Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

# Compiling Language Definitions:
# The ASF+SDF Compiler

M.G.J. van den Brand[1], J. Heering[1], P. Klint[1,2], P.A. Olivier[1]

[1]CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

{markvdb,jan,paulk,olivierp}@cwi.nl

[2]University of Amsterdam

Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

ABSTRACT

The ASF+SDF Meta-Environment is an interactive language development environment whose main application areas are definition of domain-specific languages, generation of program analysis and transformation tools, production of software renovation tools, and general specification and prototyping. 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.

## 1. INTRODUCTION

ASF+SDF [Bergstra et al. 1989; van Deursen et al. 1996] is the metalanguage of the ASF+SDF Meta-Environment [Klint 1993], an interactive environment for the development of domain-specific and general purpose programming languages, covering parsing, typechecking, translation, transformation, and execution of programs.

SDF [Heering et al. 1989], the syntax definition component of ASF+SDF, is a BNF-like formalism for defining the lexical, context-free and abstract syntax of languages. The implementation of SDF is beyond the scope of this article. Suffice it to say, its implementation supports interactive syntax development and fully

general context-free parsing by means of scanner and parser generators that are both lazy (just-in-time) and incremental [Heering et al. 1990; Heering et al. 1992; Heering et al. 1994]. SDF is currently being superseded by SDF2 [Visser 1997], whose main feature is a very close integration of lexical and context-free syntax. This is reflected in its implementation by the use of scannerless parsing.

The semantics definition component of ASF+SDF, which is an outgrowth of the algebraic specification formalism ASF [Bergstra et al. 1989], 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. It 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 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 typecheckers 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.

ASF+SDF's main application areas are

—Definition of domain-specific languages

—Generation of program analysis and transformation tools

—Production of software renovation tools

—General specification and prototyping.

Table 1 gives details and further references.

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 here.

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 [Moreau and Kirchner 1998], Haskell [Peyton Jones et al. 1993; Peyton Jones 1996], Opal [Didrich et al. 1994], and SML [Appel 1992].

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. Disre-

**Domain-Specific Languages**

—Risla [van den Brand et al. 1996; van Deursen and Klint 1998] (financial product specification)
—Box [van den Brand and Visser 1996] (prettyprinting)
—EURIS [Groote et al. 1995] (railroad safety)
—Action Semantics [van Deursen 1994] (programming language semantics)
—Dahl [Moonen 1997] (dataflow analysis)
—Manifold [Rutten and Thiébaux 1992], ToolBus [Bergstra and Klint 1998] (coordination languages)
—ALMA-0 [Apt et al. 1998] (backtracking and search)

**Program Analysis**

—Typechecking of Pascal [van Deursen et al. 1996, Chapter 2]
—Typechecking and execution of CLaX [Dinesh and Tip 1992; Dinesh and Tip 1997]
—Type inference, object identification, and documentation for Cobol [van den Brand et al. 2000; van Deursen and Moonen 1998; van Deursen and Kuipers 1998; van Deursen and Kuipers 1999]

**Program Transformation**

—Interactive program transformation for Clean [van den Brand et al. 1995] and Prolog [Brunekreef 1996]
—Automatic program transformation for C++ [Dinesh et al. 1998]

**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 den Brand et al. 1996; van den Brand et al. 1997; van Deursen et al. 1999]
—Automatic program transformation for restructurering of Cobol programs (including embedded languages like SQL and CICS) [van den Brand et al. 1997; van den Brand et al. 1998; Sellink et al. 1999]
—Derivation of language descriptions from compilers and on-line manuals [Sellink and Verhoef 1999; Sellink and Verhoef 2000]

**Specification and Prototyping of New Applications and Tools**

—PIM [Field 1992; Bergstra et al. 1997] (compiler toolkit)
—μCRL [Hillebrand 1996] (proof checking and simulation toolkit)
—Components of the ASF+SDF Meta-Environment itself [van den Brand et al. 1997] (including a parser generator, a prettyprinter generator, and the ASF+SDF compiler described in this article)

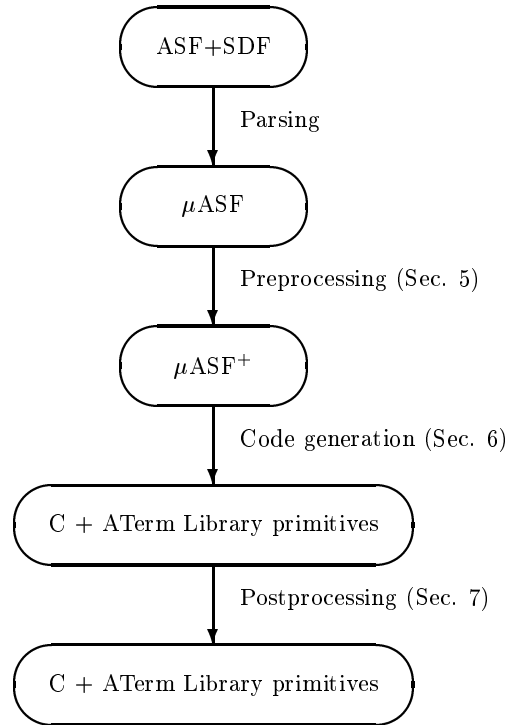Table 1.    Main application areas of the ASF+SDF Meta-Environment.

```
          ┌─────────────┐
          │   ASF+SDF   │
          └─────────────┘
                 │
                 │ Parsing
                 ▼
          ┌─────────────┐
          │    μASF     │
          └─────────────┘
                 │
                 │ Preprocessing (Sec. 5)
                 ▼
          ┌─────────────┐
          │    μASF⁺    │
          └─────────────┘
                 │
                 │ Code generation (Sec. 6)
                 ▼
    ┌───────────────────────────┐
    │ C + ATerm Library primitives │
    └───────────────────────────┘
                 │
                 │ Postprocessing (Sec. 7)
                 ▼
    ┌───────────────────────────┐
    │ C + ATerm Library primitives │
    └───────────────────────────┘
```

Fig. 1.    General layout of the ASF+SDF compiler.

garding 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.

—It has to support separate compilation of ASF+SDF modules. For large language definitions, modularization and separate compilation are as important as for conventional programs.

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

## 2. 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 Figure 1. The following compiler phases 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, which is 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 $\mu$ASF, an abstract syntax version of ASF+SDF.

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

—Code generation. The compiler generates C extended with calls to the *ATerm library*, a run-time library for term manipulation and storage. Each $\mu$ASF 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.

## 3. 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; Walters 1997] 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.

In the following subsections we discuss the arguments in favor of generating C rather than native code, the choice of ASF+SDF as an implementation language for the compiler, some pitfalls in the areas of high-level transformations and abstract machine interfaces, the importance of a proper organization of term storage, and some issues related to separate compilation.

### 3.1 Choice of C as Target Language

Generating C code is an efficient way to achieve portability. 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.

## 3.2 Choice of ASF+SDF as Implementation Language

Not unexpectedly, 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.

## 3.3 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.

For this reason, our compiler does not generate code for the Abstract Rewrite Machine (ARM), which was originally developed for ASF+SDF and then used in the compiler for the equational programming language Epic [Fokkink et al. 1998]. ARM is based on the notion of *minimal term rewriting system* (MTRS). An MTRS consists of unconditional rewrite rules in so-called *minimal form* [Fokkink et al. 1998, Definition 3.1.1]. ARM thus requires a high-level transformation phase to simplify the rules that are not in this form and to eliminate the conditions (if any) [Fokkink et al. 1998, p. 681]. Furthermore, ARM does not support list matching, so rules with lists have to be transformed to minimal rewrite rules as well. Although these transformations are possible, they have turned out to be counterproductive in the ASF+SDF compiler, and with C taking care of portability, ARM's main purpose was lost. In fact, rather than breaking rules down into smaller ones, the ASF+SDF compiler tries to combine rules into larger ones as much as possible during preprocessing.

Our experience with ARM is not unique. 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. The factors involved in this trade-off 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 the instructive 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. 6.1). 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. 8) show the Clean compiler and the ASF+SDF compiler to generate code with comparable performance.

### 3.4 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. 6.1.

### 3.5 Separate Compilation

For large modularized language definitions, separate compilation is as important as it is for large modularized programs. Fully separate compilation of ASF+SDF modules is hard since the rewrite rules defining an ASF+SDF function may be scattered over several modules and each ASF+SDF function has to correspond to a single C function in the generated code for reasons of efficiency. Fortunately, the number of modules contributing to the definition of an ASF+SDF function is usually very small, so a useful approximation to separate compilation of ASF+SDF modules can still be obtained.

### 4. 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. A more detailed semantics by example of $\mu$ASF, which helped to answer the questions that emerged while the compiler was being written, is given by Bergstra and van den Brand [2000]. For the use of ASF+SDF (including SDF) see [van Deursen et al. 1996].

Since we do not go into the syntax definition component SDF, we will use a running example written in $\mu$ASF, the abstract syntax (prefix notation only) of ASF+SDF. Consider the definition of a simple type environment in Figure 2. The 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

```
module Type-environment
signature
  nil-type        {constructor};
  pair(_,_)       {constructor};
  type-env(_)     {constructor};
  lookup(_,_);
  add-to(_,_,_)
rules

[l-1]   lookup(Id,type-env(conc(*Pair1,conc(pair(Id,Type),*Pair2))))
              = Type;

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

[at-1] add-to(Id,Type1,type-env(conc(pair(Id,Type2),*Pair1)))
              = type-env(conc(pair(Id,Type1),*Pair1));

[at-2] Id1 != Id2 &
       add-to(Id1,Type1,type-env(*Pair1)) == type-env(*Pair2)
         ==>
       add-to(Id1,Type1,type-env(conc(pair(Id2,Type2),*Pair1)))
              = type-env(conc(pair(Id2,Type2),*Pair2));

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

Fig. 2.   Definition of a simple type environment in $\mu$ASF, the abstract syntax (prefix notation only) version of ASF+SDF produced by the parsing phase.

conversion to $\mu$ASF. 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 are first-order, i.e., they cannot have arguments, and 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 2. The example contains a single conditional rule [at-2] with both a negative and a positive condition, and a single default rule [l-2].

With an appropriate user-defined syntax, the ASF+SDF version of rule [at-1] would get the more natural look

```
[at-1] add (Id,Type1) to {(Id,Type2),Pair1*} = {(Id,Type1),Pair1*};
```

and similarly for the other rules. In the following sections we explain the various types of rules in more detail.

| = | 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 2.    The predefined symbols used in μASF rewrite rules.

### 4.1 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 redex if its left-hand side matches the redex and its conditions (if any) succeed after substitution of the values found during matching.

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.

Consider rule [at-2] in Fig. 2 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 for the variables Id1, Type1, Id2, Type2, and *Pair1.

(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 *Pair2 in its right-hand side. The value of *Pair2 is obtained by matching the right-hand side with the normalized left-hand side. Since *Pair2 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 `*Pair2` found in step 3.

## 4.2 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. 4.1) 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. 2 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 `*Pair1` and `*Pair2` 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].

## 4.3 Default Rules

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

## 4.4 Constructors

A (free) constructor is a function that does not occur at the outermost position in the left-hand side of a rewrite rule. A term consisting solely of constructors is in normal form. In ASF+SDF the rules defining a function may be scattered over many modules, so this is a global property. The `constructor` attribute supplies this information locally in a module, thus improving readability and facilitating separate compilation of modules. In Fig. 2, the functions `nil-type`, `pair`, and `type-env` are declared as constructors. As mentioned before, the built-in list constructors `list`, `conc`,[1] and `null` need not be declared. Omitting constructor attributes is not a fatal error, but may result in less readable ASF+SDF definitions as well as less efficient code. Some of the compiler optimizations depend on constructor attributes being present in the ASF+SDF source.

---

[1]The associativity of `conc` is taken care of by list matching, otherwise it is a free constructor.

### 4.5 Modules

ASF+SDF's only module operation is `import`.[2] As mentioned in Sec. 3.5, separate compilation of modules is an important design issue.

### 4.6 Rewriting Strategies

ASF+SDF is a strict language based on innermost rewriting (call-by-value). With few exceptions, practical experience with ASF+SDF over the past ten years has shown innermost rewriting to be a good choice for several reasons:

—Most users are familiar with call-by-value from C and other imperative languages.

—It is consistent with the semantics of ASF+SDF's default rules (Sec. 4.3).

—Its behavior is more predictable than that of other strategies, an important consideration when rewrite systems become large.

—No strictness annotations need to be added by the user to improve the quality of the code generated by the compiler. This is an advantage in view of the fact that "inserting these strictness annotations correctly can be a fine art" [Hartel et al. 1996, p. 651].

—It 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. 6.1).

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.

## 5. PREPROCESSING

Figure 3 is a refinement of Figure 1 showing the preprocessing steps as well as other actions performed in later phases of the compiler. The output language of the preprocessing phase is $\mu ASF^+$, which is $\mu ASF$ with the additional constructs shown in Table 3. Their purpose will become clear later on when the preprocessing (Sec. 5) and code generation (Sec. 6) are discussed. Some of them, like nested rules, the `else`-construct, and the assignment, might very well be added to ASF+SDF itself, but this remains to be done.

We now discuss the various preprocessing steps in more detail. As noted in Sec. 3.3, they have to be chosen judiciously to prevent them from becoming counterproductive, especially if their purpose is to reduce the number of different constructs that have to be handled by the code generator. Each step has to preserve the innermost rewriting strategy[3] as well as the backtracking behavior of list matching.

---

[2]The parameterization and renaming operations of ASF [Bergstra et al. 1989] are not available in the current implementation of ASF+SDF.

[3]Function arguments annotated with the `delay` attribute (Sec. 4.6) have to be taken into account as well, but will be ignored in this article for the sake of readability.
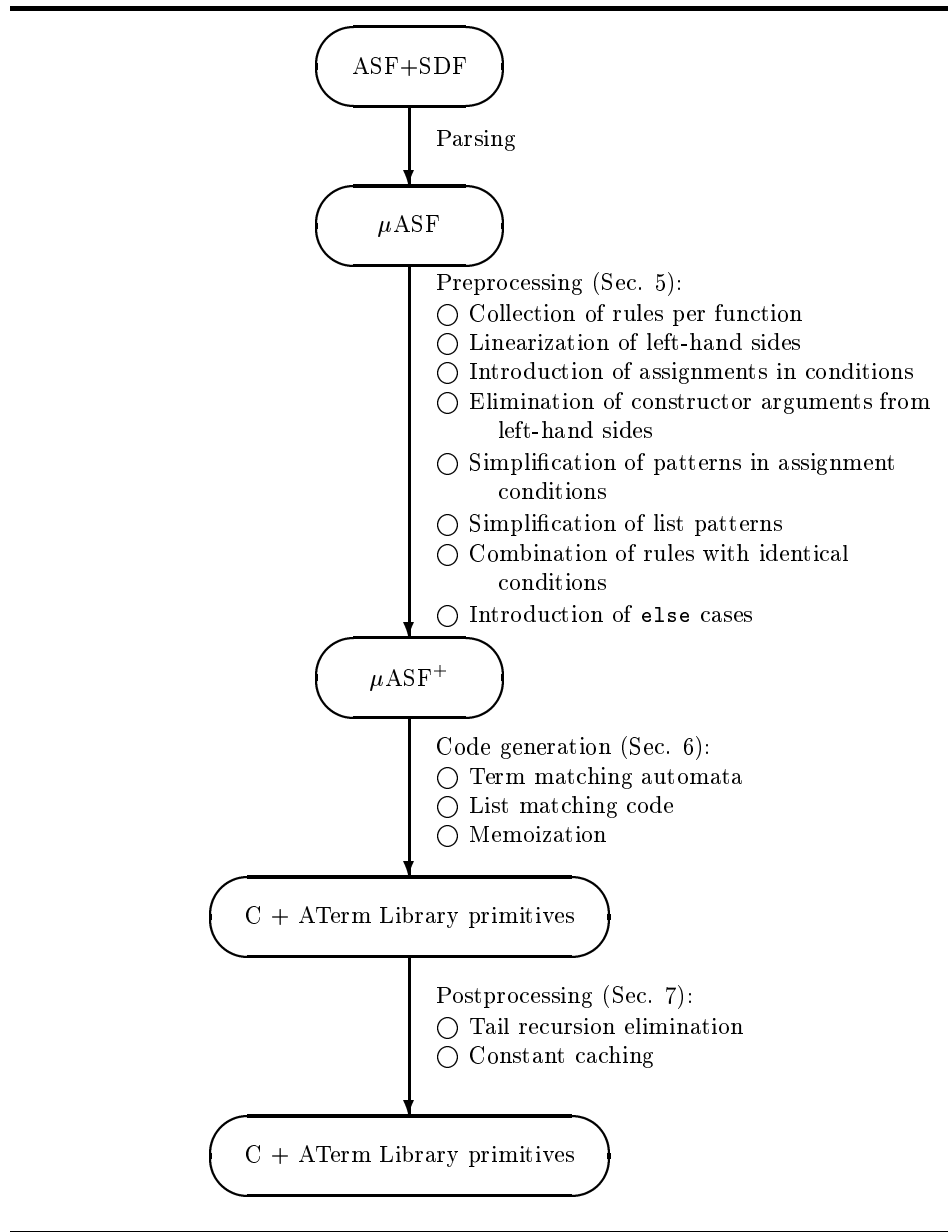
```
                        ╭─────────────╮
                        │   ASF+SDF   │
                        ╰─────────────╯
                               │
                               │  Parsing
                               ▼
                        ╭─────────────╮
                        │    μASF      │
                        ╰─────────────╯
                               │
                               │  Preprocessing (Sec. 5):
                               │  ◯ Collection of rules per function
                               │  ◯ Linearization of left-hand sides
                               │  ◯ Introduction of assignments in conditions
                               │  ◯ Elimination of constructor arguments from
                               │       left-hand sides
                               │  ◯ Simplification of patterns in assignment
                               │       conditions
                               │  ◯ Simplification of list patterns
                               │  ◯ Combination of rules with identical
                               │       conditions
                               │  ◯ Introduction of else cases
                               ▼
                        ╭─────────────╮
                        │   μASF+      │
                        ╰─────────────╯
                               │
                               │  Code generation (Sec. 6):
                               │  ◯ Term matching automata
                               │  ◯ List matching code
                               │  ◯ Memoization
                               ▼
                 ╭───────────────────────────╮
                 │  C + ATerm Library primitives │
                 ╰───────────────────────────╯
                               │
                               │  Postprocessing (Sec. 7):
                               │  ◯ Tail recursion elimination
                               │  ◯ Constant caching
                               ▼
                 ╭───────────────────────────╮
                 │  C + ATerm Library primitives │
                 ╰───────────────────────────╯
```

Fig. 3.    Layout of the ASF+SDF compiler. This is a refinement of Figure 1.

| | |
|---|---|
| := | 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 3.    Additional predefined symbols of $\mu$ASF$^{+}$.

## 5.1 Collection of Rules per Function

As mentioned in Sec. 3.5, fully separate compilation of ASF+SDF modules is hampered by the fact that the rewrite rules for a function can be scattered over several modules. Given a top module for which an executable has to be generated, the preprocessing phase starts by traversing the top module 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. The rules collected for each function together with the corresponding function declaration from the signature are made into a new $\mu$ASF module.[4] When a rewrite rule is changed, only the module containing the function actually affected is recompiled. This yields a useful approximation to separate compilation because the number of modules involved is usually limited ($< 100$) and the number of modules contributing to the definition of a function is usually very small. Still, the full specification has to be scanned for the rare cases a function is not completely defined in a single module, and a function attribute ruling this out would be a useful addition to ASF+SDF.

## 5.2 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. Innermost rewriting guarantees that these conditions do not cause spurious rewrite steps not done by the original non-linear match.[5]

For example, rules [l-1] and [at-1] in Fig. 2 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),*Pair1)))
        = type-env(conc(pair(Id,Type1),*Pair1))
```

---

[4]For reasons of efficiency, constructor functions (which can never occur at the outermost position of a left-hand side) are not made into separate modules. Instead, the constructors defined in a module are kept together and made into a single new module.

[5]Non-linearities involving function arguments annotated with the delay attribute are not allowed.

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

Linearization has pros and cons. On the one hand, it 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. 6.1. On the other hand, in rare cases it may also cause inefficiencies. Consider, for instance, a rule `f(X,X,lp) = ...` with complicated list pattern `lp`. A straightforward implementation would first check the equality of the values obtained for the first two arguments of `f` before proceeding with the matching of `lp`. A straightforward implementation of the transformed rule

```
X == X1 ==> f(X,X1,lp) = ...
```

as currently generated by the compiler postpones the equality check and does a full match of `f(X,X1,lp)` first. This is inefficient if the full match succeeds with unequal values for `X` and `X1`.

## 5.3 Introduction of Assignments in Conditions

As explained in Sec. 4.1, one side of a positive condition may contain variables that are uninstantiated at the time the condition is evaluated. 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 uninstantiated variables is not normalized before matching. To flag this case to the code generation phase, the $\mu$ASF equality is replaced by the $\mu$ASF$^+$ assignment. If necessary, the left- and right-hand side of the original condition are interchanged.

Rule `[at-2]` in Fig. 2 is of this kind since its second condition contains the new list variable `*Pair2`. It would be transformed into

```
[at-2'] Id1 != Id2 &
        type-env(*Pair2) := add-to(Id1,Type1,type-env(*Pair1))
           ==>
        add-to(Id1,Type1,type-env(conc(pair(Id2,Type2),*Pair1)))
              = type-env(conc(pair(Id2,Type2),*Pair2)).
```

## 5.4 Elimination of Constructor Arguments from Left-Hand Sides

Complex arguments consisting solely of constructors are eliminated from left-hand sides of rules and moved to assignment conditions. Let `f(...,ct,...) = ...` be such a rule with complex constructor term `ct`. It is transformed to

```
X := ct ==> f(...,X,...) = ....
```

This transformation simplifies the matching automaton by replacing the matching of `ct` by a simple pointer equality check (this will become clear later). Since the value of `X` is not evaluated and `ct` is already in normal form, it does not introduce spurious rewrite steps not done by the original rule.

### 5.5 Simplification of Patterns in Assignment Conditions

If not already in the right form, assignment conditions will be broken up into several new assignment conditions in such a way that the patterns making up their left-hand sides consist of a single variable, a single constant, or a single function symbol with only variables as arguments. This transformation has no effect on the performance or even the structure of the corresponding matching automaton, but makes its generation easier.

Rule [at-2'] has an assignment condition whose left-hand side is already in the right form, so we give another example. The rule

```
g(h(a),Z) := k(X) ==> f(X,Y) = ...
```

is transformed into

```
g(H,Z) := k(X) & h(A) := H & a := A ==> f(X,Y) = ....
```

In both the original and the transformed version, the instantiated right-hand side `k(X)` is normalized before the assignment is evaluated by matching with its left-hand side. Hence, the values obtained for `H` and `A` (if any) by matching must themselves be normal forms, and the second and third assignment cannot introduce spurious rewrite steps not done by the original assignment.

### 5.6 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, but may occasionally cause inefficiencies similar to those that may be caused by linearization (Sec. 5.2).

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,*Pair1)))
               = type-env(conc(pair(Id,Type1),*Pair1))
```

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 $\mu$ASF$^+$ list functions in Table 3. For example, [at-1''] is transformed into

```
[at-1'''] t := non_empty_list(*Pair) &
          P := list_head(*Pair) &
          *Pair1 := list_tail(*Pair) &
          pair(Id1,Type2) := P &
          Id == Id1
            ==>
```

```
add-to(Id,Type1,type-env(*Pair))
       = type-env(conc(pair(Id,Type1),*Pair1)),
```

where `t` is the boolean value `true` (Table 3), and similarly for `[at-2'']`.

## 5.7 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(*Pair) &
          P := list_head(*Pair) &
          *Pair1 := list_tail(*Pair) &
          pair(Id1,Type2) := P
            ==>
          add-to(Id,Type1,type-env(*Pair)) =
          {
          Id == Id1
            ==>
          type-env(conc(pair(Id,Type1),*Pair1));

          Id != Id1 &
          type-env(*Pair2) := add-to(Id,Type1,type-env(*Pair1))
            ==>
          type-env(conc(pair(Id1,Type2),*Pair2))
          },
```

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

## 5.8 Introduction of `else` Cases

$\mu$ASF$^+$ 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(*Pair) &
          P := list_head(*Pair) &
          *Pair1 := list_tail(*Pair) &
          pair(Id1,Type2) := P
            ==>
          add-to(Id,Type1,type-env(*Pair)) =
          {
          Id == Id1
            ==>
          type-env(conc(pair(Id,Type1),*Pair1))
            else
          type-env(*Pair2) := add-to(Id,Type1,type-env(*Pair1))
            ==>
          type-env(conc(pair(Id1,Type2),*Pair2))
          }.
```

| | |
|---|---|
| `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` |
| `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 4.    Selected ATerm library functions.

## 6. CODE GENERATION

### 6.1 The ATerm Library

6.1.1 *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 4. Many of them correspond directly to predefined symbols of $\mu$ASF (Table 2) and $\mu$ASF$^+$ (Table 3). Examples of actual code using them is given in Sec. 6.2 and Sec. 6.3.

6.1.2 *Term Storage.* The decision to store terms uniquely, which was briefly discussed in Sec. 3.4, 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 4) 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,[6] the modest (but not negligible) time overhead at term construction time is one hash table lookup.

We get two returns on this investment. First, the amount of space gained by

---

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

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. Second, `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. 6.4).

6.1.3 *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.

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, Kamperman, and Walters [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.

6.1.4 *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, there is no portable and efficient way in C to detect when local variables are no longer in use. 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.

## 6.2 Matching

6.2.1 *Term Matching.* After collecting the rules making up a function definition (Sec. 5.1), 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]. For reasons of separate compilation, each generated C function has its own local matching automaton, unlike, for instance, the compiler for the Elan rewriting logic language [Moreau and Kirchner 1998], which generates a single large 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. 6.1.2). If neither left-hand side matches,

the appropriate normal form is constructed by ATerm library function `make_nf3`
(Table 4).

Slightly better code could be obtained by dropping the left-to-right bias of the
generated automaton[7] 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)
}.
```

6.2.2 *List Matching.* As was pointed out in Sec. 5.6, 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. 4.2).

Consider the ASF+SDF rule

```
[s-1] {Id0*,Id,Id1*,Id,Id2*} = {Id0*,Id,Id1*,Id2*},
```

which makes lists into sets by removing elements that occur more than once. Its
$\mu$ASF representation would be

```
[s-1] set(conc(*Id0,conc(Id,conc(*Id1,conc(Id,*Id2))))) =
      set(conc(*Id0,conc(Id,conc(*Id1,*Id2)))),
```

where `set` is some prefix representation of the user-defined accolade notation for sets
used in the ASF+SDF rule, and `conc` is the predefined associative list concatenation
of $\mu$ASF. Each application of [s-1] picks up the leftmost pair of elements occurring
more than once in variable `Id` and keeps only a single occurrence in its right-hand
side. List variables `*Id1`, `*Id2`, and `*Id3`, each of which can match the empty list,
are used to pick up and transfer the context.

Since rule [s-1] is nonlinear, it is first transformed to

```
[s-1'] Id == Id'
         ==>
       set(conc(*Id0,conc(Id,conc(*Id1,conc(Id',*Id2))))) =
       set(conc(*Id0,conc(Id,conc(*Id1,*Id2))))
```

by the preprocessor. The C code generated for rule [s-1'] is shown in Fig. 4.
It consists of two nested while-loops, which try successive values for the three list

---

[7]Nedjah, Walter, and Eldridge [1997] discuss optimization of the matching automaton under a
left-to-right constraint.

```
ATerm set(ATerm arg0) {
  ATerm tmp_0 = arg0;                     /* cursor in argument list */
  ATerm tmp_1[2];                         /* *Id0 (begin and end cursor) */
  tmp_1[0] = tmp_0;
  tmp_1[1] = tmp_0;
  while(not_empty_list(tmp_0)) {
    ATerm tmp_2[2];                       /* *Id1 (begin and end cursor) */
    ATerm tmp_3 = list_head(tmp_0);       /* Id */
    tmp_0 = list_tail(tmp_0);
    tmp_2[0] = tmp_0;
    tmp_2[1] = tmp_0;
    while(not_empty_list(tmp_0)) {
      ATerm tmp_4 = list_head(tmp_0);     /* Id' */
      tmp_0 = list_tail(tmp_0);
      if(term_equal(tmp_3, tmp_4)) {      /* Id = Id' */
        return set(conc(slice(tmp_1[0], tmp_1[1]),
                   conc(tmp_3, conc(slice(tmp_2[0],tmp_2[1]), tmp_0))));
      }
      tmp_2[1] = list_tail(tmp_2[1]);
      tmp_0 = tmp_2[1];
    }
    tmp_1[1] = list_tail(tmp_1[1]);
    tmp_0 = tmp_1[1];
  }
  return make_nf1(setsym,arg0);
}
```

Fig. 4.   Code generated for rule [s-1'].

variables. The various ATerm functions used in it are listed in Table 4. The condition is checked in the body of the innermost loop.

Rule [s-1] is applied as often as needed to reach a normal form containing each element only once, but each application is independent of the previous one, starting from the beginning of the set rather than at the position where the previous application left off. This leaves room for further optimization, but its implementation in sufficiently general form to be effective has turned out to be hard [Vinju 1999].

## 6.3 Evaluation of Conditions and Right-Hand Sides

The code generated for rule [at-1-2'] (Sec. 5.8) is shown in Fig. 5. Before execution starts, *extfun1 and *extfun2 are linked dynamically to, respectively, C functions type_env and pair. The reasons for doing this at run-time are explained in Sec. 6.5. As in the previous example, the various ATerm functions used in the code are listed in Table 4. The $\mu$ASF$^+$ else of the rule corresponds to the first else in the C code.

```
ATerm add_to(ATerm arg0, ATerm arg1, ATerm arg2)
{
  ATerm tmp[6];
  if (check_sym(arg2, extfun1_sym)) {      /* arg2 = type-env(*Pair) */
    ATerm atmp20 = arg_0(arg2);
    if (not_empty_list(atmp20)) {          /* t := non_empty_list(*Pair) */
      tmp[0] = list_head(atmp20);          /* P := list_head(*Pair) */
      tmp[1] = list_tail(atmp20);          /* *Pair1 := list_tail(*Pair) */
      if (check_sym(tmp[0], extfun2_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 (*extfun1)(conc((*extfun2)(arg0, arg1), tmp[1]));


        }
        else {
          tmp[4] = add_to(arg0, arg1, (*extfun1)(tmp[1]));
                        /* tmp[4] = add-to(Id,Type1,type-env(*Pair1)) */
          if (check_sym(tmp[4], extfun1_sym)) {
                                        /* tmp[4] = type-env(*Pair) */
            tmp[5] = arg_0(tmp[4]);
            return (*extfun1)(conc((*extfun2)(tmp[2], tmp[3]), tmp[5]));
          }
        }
      }
    }
    else {
      return (*extfun1)(make_list((*extfun2)(arg0, arg1)));
    }
  }
  return make_nf3(extfun1_sym, arg0, arg1, arg2);
}
```

Fig. 5.   Code generated for rule [at-1-2'].

## 6.4 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[8] along with the corresponding result (normal form) once it
has been computed. When called with a "known" set of arguments, the result is
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. 6.1.2)
combines very well with memoization. Since memo tables tend to contain many

---

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

| register_prod(prod, funptr, symbol) | Add C function pointer funptr and unique symbol symbol generated for function with ASF+SDF identifier prod to symbol table |
|---|---|
| lookup_func(prod) | Get C function pointer for function with ASF+SDF identifier prod |
| lookup_sym(prod) | Get symbol for function with ASF+SDF identifier prod |
| lookup_prod(symbol) | Return ASF+SDF identifier of symbol symbol |

Table 5.    ATerm library functions used for dynamic linking.

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.

### 6.5 Dynamic Linking of ASF+SDF Function Identifiers

Because of the user-defined syntax, an ASF+SDF function identifier corresponds to an SDF grammar production (which is similar to a BNF rule). Mapping such rules to C function identifiers directly is not possible because of length and character set restrictions. To circumvent this problem, we adopted a dynamic linking approach for function identifiers in addition to the usual static linking.

More specifically, for each C file M the compiler maps ASF+SDF function identifiers (productions) to C function identifiers whose uniqueness is not guaranteed beyond the scope of M. This does not require global knowledge. The compiler also generates additional functions register_M and lookup_M for each C file M. These are executed before actual rewriting starts and perform the dynamic linking on the basis of the ASF+SDF function identifiers. For each function defined in M, register_M stores the ASF+SDF identifier along with the corresponding unique C function pointer supplied by the preceding static linkage editing phase in a symbol table using ATerm function register_prod (Table 5). For each external function called from M, lookup_M then obtains a pointer from the symbol table on the basis of the ASF+SDF identifier using ATerm library function lookup_func.

### 7. 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 $\mu\text{ASF}^+$.

| Language | Type of language and semantic characteristics | Compiled to |
|---|---|---|
| ASF+SDF | Language definition formalism<br>• First-order<br>• Strict<br>• Conditional (both pos and neg)<br>• Default rules<br>• A-rewriting (lists) | C |
| Clean<br>[Plasmeijer and van Eekelen 1994]<br>[Smetsers et al. 1991] | Functional language<br>• Higher-order<br>• Lazy<br>• Strictness annotations<br>• Polymorphic typing | Native code via ABC abstract graph rewriting machine |
| Elan<br>[Moreau and Kirchner 1998] | Rewriting logic language<br>• First-order<br>• Strategy specification<br>• AC-rewriting | C |
| Haskell<br>[Peyton Jones et al. 1993]<br>[Peyton Jones 1996] | Functional language<br>• Higher-order<br>• Lazy<br>• Strictness annotations<br>• Polymorphic typing | C |
| Opal<br>[Didrich et al. 1994] | Algebraic programming language<br>• Higher-order<br>• Strict | C |
| SML<br>[Appel 1992] | Functional language<br>• Higher-order<br>• Strict<br>• Polymorphic typing | Native code |

Table 6.    Languages used in the benchmarking of the ASF+SDF compiler.

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 $\mu$ASF interpreter that can be called by the compiler exists, such normalizations potentially require the full definition to be available. This is in conflict with the requirement of separate compilation.

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

## 8.  BENCHMARKING

Table 6 lists some of the semantic features of the languages used in the benchmarking of the ASF+SDF compiler. Modularization aspects are not included. Although the languages listed are all based on some form of rewriting, their authors do not use the same terminology to classify them as can be seen in the second column. At least to some extent, this reflects a difference in orientation and purpose.

Section 8.1 gives results of three benchmarks comparing the compilers for the languages listed in Table 6.  Section 8.2 gives results for two large ASF+SDF definitions.

### 8.1  Three Small Benchmarks

All three benchmarks are based on the normalization of expressions $2^n$ mod 17, with $17 \leq n \leq 23$, where the natural numbers involved are in successor representation (unary representation). They are synthetic benchmarks yielding rewrite intensive computations. 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 [Olivier 1999].

Note that these benchmarks were primarily designed to evaluate specific implementation aspects, such as the effect of subterm sharing, lazy evaluation, and the like. They do not provide an overall comparison of the various systems. Also note that some systems failed to compute results for the full range $17 \leq n \leq 23$. In those cases, the corresponding graph ends prematurely. The possibility to switch subterm sharing off was added to the ASF+SDF compiler only for the purpose of benchmarking. It is not a standard compiler option. Measurements were performed on a SUN ULTRA SPARC-5 (270 MHz) with 512 MB of memory.

8.1.1  *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. 6. The differences between ASF+SDF, Clean, Haskell, and SML are small. Even in this case, maximal subterm sharing is effective in the sense that ASF+SDF without sharing performs less well, largely as a consequence of the less efficient evaluation of `term_equal` (Sec. 6.1.2), but it does not yield a speed-up with respect to Clean, Haskell, and SML. This shows maximal subterm sharing to be an effective substitute for the sophisticated optimization techniques used by some of the other compilers. This is further confirmed by the following two benchmarks.

8.1.2  *The* `evalexp` *Benchmark.*  The second benchmark is called `evalexp` and is based on an algorithm that uses a lot of memory when a typical strict implementation is used. Using a lazy implementation, the amount of memory needed is relatively small.

Memory usage is shown in Figure 7. Clearly, strict implementations that do not use maximal subterm sharing cannot cope with the excessive memory requirements of this benchmark, but ASF+SDF and Clean (lazy) have no problems whatsoever.

Execution times are plotted in Figure 8. Only Clean (lazy) is faster than ASF+SDF, but the differences are small.

8.1.3  *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 imple-

Fig. 6. Execution times for the evalsym benchmark

Centrum voor Wiskunde en Informatica

**REPORT**RAPPORT

Asf+Sdf

3000

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Fig. 7. Memory usage for the evalexp benchmark

Centrum voor Wiskunde en Informatica

**REPORT**RAPPORT

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

140000

Asf+Sdf

Fig. 8.   Execution times for the `evalexp` benchmark

Centrum voor Wiskunde en Informatica

**REPORT**RAPPORT

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Fig. 9.  Memory usage for the `evaltree` benchmark

Asf+Sdf

Centrum voor

REPOR

Centrum voor Wiskunde en Informatica

**REPORT***RAPPORT*

Compiling Language Definitions: The ASF+SDF Compi

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

140000

Copyri
P.O.
Kr

Fig. 10.  Execution times for the `evalfree` benchmark

Centrum voor Wiskunde en Informatica

**REPORT**RAPPORT

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

Compiling Language Definitions: The ASF+SDF Compiler

M.G.J. van den Brand, J. Heering, P. Klint, P.A. Olivier

Software Engineering (SEN)

**SEN-R0014 May 31, 2000**

| Definition | ASF+SDF (rules) | ASF+SDF (lines) | Generated C code (lines) | ASF+SDF to C compilation time (s) | C compilation time (s) |
|---|---|---|---|---|---|
| ASF+SDF compiler | 1876 | 8699 | 85185 | 216 | 323 |
| Risla expander | 1082 | 7169 | 46787 | 168 | 531 |

Table 7.    Size and compilation time for two large ASF+SDF definitions.

| Application | Time (s) | Memory (MB) |
|---|---|---|
| ASF+SDF compiler (with sharing) | 216 | 16 |
| ASF+SDF compiler (without sharing) | 661 | 117 |
| Risla expansion (with sharing) | 9 | 8 |
| Risla expansion (without sharing) | 18 | 13 |

Table 8.    Performance of two large ASF+SDF definitions with and without maximal subterm sharing.

mentations. Figure 9 shows that neither the lazy nor the strict implementations can cope with the memory requirements of this benchmark. ASF+SDF is the only one that scales up for $n > 20$. It can keep memory requirements at an acceptable level due to its maximal subterm sharing. The execution times are shown in Figure 10.

## 8.2 Two Large ASF+SDF Definitions

Table 7 gives some statistics for two large ASF+SDF definitions whose performance is shown in Table 8. The ASF+SDF compiler was written in ASF+SDF itself, so the top entry in the fourth column of Table 7 gives the self-compilation time. The language Risla is a domain-specific language for loans, mortgages, and other financial products offered by banks [van den Brand et al. 1996; van Deursen and Klint 1998]. The expander is the first phase of the Risla implementation. It brings Risla specifications in normal form by eliminating their modular structure (if any) [Arnold et al. 1995]. The C compilation times in the last column were obtained using SUN's native C compiler with maximal optimizations.

Table 8 gives performance figures for the compiled versions both with and without maximal subterm sharing of ATerms (Sec. 6.1.2). The time obtained for the ASF+SDF compiler with sharing is, of course, again the self-compilation time.

## 9. CONCLUSIONS AND FURTHER WORK

The ASF+SDF compiler generates high quality C code in a relatively straightforward 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. 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. 6.2.2, 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.

## ACKNOWLEDGMENTS

## 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. Technical Report 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.

ARNOLD, B. R. T., VAN DEURSEN, A., AND RES, M. 1995. An algebraic specification of a language for describing financial products. In M. WIRSING Ed., *ICSE-17 Workshop on Formal Methods Application in Software Engineering* (April 1995), pp. 6–13. IEEE.

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

BERGSTRA, J. A. AND VAN DEN BRAND, M. G. J. 2000. Syntax and semantics of a high-level intermediate representation of ASF+SDF. Technical report, CWI, Amsterdam. To appear.

BERGSTRA, J. A., DINESH, T. B., FIELD, J., AND HEERING, J. 1997. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems 19*, 5 (September), 639–684.

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 & Experience 30*, 259–291.

VAN DEN BRAND, M. G. J., VAN DEURSEN, A., KLINT, P., KLUSENER, S., AND VAN DER MEULEN, E. A. 1996. Industrial applications of ASF+SDF. In M. WIRSING AND M. NIVAT Eds., *Algebraic Methodology and Software Technology (AMAST '96)*, Volume 1101 of *Lecture Notes in Computer Science* (1996). Springer-Verlag.

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 *Proceed-*

*ings of ASF+SDF '95*, Technical Report P9504 (1995), pp. 29–52. Programming Research Group, University of Amsterdam.

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

VAN DEN BRAND, M. G. J., KLINT, P., AND VERHOEF, C.    1996.    Core technologies for system renovation. In K. G. JEFFERY, J. KRÁL, AND M. BARTOŠEK Eds., *Theory and Practice of Informatics (SOFSEM '96)*, Volume 1175 of *Lecture Notes in Computer Science* (1996), pp. 235–255. Springer-Verlag.

VAN DEN BRAND, M. G. J., KLINT, P., AND VERHOEF, C.    1997.    Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices 32*, 2, 54–61.

VAN DEN BRAND, M. G. J., KUIPERS, T., MOONEN, L., AND OLIVIER, P. A.    1997.    Implementation of a prototype for the new ASF+SDF meta-environment. In M. P. A. SELLINK Ed., *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF '97)*, Electronic Workshops in Computing (1997). Springer/British Computer Society.

VAN DEN BRAND, M. G. J., SELLINK, A., AND VERHOEF, C.    1998.    Control flow normalization for COBOL/CICS legacy systems. In *Euromicro Software Maintenance and Reengineering* (1998). IEEE Computer Society.

VAN DEN BRAND, M. G. J., SELLINK, M., AND VERHOEF, C.    1997.    Reengineering COBOL software implies specification of underlying dialects. Technical Report P9702, University of Amsterdam, Programming Research Group.

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. AND VISSER, E.    1996.    Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology 5*, 1–41.

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

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 J. VAN LEEUWEN Ed., *Handbook of Theoretical Computer Science, volume B*, pp. 243–320. Elsevier Science Publishers.

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*, Volume 5 of *AMAST Series in Computing*. 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 J.-P. FINANCE Ed., *Fundamental Approaches to Software Engineering (FASE '99)*, Volume 1577 of *Lecture Notes in Computer Science* (1999), pp. 1–21. Springer-Verlag.

VAN DEURSEN, A. AND KUIPERS, T.    1998.    Rapid system understanding: Two COBOL case studies. In S. TILLEY AND G. VISAGGIO Eds., *Sixth International Workshop on Program Comprehension (IWPC '98)* (1998), pp. 90–98. IEEE Computer Society.

VAN DEURSEN, A. AND KUIPERS, T.    1999.    Finding objects using cluster and concept analysis. In *Proceedings 21st International Conference on Software Engineering (ICSE '99)* (1999). ACM. To appear.

VAN DEURSEN, A. AND MOONEN, L.    1998.    Type inference for COBOL systems. In I. BAXTER, A. QUILICI, AND C. VERHOEF Eds., *Proceedings 5th Working Conference on Reverse Engineering* (1998), pp. 220–230. IEEE Computer Society.

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

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

DINESH, T. B., HAVERAAEN, M., AND HEERING, J.   1998.   An algebraic programming style for numerical software and its optimization. Technical Report SEN-R9844, CWI, Amsterdam. ACM CoRR E-print Server xxx.lanl.gov/abs/cs.SE/9903002.

DINESH, T. B. AND TIP, F.   1992.   Animators and error reporters for generated programming environments. Report CS-R9253, CWI, Amsterdam.

DINESH, T. B. AND TIP, F.   1997.   A slicing-based approach for locating type errors. In *Proceedings USENIX Conference on Domain-Specific Languages* (1997).

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, June 1992), pp. 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.

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)* (1995), pp. 57–68. IEEE.

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.

HEERING, J., KLINT, P., AND REKERS, J.   1990.   Incremental generation of parsers. *IEEE Transactions on Software Engineering SE-16*, 1344–1351.

HEERING, J., KLINT, P., AND REKERS, J.   1992.   Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems 14*, 490–520.

HEERING, J., KLINT, P., AND REKERS, J.   1994.   Lazy and incremental program generation. *ACM Transactions on Programming Languages and Systems 16*, 1010–1023.

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.

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 P. LESCANNE Ed., *Rewriting Techniques and Applications (RTA '87)*, Volume 256 of *Lecture Notes in Computer Science* (1987), pp. 25–41. Springer-Verlag.

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 S. ABRAMSKY, D. GABBAY, AND T. S. E. MAIBAUM Eds., *Handbook of Logic in Computer Science, volume 2*, pp. 1–116. Oxford University Press.

VAN DER MEULEN, E. A. 1996. Incremental typechecking. In A. VAN DEURSEN, J. HEERING, AND P. KLINT Eds., *Language Prototyping: An Algebraic Specification Approach*, Volume 5 of *AMAST Series in Computing*, pp. 199–248. World Scientific.

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

MOREAU, P.-E. AND KIRCHNER, H. 1998. A compiler for rewrite programs in associative-commutative theories. In C. PALAMIDESSI, H. GLASER, AND K. MEINKE Eds., *Principles of Declarative Programming (PLILP '98/ALP '98)*, Volume 1490 of *Lecture Notes in Computer Science* (1998), pp. 230–249. Springer-Verlag.

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 M. HANUS, J. HEERING, AND K. MEINKE Eds., *Algebraic and Logic Programming (ALP '97/HOA '97)*, Volume 1298 of *Lecture Notes in Computer Science* (1997), pp. 273–286. Springer-Verlag.

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

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

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* (1993), pp. 249–257. DTI/SERC.

PEYTON JONES, S. L., NORDIN, T., AND OLIVA, D. 1998. `C--`: A portable assembly language. In C. CLACK, K. HAMMOND, AND T. DAVIE Eds., *Implementation of Functional Languages (IFL '97)*, Volume 1467 of *Lecture Notes in Computer Science* (1998), pp. 1–19. Springer-Verlag.

PLASMEIJER, M. J. AND VAN EEKELEN, M. C. J. D. 1994. Concurrent CLEAN—version 1.0—Language refence manual. Technical report, 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. Technical Report CS-R9269, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

SELLINK, M. P. A., SNEED, H. M., AND VERHOEF, C. 1999. Restructuring of COBOL/CICS legacy systems. In P. NESI AND C. VERHOEF Eds., *Proceedings of the Third European Conference on Maintenance and Reengineering* (1999), pp. 72–82. IEEE Computer Society. adam.wins.uva.nl/~x/cics/cics.html.

SELLINK, M. P. A. AND VERHOEF, C. 1999. Generation of software renovation factories from compilers. In H. YANG AND L. WHITE Eds., *International Conference on Software Maintenance (ICSM '99)* (1999), pp. 245–255. IEEE Computer Society. adam.wins.uva.nl/~x/com/com.html.

SELLINK, M. P. A. AND VERHOEF, C. 2000. Development, assessment, and reengineering of language descriptions. In J. EBERT AND C. VERHOEF Eds., *Fouth European Conference on Software Maintenance and Reengineering* (2000). 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 J. HUGHES Ed., *Functional Programming and Computer Architecture (FPCA '91)*, Volume 524 of *Lecture Notes in Computer Science* (1991), pp. 592–617. Springer-Verlag.

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

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. Master'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. www.cs.uu.nl/∼visser/publications/ftp/Vis97.thesis.ps.gz.

WALTERS, H. R.  1997.    Epic and ARM—User's guide. Technical Report SEN-R9724, CWI, Amsterdam.