Basic Research in Computer Science

BRICS RS-94-5 P. D. Mosses: Unified Algebras and Abstract Syntax

Unified Algebras and Abstract Syntax

Peter D. Mosses

BRICS Report Series **RS-94-5**

ISSN 0909-0878 **March 1994**

See back inner page for a list of recent publications in the BRICS Report Series. 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 WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Unified Algebras and Abstract Syntax*

Peter D. Mosses

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

Abstract

We consider the algebraic specification of abstract syntax in the framework of unified algebras. We illustrate the expressiveness of unified algebraic specifications, and provide a grammar-like notation for specifying abstract syntax, particularly attractive for use in semantic descriptions of full-scale programming languages.

1 Introduction

The algebraic specification framework of *unified algebras* is somewhat unorthodox: both individuals and sorts are treated as values, so operations can be applied to sorts as well as to individuals. Moreover, no distinction is made between a singleton sort and its only element. The empty sort serves as a convenient representation of undefinedness.

Signatures of unified algebras are merely ranked alphabets. Axioms of specifications are (definite) Horn clauses involving equality, sort inclusion, and individual inclusion. The usual functionalities of operations, which are signature components in most frameworks, can easily be specified as axioms. Models of unified specifications are distributive lattices with bottoms, equipped with a distinguished (usually discrete) subset of

*To appear in: *Recent Trends in Data Type Specification* (ed. F. Orejas), Lecture Notes in Computer Science 785, Springer-Verlag, 1994. Citations of this work should refer only to the LNCS publication, which is identical to the present report (up to formatting details).

†Basic Research in Computer Science, a Centre of the Danish National Research Foundation.

individuals, together with inclusion-preserving functions. Specifications have initial models. Initial constraints can be specified, providing a simple form of parameterization. For formal details, see [10]; for examples, see [12, Appendix E].

Unified algebras are claimed to have significant advantages over conventional frameworks, in particular concerning the treatment of polymorphism and genericity. Here, we show how unified algebras allow the specification of abstract data types that provide an elegant and flexible treatment of *abstract syntax*.

Abstract syntax is used primarily in formal semantic descriptions of programming languages; other applications include syntax-directed editing and program transformation systems. Essentially, abstract syntax ignores details concerned with unambiguous parsing and lexical symbols, and focuses on the compositional tree structure of parsed phrases. Thereby it provides a simple interface between (context-free) concrete syntax and semantics.

There are various ways of specifying abstract syntax. With some of them, for instance McCarthy's original formulation [9], specifications resemble ordinary algebraic specifications of constructor, selector, and discriminator operations. With others, they resemble ordinary context-free grammars. It has also been shown how to transform grammars into many-sorted algebraic specifications, obtaining the corresponding abstract syntax as initial many-sorted algebras [3].

What is 'wrong' with these previous approaches to specifying abstract syntax? Well, the ones that look like ordinary algebraic specifications are not sufficiently perspicuous when used on large-scale programming languages. The ones that look like ordinary grammars are usually quite perspicuous, but their algebraic interpretation is somewhat clumsy; they are also rather *rigid*, in that nonterminal symbols cannot be replaced by the corresponding alternatives without disturbing the meaning of the specification.

We shall see that with unified algebras, a grammar is *itself* a set of axioms for an algebraic specification: there is no need for any transformation. The advantages of this approach include: it is straightforward to allow algebraic sort constructors corresponding to *regular expressions*; abstract syntax is naturally *order-sorted*; nonterminal symbols can be *substituted* by their alternatives without disturbing the specified algebra; and the *micro-syntax* of lexical symbols can be accommodated without

bother.

The contribution of this paper is threefold. It illustrates the use of the expressiveness of unified algebraic specifications, thereby motivating this framework in relation to conventional frameworks. It provides an algebraic notation for specifying abstract syntax that is particularly attractive for use in semantic descriptions of realistic, large-scale programming languages. Finally, the algebraic sort constructors given here are generally useful in specifications of abstract data types: they allow a simple treatment of operations with variable numbers of arguments, e.g., an operation that constructs a list from n components for any n.

The paper is organized as follows. Section 2 summarizes the notation used in unified algebraic specifications, defines the notion of a unified algebra, and discusses constraints. Section 3 considers previous approaches to abstract syntax. Section 4 gives a unified algebraic specification of an abstract data type of trees. Section 5 proves that the given specification is consistent, by defining a nontrivial model; it also considers initial models of the specification. Section 6 provides a simple illustration of the unified algebraic specification of abstract syntax, and argues that the approach has some beneficial pragmatic qualities that previous approaches lack.

2 Unified Algebras

The following definitions are from [10] (except that 'elements' are now called 'individuals').

Definition 1 *A* signature Σ *for a unified algebra is a set of operation* symbols*, each symbol having the* rank *determined by the number of placeholder signs '_' that it contains. Signatures of unified algebras always include the symbols* $\_ \mid \_$ *and* $\_ \,\&\, \_$ *of rank 2, and the symbol* nothing *of rank 0.*

Definition 2 *A* unified algebra A *has a* universe*, which is a distributive lattice [4] with a bottom value, together with a distinguished subset* $I_A \subseteq A$ *of* individuals*. The partial order of the lattice is denoted by* $\leq_A$*. For each operation symbol* f *in the signature* Σ *of* A*, there is a monotone (total) function* f_A *on the universe of the lattice, with* $\_ \mid \__A$ *being the join operation of the lattice,* $\_ \,\&\, \__A$ *the meet, and* $\mathsf{nothing}_A$ *the bottom of the lattice.*

Notice that the operations are not required to be strict or additive, nor to preserve the property of individuality. Moreover, we do not insist that the lattices serving as universes of unified algebras have tops.

Technically, the framework of unified algebras is 'unsorted'. However, *all* the values in the universe of a unified algebra may be thought of as sorts, with the individuals corresponding to singleton sorts. The partial order of the lattice represents sort inclusion; join is sort union and meet is sort intersection. The individuals do *not* have to be the atoms of the lattice, just above the bottom: for instance, the meet of two individuals is below both of them, but need not be identified with the bottom value. Those values that do not include any individuals at all, such as the bottom value, are vacuous sorts, often representing the lack of a proper result that arises from applying an operation to unintended arguments. A special case of a unified algebra is a *power algebra*, whose universe is a power set, ordered by set inclusion, with the singletons as individuals [10].

Definition 3 *The* axioms *of a unified algebraic specification are definite Horn clauses, written* $e_1; \cdots; e_n \Rightarrow e$*, where the antecedents* $e_1, \cdots, e_n$ *and the consequent* e *are equations* $t_1 = t_2$*, inclusions* $t_1 \leq t_2$*, or* individual *inclusions* $t_1 : t_2$ *between terms. The terms are built from the operation symbols in the signature of the specification and from* variables*. Each axiom is treated as if all the variables occurring in it are universally quantified.*

For perspicuity, we use *mixfix* notation when writing terms, replacing the place-holder signs '_' in symbols by the arguments and inserting grouping parentheses when necessary for disambiguation. It is convenient to assume that infixes associate to the left, so that further parentheses can be omitted.

Definition 4 *An axiom* $e_1; \cdots; e_n \Rightarrow e$ *of a unified specification is* satisfied *by a unified algebra* A *(of the same signature) if whenever all the antecedents* $e_1, \cdots, e_n$ *hold in* A*, so does the consequent* e*. An equation* $t_1 = t_2$ *holds in* A *when the terms* t_1*,* t_2 *have identical values (whether or not these values are individuals, proper sorts, or vacuous). An inclusion* $t_1 \leq t_2$ *holds when the value of* t_1 *is below that of* t_2 *in the partial order of the sort lattice. An individual inclusion* $t_1 : t_2$ *holds when the value of* t_1 *is not only included in that of* t_2*, but also in the distinguished subset of individuals* I_A*. A unified algebra that satisfies all the axioms of a specification is called a* model *for the specification.*

Unified algebraic specifications always have initial models, because they are essentially just unsorted Horn clause logic (with equality) specifications: the structure of unified algebras is entirely captured by a set of Horn clauses, given in the Appendix. One reason for not restricting attention to the power algebras mentioned above is that then specifications—even very simple ones—would fail to have initial models. For example, let a and b be specified to be individuals, and let c : a | b, so that c is an *individual* included in the union a | b; since individuals are singleton sets in power algebras, this forces *either* c=a *or* c=b, and there is clearly no initial model for the specification.

Although it can be shown that unified algebras provide a *liberal* institution, with the usual notion of reduct functor, it is problematic to define useful constraints in unsorted frameworks, because the ordinary reduct functor only forgets operations—never values. However, by using a *more forgetful* reduct functor (treating all ground terms as if they were sorts) one can simulate the way that many-sorted and order-sorted forgetful functors deal with values, thereby providing *bounded data constraints* whose effect is similar to that of ordinary data constraints in conventional frameworks. See [10] for the details.

3 Abstract Syntax

Let us briefly consider some previous approaches to abstract syntax. At the end of this section, we shall discuss the relationship between concrete and abstract syntax.

McCarthy [9] was the first to formulate a notion of abstract syntax. There he proposed the use of syntax that differs from context-free grammars (BNF) by being "analytic rather than synthetic; it tells how to take a program apart, rather than how to put it together". The syntax should also be independent of the notation used to represent sums, etc., in program texts.

This idea is realized by introducing predicates to distinguish between different constructs. For simple arithmetic expressions, one might have the predicates isvar(t), isconst(t), issum(t), and isprod(t). For each predicate, one introduces selector functions, such as addend(t) and augend(t) for terms t satisfying issum(t). Finiteness of terms can be expressed by the convergence of a recursively defined predicate expressed using the

introduced predicates and selector functions.

McCarthy also considers "languages which have both an analytic and a synthetic syntax satisfying certain relations". The synthetic syntax uses constructor functions, such as mksum(t,u) and the relations are specified by equations, such as addend(mksum$(t,u)) = t$.

The specification of abstract syntax in Meta-IV, the meta-notation of VDM [2], exploits a systematic naming convention for predicates and constructor functions: is-$A(o)$ tests whether an object o is of type A, mk-$A(o_1,\ldots,o_n)$ constructs objects of type A from appropriate arguments. Types are specified in a notation close to BNF. E.g., for arithmetic expressions one may specify:

Expr = Var | Const | Sum | Prod
Sum :: Expr Expr
...

Equations are interpreted as domain equations. The | stands for simple, nondiscriminated union; the :: ensures that objects created by the corresponding constructor function are distinct from those created by other constructors. Here the constructor function mk-Sum(l,r) is implicitly declared, and one can select the two subexpressions of a sum s by pattern-matching, as in let mk-Sum$(l,r) = s$ in .... Meta-IV also allows explicit selector functions to be introduced, as in:

Sum :: s-left:Expr s-right:Expr .

Meta-IV goes on to allow the use of domain operators for tuples (A*, A+), optional domains ([A]), power sets (A-set), finite maps, partial functions, and total functions.

The form of abstract syntax used in connection with syntax-directed editing in [5, 15] is based on *phyla* and constructor operators. A phylum is simply a set of terms, and the operators map terms to terms. Phyla may not overlap. Specifications may be factored, so arithmetic expressions could be specified as:

expr = Var(ID) | Const(INT) | Sum, Prod(expr,expr)

where ID and INT are predefined lexical phyla, specified using regular expressions. In the ASF+SDF formalism [6], abstract syntax is derived from the context-free grammar that is used to specify concrete syntax, for example:

ID → EXP
NAT → EXP
EXP "+" EXP → EXP {left}
EXP "∗" EXP → EXP {left}
"(" EXP ")" → EXP {bracket}

Abstract syntax trees are then generated automatically from parsed strings.

Denotational semantics (see [16] for a comprehensive text, or [11] for an introduction) has exploited various meta-notations for specifying abstract syntax. Scott and Strachey [17] originally paid scant respect to syntax: they used ambiguous, indexed grammars, written in a variant of BNF; they assumed that languages come equipped with grouping parentheses so that the compositional structure of a phrase could always be made precise when necessary. For example:

I ∈ Iden
K ∈ Const
E ∈ Expr
O ∈ Oper
E ::= I | K | E_0 O E_1 | (E)
O ::= + | *

Later Stoy [18] interpreted such grammars as defining sets of parse trees, and pointed out that they may be just as abstract as McCarthy's abstract syntax, if the grammar is chosen appropriately.

The initial algebra approach to abstract syntax (and semantics) [3] shows the existence of initial algebras for each many-sorted signature Σ using a concrete representation of trees. By identifying abstract syntax with an isomorphism class of initial algebras, independence from representational details is obtained, and algebraic homomorphisms from abstract syntax to target algebras having the same signature are uniquely defined. To specify an abstract syntax, one merely gives the signature Σ. It was shown that one can obtain signatures systematically from context-free[1] grammars, by mapping each nonterminal A of a grammar G to a sort symbol, and regarding each production $A_0 \rightarrow u_0 A_1 u_1 \ldots A_n u_n$ (where the u_i are sequences of terminal symbols) as an operation symbol from

[1]The grammars are allowed to be infinite, so they may generate non-context-free languages.

argument sorts $A_1, \ldots, A_n$ to sort A_0. The initial algebras with this signature are essentially parse trees for derivations in G (whether or not the grammar is ambiguous).

We see that most of the approaches considered above have the emphasis on *synthetic* abstract syntax, where notation is provided for constructing abstract phrases. Moreover, several approaches directly exploit context-free grammars, where terminal symbols are used to distinguish between different constructs.

This departure from McCarthy's original concept of abstract syntax, which was primarily analytic, has been found to be beneficial in various applications. No abstractness is hereby lost—providing one doesn't insist on a precise correspondence between the symbols used in grammars for concrete syntax and for abstract syntax—because in *all* approaches, even in McCarthy's, one has to choose *some* symbols for naming operations, and it is neither more nor less abstract to choose, say, mk-Sum with prefix notation in some algebraic signature, than to choose + with infix notation in the corresponding context-free grammar.

Since the grammars used for abstract syntax are not intended for parsing, they may be ambiguous (hence simpler) and yet still specify sorts of abstract syntax trees precisely. And when one does want to relate concrete syntax to abstract syntax, the relation is much easier to see when the terminal symbols used in the grammar for abstract syntax are suggestive of the corresponding concrete symbols.

However, there is still one mismatch: in practice, the concrete syntax of real programming languages is usually specified with grammars that exploit some form of *regular expressions*, as in so-called Extended BNF. It appears that only the Meta-IV approach to abstract syntax caters for the trees with unbounded branching that naturally arise from parsing according to such grammars. And although one could generalize the translation from grammars to signatures in the initial algebra approach to cope with regular expressions, the resulting signatures would be quite messy, with a new sort for each regular expression used in the grammar.

Thus there is a need for a simple algebraic treatment of context-free grammars allowing regular expressions, for use in specifying abstract syntax. We now proceed to provide such a treatment.

4 Trees

This section gives a unified algebraic specification of an abstract data type of (finite) trees, including operations for expressing regular sets of trees.

The first line of the specification below declares the signature of the specification. (The symbols $\_ \mid \_$, $\_ \& \_$, and nothing are always implicitly in the signature.) The constant character stands for some unspecified alphabet, whose individuals are to be used as the leaves of our (otherwise unlabelled) trees. The intended interpretation of the binary operation symbol $\_\ \_$ (juxtaposition) is concatenation of sequences, and the empty parentheses are to be interpreted as the empty sequence. The unary operation symbol $\_^*$ is to be the Kleene-*, mapping any sort of tree T to a sort including precisely those individual sequences whose components are all of sort T; more generally, it may be applied to a sort of sequences of trees. Finally, $[\![\ \_\ ]\!]$ constructs an individual tree from its branches. N.B. it is *not* a semantic function itself! (If $\mathcal{F}\ \_$ is a semantic function, it will still be correct to use the familiar $\mathcal{F}[\![\ldots]\!]$ in semantic equations.) Both $\_\ \_$ and $[\![\ \_\ ]\!]$ are to extend naturally from individuals to arbitrary sorts, e.g., (a b) is to be a sort that includes all individuals $(x\ y)$ where x and var y are individuals included in sorts a and b, respectively.

introduces: character , tree , $\_\ \_$, ( ) , $\_^*$, $[\![\ \_\ ]\!]$.

(1) $(a\ b)\ c = a\ (b\ c)$.

(2) $(\,)\ a = a$.

(3) $a\ (\,) = a$.

(4) $a\ (b \mid c) = (a\ b) \mid (a\ c)$.

(5) $(a \mid b)\ c = (a\ c) \mid (b\ c)$.

(6) nothing a = nothing .

(7) a nothing = nothing .

(8) $a^* = (\,) \mid (a\ a^*)$.

(9) $a^* = (\,) \mid (a^*\ a)$.

(10) $(a\ x) \leq x \ \Rightarrow\ a^*\ x \leq x$.

(11) $(x\ a) \leq x \ \Rightarrow\ x\ a^* \leq x$.

(12) tree = character $\mid$ $[\![$ tree* $]\!]$.

(13) $[\![\ a \mid b\ ]\!] = [\![\ a\ ]\!] \mid [\![\ b\ ]\!]$.

(14) $[\![\ \mathsf{nothing}\ ]\!] = \mathsf{nothing}$.

(15) $a : \mathsf{tree}^* \ \Rightarrow\ [\![\ a\ ]\!] : \mathsf{tree}$.

(16) $(\ ) : \mathsf{tree}^*$.

(17) $a : \mathsf{tree}^*\ ;\ b : \mathsf{tree}^* \ \Rightarrow\ (a\ b) : \mathsf{tree}^*$.

Axioms (1)–(11) are taken almost straight from [7], where they (together with some of the axioms and the rules of inference stated in our Appendix) are shown to provide a complete deductive system for equations between regular sets over an alphabet.[2] Concerning the use of Horn clauses, note that no finite set of pure equations can be a base for the equational theory of regular sets, in the absence of auxiliary operation symbols. In any case, a straightforward Horn clause specification seems preferable to an intricate equational specification, at least in regard to practical reasoning on the basis of specifications.

The framework of unified algebras provides some formal abbreviations for commonly occurring patterns of axioms. Exploiting these, the above specification can be written somewhat more succinctly:

introduces: $\mathsf{character}$, tree , $\_\ \_$, $(\ )$, $\_^*$, $[\![\ \_\ ]\!]$.

$\_\ \_ :: \mathsf{tree}^*, \mathsf{tree}^* \to \mathsf{tree}^*$ (*total*, *associative*, *unit is* $(\ )$)

$a^* = (\ ) \mid (a\ a^*) = (\ ) \mid (a^*\ a)$.

$(a\ x) \leq x \ \Rightarrow\ a^*\ x \leq x$.

$(x\ a) \leq x \ \Rightarrow\ x\ a^* \leq x$.

$\mathsf{tree} = \mathsf{character} \mid [\![\ \mathsf{tree}^*\ ]\!]$.

$[\![\ \_\ ]\!] :: \mathsf{tree}^* \to \mathsf{tree}$ (*total*) .

$(\ ) : \mathsf{tree}^*$.

A 'functionality' of the form $\mathsf{f}\ \_ :: t_1 \to t_2$ is equivalent to the inclusion axiom $\mathsf{f}\ (t_1) \leq t_2$; notice that the t_i need not be constants. Monotonicity then implies $\mathsf{f}\ (x_1) \leq t_2$ for all arguments $x_1 \leq t_1$. The 'attribute' *total* on a functionality expresses that an operation is the natural (strict, additive) extension to sorts of some ordinary total operation on individuals. We don't bother to specify the functionality of $\_^*$, as it would be $\mathsf{tree} \to \mathsf{tree}^*$, making a tautology. Note that we must *not* specify $\_^*$ to be total, as it is to be neither strict nor additive, nor to map individuals to individuals!

[2] This does not imply that the axioms are complete for proving equalities between regular expressions that are allowed to use the intersection operation $\_\ \&\ \_$.

The above specifications are, in the absence of any explicit constraints, interpreted *loosely*: any unified algebra (with the declared signature) satisfying the stated axioms would be a model. Here, however, we intend the individuals of our models to be only the finite sequences of finite individual trees. Moreover, models should not equate individual trees that have different shapes or leaves. As usual with initial algebra approaches to specification, there should be no 'junk' and no 'confusion'.

All this can be specified by a *bounded data constraint* that leaves the individuals included in character open, while insisting that the values—both sorts and individuals—included in tree* be freely generated by these characters, relative to the specified axioms. Formally, the constraint consists of a *theory inclusion*. Unified algebraic specifications allow such constraints to be specified succinctly using references to modules. Here, we do not bother to introduce notation for modules. The desired constraint is simply the inclusion of the theory whose only explicit operation symbol is character, with no explicit axioms, in the theory presented by the above specification.

5 Correctness

It is easy to specify axioms that express intended properties of operations. Unfortunately, it is also easy make a mistake! For instance, one might specify an axiom that should hold only for a variable taking *individual* values, but not for vacuous or proper sorts. The possibility of instantiating the axiom with these sorts may then lead to unwelcome consequences—perhaps even to the identification of all values! (Such dangers are not special to unified algebras: the treatment of partial operations and errors in many-sorted algebraic specifications is notoriously tricky.)

In the case of trees, we have a good idea of what the intended models are—up to isomorphism—so we can check the correctness of our specification by defining a particular unified algebra and verifying that it satisfies all the axioms. We should also check that our model satisfies the specified bounded data constraint, which here ensures that models with the same individual characters form an isomorphism class.

For any set C (of characters) not containing the value λ let the unified algebra U be defined as follows. We use ordinary notation for mathematical definitions of sets, partial functions, and sequences of numbers and

functions. In particular, sequence concatenation is written $p \cdot p'$, and ε is the empty sequence. Sequences of natural numbers p in N^* represent positions in trees; the functions f in T represent finite trees by mapping positions to labels in $C \cup \lambda$, the labels of interior nodes being always λ.

$$\begin{array}{rcl}
T & = & \{f : N^* \xrightarrow{\sim} (C \cup \lambda) \mid \\
 & & \quad |\mathsf{dom}(f)| < \infty; \\
 & & \quad \forall p{\in}N^* \forall n{\in}N(p \cdot n \in \mathsf{dom}(f) \;\Rightarrow\; p \in \mathsf{dom}(f),\; f(p) = \lambda); \\
 & & \quad \forall p{\in}N^* \forall n{\in}N(p \cdot (n+1) \in \mathsf{dom}(f) \;\Rightarrow\; p \cdot n \in \mathsf{dom}(f))\} \\
S & = & T^* \\
U & = & \mathcal{P}(S) \\
\leq_U & = & \subseteq \\
I_U & = & \{\{s\} \mid s \in S\}.
\end{array}$$

For each operation symbol f in the specified signature the interpretation as a total function (of 0, 1, or 2 arguments) on U is defined as follows:

$$\begin{array}{rcl}
_ \mid __U(a,b) & = & a \cup b \\
_ \,\&\, __U(a,b) & = & a \cap b \\
\mathsf{nothing}_U & = & \emptyset \\
\mathsf{character}_U & = & \{[\varepsilon \mapsto c] \mid c \in C\} \\
\mathsf{tree}_U & = & T \\
_ \; __U(a,b) & = & \{x \cdot y \mid x \in a; y \in b\} \\
(\;)_U & = & \{\varepsilon\} \\
_^*{}_U(a) & = & \{\varepsilon\} \cup \{x_1 \cdot \ldots \cdot x_n \mid x_1, \ldots, x_n \in a; n \geq 0\} \\
[\![_]\!]_U(a) & = & \begin{cases} \emptyset, \text{ if } a = \emptyset; \\ \cup\{\mathsf{node}(f_0, \ldots, f_n) \mid \\ \quad x \in a; x = f_0 \cdot \ldots \cdot f_n; f_0, \ldots, f_n \in T\}, \text{ otherwise} \end{cases}
\end{array}$$

where we define the auxiliary functional node for each n by:

$$\mathsf{node}(f_0, \ldots, f_n)(p) \;=\; \begin{cases} \lambda, & \text{if } p = \varepsilon; \\ f_i(p'), & \text{if } p = (i \cdot p'), 0 \leq i \leq n; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The partial functions in the set T mapping sequences $n_1 \cdot \ldots \cdot n_m$ of natural numbers $n_i \in N$ represent trees whose leaves are labelled with characters $c \in C$, and whose internal nodes are unlabelled. The function

mapping only ε to a character c represents that character; that mapping only ε to λ represents the tree with no branches at all. The set S consists of sequences of functions, representing sequences of trees. (T^* is the well-known set of sequences of elements from the set T; it could be defined in terms of higher-order functions to avoid any trace of circularity, at some extra notational expense.) U is the power set consisting of all subsets of S, ordered by set inclusion, and the set I_U of individuals is the set of all singletons in U. Note that we do not take account of the natural partial order on the partial functions themselves.

The interpretation of the various operation symbols as (total) functions on U is rather straightforward, except perhaps for that of node construction: the partial function $\mathsf{node}(f_0,\ldots,f_n)$ inspects the first branch number, say i, in its argument p, and applies the corresponding subtree f_i to the rest of of p; if the argument p is the empty number sequence, it returns λ.

Proposition 1 *The structure U defined above is a unified algebra, and it satisfies axioms (1)–(17) from Section 4.*

Proof: It is easy to see that U is a distributive lattice with a bottom, and that the operation symbols $\_ \mid \_$, $\_ \& \_$, and $\mathsf{nothing}$ are interpreted correctly. The operations corresponding to $\_\ \_$ and $[\![\ \_\ ]\!]$ are defined as pointwise extensions to U of functions on S, which ensures that they are monotone, strict, additive, and map individual arguments to individuals. Thus axioms (4)–(7) and (13)–(15) are satisfied. Since the only functions in T that cannot be returned by $\mathsf{node}(f_0,\ldots,f_n)$ are precisely those that map ε to a character c, axiom (12) is satisfied. The specified sequencing operation satisfies axioms (1)–(3) because the mathematical sequencing notation does. Similarly for axioms (8) and (9).

For axiom (10), let a and x have values $\bar{a}$ and $\bar{x}$ such that $\_\ \__U(\bar{a},\bar{x}) \subseteq \bar{x}$ holds. We have to show $\_\ \__U(\_^*{}_U(\bar{a}),\bar{x}) \subseteq \bar{x}$ holds. Each element in $\_\ \__U(\_^*{}_U(\bar{a}),\bar{x})$ consists of a finite, possibly-empty sequence s of partial functions formed by concatenating sequences s_1, …, s_n contained in $\bar{a}$ with a sequence s_0 contained in $\bar{x}$. If $n = 0$, the result follows immediately. Otherwise, consider the concatenation of s_n with s_0; from $\_\ \__U(\bar{a},\bar{x}) \subseteq \bar{x}$ this must be an element of $\bar{x}$. By a simple induction we get $s \in \bar{x}$ and the result follows. By symmetry, axiom (11) is satisfied as well. □

So U satisfies the axioms of our specification, which demonstrates that the specification is consistent. But does it satisfy the bounded data

constraint given in Section 4? No, it doesn't! For consider, say, the term $[\![(\,)]\!] \,\&\, [\![\,[\![(\,)]\!]\,]\!]$. Clearly, the value of this term in U is the empty set. Almost as clearly, the equation $[\![(\,)]\!] \,\&\, [\![\,[\![(\,)]\!]\,]\!] = \mathsf{nothing}$ is not a consequence of the axioms of our specification (including those given in the Appendix). Hence U has 'confusion', so it cannot be freely generated by C relative to the specified axioms. Thinking of U as the model we are trying to characterize, we might say that our *specification* gives rise to 'junk', rather than U having 'confusion'. This specification junk consists of distinct expressible *sorts* that denote equal sets in U; however, there are no junk individuals at all. Less seriously, U has unreachable junk that cannot be expressed by ground terms, arising from the use of the (uncountable!) unrestricted powerset $\mathcal{P}(S)$, which includes various non-regular sets of character sequences, for example. This problem could be eliminated by reducing U to its smallest subalgebra.

We could try to mend this discrepancy between our specification and its intended model in several ways:

1. Add further axioms to our specification, such that all ground equations involving $\_ \,\&\, \_$ that are satisfied in U become consequences. We have recently proved a corresponding result [1] in the absence of node construction $[\![\_]\!]$, and it is conjectured that the specification concerned—and its correctness proof—can be extended to the algebra considered here. Note that the axioms given by Kozen for his *action lattices* [8] relate meet only to join, not to sequencing or *. Moreover, action lattices involve *residuation* [14], which is nonmonotonic and so cannot be used as an operation in unified algebras.

2. Relax the notion of models of bounded data constraints in unified algebras, so that *extensionally-equal* sorts may always be equated in a model. This complicates the notion of a model of a constrained specification, but it might be a viable extension of the framework.

3. Define the problem away by removing $\_ \,\&\, \_$ from unified algebras, so that models are merely semilattices. Then the problematic identities cannot be expressed, and the reduced U would presumably be a model of the specified bounded data constraint. The drawback here is that it is actually quite useful in practice to have $\_ \,\&\, \_$ available! For instance, we might want to specify that truth-values

and numbers are to be disjoint. Of course, meet could always be explicitly introduced and axiomatized when needed; but if we introduced it together with our specification of trees, our problem would promptly reappear.

Further investigation of these possibilities is out of the scope of this paper.

6 Unified Abstract Syntax

Let us now turn to the use of our notation for trees in the specification of abstract syntax. The idea is rather simple: we use sort equations to define abstract syntax as a collection of sorts of trees.

First, it is convenient to extend our specification of trees with the following notation:

introduces: string , $\_^+$, $\_^?$.

string = ⟦ character* ⟧ .

$a^+ = a\ a^*$.

$a^? = (\ ) \mid a$.

Let us also assume a definite notation for individual characters and strings. For each printing (or blank) character c, there is supposed to be a symbol 'c' that denotes the corresponding individual in character. The symbols digit and letter can then easily be specified to denote the expected subsorts of character. Moreover, the symbol "$c_1 \ldots c_n$" abbreviates ⟦ 'c_1' ... 'c_n' ⟧, thus denoting an individual in string since strings are simply trees whose branches are all characters.[3]

Now consider the following unified algebraic specification. If one ignores the double brackets ⟦...⟧, it looks just like a context-free grammar exploiting regular expressions, with terminal symbols being written in quotes. (Although the need for the double brackets below might be considered a drawback by some, their use avoids relying on obscure precedence rules for disambiguating grouping in grammar specifications. Moreover, they make it possible to specify the elimination of 'chain-nodes' in abstract syntax trees.)

[3]To specify this notation for strings formally would require a schematic presentation of an infinite signature and a corresponding set of axioms.

grammar:

Stmt = ⟦ Iden ":=" Expr ⟧ |
⟦ "begin" Stmt (";" Stmt)* "end" ⟧ |
⟦ "if" Expr "then" Stmt ("else" Stmt)$^{?}$ ⟧ .

Expr = Numl | Iden | ⟦ Expr Oper Expr ⟧ .

Oper = "+" | "−" | "∗" | "=" .

Numl = ⟦ digit^{+} ('.' digit^{+})$^{?}$ ⟧ .

Iden = ⟦ letter (letter | digit)* ⟧ .

The specification of **grammar:** at the beginning formally abbreviates the specification of our general notation for trees, characters, and strings, together with the introduction of the left hand side symbols of the equations as constants in the signature. This makes the above equations well-formed axioms. Each equation defines the interpretation of a constant to be a particular subsort of tree. In general it can also be useful to have constants standing for subsorts of tree*, for instance Stmts = Stmt (";" Stmt)*.

Observe the following properties of the specification:

- The sort ⟦ Iden ":=" Expr ⟧ includes only individual trees with three branches, the second branch being always the string ":=". In contrast, the sort⟦ "begin" Stmt (";" Stmt)* "end" ⟧ includes trees with unbounded branching.

- The sort ⟦ "if" Expr "then" Stmt ("else" Stmt)$^{?}$ ⟧ is entirely equivalent to the union ⟦ "if" Expr "then" Stmt ⟧ | ⟦ "if" Expr "then" Stmt "else" Stmt ⟧.

- Individuals (e.g., "begin") are mixed with proper sorts (e.g., Stmt) as arguments to the binary sequencing operation _ _ . It would be tedious if one had to use different symbols for an individual and the singleton sort which includes just that individual.

- The sorts Numl and Iden are *subsorts* of Expr, rather than component sorts. Of course if one really wants them as components, all one has to do is to enclose them in ⟦. . . ⟧.

- The sorts Numl and Iden are also subsorts of string. (This would not be the case for Numl is we had used the string "." instead of the character '.'.)

- The sort Iden includes the words "begin", "end", etc. It is in fact quite easy to define a subsort of Iden that is disjoint from such reserved words, since one can specify disjointness of sorts x and y using $x \,\&\, y =$ nothing. Of course the reserved words are a finite set, so the unreserved words are regular and could still be specified without the meet operation available—but the specification would then be *extremely* tedious!

- The 'organization' of the grammar does not affect the structure of the specified trees. For instance, we may replace the alternative ⟦ Expr Oper Expr ⟧ by ⟦ Expr "+" Expr ⟧ | ... | ⟦ Expr "=" Expr ⟧ without changing the semantics of the specification at all.

Assume that we restrict models of the above specification to initial models, using an empty bounded data constraint. This ensures that models only contain finite trees, and that distinct tree terms denote distinct trees—up to associativity and unit laws for the binary sequencing of branches. The class of specified models then gives us the intended abstract syntax. Note that the models contain all trees, with the sorts actually specified in the grammar denoting the expected subsorts.

Notice that the use of strings as 'terminal' components does not decrease the abstractness of our abstract syntax: using arbitrarily-chosen labels to distinguish between nodes with the same nonterminal component sorts instead would give isomorphic models. The strings used above suggest the corresponding terminal symbols that *might* be used in a corresponding concrete syntax; in practical applications, such as semantic descriptions of realistic programming languages, the mnemonic value of the strings can be extremely valuable.

The use of such abstract syntax in semantic descriptions is illustrated in [12]. (A slightly different signature is used there for sequences, requiring that sequence arguments to operations like _* be enclosed in angle brackets $\langle \ldots \rangle$ rather than ordinary parentheses. This avoids the 'invisible' operation symbol _ _, which tends to give rise to ambiguity when used together with action notation in action semantic descriptions.)

The sequencing and *-operations are also extremely convenient for reducing operations with varying numbers of arguments to unary ones. For example, list of _ :: item* → list allows list of $(x_1 \ \ldots \ x_n)$ for constructing a list with the components $x_1, \ldots, x_n$, for any $n \geq 0$.

7 Conclusion

We have specified an algebraic notation for trees and regular expressions, and shown that the axioms have a nontrivial model. This notation allows perspicuous and flexible specifications of abstract syntax for programming languages by extended context-free grammars, as has been illustrated. Terms in our tree notation can be used as sorts when specifying other operations, for instance when using semantic equations to define semantic functions, as in denotational or action semantics. This is made possible by the use of unified algebras, the expressiveness of which is fully exploited when specifying abstract syntax grammars: productions of the grammar are sort equations, and sort constructing operations are applied to mixtures of individuals and proper sorts. The notation for regular expressions can also be useful when specifying abstract data types with operations (such as list construction) that are naturally regarded as applied to ungrouped but ordered collections of arguments.

It may be desirable to generalize the notion of models of bounded data constraints, so that extensionally-equal sorts can be identified in a model even when their equality does not follow directly from the specified axioms in the logic of unified algebras. This would support the view, espoused by the author in [13], that it is really only the individuals themselves that matter, the sorts are there merely to classify the individuals. Sort inclusions are often significant, but sort equalities are usually irrelevant.

Acknowledgments Valentin Antimirov provided many useful comments during the preparation of this paper. The work reported here has been partially funded by the Danish Science Research Council project DART (5.21.08.03).

Appendix

The logic of unified algebras is given by the following definite Horn clauses. (The axioms characterizing distributive lattices could be given purely equationally, with $x \leq y$ being regarded as an abbreviation for $x \mid y = y$.) Recall from Section 2 that $x{:}y$ holds when x is not only included in the sort y but also x is an *individual*.

(1) $a = b \;;\; b = c \;\Rightarrow\; a = c$. $\quad a = b \;\Rightarrow\; b = a$. $\quad a = a$.

(2) $a \leq b \;;\; b \leq c \;\Rightarrow\; a \leq c$. $\quad a \leq a$.
$a \leq b \;;\; b \leq a \;\Rightarrow\; a = b$. $\quad \mathsf{nothing} \leq a$.

(3) $a : a \;;\; a \leq b \;\Rightarrow\; a : b$. $\quad a : b \;\Rightarrow\; a : a$.
$a : b \;\Rightarrow\; a \leq b$. $\quad a : \mathsf{nothing} \;\Rightarrow\; b = c$.

The last axiom above ensures that nothing is vacuous, except in the trivial one-point model. An alternative would be to permit falsity as a consequent in our Horn clauses, and expect initial models to exist only when some model exists.

(4) $a \mid (b \mid c) = (a \mid b) \mid c$. $\quad a \mid b = b \mid a$. $\quad a \mid a = a$.
$a \mid \mathsf{nothing} = a$.
$a \leq c \;;\; b \leq c \;\Rightarrow\; a \mid b \leq c$. $\quad a \leq a \mid b$.

(5) $a \,\&\, (b \,\&\, c) = (a \,\&\, b) \,\&\, c$. $\quad a \,\&\, b = b \,\&\, a$. $\quad a \,\&\, a = a$.
$a \,\&\, \mathsf{nothing} = \mathsf{nothing}$.
$c \leq a \;;\; c \leq b \;\Rightarrow\; c \leq a \,\&\, b$. $\quad a \,\&\, b \leq a$.

(6) $a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$. $\quad a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$.

Furthermore, for each operation symbol f of rank n the following axioms are provided, for $i = 1$ to n:

$$x_i \leq x_i' \;\Rightarrow\; f(\!| x_1, \ldots, x_i, \ldots, x_n |\!) \leq f(\!| x_1, \ldots, x_i', \ldots, x_n |\!) \;.$$

Finally, the inference rules of Horn clause logic with equality are simply: *Modus Ponens* (from the formulae $e_1, \ldots, e_m$ and the clause $e_1; \ldots; e_m \Rightarrow e$ infer e); the substitutivity of terms proved equal; and the instantiation of clauses by substituting terms for variables.

References

[1] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. Technical Monograph DAIMI PB-461, Computer Science Dept., Aarhus University, 1993. A short version is to appear in Proc. Conf. on Developments in Language Theory, ed. A. Salomaa, World Scientific Publ.

[2] D. Bjørner and C. B. Jones, editors. *Formal Specification & Software Development*. Prentice-Hall, 1982.

[3] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24:68–95, 1977.

[4] G. Grätzer. *Lattice Theory: First Concepts and Distributive Lattices*. W. H. Freeman & Co., 1971.

[5] G. Kahn et al. Metal: A formalism to specify formalisms. *Sci. Comput. Programming*, 3:151–188, 1983.

[6] P. Klint. A meta-environment for generating programming environments. In *Algebraic Methods II: Theory, Tools, and Applications*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1991.

[7] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *LICS'91, Proc. 6th Ann. Symp. on Logic in Computer Science*, pages 214–225. IEEE, 1991.

[8] D. Kozen. On action algebras. Technical Monograph DAIMI PB–381, Computer Science Dept., Aarhus University, 1992.

[9] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 62, Proc. IFIP Congress 62*, pages 21–28. North-Holland, 1962.

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

[11] 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.

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

[13] P. D. Mosses. The use of sorts in algebraic specifications. In *Proc. 8th Workshop on Abstract Data Types and 3rd COMPASS Workshop*, volume 655 of *Lecture Notes in Computer Science*, pages 66–91. Springer-Verlag, 1993.

[14] V. Pratt. Action logic and pure induction. In *Logics in AI, Proc. European Workshop JELIA'90*, volume 478 of *Lecture Notes in Computer Science*, pages 97–120. Springer-Verlag, 1990.

[15] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, third edition, 1989.

[16] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn & Bacon, 1986.

[17] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

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

Recent Publications in the BRICS Report Series

RS-94-1 **Glynn Winskel.** ***Semantics, Algorithmics and Logic: Basic Research in Computer Science. BRICS Inaugural Talk*****. February 1994.**

RS-94-2 **Alexander E. Andreev.** ***Complexity of Nondeterministic Functions*****. February 1994.**

RS-94-3 **Uffe H. Engberg and Glynn Winskel.** ***Linear Logic on Petri Nets*****. February 1994.**

RS-94-4 **Nils Klarlund and Michael I. Schwartzbach.** ***Graphs and Decidable Transductions based on Edge Constraints*****. February 1994.**

RS-94-5 **Peter D. Mosses.** ***Unified Algebras and Abstract Syntax*****. March 1994.**