11], where also the overall design of action notation is motivated, and its use in action semantics is illustrated. The following comments are merely intended to give a rough grasp of the main primitive actions and combinators used in the example, prior to the presentation of their formal operational semantics in the next section. We consider the various kinds of information processing in turn.

Basic Control Flow

The basic combination A_1 and then A_2 combines the actions A_1, A_2 into a compound action that represents their normal, left-to-right sequencing, performing A_2 only when A_1 completes. complete is the unit for _ and then _.

The action A_1 or A_2 represents implementation-dependent choice between alternative actions, although if A_1, A_2 are such that one or the other of them is always bound to fail, the choice is deterministic. A falure causes the alternative currently being performed to be abandoned and, if possible, some other alternative to be performed instead, i.e., *back-tracking*.

```
| give 1
then
| unfolding
| | | | check not (the given natural is greater than 10)
| | | and then
| | | | | store the sum of (it, the natural stored in the cell bound to "s")
| | | | |          in the cell bound to "s")
| | | | and
| | | | | | give the sucessor of it
| | | | | then
| | | | | | unfold
| | or
| | | | check (the given natural is greater than 10)
| | | and then
| | | | complete
```

Box 2.1 An example of action notation

The action A_1 and A_2 represents implementation-dependent order of performance of the indivisible subactions of A_1, A_2. When these subactions cannot 'interfere' with each other, it indicates that their order of performance is simply irrelevant.

A performance of A_1 and A_2 arbitrarily interleaves the steps of performances of A_1, A_2 until both have completed, or until one of them escapes or fails. When the performance diverges, it may be 'unfair', for instance letting A_1 make infinitely-many steps but only finitely-many of A_2.

unfolding A performs A but whenever it reaches the dummy action unfold, it performs A instead. One may prefer to regard unfolding A as an abbreviation for an action, generally infinite, formed by continually substituting A for unfold in A. (To avoid syntactic 'singularities' in action terms such as unfolding unfold, substitute complete and then A instead of just A.) The action unfolding A is mostly used in the semantics of iterative constructs, with unfold occurring exactly once in A, but it can also be used with several occurrences of unfold.

Transient Information Processing

The primitive action give Y completes, giving the data yielded by evaluating the yielder Y, provided that this is an individual; it fails when Y yields nothing. The action check Y requires Y to yield a truth-value; it completes when the value is true, otherwise it fails, without committing. It is used for guarding alternatives. For instance, (check Y and then A_1) or (check not Y and then A_2) expresses a *deterministic* choice between A_1 and A_2, depending on the condition Y.

The functional action combination A_1 then A_2 represents ordinary functional composition of A_1 and A_2: the transients given to the whole action are propagated only to A_1, the transients given by A_1 on completion are given only to A_2, and only the transients given by A_2 are given by the whole action. Regarding control flow, A_1 then A_2 specifies normal sequencing, as in A_1 and then A_2. When A_1 doesn't give any transients and A_2 doesn't refer to any given transients, A_1 then A_2 may be used interchangeably with A_1 and then A_2.

The basic action combination A_1 and A_2 passes given transients to both the subactions, and concatenates the transients given by the subactions when they both complete; similarly for A_1 and then A_2. Finally, each alternative of A_1 or A_2, when performed, is given the same transients as the combination, and of course the combination gives only the transients given by the non-failing alternative performed, if any.

Whereas the data flow in A_1 then A_2 is analogous to that in ordinary function composition $g \circ f$ (at least when the functions are strict) the data flow in A_1 and A_2 is analogous to socalled *target-tupling* of functions, sometimes written $[f, g]$ and defined by $[f, g](x) = (f(x), g(x))$.

The yielder given Y yields all the data given to its evaluation, provided that this is of the data sort Y. For instance the given truth-value (where 'the' is optional) yields true or false when the given data consists of that single individual of sort truth-value. Otherwise it yields nothing. Similarly, given $Y\#n$ yields the n'th individual component of a given tuple, for $n > 0$, provided that this component is of sort Y (not illustrated in the example). The yielder 'it' yields the same as the given datum, where datum is a sort that can be specialized to include all sorts of data items.

It is primarily the presence of A_1 then A_2 in functional action notation that causes the *transience* of transient data. This combinator does not automatically make the given transients available to A_2, so unless A_1 propagates them, they simply disappear.

Scoped Information Processing

The yielder the d bound to T evaluates to the current binding for the particular token T, provided that it is of data sort d, otherwise it yields nothing. (The primitive actions and combinators provided in action notation for producing bindings are not considered in this paper.)

Stable Information Processing

The imperative action store Y_1 in Y_2 changes the data stored in the cell yielded by Y_2 to the storable data yielded by Y_1. The cell concerned must have been previously reserved (using the primitive action revere Y) othewise the storing action fails.

The yielder the d stored in Y yields the data currently stored in the cell yielded by Y, provided that it is of the sort d. Otherwise it yields nothing.

Data Operations

Various commonly-used data types (truth-values, natural numbers, stings, storage cells) are provided by a general data notation included in action notation. All data operations extend naturally to yielders, yielding the result of applying the operation concerned to the data items yielded by evaluating the arguments. For added readability, the operations 'the' and 'of' are provided; they denote the identity function on data.

Notice that we allow data operations to be applied to entire *sorts* of data—not only to individuals. The formal basis for this is provided by the framework of unified algebras [7], which is summarized in the Appendix of this paper.

So much for an informal explanation of the illustrative subset of action notation used in the example. Next we specify the intended interpretation of the notation formally.

3 Operational Semantics

We use a variant of structural operations semantic [14, 1] to define a transition system. Sequences of transitions correspond to possible performances of actions, representing program behaviours. In Section 4 we consider the definition of action equivalence in terms of transition bisimulation.

The key idea is to use a transition *function* mapping individual configurations to arbitrary *sorts* of configurations, rather than a transition *relation* between configurations. It is notationally just as easy to specify a fiction as a relation, and by allowing proper sorts (not merely individuals) as results we can still cope with nondeterminism. Moreover, we can specify the result to be a single individual when the transition from a particular configuration is deterministic, rather than leaving determinism implicit; we can even specify directly that a configuration is blocked, using a vacuous sort such as nothing! (The formal basis for using sorts as arguments and results of operations is provided by the framework of unified algebras [7], summarized in the Appendix of this paper.)

A less significant point is that we use positive *Horn clauses* instead of inference rules. The only drawback of this seems to be that meta-proofs using induction on the length of inference become less immediate, because one has to consider the inference rules for Horn clause logic, as demonstrated in [12] (see also [13]). By considering the *initial* model of the Horn clauses we obtain the effect of demanding the *least* transition relation satisfying the corresponding inference rules.

Readers who are experienced in structural operational semantics may notice below—once they have become accustomed to the notation used here—some novel techniques that improve the modularity of our description. For example, the function simplified x applies reductions to the syntactic part of a configuration after each transition.

The structural operational semantics of the full action notation used in action semantics is given in [11, Appendix C]; it is about 12 pages long. For simplicity, we here ignore information not pertinent to the performance of actions in our illustrative subset of action notation.

Abstract Syntax

Actions	**needs:**	**Yielders, Data.**
Yielders	**needs:**	**Actions, Data.**
Data	**.**	

Configurations includes: **[11] / Data Notation,**
[11] / Action Notation / * /Data.

Actions	**needs:**	**Abstract Syntax.**
States	**needs:**	**Acting.**
Commitments	**.**	

Transitions needs: Abstract Syntax, Configurations.

Actions

Simple	**needs:**	**Yielders, Data.**
Compound	**needs:**	**Actions/Simple, Data.**
Stepping		**.**
Simplifying		**.**
Unfolding		**.**
Giving		**.**

Yielders	**needs:**	**Data.**
Data	**.**	

Box 3.1 Modules

The entire specification below is formulated in the meta-notation provided by the framework of unified algebras, summarized in the Appendix. Note especially that $x \leq y$ holds when x denotes a sort included in the sort y, whereas $x : y$ also requires x to denote an *individual* value. The operations $x|y$ and $x \,\&\, y$ provide sort union and intersection, respectively. For our use of unified algebras in this paper, sorts can be regarded as sets, with individuals corresponding to singletons.

The modular structure of the specification is given in Box 3.1 (the order in which modules are presented has no formal significance).

3.1 Abstract Syntax

The grammar below specifies the abstract syntax of a *kernel* of our subset of action notation. The nonterminal symbols of the gammar are capitalized, and the terminal symbols are quoted, to avoid confusion between notation for syntactic and semantic entities. Moreover, the brackets $[\![\ldots]\!]$ denote construction of nodes in trees.

The abstract site of notation for data is left open, so the user of action notation may add nonstandard data notation. A uniform abstract syntax is adopted for applications of unary and binary data operations (which may concretely exploit mixfix notation). The operational semantics of action notation does not depend on the details of notation for data, only on the existence of its intended interpretation.

closed except Data.

grammar:

3.1.1 Actions

- Action = Simple-Action | ⟦ "unfolding" Action ⟧ |
  ⟦ Action Action-Infix Action ⟧ .

- Simple-Action = "complete" | "unfold" | ⟦ "give" Yielder ⟧ |
  ⟦ "store" Yielder "in" Yielder ⟧ .

- Action-Infix = "or" | "and" | "and then" | "then" .

3.1.2 Yielders

- Yielder = Data-Constant | ⟦ Data-Unary "⦇" Yielder "⦈" ⟧ |
  ⟦Data-Binary "⦇" Yielder "," Yielder "⦈" ⟧ |
  ⟦"given" Data⟧ |
  ⟦"the" Data "bound to" Yielder⟧ |
  ⟦"the" Data "stored in" Yielder⟧ .

3.1.3 Data

- Data = Data-Constant | ⟦ DataUnary "⦇" Data "⦈" ⟧ |
  ⟦ Data Binary "⦇" Data "," Data "⦈" ⟧ | □ .

- Data-Constant = □ .

- Data-Unary = □ .

- Data-Binary = □ .

The □'s indicate productions left open.

The only bits of of our illustrative subset of action notation not covered by the kernel are the primitive action check Y and the yielder it. These are abbreviations determined by a function expand mapping trees to kernel abstract syntax trees, defined as follows:

- expand ⟦ "check" Y : Yielder ⟧ =
  ⟦ ⟦ "give" ⟦ (expand Y) " & " "true" ⟧ ⟧ "then" "complete" ⟧ .

- expand "it" = ⟦ "given" "datum" ⟧ .

(other sorts of nodes are left unchanged).

3.2 Configurations

The configurations used in operational semantics involve *syntactic* components, representing what remains to be performed, as well as essentially semantic entities, such as storage maps.

The specifications below use standard data notation for tuples, maps, etc., defined algebraically in [11, Appendix E]. For convenience, they also use the sort of data tuples, data, and special sorts of maps, bindings and storage, which are specified in [11, Appendix B].

3.2.1 Acting

Acting is a generalization of Action. The new constructs include Terminated entities, which stand for the outcome of the performance of a subaction, and may contain transient data. They also include Action entities with attached data.

grammar:

- Acting = Terminated | Intermediate .
- Terminated = Completed | Failed .
- Completed = ⟨ "completed" data ⟩ .
- Failed = "failed" .
- Intermediate = Simple-Action |. ⟦ " unfolding" Acting ⟧ | ⟦ Acting Action-Infix Acting ⟧ | ⟨ Action data ⟩ .

The data notation ⟨_ . . . . _⟩ denotes tuple *concatenation*, as does (_ , _) below.

3.2.2 States

A state represents a point in the performance of an action. The local information corresponds to the current scoped and stable information, the transient data being incorporated in the acting component of the state. (We put bindings together with storage here only because bindings cannot change in our illustrative subset of action notation; in [11] they are treated analogously to transient data.) Note that (Action, info) $\leq$ state, by the associativity of tuple concatenation.

introduces: state , local-info , info .

(1) state = (Acting, local-info) .

(2) local-info (bindings, storage) .

(3) info = (data, local-info) .

3.2.3 Commitments

introduces: commitments , committed , uncommitted .

(1) commitments = committing | uncommitted (*individual*) .

3.3 Transition Functions

The factions specified below correspond to a *structual operational semantics*. They are, in general, *not* compositional.

closed except Data.

3.3.1 Actions

run s is the sort of final outcomes obtained by repeatedly making transitions from the intermediate state s. stepped s is the sort of intermediate or final outcomes obtained by performing only the nrst transition from the intermediate state s.

introduces: run _ , stepped _ .

- run _ :: state $\longrightarrow$ (Terminated, storage) .

(1) stepped $(A, b, s) \geq (A' : \text{intermediate}, s' : \text{storage}, c' : \text{commitment})$;
run $(A', b, s') \geq (A'' : \text{Terminated}, s'' : \text{storage}) \Rightarrow$
run $(A{:}\text{Acting},\ b{:}\text{bindings},\ s{:}\text{storage}) \geq (A'', s'')$.

(2) stepped $(A, l) \geq (A' : \text{Terminated}, s' : \text{storage}, c' : \text{commitment}) \Rightarrow$
run $(A : \text{Acting}, l : \text{local} - \text{info}) \geq (A', s')$.

• stepped _ :: state → (Acting, storage, commitment) .

(3) stepped $(A$: Terminated, l : local − info$)$ = nothing .

3.3.1.1 Simple

(1) $i = (d$: data, b : bindings, s : storage$)$; evaluated (Y, i) =
evaluated (Y, i) = nothing ⇒

stepped ([["give" Y :Yilder]], i : info) =
stepped ([["store" Y :Yilder "in" Y_2 : Yielder]], i : info) =
stepped ([["store" Y_1 :Yilder "in" Y : Yielder]], i : info) =
("failed", s, uncommitted) .

(2) stepped ("complete", d :data, b :bindings, s :storage)
= ("completed", ( ), s, uncommitted) .

(3) stepped ("unfold", d :data, b :bindings, s:storage) = nothing .

(4) evaluated $(Y, d, b, s) = d'$: data ⇒

stepped ([["give" Y :Yielder]], d :data, b :bindings, s :storage) =
("completed", d', s, uncommitted) .

(5) evaluated $(Y_1, d, b, s) = v$: storable ;
evaluated $(Y_2, d, b, s) = c$: cell ⇒

stepped ([["store" Y_1 "in" Y_2]], d :data, b :bindings, s :storage) =
if c is in mapped-set of s
then ("completed", ( ), overlay (map c to v, s), committed)
else ("failed", s, uncommitted) .

3.3.1.2 Compound

introduces: simplified _ , unfolded _ , given _ .

Stepping

(1) $\mathsf{stepped}\ (\llbracket\text{“unfolding”}\ A\ \mathsf{:Action}\rrbracket, d\ \mathsf{:data},\ b\ \mathsf{:bindings},\ s\ \mathsf{:storage}) =$
$\mathsf{given}\ (\mathsf{unfolded}\ (A, \llbracket\ \text{“unfolding”}\ A\rrbracket), d), s,\ \mathsf{uncommitted})\ .$

(2) $\mathsf{stepped}\ (A_1, l) \geq (A_1'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ c'\ \mathsf{:commitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ \mathsf{Intermediate}\ (\text{“and then”}\ |\ \text{“then”})\ \mathsf{Intermediate}\ \rrbracket\ |$
$\llbracket\ \mathsf{Intermediate}\ \text{“and”}\ (\mathsf{Intermediate}\ |\ \mathsf{Completed})\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (\mathsf{simplified}\ \llbracket A_1'\ O\ A_2\rrbracket, s', c')\ .$

(3) $\mathsf{stepped}\ (A_2, l) \geq (A_2'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ c'\ \mathsf{:commitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ (\mathsf{Intermediate}\ |\ \mathsf{Completed})\ \text{“and”}\ \mathsf{Intermediate}\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (\mathsf{simplified}\ \llbracket A_1\ O\ A_2'\rrbracket, s', c')\ .$

(4) $\mathsf{stepped}\ (A_1, l) \geq (A_1'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ \mathsf{uncommitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ \mathsf{Intermediate}\ \text{“or”}\ \mathsf{Intermediate}\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (\mathsf{simplified}\ \llbracket A_1'\ O\ A_2\rrbracket, s',\ \mathsf{uncommitted})\ .$

(5) $\mathsf{stepped}\ (A_2, l) \geq (A_2'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ \mathsf{uncommitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ \mathsf{Intermediate}\ \text{“or”}\ \mathsf{Intermediate}\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (\mathsf{simplified}\ \llbracket A_1\ O\ A_2'\rrbracket, s',\ \mathsf{uncommitted})\ .$

(6) $\mathsf{stepped}\ (A_1, l) \geq (A_1'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ c\ \mathsf{:commitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ \mathsf{Intermediate}\ \text{“or”}\ \mathsf{Intermediate}\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (A_1', s', c')\ .$

(7) $\mathsf{stepped}\ (A_2, l) \geq (A_2'\ \mathsf{:Acting},\ s'\ \mathsf{:storage},\ c\ \mathsf{:commitment})\ ;$
$\llbracket A_1\ O\ A_2\rrbracket : \llbracket\ \mathsf{Intermediate}\ \text{“or”}\ \mathsf{Intermediate}\ \rrbracket \Rightarrow$

$\mathsf{stepped}(\llbracket A_1\ O\ A_2\rrbracket, l\ \mathsf{:local\text{-}info}) \geq (A_2', s', c')\ .$

Simplifying

The function simplified is only applied to an intermediate compound acting A where an immediate component of A is the acting part of the result of applying stepped. The result is an acting equivalent to A, simplified for instance by propagating “failed”. The specification of simplified $\llbracket A_1\ O\ A_2\rrbracket$ when both

A_1 and A_2 are terminated shows how the flow of transient information out of actions is determined by the various combinators.

- simplified _ :: Acting $\rightarrow$ Acting .

(1) $[\![A_1' \ O \ A_2]\!]$: $[\![$ Failed (“and then” | “then”) Intermediate $]\!]$ |
$[\![$ Failed (“and” (intermediate | completed $]\!]$ |
$[\![$ completed “or” Interdediate $]\!]$ $\Rightarrow$
simplified $[\![A_1' \ O \ A_2]\!] = A_1'$.

(2) $[\![A_1 \ O \ A_2']\!]$: $[\![$ (Intermediate | Completed) “and” Failed $]\!]$ |
$[\![$ Intermediate “or” Completed $]\!]$ $\Rightarrow$
simplified $[\![A_1 \ O \ A_2']\!] = A_2'$.

(3) $[\![A_1' \ O \ A_2']\!]$: $[\![$ (Intermediate Action-Infix Intermediate $]\!]$ $\Rightarrow$
simplified $[\![A_1' \ O \ A_2']\!] = [\![A_1' \ O \ A_2']\!]$.

(4) simplified $[\![$ “failed” “or” A_2 : Intermediate $]\!] = A_2$.

(5) simplified $[\![A_1$: Intermediate “or” “failed” $]\!] = A_1$.

(6) simplified $[\![$ “completed” d_1 : data “and” “completed” d_2 : data $]\!]$ =
$\langle$ “completed” $(d_1, d_2)\rangle$.

(7) simplified $[\![A_1$: Completed “and then” A_2 :Intermediate $]\!] = [\![A_1$ “and” $A_2]\!]$.

(8) simplified $[\![$ “completed” d_1 : data “then” A_2 : Intermediate $]\!]$ =
given (A_2, d_1) .

Unfolding

unfolded $(A, [\![$ “unfolding” $A]\!])$ is used to replate free occurences of the dummy action unfold by $[\![$ “unfolding” $A]\!]$ before performing A. Each unfolding takes a step, so performing $[\![$ “unfolding” “unfold” $]\!]$ takes infinitely-many steps.

- unfolded _ :: (Action, Action) $\rightarrow$ Action .

(1) unfolded (A_1 : Simple-Action, A_0 : Action) =
If A_1 “unfold” then $[\![$ “unfolding” $A_0]\!]$ else A_1 .

(2) $([\![$ unfolding A_1 : Action $]\!], A_0) = [\![$ "unfolding" $A_1]\!]$.

(3) unfolded $([\![A_1$: Action O : Action-Infix A_2 : Action $]\!], A_0 =$
$[\![($unfolded (A_1, A_0) O (unfolded $(A_2, A_0))]\!]$.

Giving

given (A, d) is used to freeze the initial transient data d given to A. The specification of given shows clearly how the flow of data into actions is determined by the various combinators.

- given _ :: (Action, data) $\rightarrow$ Action .

(1) given $(A$: Terminated, d : data$) = A$.

(2) A : Simple-Action $|[\![$ "unfolding" Action $\Rightarrow$.
given $(A, d$:data$) = (A, d)$.

(3) O : "or" | "and" | "and then" $\Rightarrow$
given $([\![A_1$: Acting O A_2 : Acting $]\!], d$: data$) =$
$[\![$ (given $(A_1, d))$ O (given $(A_2, d))]\!]$.

(4) given $([\![A_1$: Acting "then" A_2 : Acting$]\!], d$:data$) =$
$[\![($given $(A_1, d))$ "then" $A_2]\!]$.

3.3.2 Yielders

The evaluation of yielders is compositional, but note that yielders occurring in the action of an abstraction do *not* get evaluated.

introduces: evaluated _ .

- evaluated _ :: (Yielder, info) $\rightarrow$ data .

(1) evaluated $(Y$: Data-Constant, i : info$) =$ entity Y .
(2) evaluated $([\![O$: Data-Unary "$(\!|$" Y : Yielder "$|\!)$"$]\!] i$: info$) =$
unary-operation O (evaluated) $(Y, i))$.
(3) evaluated $([\![O$: Data-Binary "$(\!|$" Y_1 : Yielder "," Y_2 : Yielder "$|\!)$" $]\!] i$: info$) =$
binary-operation O (evaluated) $(Y_1, i))$ (evaluated) $(Y_2, i))$.

(4) evaluated ($[\![$ "given" D : Data "#" n : natural$]\!]$ d : data, b : bindings,
s : storage) = entity D & component#n of d .
(5) evaluated ($[\![$ "the" D : Data "bound to" Y : Yielder$]\!]$ d : data, b : bindings,
s : storage) = entity D & (b at evaluated (Y, d, b, s)) .
(6) evaluated ($[\![$ "the" D : Data "stored in" Y : Yielder$]\!]$ d : data, b : bindings,
s : storage) = entity D & (s at evaluated (Y, d, b, s)) .

3.3.3 Data

For a data term d with abstract syntax D, we expect entity $D = d$. Given the full specification of Data, the corresponding semantic equations could be generated automatically.

introduces: entity _, unary-operation _ _, binary-operation _ _ _.

- entity _ :: Data → data .

(1) entity $[\![O$: Data-Unary "(|" D : Data "|)" $]\!]$ = unary-operation O (entity D) .
(1) entity $[\![O$: Data-Binary "(|" D_1 : Data "," D_2 : Data "|)" $]\!]$ =
binary-operation O (entity D_1) (entity D_2).

- unary-operation _ _ :: Data-Unary, data → data → data .
- binary-operation _ _ _ :: Data-Binary, data → data , data → data.

4 Action Equivalence

The operational semantics of Action Notation determines the processing possibilities of each action. But this does not, by itself, provide a useful notion of *equivalence* between actions. For if two compound actions have exactly the same processing possibilities, it is easy to see that they must have the same compositional structure.

From a user's point of view, however, two actions may be considered equivalent whenever there is no conclusive *test* that reveals the differences in their processing possibilities. A test on an action may consist of performing it in a particular action *context*, and checking that it completes; diverging tests may be regarded as inconclusive. (In the full action notation, one could also test the communication behaviour arising when an action is performed by a distributed system of agents)

We expect the testing equivalence of actions to include various algebraic laws, such as associativity of the action combinators Moreover, we expect it to be a *congruence*, i.e., preserved by the combinators. Then the laws can be used in algebraic reasoning to show that various compound actions are equivalent, perhaps justifying a simple *program transforation rule* for some language on the basis of its action semantics.

Unfortunately, it is difficult to verify directly that a testing equivalence includes particular laws: one would have to consider all possible tests on the actions involved in the laws! Instead, we define another, smiler equivalence called *bisimulation*, which can more easily be shown to include the intended laws. Here (in contrast to CCS [5]) bisimulation is actually a conguence, and it follows easily that it is included in the contextual testing equivalence.

The techniques used here were developed by Park, Milner, de Nicola, and Hennessy, mainly in connection with studies of the specification calculus CCS. The notation and presentation below follow [6], although note that here we have to deal with local information and commitments, as well as actions.

First we define transition relations on states:

Definition 4.1 *For each* c : $\mathsf{commitment}$ *let* $\xrightarrow{c} \subseteq \mathsf{state} \times \mathsf{state}$ *be the* state transition relation *detemined by* $\mathsf{stepped}$ _ *as follows:*

$$(A, b, s) \xrightarrow{c} (A', b, s') \textit{ iff } \mathsf{stepped}\ (A, b, s) \geq (A', s', c).$$

where A, A' : Acting ; b : $\mathsf{bindings}$; s, s' : $\mathsf{storage}$; c : $\mathsf{commitment}$. *When* c *is* $\mathsf{uncommitted}$ *we write* $\xrightarrow{\cdot}$ *instead of* $\xrightarrow{c}$ *(and then always* $s = s'$*).*

Further, for each c : $\mathsf{commitment}$ *let* $\overset{c}{\Longrightarrow}$ $\mathsf{state} \times \mathsf{state}$ *be the* observable *state transition relation defined by*

$$\overset{c}{\Longrightarrow} = \xrightarrow{\cdot}^{*} \xrightarrow{c} \xrightarrow{\cdot}^{*}$$

(whem R_1R_2 *denotes the composition of relations* R_1, R_2 *and* R^* *denotes the reflexive transitive closure of* R*).*

Now we insider relations on actions, i.e., elements of Action:

Definition 4.2 *Let $\mathcal{H}$ be the function over binary relations $R \subseteq$ Action $\times$ Action such that $(A_1, A_2) \in \mathcal{H}(R)$ iff, for all l :*local-info,

- *Whenever $(A_1, l) \xrightarrow{c} (A_1', l')$ then,*
  - $(A_2, l) \overset{c}{\Longrightarrow} (A_1', l')$, *if* A_1' : Terminated,
  - *for some A_2' with $(A_1', A_2') \in R, (A_2, l) \overset{c}{\Longrightarrow} (A_2', l')$, otherwise;*
- *Whenever $(A_2, l) \xrightarrow{c} (A_2', l')$ then,*
  - $(A_1, l) \overset{c}{\Longrightarrow} (A_2', l')$, *if* A_2' : Terminated,
  - *for some A_1' with $(A_1', A_2') \in R, (A_1, l) \overset{c}{\Longrightarrow} (A_1', l')$, otherwise.*

Definition 4.3 $R \subseteq$ Action $\times$ Action *is a* bisimulation *if* $R \subseteq \mathcal{H}(R)$.

Let $\approx = \bigcup\{R \mid R \text{ is a bisimulation}\}$. When $A_1 \approx A_2$ we say that A_1 and A_2 are bisimilar.

Notice that two actions can only be bisimilar when they have similar transitions for *any* particular binding and storage information. In practice, this means that they must refer to exactly the same items of the current information.

Proposition 4.1 $\approx$ *is the largest bisimulation, the largest fixed point of $\mathcal{H}$, and an equivalence relation.*

Proof: Using the monotonicity of $\mathcal{H}$. See [6] for the details of a similar proof. □

Proposition 4.2 $\approx$ *is a congruence for the constructs of action notation.*

Proof: From the definitions, and by constructing bisimulations containing the compound actions when subactions are bisimilar. For example, consider the combinator A' then A : we have to show that whenever $A_1 \approx A_2$ we get also $(A_1 \text{ then } A) \approx (A_2 \text{ then } A)$, and similarly for the other argument of _then_. It is enough to show that $\{(A_1 \text{ then } A, A_2 \text{ then } A) \mid A_1 \approx A_2; A, A_1, A_2 : \text{Action}\}$ is a bisimulation. For any l:local-info, and any transition $(A_1, l) \xrightarrow{c} (A_1', l')$ to a *terminated state*, we have $(A_2, l) \overset{c}{\Rightarrow} (A_1', l')$ and the result follows immediately. Similarly for transitions to *intermediate* states, only now $(A_2, l) \stackrel{c}{\Longrightarrow} (A'_2, l')$ for some A_2 with $A'_1 \approx A'_2$. $\square$

The associativity of all our binary action combinators, the idempotence of the choice combinator 'or', and the unit properties of $\mathsf{complete}$ for both the basic sequencing combinator '$\mathsf{and\ then}$' and for interleaving 'and', as well as various other simple algebraic laws, can be shown just as easily. More usefully, these same laws (and others) can also be shown to hold for the bisimulation equivalence defined for the full action notation in [11].

5 Conclusion

We have seen how a unified metanotation can be used to define a structural operational semantics for a simple subset of action notation. A straightforward definition of bisimulation equivalence provides some essential algebraic laws for action equivalence. The extension to the full action notation, including concurrent action performance with asynchronous message-passing and process creation, can be found in [11], as can a fuller description of the unified metanotation.

The author welcomes comments on this work, and suggestions for how best to increase the strength of action theory.

Acknowledgments David Watt collaborated on the development of action notation. The work reported here has been partially funded by the Danish Science Research Council project DART (5.21.08.03).

Appendix: A Unified Meta-Notation

The metanotation summarized below is a subset of that used in [11].

Metanotation is for specifying formal notation: what symbols are used, how they may be put together, and their intended interpretation.

Our metanotation here supports a *unified* treatment of sorts and individuals (i.e., types and objects): an individual is treated as a special case of a sort. Thus operations can be applied to sorts as well as individuals. A vacuous sort represents the lack of an individual, in particular the *undefined* result of a partial operation. Sorts may be related by inclusion; sort equality is just mutual inclusion. But a sort is not determined just by the set of individuals that it includes: it has an *intension*, stemming from the way it is expressed. For example, the sort of those natural numbers that are in the range of the successor operation may be distinct from the sort of those that have a well-defined reciprocal, even though their sets of individuals are the same.

The meta-notation provides (positive) Horn clauses and (initial) constraints—explained below—for speciffying the intended interpretation of symbols. Specifications may be divided into mutually-dependent and nested modules, presented incrementally in any order.

A model of a specification consists of a distributive lattice of sorts with a bottom, a distinguished subset of individuals, and a monotonic function on the lattice for each operation, such that all the specified clauses and constraints are satisfied. See [7] for the formal details.

Vocabulary

The vocabulary of the meta-notation consists of constant and operation symbols, variables, titles, and special marks.

Symbols are of two forms: quoted or unquoted. Quoted symbols always stand for constants. In unquoted symbols the underline character _ indicates the positions of arguments. Symbols without _ *always* stand for constants. Symbols are written here in this saris-serif font. An operation symbol is classified as an *infix* when it both starts and ends with a _ , and as a *postfix* or *postfix* when it only ends, respectively starts, with a _ . It is called an *outfix* when _ only occurs internally.

There is one built-in constant symbol, nothing, and there are two built-in infix operation symbols, _ | _ , _ & _ .

Variables are sequences of letters, here written in *this italic font*, optionally followed by primes ′ and/or a numerical subscript or suffix.

Titles are sequences of words, here capitalized and written in **This Bold Font**.

A pair of grouping parentheses ( ) may be replaced by a vertical line to the left of the grouped material. Reference numbers for parts of specifications have no formal significance.

Sentences

A sentence is essentially a Horn clause involving formulae that assert equality, sort inclusion, or individual inclusion between the values of terms. The variables occurring in the terms range over all values, not only over individuals. The universal quantification is left implicit.

Terms

Terms consist essentially of constant symbols, variables, and applications of operation symbols to subterms. We use *mixfix* notation, writing the application of an operation symbol $S_0\_\ldots\_S_n$ to terms $T_1,\ldots,T_n$ as $S_0T_1\ldots T_nS_n$. Infixes have weaker precedence than prefixes, which themselves have weaker precedence than postties. Moreover, infixes are grouped to the left, so we may write x | y | z without parentheses. Grouping parentheses ( ) may be freely inserted for further disambiguation.

The value of a term is determined by the interpretation of the variables that occur in it. Such a value may be an individual (which is regarded as a special kind of sort), a vacuous sort, or a proper sort that includes some individuals.

The value of the constant nothing is a vacuous sort, included in all other sorts. Operations map sorts to sorts, preserving sort inclusion. _ | _ is sort union and _ & _ is sort intersection; they are the join and meet, respectively, of the sort lattice, and enjoy the usual properties of set union and intersection: associativity, commutativity, idempotency, and distribution over each other (De Morgan's laws). Moreover, nothing is the unit for _ | _. There is no point in having a unit for _ & _ , as it would be a sort that includes everything.

Formulae

$T_1 = T_2$ asserts that the values of the terms T_1 and T_2 are the same (individuals or sorts).

$T_1 \leq T_2$ asserts that the value of the term T_1 is a subsort of that of the term T_2; so does $T_2 \geq T_1$. Sort inclusion is the partial order of the sort lattice.

$T_1 : T_2$ asserts that the value of the term T_1 is an individual included in the (sort) value of the term T_2.

The mark $\Box$ (read as 'filled in later') in a term abbreviates the other side of the enclosing equation. Thus $T_2 = T_1 \mid \Box$ specifies the same as $T_2 = T_1 \mid T_2$ (which is equivalent to $T_2 \geq T$).

The mark *disjoint* following an equation or inclusion $T = T_1 \mid \ldots \mid T_n$ abbreviates equations asserting vacuity of the pairwise intersections of the T_i. The mark *individual* abbreviates equations asserting that each T_i is an individual, as well as their disjointness.

$F_1; \ldots ; F_n$ is the conjunction of the formulae $F_1, \ldots , F_n$. Conjunctions with a common term may be abbreviated, e.g., $x, y : x$ abbreviates $x : z \; ; \; y : z$ and $x : y = z$ abbreviates $x : y \; ; \; y = z$.

Clauses

A (generalized positive Horn) clause $F_1; \ldots ; F_m \Rightarrow C_1; \ldots ; C_n$, where $m, n \geq 1$, asserts that whenever all the antecedent formulae F_i hold, so do *all* the consequent clauses (or formulae) C_j. Note that clauses cannot be nested to the left of $\Rightarrow$, so $F_1 \Rightarrow F_2 \Rightarrow F_3$ is unambiguously grouped as $F_1 \Rightarrow (F_2 \Rightarrow F_3)$.

We restrict the interpretation of a variable V to individuals of some sort T in a clause C by specifying $V : T \Rightarrow C$. Alternatively we may simply replace some occurrence of V as an argument in C by $V : T$. We restrict V to subsorts of T by writing $V \leq T$ instead of $V : T$.

Functionalities

A functionality clause $S :: T_1, \ldots , T_n \to T$ specifies that the value of any application of the operation S is included in T whenever the values of the argument terms are included in the T_i. It does *not* by itself indicate whether the value might be an individual, a proper sort, or a vacuous sort.

Such a functionality may be augmented by the some *attributes*, for example *'total'* which abbreviates a clause asserting that the operation is a natural extension of an ordinary total operation on individuals to proper (and vacuous) sorts. It is straightforward to translate ordinary many-sorted algebraic specifications into our metanotation using functionalities and attributes; similarly for order-sorted specifications [2] written in OBJ3 [3].

Specifications

A modular specification S is of the form $B\ M_1 \ldots M_n$, where B is a basic specification, and the M_i are modules. Either B or the M_i (but not both) may be absent. B is inherited by all the M_i.

Each symbol stands for the same value or operation throughout a specification—except for symbols introduced *privately*. All the symbols (but not the variables) used in a module have to be explicitly introduced: either in the module itself, or in an outer basic specification, or in a referenced module.

Basic Specifications

A basic specification B may introduce symbols, assert sentences, and impose (initial) constraints on subspecifications. The metanotation for basic specifications is as follows.

introduces: $O_1, \ldots, O_n$. introduces the indicated symbols, which stand for constants and/or operations. Also **privately introduces:** $O_1, \ldots, O_n$. introduces the indicated symbols, but here the enclosing module translates them to *new* symbols, so that they cannot clash with symbols specified in other modules.

C . asserts the clause C as an axiom, to hold for any assignment of values to the variables that occur in it. Omitting C gives the empty specification, made visible by a period.

$B_1, \ldots, B_n$ specifies all that the basic specifications $B_1, \ldots, B_n$. specify, i.e., it is their union. The order of the B_i is irrelevant, so symbols may be used before they are introduced.

includes: $R_1, \ldots, R_n$. specifies the same as all the modules indicated by the references R_i. **needs:** $R_1, \ldots, R_n$. is similar to **includes:** $R_1, \ldots, R_n$. , except that it is not transitive: symbols introduced in the modules referenced by the R_i are not regarded as being automatically available for use in modules that reference the enclosing module.

grammar: S augments the basic specification S with standard specifications of strings and syntax trees (from [11, Appendix E]), and with the introduction of each constant symbol that occurs as the left hand side of an equation in S. Similarly when S is a series of modules.

closed . specifies the constraint that the enclosing module is to have a *standard* (i.e., initial) interpretation. This means that it must be possible, using the specified symbols, to express every *individual* that is included in some expressible sort (*no junk*), and moreover that terms have equal/included/individual values only when that logically follows from the specified axioms (*no confusion*). **closed except** $R_1, \ldots, R_n$. specifies a similar constraint, but leaves the (sub)modules referenced by the R_i open, so that they may be specialized in extensions of the specification. **open** . merely indicates that the enclosing module is not to be closed.

Modules

A module M is of the form $I\ S$, where I is a title that identifies the specification S.

Modules may also be nested, in which case an inner module inherits the basic specifications of all the enclosing modules, and the series of titles that identifies the immediately enclosing module.

Parameterization of modules is rather implicit: unconstrained submodules, specified as **open** . , can always be specialized.

A series of titles $I_1/\ldots/I_n$ refers to a module (together with all its submodules). A common prefix of the titles of the enclosing module and of the referenced module may be omitted. In particular, sibling modules in a nest can be referenced using single titles. $R/^*$ refers to all submodules of R.

References

[1] E. Astesiano. Inductive and operational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts,* IFIP State-of-the-Art Report, pages 51–136. Springer-Verlag, 1991.

[2] J. A. Goguen and J. Meseguer. Order-sorted algebra: Algebraic theory of polymorphism. *Journal of Symbolic Logic,* 51:844–845, 1986. Abstract.

[3] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Lab., SRI International, 1988.

[4] G. Kahn. Natural semantics. In *STACS'87, Proc. Symp. on Theoretical Aspects of Computer Science,* number 247 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[5] R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[6] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science,* volume B, chapter 19. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[7] P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science,* pages 304–312. IEEE, 1989.

[8] P. D. Mosses. Unified algebra and modules. In *POPL'89, Proc. 16th Ann. ACM Symp. on Principles of Progamming Languages,* pages 329–343. ACM, 1989.

[9] P. D. Mosses. Denotational semantics. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science,* volume B, chapter 11. Elsevier Science Publishers, Amsterdam; and MIT Press, 1990.

[10] P. D. Mosses. A practical introduction to denotational semantics. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts,* IFIP Stater-of-the-art Report, pages 1–49. Springer-Verlag, 1991.

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

[12] J. Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Aarhus University, 1992.

[13] J. Palsberg. A provably correct compiler generator. In *ESOP'92, Proc. European Symposium on Programming, Rennes,* number 582 in Lecture Notes in Computer Science, pages 418–434. SpringerVerlag, 1992.

[14] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN–19, Computer Science Dept., Aarhus University, 1981. Now available only from University of Edinburgh.

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