1992

S.M. Eker

A comparison of OBJ3 and ASF+SDF

Computer Science/Department of Software Technology Report CS-R9223 July

A Comparison of OBJ3 and ASF+SDF

S. M. Eker[0]

CWI, P.O.Box 4079, 1009 AB Amsterdam.

Abstract

We contrast the features of two current algebraic specification languages and compare their performance on a suite of benchmark programs.

Key Words & Phrases: Algebraic Specification, Term Rewriting, Benchmarks.
1991 Mathematics Subject Classification: 68Q15 [**Software**]: Programming Languages; 68Q42 [**Theory of Computing**]: Rewriting Systems; 68Q65 [**Theory of Computing**]: Abstract Data Types, Algebraic Specification.
1991 CR Categories: D.3.3 [**Programming Languages**]: Language Constructs and Features.
Note: This work was done while the author was an ERCIM Research Fellow.

1 Introduction

Algebraic specification languages offer a declarative programming pardigm which has simpler semantics than that of higher order functional languages such as for example ML. An algebraic specification language essentially allows the programmer to write down a many sorted signature (i.e. a declaration of sorts, constants and operations) together with a set of (conditional) equations that define the operations. The denotational semantics of a program is given by the initial algebra construction [7, 3]. The operational semantics of a program is given by term rewriting [2]. The simplicity of the semantics facilitates formal reasoning about both the meaning of a program (by equational logic and structural induction) and its runtime behaviour (by termination proofs and confluence checks).

On top of the basic signature plus equations model, practical algebraic specification languages provide extensions to support the construction of large readable programs. Such extensions include modules, user definable syntax and enhanced pattern matching. Two current algebraic specification languages are OBJ3 and ASF+SDF. OBJ3 is the latest in a series of algebraic specification languages developed at SRI [5] and is based on a refinement of many sorted algebra called order sorted algebra [4]. ASF+SDF is the fusion of the Algebraic Specification Formalism and the Syntax Definition Formalism developed at CWI [1]. This report outlines the main differences in the two languages and compares the performance of the currently available implementations.

2 Modularity

Both languages support the hierarchical division of programs into modules with later modules importing earlier ones.

OBJ3 has has two kinds of modules; objects and theories. The distinction between the two kinds is mostly a semantic one—objects denote initial algebras while theories denote varieties. In practice objects are used to hold executable rewrite rules and theories are used to express interfaces between modules. An object may be parameterised by a theory and then instantiated as required by binding a second object to the theory via a view. For example a theory T might contain a sort S and an operation $+ : S \times S \to S$ together with an equation (or attribute) that asserts that $+$ is associative. An object

[0]From 15th July 1992 the author's address will be INRIA-LORRAINE, Campus Scientifique, 615 rue du Jardin Botanique - BP 101, F-54602 VILLERS-LES-NANCY CEDEX, France.

Report CS-R9223
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

L which is parameterised by T which defines lists over S and a summation operation which sums the elements of a list. Now L could be instantiated by binding T to an object defining the natural numbers to get to provide lists of natural numbers together with a summation operator.

As well as instantiation, OBJ3 provides module sums and renaming. The contents of one module may be imported into another module by one of four importation modes. From a semantic point of view the importation modes are distinguished by the effect the importing module has on the imported modules denotational semantics. From a programming point of view the choice of importation mode affects the recomputation of evaluation strategies, the sharing of modules, the generation of completion equations for OBJ3's class rewriting mechanism and the collapsing of identically named and ranked operations. When a module is imported its variables are hidden by default although they can be made visible by using the 'vars-of' construct

In contrast the ASF+SDF modularisation facility is straightforward but unsophisticated. There is just a single kind of module and a single importation mode with no parameterisation. All entities (sorts, lexical syntax, constants, operations and variables) can be optionally exported to or hidden from an importing module.

3 User definable syntax

Both OBJ3 and ASF+SDF provide user definable syntax for operations and constants. In OBJ3 the syntax of an operation is given by a template with the argument positions indicated by underscores; for example:

```
op _+_ : Int Int -> Int .
```

The underscores may be omitted in which case the default 'prefix with arguments in parentheses' is assumed which is convenient when writing large programs. OBJ3 does not currently support user definable syntax at the lexical level; A few single characters are treated as symbols in their own right and all other symbols must be delimited by white space. ASF+SDF allows the user to define both lexical and context free syntax. At the lexical level it is possible to access the individual characters in a multi-character symbol from within a program. At the context-free level the syntax of operations can be defined in a way analogous to that in OBJ3; for example:

```
INT "+" INT -> INT
```

With user definable syntax comes the problem of disambiguating expressions containing infix operations. There are three mechanisms for this—bracketing, priorities and associativity—which are implemented in both languages.

Each OBJ3 operation has a precedence given by a small non-negative integer together with a gathering pattern which determines for each argument position where the position may be filled by a term whose head symbol has lower (or equal) precedence than the operation itself. The precedence and gathering pattern for each operation may be defined explicitly by the user, otherwise sensible defaults are used. Associative operations such as '+' can be declared to be associative using the 'assoc' attribute. Then terms such as $a + b + c$ are no longer ambiguous since they refer to the equivalence class containing $(a + b) + c$ and $a + (b + c)$. Note that as well as disambiguating the concrete syntax the 'assoc' attribute causes matching modulo associativity to be used when rewriting such terms. Finally the precedence and gathering pattern mechanism can be explicitly overridden at any time by the usual convention of grouping subterms by parentheses.

In ASF+SDF the relative priorities of operations are determined by a (possibly empty) strict partial order explicitly given by the user. Associative operations such as '+' can be declared to be either left or right associative to disambiguate terms such as $a + b + c$. It is also possible to declare operations as nonassociative to rule out certain parse trees. Left and right associativity and nonassociativity declarations can also be made for groups of operations. There is also a somewhat complicated notion of extending the partial ordering on operations to parse trees. Finally the priority and associativity mechanisms can be explicitly overridden by bracket functions; for example declaring

```
"(" INT ")" -> INT {bracket}
```

allows the placement of parentheses around expressions of sort INT. Bracket operations only exist in the concrete syntax and do not occur in abstract syntax trees. They are more flexible than OBJ3's parentheses however they have this disadvantage that they must be declared explicitly for each sort.

4 Term rewriting

4.1 Matching

Both languages have extensions to normal term matching which allow more elegant specifications (and in the case of OBJ enhance its potential as a theorem prover) at the cost of worst case exponential time complexity.

OBJ3 allows congruence class rewriting where equivalence classes of terms rather than terms are rewritten. These equivalence classes are formed by the equivalence relation produced by one or more of the axioms for associativity, commutativity and identity. These are specified by attributes attached to each operation symbol. For example the operation + on integers might be declared

```
op _+_ : Int Int -> Int [assoc comm id: 0] .
```

This obtains the effect of having the rules $(x+y)+z = x+(y+z)$, $x+y = y+x$ and $x+0 = x$ usable in either direction. This is very useful for obtaining systems of rules that are confluent on open terms (or terms in which unconstrained constants play the role of free variables). It is also useful for defining associative lists using a symmetric append operation as opposed to asymmetric lists with the more usual 'cons' operation. The class rewriting is implemented by a combination of matching algorithms (which perform term pattern matching modulo a set of equations) and completion (which adds new rules). Since OBJ3 uses order-sorted algebra its matching algorithms have to handle the case where the sort of subterms in a subject are raised to allow a match to take place; for example to allow a rule for manipulating integer expressions to be used to rewrite natural expressions.

ASF+SDF has a rather more restricted facility in the form of built in associative lists. For example

```
{INT ","} *
```

refers to a sort that consists of (possibly empty) lists of integers separated by commas. Non-empty list sorts and list sorts without separators are also available. Variables may range over list sorts and list sorts may be use in the domain but not the range of an operation. List sorts are implemented by a restricted form of associative matching.

4.2 Reduction strategy

Each operation in a OBJ3 program has its own reduction strategy which determines the order in which its arguments are reduced and whether they are reduced before reductions on a term headed by the operation take place. This strategy can be explicitly provided by the user or a default strategy can be computed by the OBJ3 interpreter. The default strategy is usually innermost first. However an optimised strategy may be computed for operations which have a tail recursive rule. Explicit strategies can be useful in avoiding needless reduction of arguments that will never be used. For example the built in 'if-then-else-fi' operation has a strategy which forces evaluation of the first argument and then tries to reduce the term headed by 'if-then-else-fi', which if successful will remove one of the other arguments. There is no defined preference among rules that start with the same head symbol. OBJ3 also provides memoization on a per operation basis which stores the results of previous reductions in a hash table to avoid doing the same reduction twice.

In ASF+SDF the reduction strategy is leftmost innermost first however the interpreter will sometime perform an optimisation to try and avoid evaluating arguments which may not be needed. Some control over which rule will be applied at a subterm is available in the form of default equations which will only be applied after any non-default (conditional) equations have been tried and failed.

4.3 Conditional equations

Both languages provide conditional equations which are implemented by backtracking on failure. However this backtracking is not as general or as powerful as in Prolog because when a condition succeeds the rewrite rule guarded by the condition is applied and no further backtracking above this level can take place.

In OBJ3 the condition may be any expression that evaluates to the built in sort `Bool`. Polymorphic operations for syntactic equality and inequality are provided. Left hand sides which contain operations with attributes can potentially match in many ways and all of these matches are tried until one causes the condition to evaluate to 'true'. All variables occurring in the condition must appear in the left hand side of the rule.

In ASF+SDF the condition consists of a conjunction of equalities and inequalities. Unlike OBJ3 the introduction of new variables not occurring in the left hand side of the rule is allowed in the condition. The equalities and inequalities are evaluated in left to right order. At most one side of an inequality may contain uninstantiated variables. This is a powerful feature since it allows information produced by evaluating the condition to be bound to an uninstantiated variable and then used in the right hand side of the rule. Since operations may not return list sorts it is usual to circumvent this restriction by using inclusion operations from a list sort in a non-list sort which may be returned by an operation. Conditional equations provide the only way of stripping off these inclusion operations to allow terms of the original list sort to be accessed and concatenated.

5 Programming environment

The current version of OBJ3 provides only a scrolling text user interface. Debugging facilities are available in the form of a parse command to check the syntax and typing of an OBJ3 term and rewrite tracing with options to control how much information is shown. There is also a facility to apply rewrite rules one at a time at selected locations in the current term. Access to the underlying Kyoto Common Lisp sytem is available by using 'built in' equations whose right hand sides are lisp expressions. OBJ3 has a 'prelude' which contains definitions for a number of predefined sorts including booleans, naturals, integers, rationals and floating point numbers and these definitions make use of the lisp system for efficiency. An X-Windows based theorem proving environment built on top of two copies of OBJ3 is under development [6].

ASF+SDF provides a X-Windows user interface with a syntax directed editor that may be instantiated for modules and terms as required. There is also a debugger which supports rewrite tracing with break patterns and the option to interrupt tracing whenever a particular rule is applied. Access to the underlying LeLisp system is provided by means of hybrid functions which are implemented by lisp expressions. A compiler to convert ASF+SDF programs into 'C' programs which itself is written in ASF+SDF is under development [8].

6 Performance

This section examines the performance of the term rewrite engines in the current implementations of OBJ3 Release 2.0 and ASF+SDF. The timings in this section were made on a SPARCstation 1 with 16Mb RAM. The OBJ3 timings were obtained using a lisp function supplied by T. Winkler (coauthor and current maintainer of OBJ3). The ASF+SDF timings were obtained using the trace option '(set #:EQM:trace t)'. In both cases the timing refers only to the term rewriting and excludes parsing the input and formatting the output. All timings are in milliseconds and are rounded to the nearest ten milliseconds[1]. It was noticeable on both systems that reductions would occasionally take far longer than usual—presumably due to garbage collection. To minimise this effect each reduction was tried five times and the lowest of the five timings is recorded in the tables below. The number of rewrite steps reported by each system for each reduction is also recorded. When the set of rules contains conditional

[1] Which appears to be the resolution of the ASF+SDF timed trace facility.

equations ASF+SDF returns two values the number of steps actually taken (shown first) and the number of attempts (shown in brackets).

6.1 Factorials

This is a fairly standard test (for example see [9]) of raw rewriting speed. Here the natural numbers are represented in unary notation using the constant zero and a successor function. The functions addition, multiplication and factorial have their usual primitive recursive definitions. Here n in the table refers to the number whose factorial is computed.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
3	26	30	26	10
4	72	100	72	40
5	324	480	324	190

6.2 Naive lists

Lists are the most important data structure in functional and logic programming. The naive approach to algebraically defining the sort L of lists over a sort D is to have a constant *nil* for the empty list a constructor operation with type $D \times L \rightarrow L$ for building new lists.

We consider two operations on naive lists of symbols from $\{a, b, c, d, e\}$. The first of these is the replacement of occurrences of a by b. This can be specified using single operation as the only auxiliary operation used is that of adding a new symbol at the start of a list and that is already available as the list constructor operation. The test data consists of lists from $(ababbcbdbe)^+$. The n in the table refers to the total length of the list.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	37	70	21(25)	10
40	73	150	41(49)	40
60	109	200	61(73)	70
80	145	280	81(97)	100
100	181	370	101(121)	120

The second operation is list reversing. This cannot be done with a single operation of type $L \rightarrow L$. The specification we use here is the obvious one which uses an auxiliary 'append' function for appending an element to the end of a list. It has quadratic running time—an alternative specification of a list reversing operation on naive lists which has linear running time is given in [10]. The test data is as before.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	231	400	231	160
40	861	1550	861	610
60	1891	3450	1891	1330
80	3321	6020	3321	2350
100	5151	9920	5151	3620

6.3 Associative lists

Both OBJ3 and ASF+SDF have provision for associative lists which are symmetric with respect to their head and tail. In OBJ3 associative lists are specified using a constructor operator which appends two lists and which has the associative and identity attributes (the identity element is the empty list). In ASF+SDF lists sorts are part of the language. The operations and test data are as before. The results for replacement on associative lists are:

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	38	3450	21(39)	40
40	74	15880	41(75)	100
60	110	43350	61(111)	160
80	146	93070	81(147)	240
100	182	172850	101(183)	330

With associative lists, reversal may be specified without the need for an auxiliary operation. The results for reversing associative lists are:

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	22	2870	21(22)	20
40	42	11580	41(42)	80
60	62	29480	61(62)	130
80	82	59320	81(82)	190
100	102	104920	101(102)	270

6.4 Non-empty associative lists

To avoid the need for identity completion in OBJ3 we can work with non-empty associative lists. Non-empty list are also available as a built in feature ASF+SDF. However non-empty lists have the disadvantage in an inductive definition with a condition such as the list replacement example, the condition appears in both the induction and basis cases where as with the potentially empty associative lists this condition only appears in the induction case. The operations and test data are as before. The results for replacement on non-empty associative lists are:

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	36	370	20(54)	40
40	72	1950	40(110)	90
60	108	5100	60(166)	160
80	144	10120	80(222)	240
100	180	18580	100(278)	350

The results for reversing non-empty associative lists are:

	OBJ3		ASF+SDF	
n	steps	time	steps	time
20	20	170	20(39)	30
40	40	430	40(79)	70
60	60	850	60(119)	130
80	80	1350	80(159)	220
100	100	2020	100(199)	280

6.5 Merge sorting

Merge sorting is straightforward way of sorting lists. In order to have a large ordered set of items without explicitly defining the ordering on each pair we use the set of nonempty strings $\{a, b\}^+$ and lexicographic order. Non-empty associative lists are used for the strings and naive lists are used for the lists of strings. Notice that in the inductive case of the 'merge' operation we want to avoid having to evaluate the test on the leading strings twice. In OBJ3 this is achieved by using the built in 'if then else fi' operation which evaluates its first argument eagerly and its other two arguments lazily. In ASF+SDF this is achieved by using a conditional equation for the positive branch followed by a default equation for the negative branch. The input data is formed by repetitions of the list of strings

$$abab, babb, abaa, bbbb, bbba, aaab, bbaa, aaaa, aabb, baba$$

The n in the table refers to the number of strings in the list.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
10	228	980	194(462)	260
20	593	2620	506(1406)	700
30	1032	4750	880(2521)	1220
40	1482	7430	1266(3774)	1790
50	1962	9580	1677(5104))	2410

6.6 Towers of Hanoi

This is a well known problem with an elegant recursive solution. Although it would be more natural to use associative lists to build the solution we used naive lists with a auxiliary concatenation operations in both OBJ3 and ASF+SDF version because as we saw above long associatives lists rewrite inefficiently in OBJ3. We also avoid using OBJ3's built in numbers and subsorts as these have no direct equivalents in ASF+SDF. The n in the table refers to the number of disks to be moved.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
3	62	80	55(132)	40
4	138	200	123(284)	110
5	298	450	267(644)	260
6	634	950	571(1280)	500
7	1338	1950	1211(2792)	1090
8	2810	4230	2555(5460)	2200

6.7 Missionaries and Cannibals

This is a well known problem of finding the shortest sequence of moves for getting a group of missionaries and cannibals across a river in a two man boat, subject to the constraint that the number of cannibals must never out number missionaries on either bank of the river (unless there are no missionaries on that bank). The solution we use is a straight forward depth first search where we keep track of the shortest non-failure solution to date. To avoid an infinite loop we maintain a list of positions visited so far in each branch of the search tree. We use a list to hold the moves taken and we keep the set of ten potential moves in a constant list. All lists are naive lists to avoid inefficient matching with OBJ3. The n in the table refers to the number of pairs of missionaries and cannibals.

	OBJ3		ASF+SDF	
n	steps	time	steps	time
1	2210	5480	1281(1852)	980
2	24943	63130	13775(21101)	12790
3	36953	87400	20485(33076)	20040

Acknowledgement

I thank T. B. Dinesh and Paul Klint for their comments on drafts of this report.

References

[1] The ASF+SDF Meta-environment User's Guide, 1992. Draft.

[2] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[3] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1 - Equations and Initial Semantics.* EATCS Monograph Series Vol 6, Springer-Verlag, Springer-Verlag.

[4] Joseph Goguen and José Meseguer. ORDER SORTED ALGEBRA I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions, and Partial Orderings. Technical Report SRI-CSL-89-10, Computer Science Laboratory, SRI, 1989.

[5] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ3. In J. A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ.* Cambridge University Press, (to appear).

[6] J. A. Goguen et al. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society,* A(339), 1992.

[7] ADJ (J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright). An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in Programming Methodology: IV Data Structuring,* pages 80–149. Prentice-Hall, Englewood Cliffs, 1978.

[8] J. F. Th. Kamperman and H. R. Walters. Efficient execution of algebraic specifications: A self-fulfilling prophecy, (in preparation).

[9] Stephane Kaplan. A compiler for conditional term rewriting systems. In Pierre Lescanne, editor, *Rewriting Techniques and Applications '87,* Lecture Notes in Computer Science 256, pages 25–41. Springer-Verlag, 1987.

[10] Michael O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

A Benchmark programs

A.1 Factorials in OBJ3

```
obj FACTORIAL is
    sort Nat .
    op 0 : -> Nat .
    op s : Nat -> Nat .
    op _+_ : Nat Nat -> Nat [prec 3] .
    op _*_ : Nat Nat -> Nat [prec 2] .
    op _! : Nat -> Nat [prec 1] .

var N M : Nat .
    eq 0 + M = M .
    eq s(N) + M = s(N + M) .
    eq 0 * M = 0 .
    eq s(N) * M = N * M + M .
    eq 0 ! = s(0) .
    eq s(N) ! = s(N) * N ! .
endo
```

A.2 Factorials in ASF+SDF

```
%%      FACTORIAL

exports
    sorts NAT
    lexical syntax
        [ \t\n]              -> LAYOUT
        "0"                  -> NAT

    context-free syntax
        s "(" NAT ")"      -> NAT
        NAT "+" NAT        -> NAT
        NAT "*" NAT        -> NAT
        NAT "!"            -> NAT

    priorities
        "!" > "*" > "+"

hiddens
    variables
        [NM]               -> NAT

equations
    [1] 0 + M = M
    [2] s(N) + M = s(N + M)
    [3] 0 * M = 0
    [4] s(N) * M = N * M + M
    [5] 0! = s(0)
    [6] s(N)! = s(N) * N!
```

A.3 Naive lists in OBJ3

```
obj LIST is
    sorts Item List .
    ops a b c d e : -> Item .
    op nil : -> List .
    op _|_ : Item List -> List .
endo

obj REPLACE is
    protecting LIST .
    op repl : List -> List .

var H : Item .
var T : List .
    eq repl(a | T) = b | repl(T) .
    cq repl(H | T) = H | repl(T) if H =/= a .
    eq repl(nil) = nil .
endo

obj REVERSE is
    protecting LIST .
    op rev : List -> List .
    op append : List Item -> List .

var H I : Item .
var T : List .
    eq rev(H | T) = append(rev(T), H) .
    eq rev(nil) = nil .
    eq append(nil, I) = I | nil .
    eq append(H | T, I) = H | append(T, I) .
endo
```

A.4 Naive lists in ASF+SDF

```
%%      LIST

exports
    sorts
        ITEM LIST

    lexical syntax
        [ \t\n]                 -> LAYOUT
        [a-e]                   -> ITEM

    context-free syntax
        nil                     -> LIST
        ITEM "|" LIST           -> LIST

%%      REPLACE

imports
    LIST

exports
    context-free syntax
        repl "(" LIST ")"           -> LIST

hiddens
    variables
        H   -> ITEM
        T   -> LIST


equations
    [1] repl(a | T) = b | repl(T)
    [2] H != a ====> repl(H | T) = H | repl(T)
    [3] repl(nil) = nil

%%      REVERSE

imports
    LIST

exports
    context-free syntax
        rev "(" LIST ")"                -> LIST
        append "(" LIST "," ITEM ")"    -> LIST

hiddens
    variables
        [HI]  -> ITEM
        T     -> LIST
```

```
equations
    [1] rev(H | T) = append(rev(T), H)
    [2] rev(nil) = nil
    [3] append(nil, I) = I | nil
    [4] append(H | T, I) = H | append(T, I)
```

A.5 Associative lists in OBJ3

```
obj LIST is
    sorts Item List .
    subsort Item < List .
    ops a b c d e : -> Item .
    op nil : -> List .
    op _|_ : List List -> List [assoc id: nil] .
endo

obj REPLACE is
    protecting LIST .
    op repl : List -> List .

var H : Item .
var T : List .
    eq repl(a | T) = b | repl(T) .
    cq repl(H | T) = H | repl(T) if H =/= a .
    eq repl(nil) = nil .
endo

obj REVERSE is
    protecting LIST .
    op rev : List -> List .

var H : Item .
var T : List .
    eq rev(H | T) = rev(T) | H .
    eq rev(nil) = nil .
endo
```

A.6 Associative lists in ASF+SDF

```
%%      LIST

exports
    sorts
        ITEM LIST

    lexical syntax
        [ \t\n]                 -> LAYOUT
        [a-e]                   -> ITEM

    context-free syntax
        {ITEM "|"}*             -> LIST

%%      REPLACE
```

```
imports
    LIST

exports
    context-free syntax
        repl "(" LIST ")"                  -> LIST

hiddens
    variables
        H     -> ITEM
        [TL]  -> {ITEM "|"}*


equations
    [1] repl(T) = L ====> repl(a | T) = b | L

    [2] H != a,
        repl(T) = L
        ==============
        repl(H | T) = H | L

    [3] repl() =

%%      REVERSE

imports
    LIST

exports
    context-free syntax
        rev "(" LIST ")"                   -> LIST
        append "(" LIST "," ITEM ")"       -> LIST

hiddens
    variables
        H     -> ITEM
        [TL]  -> {ITEM "|"}*


equations
    [1] rev(T) = L ====> rev(H | T) = L | H
    [2] rev() =
```

A.7 Non-empty associative lists in OBJ3

```
obj LIST is
    sorts Item List .
    subsort Item < List .
    ops a b c d e : -> Item .
    op _|_ : List List -> List [assoc] .
endo

obj REPLACE is
    protecting LIST .
    op repl : List -> List .
```

```
var H : Item .
var T : List .
    eq repl(a | T) = b | repl(T) .
    cq repl(H | T) = H | repl(T) if H =/= a .
    eq repl(a) = b .
    cq repl(H) = H if H =/= a .
endo

obj REVERSE is
    protecting LIST .
    op rev : List -> List .

var H : Item .
var T : List .
    eq rev(H | T) = rev(T) | H .
    eq rev(H) = H .
endo
```

A.8 Non-empty associative lists in ASF+SDF

```
%%      LIST

exports
    sorts
        ITEM LIST

    lexical syntax
        [ \t\n]                -> LAYOUT
        [a-e]                  -> ITEM

    context-free syntax
        {ITEM "|"}+            -> LIST

%%      REPLACE

imports
    LIST

exports
    context-free syntax
        repl "(" LIST ")"                -> LIST

hiddens
    variables
        H     -> ITEM
        [TL]  -> {ITEM "|"}+


equations
    [1] repl(T) = L ====> repl(a | T) = b | L

    [2] H != a,
        repl(T) = L
        ==============
        repl(H | T) = H | L

    [3] repl(a) = b
    [4] H != a ====> repl(H) = H

%%      REVERSE

imports
    LIST

exports
    context-free syntax
        rev "(" LIST ")"                 -> LIST
        append "(" LIST "," ITEM ")"   -> LIST

hiddens
    variables
        H     -> ITEM
        [TL]  -> {ITEM "|"}+


equations
    [1] rev(T) = L ====> rev(H | T) = L | H
    [2] rev(H) = H
```

A.9 Mergesort in OBJ3

```
obj STRG is
    sorts Elt Strg .
    subsort Elt < Strg .
    ops a b : -> Elt .
    op __ : Strg Strg -> Strg [assoc] .
    op _>=_ : Strg Strg -> Bool .

vars E E2 : Elt .
vars S S2 : Strg .
    eq b >= a = true .
    eq a >= b = false .
    eq E >= E = true .

    eq E S >= E2 = E >= E2 .
    eq E >= E S2 = false .
    cq E >= E2 S2 = E >= E2 if E =/= E2 .

    eq E S >= E S2 = S >= S2 .
    cq E S >= E2 S2 = E >= E2 if E =/= E2 .
endo

obj MERGE is
    protecting STRG .
    sort List .
    op nil : -> List .
    op _|_ : Strg List -> List .
    op merge : List List -> List .
    ops sort odd even : List -> List .
```

```
vars S S2 : Strg .
vars L L2 : List .
    eq merge(nil, L) = L .
    eq merge(L, nil) = L .
    eq merge(S | L, S2 | L2) = if S >= S2 then
                                    S2 | merge(S | L, L2)
                                else
                                    S | merge(L, S2 | L2)
                                fi .
    eq odd(nil) = nil .
    eq odd(S | nil) = S | nil .
    eq odd(S | S2 | L) = S | odd(L) .
    eq even(nil) = nil .
    eq even(S | nil) = nil .
    eq even(S | S2 | L) = S2 | even(L) .
    eq sort(nil) = nil .
    eq sort(S | nil) = S | nil .
    eq sort(S | S2 | L) = merge(sort(odd(S | S2 | L)),
                                sort(even(S | S2 | L))) .
endo
```

A.10 Mergesort in ASF+SDF

```
%%      STRING

exports
    sorts
        BOOL ELT STRING

    lexical syntax
        [ \t\n]             -> LAYOUT
        [ab]                -> ELT

    context-free syntax
        true                -> BOOL
        false               -> BOOL
        ELT+ ">=" ELT+      -> BOOL
        "(" ELT+ ")"        -> STRING
        STRING ">=" STRING  -> BOOL
hiddens
    variables
        E [0-9]*            -> ELT
        S [0-9]*            -> ELT+


equations
    [S1] b >= a = true
    [S2] a >= b = false
    [S3] E >= E = true

    [S4] E S >= E2 = E >= E2
    [S5] E >= E S2 = false
    [S6] E != E2 ====> E >= E2 S2 = E >= E2
    [S7] E S >= E S2 = S >= S2
    [S8] E != E2 ====> E S >= E2 S2 = E >= E2

    [S9] (S) >= (S2) = S >= S2

%%      MERGE

imports
    STRING

exports
    sorts
        LIST

    context-free syntax
        nil                                     -> LIST
        STRING "|" LIST                         -> LIST
        sort "(" LIST ")"                       -> LIST
        merge "(" LIST "," LIST ")"             -> LIST
        odd "(" LIST ")"                        -> LIST
        even "(" LIST ")"                       -> LIST

hiddens
    variables
        S [0-9]*     -> STRING
        L [0-9]*     -> LIST


equations
    [M1] merge(nil, L) = L
    [M2] merge(L, nil) = L
    [M3] S >= S2 = true ====> merge(S | L, S2 | L2) = S2 | merge(S | L, L2)
[default-M4] merge(S | L, S2 | L2) = S | merge(L, S2 | L2)
    [M5] odd(nil) = nil
    [M6] odd(S | nil) = S | nil
    [M7] odd(S | S2 | L) = S | odd(L)
    [M8] even(nil) = nil
    [M9] even(S | nil) = nil
    [M10] even(S | S2 | L) = S2 | even(L)
    [M11] sort(nil) = nil
    [M12] sort(S | nil) = S | nil
    [M13] sort(S | S2 | L) = merge(sort(odd(S | S2 | L)),
                                   sort(even(S | S2 | L)))
```

A.11 Towers of Hanoi in OBJ3

```
obj DISK is
    sorts Dnum .
    ops 0 1 2 3 4 5 6 7 8 : -> Dnum .
    op dec : Dnum -> Dnum .

    eq dec(8) = 7 .  eq dec(7) = 6 .  eq dec(6) = 5 .  eq dec(5) = 4 .
    eq dec(4) = 3 .  eq dec(3) = 2 .  eq dec(2) = 1 .  eq dec(1) = 0 .
endo
```

```
obj TOWER is
    sort Tower .
    ops a b c : -> Tower .
    op other : Tower Tower -> Tower .

    eq other(a, b) = c .     eq other(b, a) = c .
    eq other(a, c) = b .     eq other(c, a) = b .
    eq other(b, c) = a .     eq other(c, b) = a .
endo

obj MOVE is
    protecting DISK + TOWER .
    sorts Move List .
    op move_from_to_ : Dnum Tower Tower -> Move .
    op nil : -> List .
    op _|_ : Move List -> List .
    op conc : List List -> List .

var H : Move .
vars T L : List .
    eq conc(nil, L) = L .
    eq conc(L, nil) = L .
    eq conc(H | T, L) = H | conc(T, L) .
endo

obj SOLVE is
    protecting MOVE .
    op solve : Tower Tower Dnum -> List .

var ORG DEST : Tower .
var D : Dnum .
    eq solve(ORG, DEST, 0) = nil .
    cq solve(ORG, DEST, D) = conc(
            solve(ORG, other(ORG, DEST), dec(D)),
            (move D from ORG to DEST) |
                solve(other(ORG, DEST), DEST, dec(D)) ) if D =/= 0 .
endo
```

A.12 Towers of Hanoi in ASF+SDF

```
%%      DISK

exports
    sorts
        DNUM

    lexical syntax
        [ \t\n]                 -> LAYOUT
        [012345678]             -> DNUM

    context-free syntax
        dec "(" DNUM ")"        -> DNUM


equations
```

```
    [1] dec(8) = 7        [2] dec(7) = 6
    [3] dec(6) = 5        [4] dec(5) = 4
    [5] dec(4) = 3        [6] dec(3) = 2
    [7] dec(2) = 1        [8] dec(1) = 0

%%      MOVE

imports
    DISK TOWER

exports
    sorts
        MOVE LIST

    context-free syntax
        move DNUM from TOWER "to" TOWER     -> MOVE
        nil                                 -> LIST
        MOVE "|" LIST                       -> LIST
        conc "(" LIST "," LIST ")"          -> LIST

hiddens
    variables
        H     -> MOVE
        [TL]  -> LIST


equations
    [1] conc(nil, L) = L
    [2] conc(L, nil) = L
    [3] conc(H | T, L) = H | conc(T, L)

%%      TOWER

exports
    sorts
        TOWER

    lexical syntax
        [ \r\n]                   -> LAYOUT
        [abc]                     -> TOWER

    context-free syntax
        other "(" TOWER "," TOWER ")"   -> TOWER


equations
    [1] other(a, b) = c      [2] other(b, a) = c
    [3] other(a, c) = b      [4] other(c, a) = b
    [5] other(b, c) = a      [6] other(c, b) = a

%%      SOLVE

imports
    MOVE
```

```
exports
    context-free syntax
        solve "(" TOWER "," TOWER "," DNUM ")" -> LIST
hiddens
    variables
        ORG     -> TOWER
        DEST    -> TOWER
        D       -> DNUM


equations
    [1] solve(ORG, DEST, 0) = nil
    [2] D != 0 ====> solve(ORG, DEST, D) = conc(
                          solve(ORG, other(ORG, DEST), dec(D)),
                          move D from ORG to DEST |
                              solve(other(ORG, DEST), DEST, dec(D)) )
```

A.13 Missionaries and Cannibals in OBJ3

```
obj INT is
    sort Int .
    ops 0 1 2 -1 -2 3 : -> Int .
    ops s p : Int -> Int .
    op _+_ : Int Int -> Int .
    op _>=_ : Int Int -> Bool .

vars X Y : Int .
    eq s(p(X)) = X .
    eq p(s(X)) = X .
    eq 0 + Y = Y .
    eq s(X) + Y = s(X + Y) .
    eq p(X) + Y = p(X + Y) .
    eq 0 >= 0 = true .
    eq 0 >= s(0) = false .
    eq s(X) >= Y = X >= p(Y) .
    eq p(X) >= Y = X >= s(Y) .
    cq 0 >= p(Y) = true if 0 >= Y == true .
    cq 0 >= s(Y) = false if 0 >= Y == false .
    eq 1 = s(0) .
    eq 2 = s(s(0)) .
    eq -1 = p(0) .
    eq -2 = p(p(0)) .
    eq 3 = s(s(s(0))) .
endo

obj PROBLEM is
    protecting INT .
    sorts Bank Text Position Move MoveList .
    ops west east : -> Bank .
    ops (missionary rows east) (two missionaries row east)
        (missionary and cannibal row east)
        (cannibal rows east) (two cannibals row east)
        (missionary rows west) (two missionaries row west)
        (missionary and cannibal row west)
        (cannibal rows west) (two cannibals row west) : -> Text .
    op {_,_,_,_,_} : Bank Int Int Int Int -> Position .
    op {_,_,_,_,_,_} : Text Int Int Int Int Bank -> Move .
    ops nil moves : -> MoveList .
    op _|_ : Move MoveList -> MoveList .

    eq moves =
        {missionary rows east, -1, 0, 1, 0, east} |
        {two missionaries row east, -2, 0, 2, 0, east} |
        {missionary and cannibal row east, -1, -1, 1, 1, east} |
        {cannibal rows east, 0, -1, 0, 1, east} |
        {two cannibals row east, 0, -2, 0, 2, east} |
        {missionary rows west, 1, 0, -1, 0, west} |
        {two missionaries row west, 2, 0, -2, 0, west} |
        {missionary and cannibal row west, 1, 1, -1, -1, west} |
        {cannibal rows west, 0, 1, 0, -1, west} |
        {two cannibals row west, 0, 2, 0, -2, west} | nil .
endo

obj SOLVE is
    protecting PROBLEM .
    sorts TextList PosList .
    op nil : -> PosList .
    op _|_ : Position PosList -> PosList .
    op member : Position PosList -> Bool .
    ops nil fail : -> TextList .
    op _|_ : Text TextList -> TextList .
    op _>=_ : TextList TextList -> Bool .
    op solve : Position Position PosList -> TextList .
    op try : MoveList Position Position PosList -> TextList .
    op apply : Move Position Position PosList -> TextList .
    op check : Text Position Position PosList -> TextList .
    op valid : Position -> Bool .
    op best : TextList TextList -> TextList .

var NP Pos Final : Position .
var Prev : PosList .
var T T2 : Text .
var TL TL2 : TextList .
var B D : Bank .
var DMW DCW DME DCE MW CW ME CE : Int .
var M : Move .
var ML : MoveList .
    eq member(NP, nil) = false .
    eq member(NP, NP | Prev) = true .
    cq member(NP, Pos | Prev) = member(NP, Prev) if NP =/= Pos .
    eq T | fail = fail .
    eq TL >= nil = true .
    eq nil >= (T | TL) = false .
    eq (T | TL) >= (T2 | TL2) = TL >= TL2 .
    eq best(fail, TL) = TL .
    eq best(TL, fail) = TL .
    cq best(TL, TL2) = (if TL >= TL2 then TL2 else TL fi)
        if TL =/= fail and TL2 =/= fail .
    eq solve(Pos, Pos, Prev) = nil .
    cq solve(Pos, Final, Prev) = try(moves, Pos, Final, Prev)
        if Pos =/= Final .
    eq try(nil, Pos, Final, Prev) = fail .
    eq try(M | ML, Pos, Final, Prev) =
        best(apply(M, Pos, Final, Prev), try(ML, Pos, Final, Prev)) .
    eq apply({T, DMW, DCW, DME, DCE, D}, {B, MW, CW, ME, CE}, Final, Prev) =
        if D =/= B then
            check(T, {D, MW + DMW, CW + DCW, ME + DME, CE + DCE}, Final, Prev)
        else
            fail
        fi .
    eq check(T, NP, Final, Prev) =
        if member(NP, Prev) == false and valid(NP) then
            T | solve(NP, Final, NP | Prev)
        else
            fail
        fi .
    eq valid({B, MW, CW, ME, CE}) =
        MW >= 0 and CW >= 0 and ME >= 0 and CE >= 0 and
        (MW == 0 or MW >= CW) and (ME == 0 or ME >= CE) .
endo
```

A.14 Missionaries and Cannibals in ASF+SDF

```
%%      BOOL

exports
    sorts BOOL
    lexical syntax
        [ \t\n] -> LAYOUT
    context-free syntax
        true -> BOOL    false -> BOOL
        BOOL and BOOL -> BOOL {left}
        BOOL or BOOL -> BOOL {left}
        "(" BOOL ")" -> BOOL {bracket}

hiddens
    variables
        B -> BOOL


equations
    [B1] true and B = B
    [B2] false and B = false
    [B3] true or B = true
    [B4] false or B = B

%%      INT

imports
    BOOL

exports
    sorts INT
    context-free syntax
```

```
        "0" -> INT    "1" -> INT    "2" -> INT
        "-1" -> INT    "-2" -> INT    "3" -> INT
        s "(" INT ")" -> INT
        p "(" INT ")" -> INT
        INT "+" INT -> INT {left}
        INT ">=" INT -> BOOL
        zero "(" INT ")" -> BOOL


hiddens
    variables
        X -> INT    Y -> INT


equations
    [I1] s(p(X)) = X
    [I2] p(s(X)) = X
    [I3] 0 + Y = Y
    [I4] s(X) + Y = s(X + Y)
    [I5] p(X) + Y = p(X + Y)
    [I6] 0 >= 0 = true
    [I7] 0 >= s(0) = false
    [I8] s(X) >= Y = X >= p(Y)
    [I9] p(X) >= Y = X >= s(Y)
    [I10] 0 >= Y = true ====> 0 >= p(Y) = true
    [I11] 0 >= Y = false ====> 0 >= s(Y) = false
    [I12] 1 = s(0)
    [I13] 2 = s(s(0))
    [I14] -1 = p(0)
    [I15] -2 = p(p(0))
    [I16] 3 = s(s(s(0)))
    [I17] zero(0) = true
    [I18] X != 0 ====> zero(X) = false

%%      PROBLEM

imports
    INT

exports
    sorts
        BANK TEXT POSITION MOVE MOVELIST
    context-free syntax
        west -> BANK    east -> BANK
        "missionary rows east" -> TEXT    "two missionaries row east" -> TEXT
        "missionary and cannibal row east" -> TEXT
        "cannibal rows east" -> TEXT      "two cannibals row east" -> TEXT
        "missionary rows west" -> TEXT    "two missionaries row west" -> TEXT
        "missionary and cannibal row west" -> TEXT
        "cannibal rows west" -> TEXT      "two cannibals row west" -> TEXT
        "{" BANK "," INT "," INT "," INT "," INT "}"            -> POSITION
        "{" TEXT "," INT "," INT "," INT "," INT "," BANK "}"   -> MOVE
        nil                  -> MOVELIST
        MOVE "|" MOVELIST    -> MOVELIST
        moves                -> MOVELIST
```

```
equations
    [M0] moves = {missionary rows east, -1, 0, 1, 0, east} |
                 {two missionaries row east, -2, 0, 2, 0, east} |
                 {missionary and cannibal row east, -1, -1, 1, 1, east} |
                 {cannibal rows east, 0, -1, 0, 1, east} |
                 {two cannibals row east, 0, -2, 0, 2, east} |
                 {missionary rows west, 1, 0, -1, 0, west} |
                 {two missionaries row west, 2, 0, -2, 0, west} |
                 {missionary and cannibal row west, 1, 1, -1, -1, west} |
                 {cannibal rows west, 0, 1, 0, -1, west} |
                 {two cannibals row west, 0, 2, 0, -2, west} | nil

%%      SOLVE

imports
    PROBLEM

exports
    sorts
        TEXTLIST POSLIST

    context-free syntax
        nil                    -> POSLIST
        POSITION "|" POSLIST -> POSLIST
        member "(" POSITION "," POSLIST ")" -> BOOL
        nil                  -> TEXTLIST
        fail                 -> TEXTLIST
        TEXT "|" TEXTLIST  -> TEXTLIST
        TEXTLIST ">=" TEXTLIST  -> BOOL
        solve "(" POSITION "," POSITION "," POSLIST ")"               -> TEXTLIST
        try "(" MOVELIST "," POSITION "," POSITION "," POSLIST ")"  -> TEXTLIST
        apply "(" MOVE "," POSITION "," POSITION "," POSLIST ")"    -> TEXTLIST
        best "(" TEXTLIST "," TEXTLIST ")"                          -> TEXTLIST
        valid "(" POSITION ")" -> BOOL

hiddens
    variables
        NP -> POSITION    Pos -> POSITION    Final -> POSITION
        Prev -> POSLIST
        T [0-9]* -> TEXT
        TL [0-9]* -> TEXTLIST
        B -> BANK    D -> BANK
        D [MC] [WE] -> INT    [MC] [WE] -> INT
        M -> MOVE
        ML -> MOVELIST


equations
    [M0] member(NP, nil) = false
    [M1] member(NP, NP | Prev) = true
    [M1] NP != Pos ====> member(NP, Pos | Prev) = member(NP, Prev)
    [L0] T | fail = fail
    [L1] TL >= nil = true
    [L2] nil >= T | TL = false
    [L3] T | TL >= T2 | TL2 = TL >= TL2
    [B0] best(fail, TL) = TL
    [B1] best(TL, fail) = TL

    [B2] TL != fail, TL2 != fail, TL >= TL2 = true
         =========================================
         best(TL, TL2) = TL2

[default-B3] best(TL, TL2) = TL
    [S0] solve(Pos, Pos, Prev) = nil

    [S1] Pos != Final
         =======================================================
         solve(Pos, Final, Prev) = try(moves, Pos, Final, Prev)

    [S2] try(nil, Pos, Final, Prev) = fail
    [S3] try(M | ML, Pos, Final, Prev) =
             best(apply(M, Pos, Final, Prev), try(ML, Pos, Final, Prev))

    [S4] D != B, NP = {D, MW + DMW, CW + DCW, ME + DME, CE + DCE},
         member(NP, Prev) = false, valid(NP) = true
         ==========================================================
         apply({T, DMW, DCW, DME, DCE, D}, {B, MW, CW, ME, CE}, Final, Prev) =
             T | solve(NP, Final, NP | Prev)

[default-S5] apply({T, DMW, DCW, DME, DCE, D},
                {B, MW, CW, ME, CE}, Final, Prev) = fail

    [S6] valid({B, MW, CW, ME, CE}) =
             MW >= 0 and CW >= 0 and ME >= 0 and CE >= 0 and
             (zero(MW) or MW >= CW) and (zero(ME) or ME >= CE)
```