Compiling Language Definitions: The ASF+SDF Compiler

M. G. J. VAN DEN BRAND
CWI and Vrije Universiteit
J. HEERING
CWI
P. KLINT
CWI and University of Amsterdam
and
P. A. OLIVIER
CWI

The ASF+SDF Meta-Environment is an interactive language development environment whose main application areas are definition and implementation of domain-specific languages, generation of program analysis and transformation tools, and production of software renovation tools. It uses conditional rewrite rules to define the dynamic semantics and other tool-oriented aspects of languages, so the effectiveness of the generated tools is critically dependent on the quality of the rewrite rule implementation.

The ASF+SDF rewrite rule compiler generates C code, thus taking advantage of C's portability and the sophisticated optimization capabilities of current C compilers as well as avoiding potential abstract machine interface bottlenecks. It can handle large (10 000+ rule) language definitions and uses an efficient run-time storage scheme capable of handling large (1 000 000+ node) terms. Term storage uses maximal subterm sharing (hash-consing), which turns out to be more effective in the case of ASF+SDF than in Lisp or SML. Extensive benchmarking has shown the time and space performance of the generated code to be as good as or better than that of the best current rewrite rule and functional language compilers.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; F.4.2 [**Mathematical Logic and Formal Languages**]: Rewriting Systems

1. INTRODUCTION

ASF+SDF [Bergstra et al. 1989; van Deursen et al. 1996] is the metalanguage of the ASF+SDF Meta-Environment [Klint 1993; van den Brand et al. 2001], an interactive environment for the development of languages and language-oriented tools, covering parsing, typechecking, translation, transformation, and execution of programs.

ASF+SDF is a modular specification formalism based on the Algebraic Specification Formalism ASF and the Syntax Definition Formalism SDF. The latter is a BNF-like formalism for defining the lexical, context-free and abstract syntax of languages [Heering et al. 1989; Visser 1997], featuring a close integration of lexical and context-free syntax. The implementation of SDF is beyond the scope of this article. Suffice it to say, it supports fully general context-free parsing without a separate lexical scanning phase.

The ASF component of ASF+SDF uses rewrite rules to describe the semantics of languages. Such semantics may be static (typechecking) or dynamic. The latter may have an interpretive or translational character, it may include program transformations, and so on. These are all described in terms of rewrite rules whose left- and right-hand sides are sentences in the language defined by the SDF-part of the language definition.

Rewriting is the simplification of algebraic expressions or terms everybody is familiar with. It is ubiquitous in (computer) algebra as well as in algebraic semantics and algebraic specification. Rewriting is also important in functional programming, program transformation and optimization, and equational theorem proving. Useful theoretical surveys of rewriting are [Klop 1992; Dershowitz and Jouannaud 1990], but we assume only a basic understanding of rewrite systems on the part of the reader. In addition to regular rewrite rules, ASF+SDF features conditional rewrite rules, associative (flat) lists, and default rules. These will be explained below.

ASF+SDF's current application areas are

—Definition and implementation of domain-specific languages

—Generation of program analysis and transformation tools

—Production of software renovation tools.

Table I gives details and further references. Another application area is definition and implementation of general purpose programming languages, but we have accumulated relatively little experience in this area so far.

Each ASF+SDF module defines the syntax and semantics of a language or language fragment, ranging from the simple language of boolean expressions or expressions involving type environments to (part of) Cobol or Java. Correspondingly, the semantics may range from the simple evaluation of boolean expressions or expressions involving type environments to the restructuring of Cobol programs, the typechecking of Java programs, or the execution behavior of a domain-specific or general purpose language.

The contributions of the ASF+SDF formalism and its implementation are twofold:

—A seamless integration of syntax and semantics. The rewrite rules defining the semantics may use both the concrete syntax of the language to be defined as

---

Domain-Specific Languages

—Risla [van Deursen and Klint 1998] (financial product specification)
—EURIS [Groote et al. 1995] (railroad safety)
—Action Semantics [van Deursen 1994] (programming language semantics)
—Manifold [Rutten and Thiébaux 1992], ToolBus [Bergstra and Klint 1998] (coordination languages)
—ALMA-0 [Apt et al. 1998] (backtracking and search)
—Languages of the ASF+SDF Meta-Environment itself [van den Brand et al. 2001]:
  —SDF (syntax definition)
  —Box (prettyprinting specification)
  —ASF+SDF (language definition—this article)

Program Analysis

—Typechecking of Pascal [van Deursen et al. 1996, Chapter 2]
—Typechecking and execution of CLaX [Tip and Dinesh 2001]
—μCRL [Hillebrand 1996] (proof checking and simulation toolkit)
—Dahl [Moonen 1997] (dataflow analysis framework)
—Type inference, object identification, and documentation generation for Cobol [van Deursen and Moonen 2000]

Program Transformation

—Interactive program transformation for Clean [van den Brand et al. 1995] and Prolog [Brunekreef 1996]
—PIM [Field 1992] (compiler toolkit)
—Automatic program transformation for C++ [Dinesh et al. 2001]

Software Renovation

—Description of the multiplicity of languages and dialects encountered in software renovation applications such as Cobol (including embedded languages like SQL and CICS) [van Deursen et al. 1999]
—Automatic program transformation for restructuring of Cobol programs (including embedded languages like SQL and CICS) [van den Brand et al. 2000]
—Extraction of grammars from compilers and on-line manuals [Sellink and Verhoef 2000]

Table I.   Current application areas of the ASF+SDF Meta-Environment.

well as user-defined concrete syntax for semantic operations. Both are covered by SDF.
—A uniform approach to the definition of data types with user-defined notation and languages with operations like typechecking, execution, and transformation.

ASF+SDF is more expressive than attribute grammars, which it includes as the subclass of definitions that are non-circular primitive recursive schemes (NPRSs) [Courcelle and Franchi-Zannettacci 1982]. This is the natural style for most type-checkers and translators. Using this correspondence, van der Meulen [1996] has transferred incremental evaluation methods originally developed for attribute grammars to NPRS-style ASF+SDF definitions.

We describe the current ASF+SDF compiler and compare its performance with that of other rewrite system and functional language compilers we were able to run, namely, Clean [Plasmeijer and van Eekelen 1994; Smetsers et al. 1991], Elan [Kirchner and Moreau 2001], Haskell [Peyton Jones et al. 1993; Peyton Jones 1996], Maude [Clavel et al. 1999], Opal [Didrich et al. 1994], and SML [Appel 1992].

The effectiveness of the tools generated by the ASF+SDF Meta-Environment is critically dependent on the quality of the rewriting implementation. The original interpretive implementation left room for improvement. Its author, inspired by earlier rewrite compilation work of Kaplan [1987], sketched a more efficient compilational scheme [Dik 1989] that ultimately served as a basis for the compiler described in this article.

The real-world character of ASF+SDF applications has important consequences for the compiler:

—It must be able to handle ASF+SDF definitions of up to 50 000 lines. Disregarding layout and syntax declarations (SDF-parts), this corresponds to 10 000 (conditional) rewrite rules.

—It must include optimizations for the major sources of inefficiency encountered in practice.

This article is organized as follows: brief survey of the ASF+SDF language (Sec. 2); general compilation scheme (Sec. 3); major design considerations for the ASF+SDF compiler (Sec. 4); the μASF abstract syntax representation (Sec. 5); preprocessing (Sec. 6); code generation (Sec. 7); postprocessing (Sec. 8); benchmarking (Sec. 9); conclusions and further work (Sec. 10). Related work is discussed at appropriate points throughout the text rather than in a separate section.

2. BRIEF SURVEY OF THE ASF+SDF LANGUAGE

In addition to regular rewrite rules, ASF+SDF features conditional rewrite rules, associative (flat) lists, default rules, and simple modularization. In our discussion of these features we will emphasize issues affecting their compilation rather than issues of language design. For the use of ASF+SDF see [van Deursen et al. 1996].

2.1 Syntax Definitions

An ASF+SDF module can define arbitrary lexical and context-free syntax. An example of the former is shown in Fig. 1.[1] It defines sort ID for identifiers using a regular expression involving character classes. It imports module Layout, which defines lexical syntax for white space, comments, etc. (not shown). Furthermore, again using a regular expression, it defines a set of variables of sort ID. All definitions (including the variable declarations) are exported from the module, meaning they are available in modules importing it.

A simple context-free syntax for statements is shown in Fig. 2. It imports modules Identifiers and Expressions (not shown). In this way, it obtains definitions for sorts ID and EXP. It then defines sort STAT for single statements and STATS for

---

[1]To bring out the correspondence with later phases of the compilation process, we present specifications as typed in by the specification writer rather than in the typeset form produced by the ASF+SDF Meta-Environment.

```
module Identifiers

imports Layout
exports
  sorts ID
  lexical syntax
    [a-z] [a-z0-9]* -> ID
  variables
    "Id" [0-9\']*   -> ID
```

Fig. 1.   Definition of identifiers in ASF+SDF.

```
module Statements

imports Identifiers Expressions
exports
  sorts STAT STATS
  context-free syntax
    ID ":=" EXP                -> STAT
    "if" EXP "then" STATS "fi"  -> STAT
    "while" EXP "do" STATS "od" -> STAT
    {STAT ";"}+                -> STATS

  variables
    "Stat"[0-9\']*             -> STAT
```

Fig. 2.   Definition of statements in ASF+SDF.

```
module Types

imports Layout
exports
  sorts TYPE
  context-free syntax
    "int"        -> TYPE
    "real"       -> TYPE
    "string"     -> TYPE
    "nil-type"   -> TYPE

  variables
    "Type" [0-9\'] -> TYPE
```

Fig. 3.   Definition of type constants in ASF+SDF.

lists of statements. List constructs like {STAT ";"}+ denote *separator lists*. In this case, a list of statements consists of one or more statements separated by semicolons (but no semicolon at the end).

2.2 Conditional Rewrite Rules

We assume throughout that the terms being rewritten are *ground terms*, i.e., terms without variables. A rule is applicable to a subterm if its left-hand side matches the subterm, and its *conditions* (if any) succeed after substitution of the values found during matching. Such a subterm is called a *redex* for that particular rule. The process of exhaustively rewriting a term is called *normalization*. The resulting term is called a *normal form* (if normalization terminates). Conditions may be *positive* (equalities) or *negative* (inequalities).

Negative conditions succeed if both sides are syntactically different after normalization. Otherwise they fail. They are not allowed to contain variables not already occurring in the left-hand side of the rule or in a preceding positive condition. This means both sides of a negative condition are ground terms at the time the condition is evaluated.

Positive conditions succeed if both sides are syntactically equal after normalization. Otherwise they fail. One side of a positive condition may contain one or more new variables not already occurring in the left-hand side of the rule or in a preceding positive condition. This means one side of a positive condition need not be a ground term at the time it is evaluated, but may contain existentially quantified variables. Their value is obtained by matching the side they occur in with the other side after the latter has been normalized. The side containing the variables is not normalized before matching.

Variables occurring in the right-hand side of the rule must occur in the left-hand side or in a positive condition, so the right-hand side is a ground term at the time it is substituted for the redex.

As running example we will use a definition of the "language" of type environments (Fig. 4). From the viewpoint of ASF+SDF this is just a (small) language definition. As explained in Sec. 1, ASF+SDF does not distinguish data types with user-defined notation from language definitions with operations like typechecking, execution, and transformation.

Module `Type-environments` defines a type environment (sort `TENV`) as a list of pairs, where each pair (sort `PAIR`) consists of an identifier (sort `ID`) and a type (sort `TYPE`). The latter is defined in module `Types` (Fig. 3), which is imported by `Type-environments`. It defines bracket notations for pairs and lists of pairs as well as appropriate distfix notation for the operations `lookup` and `add`. A sample sentence of the type environment language would be

```
add d with real to [(a:int),(b:real),(c:string)].
```

The semantics of the language is defined by rewrite rules for the operations `lookup` and `add`. Consider rule `[at-2]` in Fig. 4 keeping the above in mind. Its application proceeds as follows:

(1) Find a redex matching the left-hand side of the rule (if any). This yields values

```
module Type-environments

imports Identifiers Types

exports
  sorts TENV PAIR
  context-free syntax
    "(" ID ":" TYPE ")"          -> PAIR
    "[" {PAIR ","}* "]"          -> TENV

    "lookup" ID "in" TENV        -> TYPE
    "add" ID "with" TYPE "to" TENV -> TENV

  variables
    "Pair" [0-9]*                -> PAIR
    "Pairs" [0-9]*               -> {PAIR ","}*
    "Tenv" [0-9]*                -> TENV

equations
  [l-1]  lookup Id in [Pairs1, (Id:Type), Pairs2] = Type

  [default-l-2]
        lookup Id in Tenv = nil-type

  [at-1] add Id with Type1 to [(Id:Type2), Pairs] = [(Id:Type1), Pairs]

  [at-2]                           Id1 != Id2,
               add Id1 with Type1 to [Pairs1] = [Pairs2]
        ================================================================
        add Id1 with Type1 to [(Id2:Type2), Pairs1] = [(Id2:Type2), Pairs2]

  [at-3] add Id with Type to [] = [(Id:Type)]
```

Fig. 4.   Definition of a simple type environment in ASF+SDF.

for the variables Id1, Type1, Id2, Type2, and Pairs1.

(2) Evaluate the first condition. This amounts to a simple syntactic inequality check of the two identifiers picked up in step 1. If the condition succeeds, evaluate the second one. Otherwise, the rule does not apply.

(3) Evaluate the second condition. This is a positive condition containing the new list variable Pairs2 in its right-hand side. The value of Pairs2 is obtained by matching the right-hand side with the normalized left-hand side. Since Pairs2 is a list variable, this involves list matching, which is explained below. In this particular case, the match always succeeds.

(4) Finally, replace the redex with the right-hand side of the rule after substituting the values of Id2 and Type2 found in step 1 and the value of Pairs2 found in step 3.

2.3   Lists

ASF+SDF lists are associative (flat) and list matching is the same as string matching. Unlike a term pattern, a list pattern may match a redex in more than one way. This may lead to backtracking within the scope of the rule containing the list pattern in the following two closely related cases:

—A rewrite rule containing a list pattern in its left-hand side might use conditions to select an appropriate match from the various possibilities.

—A rewrite rule containing a list pattern with new variables in a positive condition (Sec. 2.2) might use additional conditions to select an appropriate match from the various possibilities.

List matching may be used to avoid the explicit traversal of structures. Rule [l-1] in Fig. 4 illustrates this. It does not traverse the type environment explicitly, but picks an occurrence (if any) of the identifier it is looking for using two list variables `Pairs1` and `Pairs2` to match its context. The actual traversal code is generated by the compiler. In general, however, there is a price to be paid. While term matching is linear, string matching is NP-complete [Benanav et al. 1985]. Hence, list matching is NP-complete as well. It remains an important source of inefficiency in the execution of ASF+SDF definitions [Vinju 1999].

2.4   Default Rules

A default rule has lower priority than ordinary rules in the sense that it can be applied to a redex only if all ordinary rules are exhausted. In Fig. 4, `lookup` uses default rule [default-l-2] to return `nil-type` if rule [l-1] fails to find the identifier it is looking for.

2.5   Modules

ASF+SDF supports *import*, *renaming*, and *parameterization*. Renaming corresponds to replacing a syntax rule with another one and replacing the corresponding textual instances. Modularization is eliminated at the preprocessing level. An ASF+SDF function definition may be distributed over several modules. Since the compiler maps ASF+SDF functions to C functions, this hampers separate compilation. The full specification has to be scanned for each function.

2.6   Rewriting Strategies

ASF+SDF is a strict language based on innermost rewriting (call-by-value). This facilitates compilation to and interfacing with C and other imperative languages. In particular, it allows ASF+SDF functions to be mapped directly to C functions and intermediate results produced during term rewriting to be stored in an efficient way (Sec. 7.1). We also encountered cases (conditionals, for instance) where innermost rewriting proved unsatisfactory. In such cases, rewriting of specific function arguments can be delayed by annotating them with the `delay` attribute. See [Bergstra and van den Brand 2000] for details.

3.   GENERAL COMPILATION SCHEME

Before we discuss the major design issues, it is useful for the reader to understand the general layout of the compiler as shown in Fig. 5. The following compiler phases
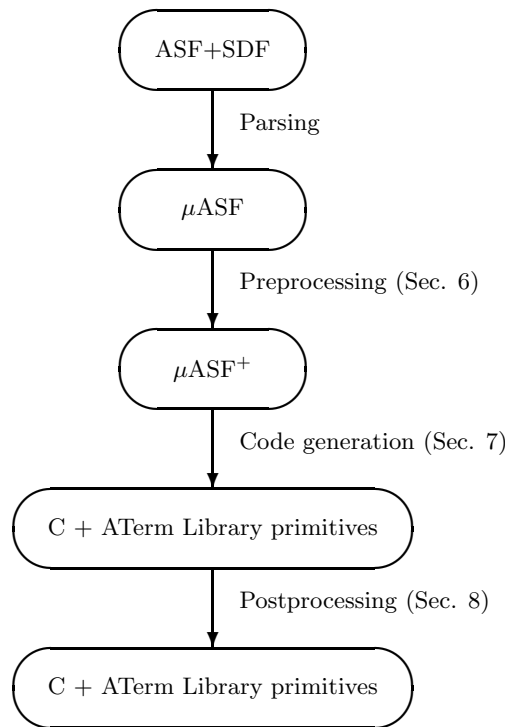
Fig. 5. General layout of the ASF+SDF compiler.

can be distinguished:

—Parsing. Since the syntax of ASF+SDF-definitions is largely defined by their SDF-part, parsing them is a nontrivial two-pass process, whose details are beyond the scope of this article. Suffice it to say, this phase yields an abstract syntax representation of the input definition as usual. As indicated in the second box from the top, the parser's output formalism is μASF, an abstract syntax version of ASF+SDF.

—Preprocessing. This is performed on the μASF representation, which is very close to the source level. Typical examples are detection of variable bindings (assignments) in conditions and introduction of elses for pairs of conditional rewrite rules with identical left-hand sides and complementary conditions. The output formalism of this phase is μASF$^+$, a superset of μASF.

—Code generation. The compiler generates C extended with calls to the *ATerm library*, a run-time library for term manipulation and storage. Each μASF function is compiled to a separate C function. The right-hand side of a rewrite rule is translated directly to function calls if necessary. Term matching is compiled to a finite automaton. List matching code depends on the complexity of the pattern involved. A few special list patterns that do not need backtracking are eliminated by transforming them to equivalent term patterns in the preprocessing phase, but

the majority is compiled to special code.

—Postprocessing. This is performed on the C code generated in the previous phase. A typical example is constant caching.

4. MAJOR DESIGN CONSIDERATIONS

The design of the compiler was influenced by the experience gained in previous compiler activities within the ASF+SDF project itself [Dik 1989; Fokkink et al. 1998; Hendriks 1991; Kamperman 1996] as well as in various functional language and Prolog compiler projects elsewhere. The surveys [Hartel et al. 1996] on functional language compilation and [van Roy 1993] on Prolog compilation were particularly helpful.

4.1 Choice of C as Target Language

Generating C code is an efficient way to achieve portability as well as interoperability with C programs. Folk wisdom has it that C code is 2–3 times slower than native code, but this is not borne out by the "Pseudoknot" benchmark results reported in [Hartel et al. 1996, Table 9], where the best functional language and rewrite system compilers generate C code. The probable reason is that many C compilers perform sophisticated optimizations [Muchnick 1997], although this raises the issue of tuning the generated C code to the optimizations done by different C compilers. At least in our case, the fact that C is in some respects less than ideal as a compiler target [Peyton Jones et al. 1998] does not invalidate these favorable observations.

4.2 Choice of ASF+SDF as Implementation Language

Not unexpectedly in view of its application domain, large parts of the compiler can be expressed very naturally in ASF+SDF, so it was decided to write the compiler in its own source language. Since the compiler is fairly large, self-compilation is an interesting benchmark.

4.3 Pitfalls in High-Level Transformations and Abstract Machine Interfaces—The Bottleneck Effect

High-level transformations have to be applied with extreme care, especially if their purpose is to simplify the compiler by reducing the number of different constructs that have to be handled later on. For instance, by first transforming conditional rewrite rules to unconditional ones or associative list matching to term matching, the compiler can be simplified considerably, but at the expense of a serious degradation in the performance of the generated code. Similarly, transformation of default rules (which can be applied only when all other rules fail) to sets of ordinary rewrite rules that catch the same cases would lead to very inefficient code. These transformations would perhaps be appropriate in a formal semantics of ASF+SDF, but in a compiler they cause a bottleneck whose effect is hard to undo at a later stage.

Since it would require a high-level transformation phase of the above kind, the compiler does not generate code for the Abstract Rewrite Machine (ARM) [Fokkink et al. 1998], which was developed especially for use in rewrite system compilers. In fact, any fixed abstract machine interface is a potential bottleneck in the compilation process. The modularization advantage gained by introducing it may be offset by a serious loss in opportunities for generating efficient code. This happens when,

in the words of Franz [1994, Sec. 2], "the code generator effectively needs to reconstruct at considerable expense, information that was more easily accessible in the front-end, but lost in the transition to the intermediate representation."

The factors involved in the use of an abstract machine have a qualitatively different character. The abstract machine interface facilitates *construction* and *verification* of the compiler, but possibly at the expense of the *performance of the generated code*. See also the discussion in [van Roy 1993, Sec. 2.4] on the pros and cons of the use of the Warren Abstract Machine (WAM) in Prolog compilers. Although the bottleneck effect is hard to describe in quantitative terms, it has to be taken seriously, the more so since the elegance of the abstract machine approach is not conducive to a thorough analysis of its performance in terms of overall compiler quality.

Of course, C also acts as an abstract machine interface, but, compared with ARM or other abstract machines, it is much less specialized and more flexible, acting proportionally less as a bottleneck. The compiler does not simply generate C, however, but C extended with calls to the ATerm library, a run-time library for term manipulation and storage (Sec. 7.1). C cannot be changed, but the ATerm library can be adapted to prevent it from becoming an obstacle to further code improvement, should the need arise. We note, however, that the fact that the ATerm library interface is made available as an API to users outside the compiler makes it harder to adapt.

Although we feel these to be useful guidelines, they have to be applied with care. Their validity is not absolute, but depends on many details of the actual implementation under consideration. The compiler for the lazy functional language Clean [Plasmeijer and van Eekelen 1994; Smetsers et al. 1991], for instance, generates native code via an abstract graph rewriting machine, contravening several of our guidelines. Nevertheless, our benchmarks (Sec. 9) show the Clean compiler and the ASF+SDF compiler to generate code with comparable performance.

4.4 Organization of Term Storage

ASF+SDF applications may involve rewriting of large terms ($> 10^6$ nodes). Usually, this requires constructing and matching many intermediate results and the proper organization of term storage becomes critical to the run-time performance of the term datatype provided by the ATerm library and, as a consequence, to the run-time performance of the generated code as a whole. Fortunately, intermediate results created during rewriting tend to have a lot of overlap. This suggests use of a space saving scheme where terms are created only when they do not yet exist. The various trade-offs involved in this choice are discussed in Sec. 7.1.

5. THE μASF ABSTRACT SYNTAX REPRESENTATION

μASF is the abstract syntax representation (prefix notation only) of ASF+SDF produced by the parsing phase (Fig. 5). As such, it is never written by the user. A semantics by example of μASF, which helped to answer the questions that emerged while the compiler was being written, is given in [Bergstra and van den Brand 2000].

The μASF representation of the simple type environment of Fig. 4 after textual expansion of its imports is shown in Fig. 6. There are some points to be noted. The

```
module Type-environments
signature
  int
  real
  string
  nil-type
  pair(_,_)
  type-env(_)
  lookup(_,_);
  add-to(_,_,_)
rules

[l-1]  lookup(Id,type-env(conc(*Pairs1,conc(pair(Id,Type),*Pairs2))))
            = Type;

[l-2]  default: lookup(Id,Tenv)
            = nil-type;

[at-1] add-to(Id,Type1,type-env(conc(pair(Id,Type2),*Pairs1)))
            = type-env(conc(pair(Id,Type1),*Pairs1));

[at-2] Id1 != Id2 &
       add-to(Id1,Type1,type-env(*Pairs1)) == type-env(*Pairs2)
         ==>
       add-to(Id1,Type1,type-env(conc(pair(Id2,Type2),*Pairs1)))
            = type-env(conc(pair(Id2,Type2),*Pairs2));

[at-3] add-to(Id,Type,type-env(null))
            = type-env(list(pair(Id,Type)))
```

Fig. 6. μASF version of the simple type environment of Fig. 4.

functions and constants used in the rules are declared in the signature section with their argument positions (if any) indicated by underscores. Although ASF+SDF is a many-sorted formalism, the sorts can be dispensed with after parsing and conversion to μASF. The predefined list constructors `list` (conversion to single element list), `conc` (associative list concatenation), and `null` (the empty list) need not be declared.

Symbols starting with a capital are variables. These need not be declared in the signature. List variables are prefixed with a "*" if they can match the empty list or with a "+" if they cannot. The predefined symbols used in the rules are listed in Table II. The =, ==, and != operators have higher precedence than & and ==>.

6. PREPROCESSING

Fig. 7 is a refinement of Fig. 5 showing the preprocessing steps as well as other actions performed in later phases of the compiler. The output language of the preprocessing phase is μASF^{+}, which is μASF with the additional constructs shown
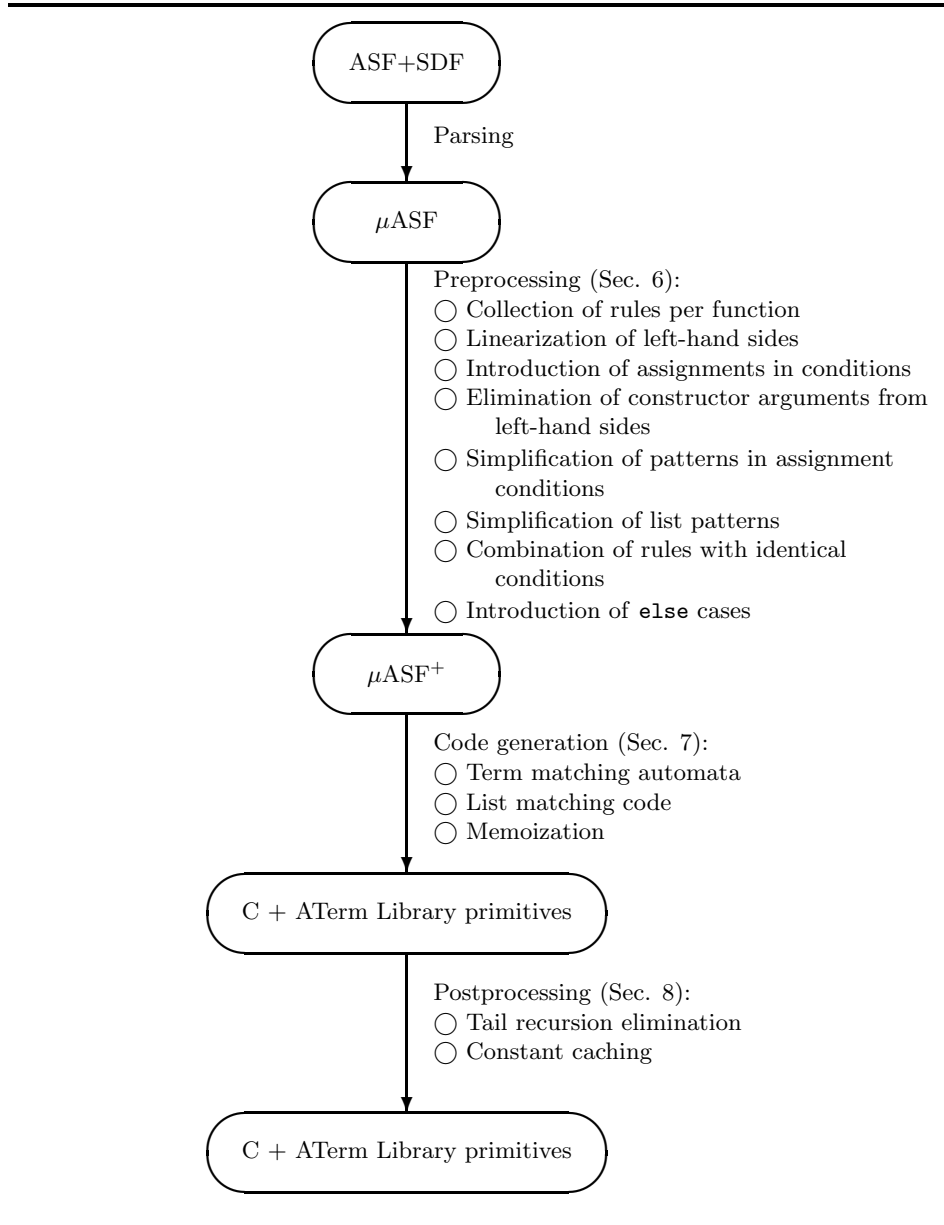
Fig. 7.   Layout of the ASF+SDF compiler. This is a refinement of Figure 5.

`=`	left-to-right rewrite
`==`	equality in positive condition
`!=`	inequality in negative condition
`&`	conjunction of conditions
`==>`	implication
`default:`	default rule flag
`list`	conversion to single element list
`conc`	associative list concatenation
`null`	empty list

Table II.   The predefined symbols used in μASF rewrite rules.

`:=`	assignment
`{ }`	nesting of rules
`else`	alternative
`list_head`	first element of list
`list_tail`	tail of list
`list_last`	last element of list
`list_prefix`	prefix of list
`not_empty_list`	list-not-empty predicate
`t, f`	true, false

Table III.   Additional predefined symbols of μASF$^+$.

in Table III. Their purpose will become clear later on when the preprocessing (Sec. 6) and code generation (Sec. 7) are discussed. Some of them, like nested rules and the `else`-construct, are similar to constructs available in functional languages and might very well be added to ASF+SDF, but this remains to be done.

We now discuss the main preprocessing steps in more detail. As noted in Sec. 4.3, they have to be chosen judiciously to prevent them from becoming counterproductive. Each step has to preserve the innermost rewriting strategy[2] as well as the backtracking behavior of list matching.

6.1 Collection of Rules per Function

An ASF+SDF function definition may be distributed over several modules (Sec. 2.5). The preprocessing phase starts by traversing the top module for which code has to be generated and all modules directly and indirectly imported by it, collecting the rewrite rules for each function declared in its signature, i.e., the rules whose left-hand side has the function as its outermost symbol. Functions without any such rules are marked as constructors in the μASF$^+$ intermediate representation.

6.2 Linearization of Left-Hand Sides

A rewrite rule is *non-linear* if its left-hand side contains more than one occurrence of the same variable. Different occurrences of the same variable have to obtain the same value during matching, so non-linearity amounts to an implicit equality check. Non-linearities are eliminated by adding appropriate positive conditions.

---

[2]Function arguments annotated with the `delay` attribute (Sec. 2.6) have to be taken into account as well, but will be ignored in this article for the sake of readability.

Innermost rewriting guarantees that these conditions do not cause spurious rewrite steps not done by the original non-linear match.

For example, rules [l-1] and [at-1] in Fig. 6 are non-linear since variable Id occurs twice in their left-hand side. Rule [at-1] would be transformed into

```
[at-1'] Id == Id1
          ==>
        add-to(Id,Type1,type-env(conc(pair(Id1,Type2),*Pairs1)))
              = type-env(conc(pair(Id,Type1),*Pairs1))
```

with new variable Id1 not already occurring in the original rule, and similarly for [l-1].

Linearization simplifies the matching automaton and enables further transformations, especially the introduction of elses if there is a corresponding rule with a negative condition as is often the case (see below). The condition is implemented very efficiently as a pointer equality check as will be explained in Sec. 7.1.

6.3 Introduction of Assignments in Conditions

As explained in Sec. 2.2, one side of a positive condition may contain variables that are uninstantiated at the time the condition is evaluated. A value is assigned to them by matching the side they occur in with the other side after the latter has been normalized. The side containing the uninstantiated variables is not normalized before matching. To flag this case to the code generation phase, the μASF equality is replaced by the μASF$^+$ assignment. If necessary, the left- and right-hand side of the original condition are interchanged.

Rule [at-2] in Fig. 6 is of this kind since its second condition contains the new list variable *Pairs2. It would be transformed into

```
[at-2'] Id1 != Id2 &
        type-env(*Pairs2) := add-to(Id1,Type1,type-env(*Pairs1))
          ==>
        add-to(Id1,Type1,type-env(conc(pair(Id2,Type2),*Pairs1)))
              = type-env(conc(pair(Id2,Type2),*Pairs2)).
```

6.4 Simplification of List Patterns

To simplify the generation of list matching code, list patterns in the left-hand side of a rule or an assignment are brought in a standard form containing, apart from the list constructors list and conc, only variables and constants. Other more complicated subpatterns are replaced by new variables that are evaluated in new assignment conditions. This transformation preserves the backtracking behavior of list matching.

Rule [at-1'], for example, will be transformed into

```
[at-1''] pair(Id1,Type2) := P &
         Id == Id1
           ==>
         add-to(Id,Type1,type-env(conc(P,*Pairs1)))
               = type-env(conc(pair(Id,Type1),*Pairs1))
```

and similarly for [at-2'] and [l-1].

List matching may cause backtracking, but list patterns containing only a single list variable or no list variables at all never do. In such cases, list matching can be

eliminated using the μASF$^+$ list functions in Table III. For example, `[at-1'']` is transformed into

```
[at-1'''] t := non_empty_list(*Pairs) &
          P := list_head(*Pairs) &
          *Pairs1 := list_tail(*Pairs) &
          pair(Id1,Type2) := P &
          Id == Id1
            ==>
          add-to(Id,Type1,type-env(*Pairs))
                = type-env(conc(pair(Id,Type1),*Pairs1)),
```

where `t` is the boolean value `true` (Table III), and similarly for `[at-2'']`.

6.5   Combination of Rules with Identical Conditions

Rules `[at-1''']` and `[at-2''']` resulting from the previous step have their left-hand side and first four conditions in common (up to renaming of variables). By factoring out the common elements after a suitable renaming of variables, they can be combined into the single nested rule

```
[at-1-2]  t := non_empty_list(*Pairs) &
          P := list_head(*Pairs) &
          *Pairs1 := list_tail(*Pairs) &
          pair(Id1,Type2) := P
            ==>
          add-to(Id,Type1,type-env(*Pairs)) =
          {
          Id == Id1
            ==>
          type-env(conc(pair(Id,Type1),*Pairs1));

          Id != Id1 &
          type-env(*Pairs2) := add-to(Id,Type1,type-env(*Pairs1))
            ==>
          type-env(conc(pair(Id1,Type2),*Pairs2))
          },
```

where the accolades are in μASF$^+$. The depth of nesting produced in this way may be arbitrarily large.

6.6   Introduction of `else` Cases

μASF$^+$ provides an `else` construct which is used to combine pairs of conditional rewrite rules with identical left-hand sides (up to renaming of variables) and complementary conditions. Introducing it in the result of the previous step yields

```
[at-1-2'] t := non_empty_list(*Pairs) &
          P := list_head(*Pairs) &
          *Pairs1 := list_tail(*Pairs) &
          pair(Id1,Type2) := P
            ==>
          add-to(Id,Type1,type-env(*Pairs)) =
          {
          Id == Id1
```

`term_equal(t1,t2)`	Check if terms `t1` and `t2` are equal
`make_list(t)`	Create list with `t` as single element
`conc(l1,l2)`	Concatenate lists `l1` and `l2`
`insert(t,l)`	Insert term `t` in front of list `l`
`null()`	Create empty list
`list_head(l)`	Get head of list `l`
`list_tail(l)`	Get tail of list `l`
`list_last(l)`	Get last element of list `l`
`list_prefix(l)`	Get prefix of list `l`
`not_empty_list(l)`	Check if list `l` is empty
`is_single_element(l)`	Check if list `l` has a single element
`slice(p1, p2)`	Take slice of list starting at pointer `p1` and ending at `p2`
`check_sym(t,s)`	Check if term `t` has outermost symbol `s`
`arg_i(t)`	Get `i`-th argument
`make_nfi(s,t0,...,ti-1)`	Construct normal form with outermost symbol `s` and arguments `t0,...,ti-1`

Table IV.   Selected ATerm library functions.

```
  ==>
type-env(conc(pair(Id,Type1),*Pairs1))
  else
type-env(*Pairs2) := add-to(Id,Type1,type-env(*Pairs1))
  ==>
type-env(conc(pair(Id1,Type2),*Pairs2))
}.
```

7. CODE GENERATION

7.1   The ATerm Library

7.1.1   *Introduction.* The compiler generates C extended with calls to the ATerm library, a run-time library for term manipulation and storage. In this section we discuss the ATerm library from the perspective of the compiler. For a broader viewpoint and further applications see [van den Brand et al. 1999; van den Brand et al. 2000].

Selected ATerm library functions are listed in Table IV. Many of them correspond directly to predefined symbols of μASF (Table II) and μASF$^+$ (Table III). Examples of actual code using them is given in Sec. 7.2 and Sec. 7.3.

7.1.2   *Term Storage.* The decision to store terms uniquely, is a major factor in the good run-time performance of the code generated by the compiler. If a term to be constructed during rewriting already exists, it is reused, thus guaranteeing maximal sharing. This strategy exploits the redundancy typically present in the terms built during rewriting. The sharing is transparent, so the compiler does not have to take precautions during code generation.

Maximal sharing of terms can only be maintained if the term construction functions `make_nf0`, `make_nf1`, ... (Table IV) check whether the term to be constructed already exists. This implies a search through all existing terms which must be very fast in order not to impose an unacceptable penalty on term construction. Using a hash function depending on the internal code of the function symbol and the

addresses of its arguments, `make_nf`i can quickly search for a function application before constructing it. Hence, apart from the space overhead caused by the initial allocation of a hash table of sufficient size,[3] the modest (but not negligible) time overhead at term construction time is one hash table lookup.

We get two returns on this investment:

—*Reduced memory usage.* The amount of space gained by sharing terms is usually much larger than the space used by the hash table. This is useful in itself, but it also yields a substantial reduction in (real-time) execution time.

—*Fast equality check.* Since terms are stored uniquely, `term_equal`, the equality check on terms, only has to check for pointer equality rather than structural equality. The compiler generates calls to `term_equal` in the pattern matching and condition evaluation code. For the same reason, this storage scheme combines very well with memoization (Sec. 7.4).

It turns out the increased term construction time is more than compensated for by fast equality checking and less use of space (and hence time).

7.1.3 *Shared Terms vs. Destructive Updates.* Shared terms cannot be modified without causing unpredictable side-effects, the more so since the ATerm library is not only used by compiler generated code but also by other components of the ASF+SDF Meta-Environment. Destructive updates would therefore cause unwanted side-effects throughout the system.

During rewriting by compiler generated code the immutability of terms causes no efficiency problems since they are created in a non-destructive way as a consequence of the innermost reduction strategy. Normal forms are constructed bottom-up and there is no need to perform destructive updates on a term once it has been constructed. Also, during normalization the input term itself is not modified but the normal form is constructed separately. Modification of the input term would result in graph rewriting instead of (innermost) term rewriting.

List operations like concatenation and slicing may become expensive, however, if they cannot simply modify one of their arguments. List concatenation, for instance, can only be performed using ATerm library primitives by taking the second list, successively prepending the elements of the first list to it, and returning the new list as a result.

The idea of subterm sharing is known in the Lisp community as *hash-consing* [Allen 1978]. Its success has been limited by the existence of the Lisp functions `rplaca` and `rplacd`, which modify a list destructively. HLisp (Hash Lisp) is a Lisp dialect supporting hash-consing at the language level [Terashima and Kanada 1990]. It has two kinds of list structures: "monocopy" lists with maximal sharing and "multicopy" lists without maximal sharing. Before a destructive change is made to a monocopy list, it has to be converted to a multicopy list.

ASF+SDF does not have functions like `rplaca` and `rplacd`, and the ATerm library only supports the equivalent of HLisp monocopy lists. Although the availability of destructive updates would make the code for some list operations more efficient, such cases are relatively rare. This explains why the technique of subterm sharing can be applied more successfully in ASF+SDF than in Lisp.

---

[3]Hash table overflow is not fatal, but causes allocation of a larger table followed by rehashing.

7.1.4   *Garbage Collection.* During rewriting, a large number of intermediate results is created, most of which will not be part of the end result and have to be reclaimed. There are basically three realistic alternatives for this. We will discuss their advantages and disadvantages in relation to the ATerm library. For an in-depth discussion of garbage collection in general and these three alternatives in particular, we refer the reader to Jones and Lins [1996].

Since ATerms do not contain cycles, reference counting is an obvious alternative to consider. Two problems make it unattractive, however. First and most important, there is no portable way in C to detect when local variables are no longer in use without help from the programmer. Second, the memory overhead of reference counting is large. Most ATerms can be stored in a few machine words, and it would be a waste of memory to add another word solely for the purpose of reference counting.

The other two alternatives are mark-compact and mark-sweep garbage collection. The choice of C as an implementation language is not compatible with mark-compact garbage collection since there is no portable and at the same time reliable way in C to find all local variables on the stack without help from the programmer. This means pointers to ATerms on the stack cannot be made to point to the new location of the corresponding terms after compactification. The usual solution is to "freeze" all objects that might be referenced from the stack, and only relocate objects that are not. Not being able to move all terms negates many of the advantages of mark-compact garbage collection such as decreased fragmentation and fast allocation.

The best alternative turns out to be mark-sweep garbage collection. It can be implemented efficiently in C, both in time and space, and with little or no support from the programmer [Boehm 1993]. We implemented this garbage collector from scratch, with many of the underlying ideas taken directly from Boehm's garbage collector, but tailored to the special characteristics of ATerms both to obtain better control over the garbage collection process as well as for reasons of efficiency.

Starting with the former, ATerms are always referenced from a hash table, even if they are no longer in use. Hence, the garbage collector should not scan this table for references. We also need enough control to remove an ATerm from the hash table when it is freed, otherwise the table would quickly fill up with unused term references.

As for efficiency, experience shows that typically very few ATerms are referenced from static variables or from generic datastructures on the heap. By providing a mechanism (`ATprotect`) to enable the user of the ATerm library to register references to ATerms that are not local (auto) variables, we are able to completely eliminate the expensive scan of the static data area and the heap.

We also have the advantage that almost all ATerms can be stored using only a few words of memory. This makes it convenient to base the algorithm used on only a small number of block sizes compared to a generic garbage collector that cannot make any assumptions about the sizes of the memory chunks that will be requested at run-time.

7.1.5   *Discussion.* Our positive experience with hash-consing in ASF+SDF refutes the theoretical arguments against its potential usefulness in the equational

programming language Epic mentioned by Fokkink et al. [1998, p. 701]. Also, while our experience seems to be at variance with observations made by Appel and Gonçalves [1993] in the context of SML, where sharing resulted in only slightly better execution speed and marginal space savings, both sharing schemes are actually rather different. In our scheme, terms are shared immediately at the time they are created, whereas Appel and Gonçalves delay the sharing of subterms until the next garbage collection. This minimizes the overhead at term construction time, but at the same time sacrifices the benefits (space savings and a fast equality test) of sharing terms that have not yet survived a garbage collection. The different usage patterns of terms in SML and ASF+SDF may also contribute to these seemingly contradictory observations.

7.2 Matching

7.2.1 *Term Matching.* After collecting the rules making up a function definition (Sec. 6.1), the compiler transforms their left-hand sides into a deterministic finite automaton that controls the matching of the function call at run-time, an approach originally due to Hoffmann and O'Donnell [1982]. In this way, each generated C function gets its own local matching automaton.

The semantics of ASF+SDF does not prescribe a particular way to resolve ambiguous matches, i.e., more than a single left-hand side matching the same innermost redex, so the compiler is free to choose a suitable disambiguation strategy. To obtain a deterministic matching automaton it uses the specificity order defined in [Fokkink et al. 1998, Definition 2.2.1]. Rewrite rules with more specific left-hand sides take precedence over rules whose left-hand sides are more general. Default rules correspond to "otherwise" cases in the automaton.

In the generated C code the matching automata are often hard to distinguish from the conditions of conditional rules, especially since the latter may have been generated in the preprocessing phase by the compiler itself to linearize or simplify left-hand sides.

The matching automata generated by the compiler are not necessarily optimal. We decided to keep the compiler simple, and take the suboptimal code for granted, especially since it usually does not make much difference. Consider the following two rules

```
f(a,b,c) = g(a)
f(X,b,d) = g(X),
```

where a, b, c, d are constants, and X is a variable. The compiler currently generates the following code in this case:

```
ATerm f(ATerm arg0, ATerm arg1, ATerm arg2) {
  if term_equal(arg0,a) {
    if term_equal(arg1,b) {
      if term_equal(arg2,c) {
        return g(a);
      }
    }
  }
  if term_equal(arg1,b) {
    if term_equal(arg2,d) {
```

```
          return g(arg0);
      }
  }
  return make_nf3(fsym, arg0, arg1, arg2)
},
```

where `fsym` is a constant corresponding to the function name `f`. The generated matching automaton is straightforward. It checks the arguments of each left-hand side from left to right using the ATerm library function `term_equal`, which does a simple pointer equality check (Sec. 7.1.2). If neither left-hand side matches, the appropriate normal form is constructed by ATerm library function `make_nf3` (Table IV).

Slightly better code could be obtained by dropping the left-to-right bias of the generated automaton[4] and checking `arg1` rather than `arg0` first:

```
ATerm f(ATerm arg0, ATerm arg1, ATerm arg2) {
  if term_equal(arg1,b) {
    if term_equal(arg0,a) {
      if term_equal(arg2,c) {
        return g(a);
      }
    }
    else if term_equal(arg2,d) {
      return g(arg0);
    }
  }
  return make_nf3(fsym, arg0, arg1, arg2)
}.
```

7.2.2 *List Matching.* As was pointed out in Sec. 6.4, a few simple cases of list matching that do not need backtracking are transformed to ordinary term matching in the preprocessing phase. The other cases are translated to nested while-loops. These handle the (limited form of) backtracking that may be caused by condition failure (Sec. 2.3). Further optimization of the generated code has turned out to be hard [Vinju 1999].

7.3 Evaluation of Conditions and Right-Hand Sides

The code generated for rule `[at-1-2']` (Sec. 6.6) is shown in Fig. 8. As in the previous example, the various ATerm functions used in the code are listed in Table IV. The μASF$^+$ `else` of the rule corresponds to the first `else` in the C code.

7.4 Memoization

To obtain faster code, the compiler can be instructed to memoize explicitly given ASF+SDF functions. The corresponding C functions get local hash tables to store each set of arguments[5] along with the corresponding result (normal form) once it has been computed. When called with a "known" set of arguments, the result is

---

[4]Nedjah et al. [1997] discuss optimization of the matching automaton under a left-to-right constraint.

[5]Function arguments annotated with the `delay` attribute need not be in normal form when stored in the memo table.

```
ATerm add_to(ATerm arg0, ATerm arg1, ATerm arg2)
{
  ATerm tmp[6];
  if (check_sym(arg2, type_env_sym)) {   /* arg2 = type-env(*Pairs) */
    ATerm atmp20 = arg_0(arg2);
    if (not_empty_list(atmp20)) {         /* t := non_empty_list(*Pairs) */
      tmp[0] = list_head(atmp20);         /* P := list_head(*Pairs) */
      tmp[1] = list_tail(atmp20);         /* *Pairs1 := list_tail(*Pairs) */
      if (check_sym(tmp[0], pair_sym)) { /* pair(Id1,Type2) := P */
        tmp[2] = arg_0(tmp[0]);          /* Id1 */
        tmp[3] = arg_1(tmp[0]);          /* Type2 */
        if (term_equal(arg0, tmp[2])) {  /* Id == Id1 */
          return make_nf1(type_env_sym,
                          conc(make_list(make_nf2(pair_sym, arg0, arg1)),
                               tmp[1]));

        }
        else {
          tmp[4] = add_to(arg0, arg1, make_nf1(type_env_sym, tmp[1]));
                         /* tmp[4] = add-to(Id,Type1,type-env(*Pairs1)) */
          if (check_sym(tmp[4], type_env_sym)) {
                                        /* tmp[4] = type-env(*Pairs) */
            tmp[5] = arg_0(tmp[4]);
            return make_nf1(type_env_sym,
                      conc(make_list(make_nf2(pair_sym, tmp[2], tmp[3])),
                           tmp[5]));
          }
        }
      }
    }
    else {
      return type_env(make_list(make_nf2(pair_sym, arg0, arg1)));
    }
  }
  return make_nf3(add_to_sym, arg0, arg1, arg2);
}
```

Fig. 8.   Code generated for rule [`at-1-2'`].

obtained from the memo table rather than recomputed. See also Field and Harrison [1988, Chapter 19].

Maximal subterm sharing (hash-consing) as used in the ATerm library (Sec. 7.1.2) combines very well with memoization. Since memo tables tend to contain many similar terms (function calls), memo table storage is effectively reduced by sharing. Furthermore, the check whether a set of arguments is already in the memo table is a simple equality check on the corresponding pointers. There is currently no hard limit on the size of a memo table, so the issue of replacement of table entries does not (yet) arise.

Unfortunately, since its effects may be hard to predict, memoization is something of a "fine art", not unlike adding strictness annotations to lazy functional programs. Memoization may easily become counterproductive if the memoized functions are not called with the same arguments sufficiently often, and finding the right subset of functions to memoize may require considerable experimentation and insight.

8. POSTPROCESSING

The quality of the generated C code is further improved by tail recursion elimination and constant caching. Not all C compilers are capable of tail recursion elimination, and no compiler known to us can do it if it has to produce code with symbolic debugging information, so the ASF+SDF compiler takes care of this itself. In principle, this optimization could also be done by the preprocessor if a while-construct were added to μASF$^+$.

Constant caching is a restricted form of memoization. Unlike the latter, it is performed fully automatically on ground terms occurring in right-hand sides of rules or in conditions. These may be evaluated more than once during the evaluation of a term, but since their normal form is the same each time (no side-effects), they are recognized and transformed into constants. The first time a constant is encountered during evaluation, the associated ground term is normalized and the result is assigned to the constant. In this way, the constant acts as a cache for the normal form.

There are good reasons to prefer this hybrid compile-time/run-time approach to a compile-time only approach:

—The compiler would have to normalize the ground terms in question. Although a suitable μASF interpreter that can be called by the compiler exists, such normalizations potentially require the full definition to be available.

—The resulting normal forms may be quite big, causing an enormous increase in code size.

9. BENCHMARKING

Table V lists some of the semantic features of the languages used in the benchmarking of the ASF+SDF compiler. Modularization aspects are not included. As can be seen in the second column, not all languages are of the same type. Like ASF+SDF, Elan and Maude are first-order rewriting languages, whereas Clean, Haskell, Opal, and SML are general purpose higher-order functional languages. At least to some extent, this difference in orientation and purpose complicates selection of suitable benchmark programs and interpretation of the results obtained.

Keeping this in mind, we devised benchmark programs with a highly synthetic character to evaluate specific implementation aspects, such as the effect of subterm sharing, graph rewriting, strict vs. lazy evaluation, and the like. They do not provide an overall comparison of the various systems. In selecting benchmarks suitable for ASF+SDF, we inevitably cover only a small part of the feature space of Clean, Haskell, Opal, and SML. The "Pseudoknot" benchmark [Hartel et al. 1996] would have provided broader coverage, but its emphasis on numerical (floating point) computation makes it unsuitable for ASF+SDF, which is aimed at definition of languages and language-based tool generation.

Language	Type of language and semantic characteristics	Compiled to
ASF+SDF	Language definition formalism • First-order • Strict • Conditional (both pos and neg) • Default rules • A-rewriting (lists)	C
Clean [Plasmeijer and van Eekelen 1994] [Smetsers et al. 1991]	Functional language • Higher-order • Lazy • Strictness annotations • Polymorphic typing	Native code via ABC abstract graph rewriting machine
Elan [Kirchner and Moreau 2001]	Rewriting logic language • First-order • Strategy specification • AC-rewriting	C
Haskell [Peyton Jones et al. 1993] [Peyton Jones 1996]	Functional language • Higher-order • Lazy • Strictness annotations • Polymorphic typing	C
Maude [Clavel et al. 1999]	Rewriting logic language • First-order • Reflection • AC-rewriting	Interpreted Core Maude
Opal [Didrich et al. 1994]	Algebraic programming language • Higher-order • Strict • Parametric typing	C
SML [Appel 1992]	Functional language • Higher-order • Strict • Polymorphic typing	Native code

Table V.   Languages used in the benchmarking of the ASF+SDF compiler.

Sec. 9.1 gives the results of our synthetic benchmarks for the languages listed in Table V. These figures are supplemented with results for some large ASF+SDF definitions in actual use, both with and without maximal subterm sharing in Sec. 9.2. Using profile information, the effects of maximal sharing are discussed in more detail in Sec. 9.3.

9.1   Three Synthetic Benchmarks

All three benchmarks deal with the symbolic manipulation of natural number expressions, where the natural numbers involved are in successor representation (unary representation).

The benchmarks are based on the normalization of expressions 2^n mod 17, with $17 \leq n \leq 23$. They are small programs that give rise to computations on very big terms. The fact that there are much more efficient ways to compute these expressions is of no concern here, except that this makes it easy to validate the results. The sources are available in [?].

Fig. 9.   Execution times for the `evalsym` benchmark.

Measurements for ASF+SDF were obtained both with and without maximal subterm sharing. For this purpose, the possibility to switch subterm sharing off, which is not a standard compiler option, was added. Some systems failed to compute results for the full range $17 \leq n \leq 23$. In those cases, the corresponding graph ends prematurely. Measurements were performed on a Mobile Pentium II (266 Mhz) with 128 MB of memory running Linux.

9.1.1   *The `evalsym` Benchmark.* The first benchmark is called `evalsym` and uses an algorithm that is CPU intensive, but does not use a lot of memory. The results are shown in Fig. 9. The differences between ASF+SDF, Elan, Haskell, Opal, and SML are relatively small, but Clean is about 1.7 times faster. Maude is about 5 times slower than the other systems, except for ASF+SDF without sharing. This is caused by the fact that Maude is interpreted (after translation to core Maude). The reason for including Maude in our benchmarks is the fact that, compared with other interpreters, the Maude interpreter is extremely fast.

9.1.2   *The `evalexp` Benchmark.* The second benchmark is called `evalexp`. As shown in Fig. 10, implementations that do not use some form of subterm sharing cannot cope with the excessive memory requirements of this benchmark. Opal, which is a strict language, probably achieves its good performance by the combined

Fig. 10. Memory usage for the `evalexp` benchmark.

Fig. 11. Execution times for the `evalexp` benchmark.

use of common subexpression elimination, peephole optimization, and so-called lazy compile-time reference counting garbage collection, ASF+SDF and Elan, which are also strict, both use maximal subterm sharing (Elan also uses the ATerm library), and Clean (lazy) uses lazy graph rewriting. Laziness is not enough, however, as is shown by the figures for Haskell.

Execution times are plotted in Fig. 11. Only Clean (lazy) is faster than ASF+SDF, actually about twice as fast.

9.1.3 *The* `evaltree` *Benchmark.* The third benchmark is called `evaltree` and is based on an algorithm that uses a lot of memory both with lazy and strict implementations, even those based on graph rewriting. Fig. 12 shows that only Elan and ASF+SDF scale up for $n > 20$. They can keep memory requirements at an acceptable level due to their use of maximal subterm sharing. The execution times are shown in Fig. 13. Not surprisingly, the extreme memory usage of the other systems leads to a degradation in execution times. In particular, Clean (lazy) is much slower than ASF+SDF this time.

Recall that the sharing scheme used by the ASF+SDF implementation is transparent to the rewriting process. In particular, it does not lead to graph rewriting, where multiple occurrences of a redex can be replaced in one stroke, but subterm sharing is usually not maximal. This explains why in some cases graph rewriting is faster than term rewriting with maximal subterm sharing while it is slower in others.

9.2 Some Large ASF+SDF Definitions

Table VI gives some statistics for several large ASF+SDF definitions whose performance is shown in Table VII. The ASF+SDF compiler was written in ASF+SDF itself, so the top entry in the fourth column of Table VI gives the self-compilation time.

We briefly describe the other applications. The Java servlet generator produces Java servlet code for a textual representation of a UML-like specification. The latter models data base applications, e.g., of banks and insurance companies. The generated Java servlet code implements a GUI to access the data bases via web pages.

The typesetter generates a textual representation from a box expression describing the formatting of a document in an abstract way [de Jonge 2001].

The SDF normalizer translates an SDF specification to an intermediate representation in so-called KernelSDF. This involves removal of its modular structure and simplification of complex grammar constructions. A detailed description of the operations performed during normalization can be found in [Visser 1997].

Finally, the Pico interpreter is an evaluator for the toy language Pico. The various Pico language constructions are evaluated given some value environment.

The C compilation times in the last column of Table VI were obtained using the gcc compiler with maximal optimizations on a 500 Mhz Pentium III PC running Linux.

Table VII shows the effects of maximal subterm sharing on the performance of the compiled versions. The figure in the top entry of the leftmost column is again the self-compilation time of the ASF+SDF compiler, but in this case only the time

Fig. 12.   Memory usage for the `evaltree` benchmark.

Fig. 13.   Execution times for the `evaltree` benchmark.

Definition	ASF+SDF (rules)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF to C compilation time (sec)	C compilation time (sec)
ASF+SDF compiler	1322	8373	45605	67	171
Java servlet generator	1446	12578	37193	174	179
Typesetter	607	2685	12231	12	36
SDF normalizer	941	3932	16192	20	45
Pico interpreter	200	448	2462	2	8

Table VI.   Size and compilation time for some large ASF+SDF definitions.

Application	Time (sec)		Memory (MB)	
	with sharing	without sharing	with sharing	without sharing
ASF+SDF compiler	45	155	27	134
Java servlet generator	12	50	10	34
Typesetter	10	49	5	5
SDF normalizer	8	28	8	11
Pico interpreter	20	80	4	4

Table VII. Performance of some large ASF+SDF definitions with and without maximal subterm sharing.

spent on rewriting was measured, whereas the corresponding figure in Table VI includes pre- and postprocessing phases.

9.3   The Effects of Maximal Subterm Sharing

In this section we present some further figures to shed more light on the effects of maximal subterm sharing. Figure VIII shows the results of profiling the typesetter application used in the previous section. We profiled one run with maximal subterm sharing and the standard pointer equality check, one run with sharing but with a "deep" equality check mimicking the equality check when subterm sharing is not necessarily maximal (as would be the case for graph rewriting), and a third run without sharing.

Figure VIII lists the functions that contribute most to total run time with the relative and absolute time spent by them, and the number of times they were called. In the standard configuration with sharing enabled, the ATerm library function `make_nf1` (Table IV) takes the largest percentage of the total run time, followed at a considerable distance by `lf_28` and other functions. Hence, the compiled specification spends most of its time building function applications with one argument. To be more precise, it spends most of its time *checking whether the term already exists*. If a new term has to be created, the ATerm library function `alloc_term` has to allocate space for it, but `alloc_term` is called very rarely as can be seen by comparing the number of calls to `make_nf1` and `alloc_term`. This is a clear indication of the success of sharing.

In the second section, the use of a "deep" equality causes the `term_equal` function

	Function	Rel. time (%)	Abs. time (sec)	No. of calls
Sharing and pointer equality	make_nf1	33.56	5.81	36 033 817
	lf_28	17.91	3.10	1 671 423
	make_list	10.80	1.87	10 103 016
	get_prefix	5.72	0.99	10 273 671
	lf_2	4.27	0.74	8 372 608
	...			
	alloc_term	0.12	0.02	110 047
	...			
Sharing and deep equality	make_nf1	23.93	5.64	36 033 817
	term_equal	23.42	5.52	69 603 654
	lf_28	14.76	3.48	1 671 423
	make_list	8.23	1.94	10 103 016
	insert	3.82	0.90	3 084 966
	...			
No sharing and structural equality	sweep_phase	28.59	14.69	1 222
	mark_term	18.20	9.35	2 124 199
	term_equal	11.35	5.83	72 134 451
	alloc_term	9.58	4.92	50 074 596
	lf_28	7.34	3.77	1 671 423
	mark_phase	4.73	2.43	1 222
	free_term	3.17	1.63	49 962 171
	make_nf1	3.06	1.57	36 033 817
	...			

Table VIII. Profiling information for the typesetter application. The lf_xx functions correspond to ASF+SDF functions in the typesetter specification, the other ones are term manipulation functions from the ATerm library.

to pop up at the second position in the profile, taking almost as much time as make_nf1. This causes an increase in total run time and a corresponding drop in the percentages of the other functions. This difference indicates that the replacement of a deep equality check with a pointer comparison leads to a gain of about 24% in execution time in this example.

The last section shows why performance degrades so drastically when sharing is disabled. The garbage collector has to perform extra work to reclaim space occupied by short-lived terms. This explains the rise of the sweep_phase, mark_term, mark_phase, and free_term functions. The make_nf1 function has actually become more efficient, because it does not have to check for existence of terms. It can simply call alloc_term each time and fill in the term to be created. Consequently, the make_nf1 function has dropped to the eighth position with only 3% of the time.

Note that the last profile also suggests that the ATerm garbage collector takes a large amount of time, and that this is the reason ASF+SDF performs so badly when sharing is turned off. The only reason this is not noticeable when sharing is turned on, is because in that case relatively few new terms are actually created and garbage collection plays a minor role.

10. CONCLUSIONS AND FURTHER WORK

10.1 Conclusions

The ASF+SDF compiler generates high quality C code in a relatively straight-forward way. The main factors contributing to its performance are the decisions to generate C code directly and to use a run-time term storage scheme based on maximal subterm sharing. The specific benefits of using maximal subterm sharing are:

—*Reduced memory usage.* This saves time as well. Very large terms can be processed efficiently.

—*Fast equality check.* Structural equality checks can be replaced by much more efficient pointer equality checks.

—*Space efficient memoization.* Since memo tables tend to contain many similar terms, less memo table storage is needed.

We feel our results show maximal subterm sharing to be a promising implementation technique for other term processing applications as well.

10.2 Further Work

Some possibilities for further improvement and extension are:

—Incorporation of additional preprocessing steps such as argument reordering during matching, evaluation of sufficiently simple conditions during matching in a dataflow fashion, i.e., as soon as the required values become available, and re-ordering of independent conditions.

—Optimization of repeated applications of a rule like rule `[s-1]` in Sec. 7.2.2, or of successive applications of different rules by analyzing their left- and right-hand sides. Similarly, elimination of the redex search phase in some cases ("matchless rewriting").

—Incorporation of other rewrite strategy options besides default rules and the `delay` attribute that are currently supported.

—Combination of maximal subterm sharing with graph rewriting. As shown by our benchmarks, in some cases graph rewriting is faster than term rewriting with maximal subterm sharing, while it is slower in others, so it may be worthwhile to investigate their combination.

—Use of an *incremental* garbage collector.

REFERENCES

ALLEN, J. R. 1978. *Anatomy of Lisp*. McGraw-Hill.

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press.

APPEL, A. W. AND GONÇALVES, M. J. R. 1993. Hash-consing garbage collection. Tech. Rep. CS-TR-412-93, Princeton University.

APT, K. R., BRUNEKREEF, J. J., PARTINGTON, V., AND SCHAERF, A. 1998. Alma-0: An imperative language that supports declarative programming. *ACM Transactions on Programming Languages and Systems 20*, 1014–1066.

BENANAV, D., KAPUR, D., AND NARENDRAN, P. 1985. Complexity of matching problems. In *Rewriting Techniques and Applications (RTA '85)*, J.-P. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 202. Springer-Verlag, 417–429.

BERGSTRA, J. A. AND VAN DEN BRAND, M. G. J. 2000. Syntax and semantics of a high-level intermediate representation of ASF+SDF. Tech. Rep. SEN-R0030, CWI, Amsterdam.

BERGSTRA, J. A., HEERING, J., AND KLINT, P., Eds. 1989. *Algebraic Specification*. ACM Press/Addison-Wesley.

BERGSTRA, J. A. AND KLINT, P. 1998. The discrete time ToolBus—A software coordination architecture. *Science of Computer Programming 31*, 205–229.

BOEHM, H. 1993. Space efficient conservative garbage collection. *ACM SIGPLAN Notices 28,* 6 (June), 197–206. Proceedings of the 1991 Conference on Programming Language Design and Implementation (PLDI '91).

VAN DEN BRAND, M. G. J., DE JONG, H. A., KLINT, P., AND OLIVIER, P. A. 2000. Efficient annotated terms. *Software—Practice and Experience 30*, 259–291.

VAN DEN BRAND, M. G. J., EIJKELKAMP, S. M., GELUK, D. K. A., MEIJER, OSBORNE, H. R., AND POLLING, M. J. F. 1995. Program transformations using ASF+SDF. In *Proceedings of ASF+SDF '95*. Technical Report P9504. Programming Research Group, University of Amsterdam, 29–52.

VAN DEN BRAND, M. G. J., KLINT, P., AND OLIVIER, P. A. 1999. Compilation and memory management for ASF+SDF. In *Compiler Construction (CC '99)*, S. Jähnichen, Ed. Lecture Notes in Computer Science, vol. 1575. Springer-Verlag, 198–213.

VAN DEN BRAND, M. G. J., SELLINK, M., AND VERHOEF, C. 2000. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming 36*, 209–266.

VAN DEN BRAND, M. G. J., VAN DEURSEN, A., HEERING, J., DE JONG, H. A., DE JONGE, M., KUIPERS, T., KLINT, P., MOONEN, L., OLIVIER, P. A., SCHEERDER, J., VINJU, J. J., VISSER, E., AND VISSER, J. 2001. The ASF+SDF Meta-Environment: A component-based language development environment. In *Compiler Construction (CC 2001)*, R. Wilhelm, Ed. Lecture Notes in Computer Science, vol. 2027. Springer-Verlag, 365–370.

BRUNEKREEF, J. J. 1996. A transformation tool for pure Prolog programs. In *Logic Program Synthesis and Transformation (LOPSTR '96)*, J. P. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, 130–145.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTI-OLIET, N., MESEGUER, J., AND QUESADA, J. 1999. Maude: Specification and programming in rewriting logic — Maude system documentation. Tech. rep., SRI International. March.

COURCELLE, B. AND FRANCHI-ZANNETTACCI, P. 1982. Attribute grammars and recursive program schemes I and II. *Theoretical Computer Science 17*, 163–191 and 235–257.

DERSHOWITZ, N. AND JOUANNAUD, J.-P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science, volume B*, J. van Leeuwen, Ed. Elsevier Science Publishers, 243–320.

VAN DEURSEN, A. 1994. Executable language definitions: Case studies and origin tracking techniques. Ph.D. thesis, University of Amsterdam.

VAN DEURSEN, A., HEERING, J., AND KLINT, P., Eds. 1996. *Language Prototyping*. AMAST Series in Computing, vol. 5. World Scientific.

VAN DEURSEN, A. AND KLINT, P. 1998. Little languages: Little maintenance? *Journal of Software Maintenance 10*, 75–92.

VAN DEURSEN, A., KLINT, P., AND VERHOEF, C. 1999. Research issues in the renovation of legacy systems. In *Fundamental Approaches to Software Engineering (FASE '99)*, J.-P. Finance, Ed. Lecture Notes in Computer Science, vol. 1577. Springer-Verlag, 1–21.

VAN DEURSEN, A. AND MOONEN, L. 2000. Exploring legacy systems using types. In *Proceedings of the Seventh Working Conference on Reverse Engineering*. IEEE Computer Society Press, 32–41.

DIDRICH, K., FETT, A., GERKE, C., GRIESKAMP, W., AND PEPPER, P. 1994. Opal: Design and implementation of an algebraic programming language. In *International Conference on Programming Languages and System Architectures*, J. Gutknecht, Ed. Lecture Notes in Computer Science, vol. 782. Springer-Verlag, 228–244.

DIK, C. H. S. 1989. A fast implementation of the Algebraic Specification Formalism. M.S. thesis, University of Amsterdam, Programming Research Group.

DINESH, T. B., HAVERAAEN, M., AND HEERING, J. 2001. An algebraic programming style for numerical software and its optimization. *Scientific Programming 8,* 4 (September/October). To appear; CoRR E-print Server xxx.lanl.gov/abs/cs.SE/9903002.

FIELD, A. J. AND HARRISON, P. G. 1988. *Functional Programming.* Addison-Wesley.

FIELD, J. 1992. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. San Francisco, 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

FOKKINK, W. J., KAMPERMAN, J. F. TH., AND WALTERS, H. R. 1998. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Trans. Program. Lang. Syst. 20*, 679–706.

FRANZ, M. 1994. Code-generation on-the-fly: A key to portable software. Ph.D. thesis, ETH Zurich. ftp://ftp.inf.ethz.ch/pub/publications/dissertations/th10497.ps.

GROOTE, J. F., KOORN, J. W. C., AND VAN VLIJMEN, S. F. M. 1995. The safety guaranteeing system at station Hoorn-Kersenboogaard. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*. IEEE, 57–68.

HARTEL, P. H. ET AL. 1996. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *J. Funct. Program. 6*, 621–655.

HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. 1989. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices 24,* 11, 43–75. Most recent version: ftp.cwi.nl/pub/gipe/reports/SDFManual.ps.Z.

HENDRIKS, P. R. H. 1991. Implementation of modular algebraic specifications. Ph.D. thesis, University of Amsterdam.

HILLEBRAND, J. A. 1996. Experiments in specification re-engineering. Ph.D. thesis, University of Amsterdam.

HOFFMANN, C. M. AND O'DONNELL, M. J. 1982. Pattern matching in trees. *J. ACM 29*, 68–95.

JONES, R. AND LINS, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* Wiley.

DE JONGE, M. 2001. A pretty-printer for every occassion. Tech. Rep. SEN-R0115, CWI, Amsterdam.

KAMPERMAN, J. F. TH. 1996. Compilation of term rewriting systems. Ph.D. thesis, University of Amsterdam.

KAPLAN, S. 1987. A compiler for conditional term rewriting systems. In *Rewriting Techniques and Applications (RTA '87)*, P. Lescanne, Ed. Lecture Notes in Computer Science, vol. 256. Springer-Verlag, 25–41.

KIRCHNER, H. AND MOREAU, P.-E. 2001. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *J. Funct. Program. 11,* 2, 207–251.

KLINT, P. 1993. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 176–201.

KLOP, J. W. 1992. Term rewriting systems. In *Handbook of Logic in Computer Science, volume 2*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1–116.

VAN DER MEULEN, E. A. 1996. Incremental typechecking. In *Language Prototyping: An Algebraic Specification Approach*, A. van Deursen, J. Heering, and P. Klint, Eds. AMAST Series in Computing, vol. 5. World Scientific, 199–248.

MOONEN, L. 1997. A generic architecture for data flow analysis to support reverse engineering. In *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF '97)*, M. P. A. Sellink, Ed. Electronic Workshops in Computing. Springer/British Computer Society.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

NEDJAH, N., WALTER, C. D., AND ELDRIDGE, S. E. 1997. Optimal left-to-right pattern-matching automata. In *Algebraic and Logic Programming (ALP '97/HOA '97)*, M. Hanus, J. Heering, and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1298. Springer-Verlag, 273–286.

OLIVIER, P. A. 1999. Benchmarking of functional/algebraic language implementations. http://adam.wins.uva.nl/∼olivierp/benchmark/.

PEYTON JONES, S. L. 1996. Compiling Haskell by program transformation: A report from the trenches. In *Programming Languages and Systems (ESOP '96)*, H. R. Nielson, Ed. Lecture Notes in Computer Science, vol. 1058. Springer-Verlag, 18–44.

PEYTON JONES, S. L., HALL, C. V., HAMMOND, K., PARTAIN, W. D., AND WADLER, P. L. 1993. The Glasgow Haskell compiler: A technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference (JFIT), Keele*. DTI/SERC, 249–257.

PEYTON JONES, S. L., NORDIN, T., AND OLIVA, D. 1998. C--: A portable assembly language. In *Implementation of Functional Languages (IFL '97)*, C. Clack, K. Hammond, and T. Davie, Eds. Lecture Notes in Computer Science, vol. 1467. Springer-Verlag, 1–19.

PLASMEIJER, M. J. AND VAN EEKELEN, M. C. J. D. 1994. Concurrent CLEAN—version 1.0—Language refence manual. Tech. rep., University of Nijmegen, Department of Computer Science. Draft.

RUTTEN, E. P. B. M. AND THIÉBAUX, S. 1992. Semantics of Manifold: Specification in ASF+SDF and extension. Tech. Rep. CS-R9269, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

SELLINK, M. P. A. AND VERHOEF, C. 2000. Development, assessment, and reengineering of language descriptions. In *Fouth European Conference on Software Maintenance and Reengineering*, J. Ebert and C. Verhoef, Eds. IEEE Computer Society. adam.wins.uva.nl/∼x/cale/cale.html.

SMETSERS, S., NÖCKER, E., VAN GRONINGEN, J., AND PLASMEIJER, M. J. 1991. Generating efficient code for lazy functional languages. In *Functional Programming and Computer Architecture (FPCA '91)*, J. Hughes, Ed. Lecture Notes in Computer Science, vol. 524. Springer-Verlag, 592–617.

TERASHIMA, M. AND KANADA, Y. 1990. HLisp—Its concept, implementation and applications. *Journal of Information Processing 13*, 3, 265–275.

TIP, F. AND DINESH, T. B. 2001. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology 10*, 1 (January), 5–55.

VAN ROY, P. 1993. The wonder years of sequential Prolog implementation. *J. Logic Program. 19/20*, 385–441.

VINJU, J. J. 1999. Optimizations of list matching in the ASF+SDF compiler. M.S. thesis, University of Amsterdam, Programming Research Group. www.cwi.nl/∼jurgenv/.

VISSER, E. 1997. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam. Available at www.cs.uu.nl/∼visser/publications/ftp/Vis97.thesis.ps.gz.