ISSN 0105-8517

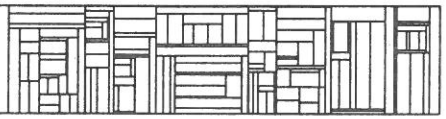
A SEMANTIC ALGEBRA FOR BINDING CONSTRUCTS

by

Peter Mosses

DAIMI PB-132
March 1981

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



A SEMANTIC ALGEBRA FOR BINDING CONSTRUCTS *

Peter Mosses

Computer Science Department
Aarhus University
Aarhus, Denmark

Abstract This paper presents a semantic algebra, suitable for use in giving the denotational semantics of various forms of declarations and binding constructs in programming languages. The emphasis of the paper is on the development of semantic descriptions which are easy to understand at an intuitive level, being based on algebraic operators corresponding to fundamental concepts of programming languages. Some familiarity with denotational semantics and abstract data types is assumed.

Keywords Programming languages, semantics, abstract data types, formalization of programming concepts, algebras, binding constructs.

1. INTRODUCTION

Denotational Semantics has been used to describe various programming languages, by giving models based on higher-order functions on Scott-domains. Techniques have been developed for representing features of programming languages in terms of the basic operations of λ-notation: application, abstraction, tupling and "tagging" (separated sums). Naturally enough, the most frequently used techniques have been those corresponding to fundamental concepts of computation, common to many programming languages: for example, function composition, representing the sequencing of actions; continuations, also representing sequencing, but allowing jumps in control; and the separation of states into (usually static) environments and (dynamic) stores. (For an introduction to the descriptive techniques of Denotational Semantics, see Gordon's (1979) book.)

The problem is that these modelling techniques seem to have taken on a life of their own. A standard denotational semantics is (generally) presented as if the meaning of each construct was just the application of higher-order functions to each other - instead of being concerned with the order of evaluation of sub-constructs, scope rules, etc. It is left to the reader of the semantics to recognize the domains of semantic values and the associated patterns of application and abstraction, in order to extract the operational implications of the semantics from the λ-notation.

For example, consider the fragment of standard denotational semantics in Table 1. Anyone familiar with some of the literature on denotational semantics will immediately recognize the familiar patterns, and conclude that the language being described is a deterministic, imperative language, perhaps having some form of jump, and probably with static scope rules for variables. A command sequence '$C_1;C_2$' is executed in left-to-right order, and the R-value of an identifier 'I' is

* To be presented at the International Colloquium on Formalization of Programming Concepts, Peniscola, Spain, April 1981.

found by a single de-referencing of the location denoted by the identifier.

Of course, it is splendid that those well-versed in denotational semantics can communicate their ideas so easily. (It was Strachey's idea that one could learn a lot about a language by simply inspecting the domain definitions in its standard denotational semantics.) But note the number of different concepts which application (in λ-notation) is representing: sequencing, static scoping, dynamic storing, looking-up in environments and stores, and passing a value to a continuation!

As well as effectively disguising operational concepts, the widespread use of applications in the standard style of denotational semantics has another unfortunate effect: it makes semantic descriptions difficult to modify. This point is of relevance when denotational semantics is used during language design and development, and also when teaching formal semantics - starting by considering a small language with simple semantics and extending it gradually to (say) ADA. Of course, a standard denotational semantics is easily extensible, so long as the new constructs are based on the same concepts as the original ones; the crunch comes when one wants to add something new, like non-determinism (and power-domains) - it can entail re-writing all the original semantic equations, unless one is clever enough to find new semantic domains which allow the original patterns of applications to be retained.

Table 1. A Fragment of Standard Denotational Semantics

θ : C = S → A
$\varkappa$: K = E → C
ρ : Env = Ide → L
σ : S = L → V
...
$\mathcal{C}$: Cmd → Env → C → S → A
$\mathcal{C}[\![C_1;C_2]\!]\rho\,\theta\sigma = \mathcal{C}[\![C_1]\!]\rho\,(\mathcal{C}[\![C_2]\!]\rho\,\theta)\sigma$
...
$\mathcal{R}$: Exp → Env → K → S → A
$\mathcal{R}[\![I]\!]\rho\,\varkappa\sigma = \varkappa\,(\sigma\,(\rho\,[\![I]\!]))\sigma$
...

Here, we use a new approach to denotational semantics, free from the above problems inherent in the standard approach. The basic idea is to insist on the abstraction, as explicit operators, of the fundamental operational concepts of computation. These operators form a so-called semantic algebra: a sort of abstract data type, with the operators generally operating on "actions" rather than on data.

The proposed approach is illustrated in detail below. One of the main features is that the "semantic equations" are now devoid of any assumptions about the structure of the semantic domains. Not only does this mean the absence of long lists of parameters to the semantic function(s), it means also that there is never any need to modify existing semantic equations when extending the described language with new constructs. The semantic equations give the meaning of each syntactic con-

struct purely in terms of the operators representing the fundamental operational concepts of computation - the semantic functions can in fact be regarded as a simple translation from the programming language to the semantic algebra.

As for the definition of the semantic algebra itself, there are two ways to go: it may be defined axiomatically, in the style of equationally-specified abstract data types; or one may take a concrete semantic algebra (a model), defining the operators as functions on Scott-domains. The choice is essentially between, respectively, ease of modification and ease of initial specification, although (even) more subjective properties like comprehensibility should also be considered.

An earlier paper (Mosses (1980)) illustrated the usefulness of equationally-specified semantic algebras in the construction of correct compilers; and the author's general preference is for equational specifications. It is also appealing to consider the semantic equations of a denotational semantics as an equational specification of an operator, rather than (or as well) as an indirect definition of a homomorphism. However, in spite of this, the semantic algebras in this paper are defined as models. The motivation is two-fold: to make this paper more digestible for those acquainted with (standard) denotational semantics; and to put the emphasis on the idea of abstracting semantic operators corresponding to fundamental concepts, rather than on the specification of these operators.

The rest of this paper is organized as follows. Section 2 introduces notation, and a basic semantic algebra which, although it may look rather strange at first, seems to be a good foundation for building up more specialized semantic algebras for particular (families of) languages. Section 3 considers the concept of binding, and shows how to cope with both static and dynamic scope rules. Section 4 specializes the basic algebras. The conclusion assesses the advantages and disadvantages of the proposed approach, and points to topics for future work.

2. A BASIC SEMANTIC ALGEBRA

We start by considering operations corresponding to some particularly fundamental concepts, such as sequencing and value-passing. These operations, together with their definitions as functions on some (Scott-) domains, give a basic semantic algebra, which can be extended with more specialized operations for the purpose of describing particular programming languages.

Actually, we do not go into the details of a model (i.e. choice of functions and domains) in this section. Our basic semantic operations are only informally described. The point is that the domains required to support the full basic semantic algebra are overly complex - at least in comparison to the domains needed for the specialized semantic algebra of Section 4, which is used for giving the semantics of a simple example language. More comments about possible models are made below.

In fact the operations of the basic semantic algebra (BS) described below are a selection from a more general semantic algebra (to be reported in a forthcoming

paper). Only those operations relevant to the semantics of our simple example language are included.

Notational Conventions

Our somewhat unconventional notation is motivated by a desire to present syntactic constructions and semantic operations in the same way - after all, the operator symbols corresponding to semantic operations give a language, and syntactic constructions may denote (compound) semantic operations. In contrast to the notation of ADJ (e.g. 1979), we make use of ordinary terminal symbols of grammars in operator symbols, and we allow distributed (mixfix) operators as well as prefix and infix ones.

Domain names are in lower case (possibly with hyphens and/or primes), e.g. cmd, a, a'; subscripts may be used to distinguish particular occurrences of domains in definitions. Domain names are used also as meta-variables, varying over values in the corresponding domains. Alphabetic parts of operator symbols are underlined, e.g. var, update. The functionalities of operator symbols are given in a modified BNF, e.g.

cmd ::= var id := num in cmd | cmd_1; cmd_2

a ::= a_1; a_2

- whether an operator symbol represents a syntactic construction or a semantic operation is determined by the domains used (here, 'cmd' is a syntactic domain, whereas 'a' is semantic).

Fundamental Operations

Now for an informal description of the basic semantic algebra, BS, whose syntax is given in Table 2. (The reader familiar with the semantic algebras of Mosses (1980) might notice some simplifications which have been made here - as well as some new uses of old symbols!)

Table 2. Syntax of BS, A Basic Semantic Algebra

Domains	a - actions,	with source $\sigma a \in \Delta^*$,	target $\tau a \in \Delta^*$	
	v - values,	with domain $\delta v \in \Delta$		
	x - variables,	with domain $\delta x \in \Delta$		
Syntax	a ::= ( )	$\sigma a =$?	$\tau a = ( )$	
	| v	( )	(δv)	
	| a_1, a_2	σa_1	$\tau a_1 \cdot \tau a_2$	$(\sigma a_1 = \sigma a_2)$
	| a_1; a_2	σa_1	$\tau a_1 \cdot \tau a_2$	$(\sigma a_1 = \sigma a_2)$
	| a_1! a_2	σa_1	τa_2	$(\tau a_1 = \sigma a_2)$
	| $x \Rightarrow a_1$	(δx)	τa_1	$(\sigma a_1 = ( ))$
	v ::= x	$\delta v = \delta x$		
	x ::= i	$\delta x =$ i	where $i \in \Delta$	
	| i_n	i	$i \in \Delta$, $n \in \{0, 1, \ldots\}$	

The central concept is that of actions, or perhaps "computations" might be better terminology. Actions (a) may consume and/or produce sequences of values, as specified by their sources (σ a) and targets (τ a); they may also have "effects". The values (v) are divided into smaller domains named by the elements of the unspecified set Δ (taking Δ as a poset would allow a hierarchy of separated sums of domains). As well as being produced and consumed by actions, values may also be referred to by semantic variables (x).

The simplest action is the null action '( )', which consumes any values passed to it, but otherwise has no effect.

Next is the action 'v', which just produces that value. (The value may depend on previously-computed values, referred to by variables x.)

The actions 'a_1, a_2' and '$a_1; a_2$' both correspond to an associative tupling of the value sequences produced by a_1 and a_2. The difference between them is that with ';', the effects of a_2 are preceded by the effects of a_1, whereas with ',', the effects (if any) of a_1 and a_2 may be interleaved. In either case, the sequences of values consumed by a_1 and a_2 are identical. (The most common use of ';' is in composing actions which neither consume nor produce any values. ',' is useful for specifying a "don't care" order of evaluation.)

The action '$a_1 ! a_2$' corresponds to function composition (and to application). All the values produced by a_1 are passed to a_2 (for consumption). Consequently, the effects of a_1 (if any) precede those of a_2. (The reader worried by the apparent commitment to a "call-by-value" discipline may be reassured by the illustration of delayed evaluation in Section 4 - the idea is just to "wrap up" an action as a value, enabling it to be passed around by other actions.)

The final basic action considered here is '$x \Rightarrow a_1$', which consumes a value, and binds the variable x to this value in a_1. (The introduction of such binding operators complicates an algebraic treatment, but this is offset by being able to do without machine-code-like "permutors" (ADJ (1979)) for re-arranging sequences of values.)

For convenience, semantic variables 'x' have been formed by underlining and (possibly) subscripting domain names $i \in \Delta$. In specialized semantic algebras used for describing particular (families of) programming languages, Δ is specified fully, and operations on values (and constants) are added to the operations in Table 2.

Syntactic Conventions

Brackets may be used to make the structure of BS terms explicit. The operations ',', ';' and '!' are all associative, allowing repeated occurrences to be written without brackets. The only other convention in BS is that the term 'a_1' in '$x \Rightarrow a_1$' is assumed to extend as far to the right as possible (cp. λ-notation).

Possible Models

If the action 'a_1, a_2' were to be dropped from BS, it would be possible to take a model based on simple Scott-domains (either "direct", or "continuation"-based).

However, when actions with effects are combined in 'a_1, a_2', the resulting potential non-determinism necessitates the introduction of power-domains (and "resumptions"). In the specialized semantic algebra LS of Section 4, ',' is allowed only on effect-free actions, permitting a model without power-domains.

Note that if '$a_1; a_2$' were to be omitted, and actions had no effects, then BS would reduce to a λ-calculus for sequences - and a "pure" model (with strict composition) could be used.

The next section extends BS with some rather general operations corresponding to the concept of bindings in programming languages. The resulting algebra, BBS, is specialized for giving the semantics of a simple example language in Section 4.

3. AN ALGEBRA FOR BINDING

Mosses (1980) gave a semantic algebra, S, and used it in giving a semantics for a small, imperative programming language. That language included the construct '*let* id *be* $aexp_1$ *in* $aexp_2$', which declared and initialized a local variable whose scope was $aexp_2$. So we might expect that S had already some suitable operations corresponding to static binding.

In fact, the "binding" operations of S were '*update*' and '*contents*' - corresponding to (dynamic) storing! These were used to "implement" static binding in the semantics of '*let* id *be* $aexp_1$ *in* $aexp_2$' by remembering the value (contents) of id before the "initialisation" to $aexp_1$, and then restoring (updating) id with this value after computing $aexp_2$. Apart from the fact that this simple technique only gives static scope rules in the absence of procedures with global variables, it can hardly be regarded as a direct way of specifying static binding. (It was used in the referenced paper only to facilitate comparison with the work of ADJ (1979) on compiler correctness.)

Let us consider the fundamental concepts of binding. The basic notion is that a declaration (or a formal parameter) specifies a binding of an identifier to some value, and subsequent occurrences of the bound identifier may refer to this value. In general, the "bound value" may be either a constant (e.g. number, address) or an unevaluated action (e.g. procedure body); and in the latter case, there may be some unresolved references to identifiers (e.g. global variables). There are basically two ways of resolving such references: static scoping, which makes use of the bindings existing at the time the value is bound; and dynamic scoping, which uses the bindings existing at the time the unresolved reference is evaluated. (These two strategies give different results when identifiers can be re-declared and bound values can have unresolved references.)

The operations of the semantic algebra BBS, whose syntax is given in Table 3, should provide a general basis for the semantics of binding constructs of programming languages. As illustrated by the specialized version of BBS used in Section 4,

it has not been assumed that static and dynamic binding strategies are mutually exclusive. Generality is achieved through allowing the "freezing" of unresolved references in a bound value at any time before they are needed in a computation.

Table 3. Syntax of BBS, a Semantic Algebra for Binding

Domains	a - actions, with source $\sigma a \in \Delta^*$, target $\tau a \in \Delta^*$				
	v - values, with domain $\delta v \in \Delta$				
	where $\{d, id\} \subseteq \Delta$, a set of domain names				
Syntax	a ::=	*delay* $\{a_1\}$	$\sigma a =$ ( )	$\tau a =$ (d)	$(\sigma a_1 = \tau a_1 = ( ))$
	|	*eval*	(d)	( )	
	|	*freeze*	(d)	(d)	
	|	*bind* $\{a_1\}$	(id, d)	τa_1	$(\sigma a_1 = ( ))$
	|	*bind-local* $\{a_1\}$	(id, d)	τa_1	$(\sigma a_1 = ( ))$
	|	*find*	(id)	(d)	
	|	*find-local*	(id)	(d)	

The following informal description of the BBS operations might suggest a model to readers familiar with (for example) Gordon's (1979) book. The domain name 'd' corresponds to "denotable values", and 'id' represents the embedding of syntactic identifiers in the semantic algebra.

The action '*delay* $\{a_1\}$' coerces an unevaluated action (a_1) to a denotable value, without freezing any references. (The restriction of a_1 to empty source and target is not essential.) The reverse of this operation is '*eval*' which converts a denotable value back into an action - without any freezing - and performs the action. The composition '*delay* $\{a_1\}$! *freeze*' gives static scopes, whereas dynamic scopes are obtained by simply refraining from using '*freeze*' at all.

The action '*bind* $\{a_1\}$' takes an (id, d) pair of values, and binds id to d for the duration of a_1. A variant on this is '*bind-local* $\{a_1\}$', in which the binding is only statically visible. The latest binding - either frozen, or dynamic - is obtained by '*find*', whereas '*find-local*' ignores dynamic bindings.

It is conjectured that the operations of BBS are sufficient for describing the scope rules of most (if not all) existing programming languages. The author would welcome any convincing counter-examples.

4. AN EXAMPLE

We conclude with a semantic description of an (artificial) language L, whose main virtue is that procedure identifiers have static scopes, whereas variable identifiers have dynamic scopes.

However, we first need to specialize BS and BBS to LS, a semantic algebra corresponding exactly to the concepts which (in our analysis, at least) underly L.

In forming LS, we add actions 'update' (for storing values) and 'contents' (for accessing values). To avoid unwanted sharing, variables are bound to unique identifiers (i.e. "locations") generated by the action 'unique', rather than bound directly to numbers.

LS is defined in Table, 4, with a convenient choice of domains for the promised model, based on "direct semantics". Note that the actions a' are (side-) effect free, which permits the definition of $'a'_1, a'_2'$ on our (relatively) simple domains.

Table 4. LS, a Semantic Algebra for L

Domains
- a - actions, with source $\sigma a \in \Delta^*$, target $\tau a \in \Delta^*$
- a' - actions without (side-) effects
- v - values, with domain $\delta v \in \Delta$
- x - variables, with domain $\delta x \in \Delta$

where $\Delta = \{id, l, n, p\}$

where
- id is identifiers
- l is locations
- n is numbers
- p is procedures

Syntax

		$\sigma a =$	$\tau a =$
a ::=	$a_1;a_2$	σa_1	$\tau a_1 \cdot \tau a_2$
|	$a_1!a_2$	σa_1	τa_2
|	$x \Rightarrow a_1$	(δx)	τa_1
|	eval	(p)	( )
|	bind-proc $\{a_1\}$	(id, p)	τa_1
|	bind-var $\{a_1\}$	(id, l)	τa_1
|	update	(l, n)	( )
|	unique	(id)	(l)
|	a'	$\sigma a'$	$\tau a'$
a' ::=	a'_1, a'_2	$\sigma a' = \sigma a'_1$	$\tau a' = \tau a'_1 \cdot \tau a'_2$
|	$a'_1 ! a'_2$	$\sigma a'_1$	$\tau a'_2$
|	v	( )	(δv)
|	delay $\{a_1\}$	( )	(p)
|	freeze-procs	(p)	(p)
|	find-proc	(id)	(p)
|	find-var	(id)	(l)
|	contents	(l)	(n)
v ::=	x	$\delta v = \delta x$	
x ::=	i	$\delta x =$ i	where $i \in \Delta$
|	i_n	i	$n \in \{0, 1, \ldots\}$

(Table 4 continued)

Semantics:

Domains
$a = [el \times ep] \to [\sigma a \times s] \to [\tau a \times s]$
$a' = [el \times ep] \to [\sigma a' \times s] \to \tau a'$

where
- el = id → l
- ep = id → p
- s = l × m where m = l → n
- l = n
- n = $\{\perp, 0, 1, \ldots\}$
- p = a + [el → ( ) × s → ( ) × s]
- id = unspecified

Operations

$(a_1;a_2)(el, ep)(\sigma a_1, s)$ . . . . . = let $(\tau a_1, s_1) = a_1(el, ep)(\sigma a_1, s)$ in
let $(\tau a_2, s_2) = a_2(el, ep)(\sigma a_1, s_1)$ in
$(\tau a_1 \cdot \tau a_2, s_2)$

$(a_1!a_2)(el, ep)(\sigma a_1, s)$ . . . . . = let $(\tau a_1, s_1) = a_1(el, ep)(\sigma a_1, s)$ in
$a_2(el, ep)(\tau a_1, s_1)$

$(x \Rightarrow a_1)(el, ep)((\delta x), s)$ . . . . = $(a_1$ with δx for $x)(el, ep)(( ), s)$

(eval)$(el, ep)((p), s)$ . . . . . = p is a → $p(el, ep)(( ), s), p(el)(( ), s)$

(bind-proc$\{a_1\}$)$(el, ep)((id, p), s)$ = $a_1(el, ep[p/id](( ), s)$

(bind-var$\{a_1\}$)$(el, ep)((id, l), s)$ = $a_1(el[l/id], ep)(( ), s)$

(update)$(el, ep)((l, n), (l_0, m))$ . . = $(( ), (l_0, m[n/l]))$

(unique)$(el, ep)((id), (l_0, m))$ . . = $((l_0), (l_0+1, m))$

$(a')(el, ep)(\sigma a', s)$ . . . . . = let $\tau a' = a'(el, ep)(\sigma a', s)$ in
$(\tau a', s)$

$(a'_1, a'_2)(el, ep)(\sigma a'_1, s)$ . . . = let $\tau a'_1 = a'_1(el, ep)(\sigma a'_1, s)$ in
let $\tau a'_2 = a'_2(el, ep)(\sigma a'_1, s)$ in
$\tau a'_1 \cdot \tau a'_2$

$(a'_1 ! a'_2)(el, ep)(\sigma a'_1, s)$ . . . . = let $\tau a'_1 = a'_1(el, ep)(\sigma a'_1, s)$ in
$a'_2(el, ep)(\tau a'_1, s)$

$(v)(el, ep)(( ), s)$ . . . . . = v

(delay$\{a_1\}$)$(el, ep)(( ), s)$ . . . = a_1

(freeze-procs)$(el, ep)((p), s)$. = p is a → $\lambda el_1 . p(el_1, ep)$, p

(find-proc)$(el, ep)((id), s)$ . . . = ep(id)

(find-var)$(el, ep)((id), s)$ . . . = el(id)

(contents)$(el, ep)((l), (l_0, m))$ . . = m(l)

Some liberties have been taken with notation in the definitions of operations, e.g. σ and τ are used to indicate sequences of (typed) bound variables, and the syntactic domain names are used both for semantic domains and as meta-variables over these! The definition of '$x \Rightarrow a_1$' has been left informal, to avoid going into some technicalities.

Table 5 gives the semantics of L, based on the operations of LS. It should not need further explanation – if it does, then this paper has not achieved its aims.

Table 5. Syntax and Semantics of L

Domains prg, cmd, id, num

Syntax

prg ::= res id in cmd

cmd ::= cmd_1; cmd_2
| id_1 := id_2
| var id := num in cmd_1
| call id
| proc id is cmd_1 in cmd_2

Semantic Functions based on LS

$\mathcal{P}$: prg → a, where $\sigma a = ( )$, $\tau a = (n)$

$\mathcal{P}[\![$res id in cmd$]\!]$ . . . . . = id ! unique !
$l \Rightarrow (id, l)$! bind-var $\{\mathcal{C}[\![$cmd$]\!]$;
l ! contents$\}$

$\mathcal{C}$: cmd → a, where $\sigma a = ( )$, $\tau a = ( )$

$\mathcal{C}[\![cmd_1; cmd_2]\!]$ . . . . . = $\mathcal{C}[\![cmd_1]\!]$;$\mathcal{C}[\![cmd_2]\!]$

$\mathcal{C}[\![id_1 := id_2]\!]$ . . . . . = (id_1 ! f: d-var,
id_2 ! find-var ! contents) ! update

$\mathcal{C}[\![$var id := num in $cmd_1]\!]$. = id ! unique !
$l \Rightarrow (l, \mathcal{N}[\![num]\!])$! update;
(id, l) ! bind-var $\{\mathcal{C}[\![cmd_1]\!]\}$

$\mathcal{C}[\![$call id$]\!]$ . . . . . = id ! find-proc ! eval

$\mathcal{C}[\![$proc id is cmd_1 in $cmd_2]\!]$ = (id,
delay $\{\mathcal{C}[\![cmd_1]\!]\}$! freeze-procs)!
bind-proc $\{\mathcal{C}[\![cmd_2]\!]\}$

$\mathcal{N}$: num → v, where $\delta v = n$, is not specified.

5. CONCLUSION

By the abstraction of operations corresponding to fundamental concepts of programming languages, we obtain denotational descriptions which do not make assumptions about choices of semantic domains. Compared with "standard" denotational descriptions, ours seem to be easy to modify and extend. Moreover, they exhibit the operational features of the described language.

The particular semantic algebras presented in this paper are not claimed to be the only suitable algebras, as regards the choice of operator symbols and the functionalities of the operations – or even as regards the underlying concepts. More experimentation with semantic algebras is needed.

Future work is to investigate the axioms satisfied by the operations of semantic algebras, and whether equational specifications could be used instead of (or as well as) domain-based models. Once the "right" semantic algebras have been developed, it would be useful to introduce a framework for expressing the hierarchical structure and modularity of these algebras – perhaps along the lines of Burstall & Goguen's (1977) specification language, Clear.

REFERENCES

ADJ: Thatcher, J.W., Wagner, E.A. & Wright, J.B.
(1979) "More on advice on structuring compilers and proving them correct", in Proc. Sixth Int. Coll. on Automata, Languages and Programming (ed. Maurer, H.A.). Lect. Notes in Comp. Sci. 71, Springer, 1979.

Burstall, R.M. & Goguen, J.A.
(1977) "Putting theories together to make specifications", in Proc. Fifth Int. Joint Conf. on Artificial Intelligence, 1977.

Gordon, M.J.C.
(1979) The Denotational Description of Programming Languages, Springer, 1979.

Mosses, P.
(1980) "A constructive approach to compiler correctness", in Proc. Seventh Int. Coll. on Automata, Languages and Programming (eds. de Bakker, J.W. & van Leeuwen, J.). Lect. Notes in Comp. Sci. 85, Springer, 1980.

DAIMI PB-107: Neil D. Jones: *Circularity Testing of Attribute Grammars Requires Exponential Time: A Simpler Proof.* — January 1980. 9 pp. — Dkr. 3.00.

DAIMI PB-108: Kurt Jensen: *A Method to Compare the Descriptive Power of Different Types of Petri Nets.* — January 1980. 26 pp. — Dkr. 4.00.

DAIMI PB-109: Ole Lehrmann Madsen: *On Defining Semantics by Means of Extended Attribute Grammars.* — January 1980. 65 pp. — Dkr. 8.00.

DAIMI PB-110: Bent Bruun Kristensen and Ole Lehrmann Madsen: *A General Algorithm for Solving a Set of Recursive Equations (Exemplified by LR-theory).* — February 1980. 24 pp. — Dkr. 5.00.

DAIMI PB-111: Zahari Zlatev: *On Solving Some Large Linear Problems by Direct Methods.* — February 1980. 47 pp. — Dkr. 7.00.

DAIMI PB-112: Zahari Zlatev: *Modified Diagonally Implicit Runge-Kutta Methods.* — February 1980. 23 pp. — Dkr. 4.00.

DAIMI PB-113: Neil D. Jones and David A. Schmidt: *Compiler Generation from Denotational Semantics.* — March 1980. 23 pp. — Dkr. 4.00.

DAIMI PB-114: Hanne Riis: *Subclasses of Attribute Grammars.* — March 1980. 104 pp. —Dkr. 14.00.

DAIMI PB-115: Bent Bruun Kristensen and Ole Lehrmann Madsen: *A Practical State Splitting Algorithm for Constructing LR-Parsers.* — March 1980. 56 pp. — Dkr. 8.00.

DAIMI PB-116: Kurt Jensen and Morten Kyng: *Petri Nets and Semantics of System Descriptions.* — April 1980. 28 pp. — Dkr. 5.00.

DAIMI PB-117: Erik Meineche Schmidt: *Space-Restricted Attribute Grammars.* — April 1980. 13 pp. — Dkr. 3.00.

DAIMI PB-118: Peter D. Mosses: *A Constructive Approach to Compiler Correctness.* — April 1980. 15 pp. — Dkr. 3.00.

List of DAIMI Publications

DAIMI PB-119: Neil D. Jones and Michael Madsen: *Attribute-influenced LR Parsing.* — April 1980. 15 pp. — Dkr. 3.00.

DAIMI PB-120: Kurt Jensen: *How to Find Invariants for Coloured Petri Nets.* — May 1980. 20 pp. — Dkr. 4.00.

DAIMI PB-121: Hanne Riis and Sven Skyum: *K-visit Attribute Grammars.* — June 1980. 19 pp. — Dkr. 3.00.

DAIMI PB-122: Zahari Zlatev: *Comparison of Two Pivotal Strategies in Sparse Plane Rotations.* — July 1980. 33 pp. — Dkr. 6.00.

DAIMI PB-123: Ole Østerby and Zahari Zlatev: *Direct Methods for Sparse Matrices.* — July 1980. 128 pp. — Dkr. 17.00.

DAIMI PB-124: Zahari Zlatev and Ole Østerby: *Absolute Stability Properties of the Explicit Linear 3-step Formulae.* — August 1980. 22 pp. — Dkr. 5.00.

DAIMI PB-125: *Software Engineering: Tools and Methods.* Proceedings of the 1978 Aarhus Workshop on Software Engineering. Nigel Derrett, Karen Møller, Mike Spier, J. Michael Bennett (eds.). — August 1980. 287 pp. — Dkr. 37.75.

DAIMI PB-126: Brian H. Mayoh: *The Meaning of Logical Programs.* — September 1980. 20 pp. — Dkr. 2.50.

DAIMI PB-127: Peter Kornerup: *Firmware Development Systems. A Survey.* — November 1980. 18 pp. — Dkr. 2.50.

DAIMI PB-128: Neil D. Jones: *Flow Analysis of Lambda Expressions.* — January 1981. 31 pp. — Dkr. 3.25.

DAIMI PB-129: Jørgen Staunstrup: *Analysis of Concurrent Algorithms.* — January 1981. 15 pp. — Dkr. 2.25.

DAIMI PB-130: David W. Matula and Peter Kornerup: *On Minimum Weight Binary Representation of Integers and Continued Fractions with Application to Computer Arithmetic.* — January 1981. 43 pp. — Dkr. 4.00.

DAIMI PB-131: Flemming Nielson: *Semantic Foundations of Data Flow Analysis.* — February 1981. 113 pp. — Dkr. 9.00.

DAIMI PB-132: Peter Mosses: *A Semantic Algebra for Binding Constructs.* — March 1981. 11 pp. — Dkr. 2.00.

DAIMI PB-133: K.R. Apt & G.D. Plotkin: *A Cook's Tour of Countable Nondeterminism.* — April 1981. 16 pp. — Dkr. 2.25.

DAIMI PB-134: Peter Kornerup & David W. Matula: *An Integrated Rational Arithmetic Unit.* — April 1981. 25 pp. — Dkr. 3.25.

DAIMI PB-135: Flemming Nielson: *A Denotational Framework for Data Flow Analysis.* — July 1981. 40 pp. — Dkr. 3.75.

DAIMI PB-136: Lars Mathiassen:
Systemudvikling og systemudviklingsmetode. — September 1981. 173 pp. — Dkr. 12.50.

DAIMI PB-137: Neil D. Jones & Henning Christiansen: *Control Flow Treatment in a Simple Semantics-Directed Compiler Generator.* — September 1981. 24 pp. — Dkr. 2.75.

DAIMI PB-138: Hanne Riis Nielson: *Computation Sequences: A Way to Characterize Subclasses of Attribute Grammars.* — November 1981. 21 pp. — Dkr. 2.50.

DAIMI PB-139: Hanne Riis Nielson: *Using Computation Sequences to Define Evaluators for Attribute Grammars.* — November 1981. 26 pp. — Dkr. 3.00.

DAIMI PB-140: Flemming Nielson: *Program Transformations in a Denotational Setting.* — November 1981. 30 pp. — Dkr. 3.25.

DAIMI PB-141: Ib Holm Sørensen: *Specification and Design of Distributed Systems.* — December 1981. 106 pp. — Dkr. 8.00.

DAIMI PB-142: Peter Møller-Nielsen and Jørgen Staunstrup: *Early Experience with a Multi-Processor Project.* — January 1982. 42 pp. — Dkr. 7.00.

DAIMI PB-143: David Harel and Dexter C. Kozen: *A Programming Language for the Inductive Sets, and Applications.* — April 1982. 17 pp. — Dkr. 2.25.

DAIMI PB-144: Thorkil Naur: *Integer Factorization.* — April 1982. 134 pp. — Dkr. 9.75.