Compilation and Memory Management for ASF+SDF

Mark van den Brand[1], Paul Klint[1,2] and Pieter Olivier[2]

[1] CWI, Department of Software Engineering
Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands
[2] University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

Abstract. Can formal specification techniques be scaled-up to industrial problems such as the development of domain-specific languages and the renovation of large COBOL systems?
We have developed a compiler for the specification formalism ASF+SDF that has been used successfully to meet such industrial challenges. This result is achieved in two ways: the compiler performs a variety of optimizations and generates efficient C code, and the compiled code uses a run-time memory management system based on maximal subterm sharing and mark-and-sweep garbage collection.
We present an overview of these techniques and evaluate their effectiveness in several benchmarks. It turns out that execution speed of compiled ASF+SDF specifications is at least as good as that of comparable systems, while memory usage is in many cases an order of magnitude smaller.

1 Introduction

Efficient implementation based on mainstream technology is a prerequisite for the application and acceptance of declarative languages or specification formalisms in real industrial settings. The main characteristic of industrial applications is their *size* and the predominant implementation consideration should therefore be the ability to handle huge problems.

In this paper we take the specification formalism ASF+SDF [5,19,15] as point of departure. Its main focus is on language prototyping and on the development of language specific tools. ASF+SDF is based on general context-free grammars for describing syntax and on conditional equations for describing semantics. In this way, one can easily describe the syntax of a (new or existing) language and specify operations on programs in that language such as static type checking, interpretation, compilation or transformation. ASF+SDF has been applied successfully in a number of industrial projects [9,11], such as the development of a domain-specific language for describing interest products (in the financial domain) [4] and a renovation factory for restructuring of COBOL code [12]. In such industrial applications, the execution speed is very important, but when processing huge COBOL programs memory usage becomes a critical issue as well. Other applications of ASF+SDF include the development of a GLR parser generator [26], an unparser generator [13], program transformation tools [14], and the compiler discussed in this paper. Other components, such as parsers, structure editors, and interpreters, are developed in ASF+SDF as well but are not (yet) compiled to C.

What are the performance standards one should strive for when writing a compiler for, in our case, an algebraic specification formalism? Experimental, comparative, studies are scarce, one notable exception is [18] where measurements are collected for various declarative programs solving a single real-world problem. In other studies it is no exception that the units of measurement (rewrite steps/second, or logical inferences/second) are ill-defined and that memory requirements are not considered due to the small size of the input problems.

In this paper, we present a compiler for ASF+SDF that performs a variety of optimizations and generates efficient C code. The compiled code uses a run-time memory management system based on maximal subterm sharing and mark-and-sweep garbage collection. The contribution of this paper is to bring the performance of executable specifications based on term rewriting into the realm of industrial applications.

In the following two subsections we will first give a quick introduction to ASF+SDF (the input language of the compiler to be described) and to μASF (the abstract intermediate representation used internally by the compiler). Next, we describe the generation of C code (Section 2) as well as memory management (Section 3). Section 4 is devoted to benchmarking. A discussion in Section 5 concludes the paper.

1.1 Specification Language: ASF+SDF

The specification formalism ASF+SDF [5,19] is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. An overview can be found in [15]. As an illustration, Figure 1 presents the definition of the Boolean datatype in ASF+SDF. ASF+SDF specifications consist of modules, each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations). The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation since arbitrary context-free grammars can be defined. The syntax defined in the SDF-part of a module can be used immediately when defining equations, the syntax in equations is thus *user-defined*.

The emphasis in this paper will be on the compilation of the equations appearing in a specification. They have the following distinctive features:

– Conditional equations with positive and negative conditions.
– Non left-linear equations.
– List matching.
– Default equations.

It is possible to execute specifications by interpreting the equations as conditional rewrite rules. The semantics of ASF+SDF is based on innermost rewriting. Default equations are tried when all other applicable equations have failed, because either the arguments did not match or one of the conditions failed.

```
imports Layout
exports
  sorts BOOL
  context-free syntax
    true                  → BOOL  {constructor}
    false                 → BOOL  {constructor}
    BOOL "|" BOOL         → BOOL  {left}
    BOOL "&" BOOL         → BOOL  {left}
    BOOL "xor" BOOL       → BOOL  {left}
    not BOOL              → BOOL
    "(" BOOL ")"          → BOOL  {bracket}
  variables
    Bool [0-9′]∗ → BOOL
  priorities
    BOOL "|"BOOL → BOOL < BOOL "xor"BOOL → BOOL <
    BOOL "&"BOOL → BOOL < notBOOL → BOOL
equations
[B1]   true | Bool  = true       [B5]   not false       = true
[B2]   false | Bool = Bool       [B6]   not true        = false
[B3]   true & Bool  = Bool       [B7]   true xor Bool   = not Bool
[B4]   false & Bool = false      [B8]   false xor Bool  = Bool
```

Fig. 1. ASF+SDF specification of the Booleans.

One of the powerful features of the ASF+SDF specification language is list matching. Figure 2 shows a single equation which removes multiple occurrences of identifiers from a set. In this example, variables with a *-superscript are list-variables that may match zero or more identifiers. The implementation of list matching may involve backtracking to find a match that satisfies the left-hand side of the rewrite rule as well as all its conditions. There is only backtracking within the scope of a rewrite rule, so if the right-hand side of the rewrite rule is normalized and this normalization fails *no* backtracking is performed to find a new match.

The development of ASF+SDF specifications is supported by an interactive programming environment, the ASF+SDF Meta-Environment [23]. In this environment specifications can be developed and tested. It provides syntax-directed editors, a parser generator, and a rewrite engine. Given this rewrite engine terms can be reduced by interpreting the equations as rewrite rules. For instance, the term `true & (false|true)` reduces to `true` when applying the equations of Figure 1.

1.2 Intermediate Representation Language: μASF

The user-defined syntax that may be used in equations poses two major implementation challenges.

imports Layout
exports
  sorts ID Set
  lexical syntax
    [a-z][a-z0-9]$*$ $\rightarrow$ ID
  context-free syntax
    "{" {ID ","}$*$ "}" $\rightarrow$ Set
hiddens
  variables
    Id "$*$"[*0-9*]$*$ $\rightarrow$ {ID ","}$*$
    Id [*0-9′*]$*$ $\rightarrow$ ID
equations

[1] $\{Id^*_0, Id, Id^*_1, Id, Id^*_2\}$
  $= \{Id^*_0, Id, Id^*_1, Id^*_2\}$

Fig. 2. The Set equation in Asf+Sdf.

```
module Booleans
signature
  true;     or(_,_);
  false;    xor(_,_);
  and(_,_); not(_);
rules
  and(true,B) = B;
  and(false,B) = false;
  or(true,B) = true;
  or(false,B) = B;
  not(true) = false;
  not(false) = true;
  xor(true,B) = not(B);
  xor(false,B) = B;
```

Fig. 3. The Booleans in μAsf.

First, how do we represent Asf+Sdf specifications as parse trees? Recall that there is no fixed grammar since the basic Asf+Sdf-grammar can be extended by the user. The solution we have adopted is to introduce the intermediate format AsFix (Asf+Sdf fixed format) which is used to represent the parse trees of the Asf+Sdf modules in a format that is easy processable by a machine. The user-defined syntax is replaced by prefix functions. The parse trees in the AsFix format are self contained.

Second, how do we represent Asf+Sdf specifications in a more abstract form that is suitable as compiler input? We use a simplified language μAsf as an intermediate representation to ease the compilation process and to perform various transformations before generating C code. μAsf is in fact a single sorted (algebraic) specification formalism that uses only prefix notation. μAsf can be considered as the abstract syntax representation of Asf+Sdf. AsFix and μAsf live on different levels, μAsf is only visible within the compiler whereas AsFix serves as exchange format between the various components, such as structure editor, parser, and compiler.

A module in μAsf consists of a module name, a list of functions, and a set of equations. The main differences between μAsf and Asf+Sdf are:

– Only prefix functions are used.
– The syntax is fixed (eliminating lexical and context-free definitions, priorities, and the like).
– Lists are represented by binary list constructors instead of the built-in list construct as in Asf+Sdf; associative matching is used to implement list matching.
– Functions are untyped, only their arity is declared.

– Identifiers starting with capitals are variables; variable declarations are not needed.

Figure 3 shows the μASF specification corresponding to the ASF+SDF specification of the Booleans given earlier in Figure 1[1]. Figure 4 shows the μASF specification of sets given earlier in Figure 2. Note that this specification is not left-linear since the variable `Id` appears twice on the left-hand side of the equation. The `{list}` function is used to mark that a term is a list. This extra function is needed to distinguish between a single element list and an ordinary term, e.g., `{list}(a)` *versus* `a` or `{list}(V)` *versus* `V`. An example of a transformation on μASF specifications is shown in Figure 5, where the non-left-linearity has been removed from the specification in Figure 4 by introducing new variables and an auxiliary condition.

```
module Set
signature
  {list}(_);
  set(_);
  conc(_,_);
  t;
rules
set({list}(conc(*Id0,
  conc(Id,conc(*Id1,
    conc(Id,*Id2)))))) =
set({list}(conc(*Id0,
 conc(Id,conc(*Id1,*Id2)))));
```

Fig. 4. μASF specification of Set.

```
module Set
signature
  {list}(_);
  set(_);
  conc(_,_);
  t;
  term-equal(_,_);
rules
term-equal(Id1,Id2) == t ==>
set({list}(conc(*Id0,
  conc(Id1,conc(*Id1,
    conc(Id2,*Id2)))))) =
set({list}(conc(*Id0,
 conc(Id1,conc(*Id1,*Id2)))));
```

Fig. 5. Left-linear specification of Set.

2 C Code Generation

The ASF compiler uses μASF as intermediate representation format and generates C code as output. The compiler consists of several independent phases that gradually simplify and transform the μASF specification and finally generate C code.

A number of transformations is performed to eliminate "complex" features such as removal of non left-linear rewrite rules, simplification of matching patterns, and the introduction of "assignment" conditions (conditions that introduce

[1] To increase the readability of the generated code in this paper, we have consistently renamed generated names by more readable ones, like **true**, **false**, etc.

new variable bindings). Some of these transformations are performed to improve the efficiency of the resulting code whereas others are performed to simplify code generation.

In the last phase of the compilation process C code is generated which implements the rewrite rules in the specification using adaptations of known techniques [22,17]. Care is taken in constructing an efficient matching automaton, identifying common and reusable (sub)expressions, and efficiently implementing list matching. For each μASF function (even the constructors) a separate C function is generated. The right-hand side of an equation is directly translated to a function call, if necessary. A detailed description of the construction of the matching automaton is beyond the scope of this paper, a full description of the construction of the matching automaton can be found in [10]. Each generated C function contains a small part of the matching automaton, so instead of building one big automaton, the automaton is split over the functions. The matching automaton respects the syntactic specificity of the arguments from left to right in the left-hand sides of the equations. Non-variable arguments are tried before the variable ones.

The datatype `ATerm` (for Annotated Term) is the most important datatype used in the generated C code. It is provided by a run-time library which takes care of the creation, manipulation, and storage of terms. ATerms consist of a function symbol and zero or more arguments, e.g., `and(true,false)`. The library provides predicates, such as `check_sym` to check whether the function symbol of a term corresponds to the given function symbol, and functions, like `make_nf`i to construct a term (normal form) given a function symbol and i arguments ($i \geq 0$). There are also access functions to obtain the i-th argument ($i \geq 0$) of a term, e.g., `arg_1(and(true,false))` yields `false`.

The usage of these term manipulation functions can be seen in Figures 6 and 7. Figure 6 shows the C code generated for the `and` function of the Booleans (also see Figures 1 and 3). This C code also illustrates the detection of reusable subexpressions. In the second `if`-statement a check is made whether the first argument of the `and`-function is equal to the term `false`. If the outcome of this test is positive, the first argument `arg0` of the `and`-function is returned rather than building a new normal form for the term `false` or calling the function `false()`. The last statement in Figure 6 is necessary to catch the case that the first argument is neither a `true` or `false` symbol, but some other Boolean normal form.

Figure 7 shows the C code generated for the Set example of Figure 2. List matching is translated into nested while loops, this is possible because of the restricted nature of the backtracking in list matching. The functions `not_empty_list`, `list_head`, `list_tail`, `conc`, and `slice` are library functions which give access to the C data structure which represents the ASF+SDF lists. In this way the generated C code needs no knowledge of the internal list structure. We can even change the internal representation of lists *without adapting the generated C code*, by just replacing the library functions. The function `term_equal` checks the equality of two terms.

```
ATerm and(ATerm arg0, ATerm arg1) {
 if(check_sym(arg0,truesym))
  return arg1;
 if(check_sym(arg0,falsesym))
  return arg0;
 return make_nf2(andsym,arg0,arg1);
}
```

Fig. 6. Generated C code for the `and` function of the Booleans.

When specifications grow larger, *separate compilation* becomes mandatory. There are two issues related to the separate compilation of Asf+Sdf specifications that deserve special attention. The first issue concerns the identification and linking of names appearing in separately compiled modules. Essentially, this amounts to the question how to translate the Asf+Sdf names into C names. This problem arises since a direct translation would generate names that are too long for C compilers and linkage editors. We have opted for a solution in which each generated C file contains a "register" function which stores at run-time for each defined function defined in this C file a mapping between the address of the generated function and the original Asf+Sdf name. In addition, each C file contains a "resolve" function which connects local function calls to the corresponding definitions based on their Asf+Sdf names. An example of registering and resolving can be found in Figure 8.

The second issue concerns the choice of a unit for separate compilation. In most programming language environments, the basic compilation unit is a file. For example, a C source file can be compiled into an object file and several object files can be joined by the linkage editor into a single executable. If we change a statement in one of the source files, that complete source file has to be recompiled and linked with the other object files.

In the case of Asf+Sdf, the natural compilation unit would be the module. However, we want to generate a single C function for each function in the specification (for efficiency reasons) but Asf+Sdf functions can be defined in specifications using multiple equations occurring in several modules. The solution is to use a single function as compilation unit and to *re-shuffle* the equations before translating the specification. Equations are thus stored depending on the module they occur in as well as on their outermost function symbol. When the user changes an equation, only those functions that are actually affected have to be recompiled into C code. The resulting C code is then compiled, and linked together with all other previously compiled functions.

3 Memory Management

At run-time, the main activities of compiled Asf+Sdf specifications are the creation and matching of large amounts of terms. Some of these terms may even

```
ATerm set(ATerm arg0) {
  if(check_sym(arg0,listsym)) {
    ATerm tmp0 = arg_0(arg0);
    ATerm tmp1[2];
    tmp1[0] = tmp0; tmp1[1] = tmp0;
    while(not_empty_list(tmp0)) {
      ATerm tmp3 = list_head(tmp0);
      tmp0 = list_tail(tmp0);
      ATerm tmp2[2];
      tmp2[0] = tmp0; tmp2[1] = tmp0;
      while(not_empty_list(tmp0)) {
        ATerm tmp4 = list_head(tmp0);
        tmp0 = list_tail(tmp0);
        if(term_equal(tmp3,tmp4))
          return set(list(conc(slice(tmp1[0],tmp1[1]),
                      conc(tmp3,conc(slice(tmp2[0],tmp2[1]),
                                     tmp0)))));
        tmp2[1] = list_tail(tmp2[1]);
        tmp0 = tmp2[1];
      }
      tmp1[1] = list_tail(tmp1[1]);
      tmp0 = tmp1[1];
    }
  }
  return make_nf1(setsym,arg0);
}
```

Fig. 7. C code for the Set specification.

be very big (more than 10^6 nodes). The amount of memory used during rewriting depends entirely on the number of terms being constructed and on the amount of storage each term occupies. In the case of innermost rewriting a lot of redundant (intermediate) terms are constructed.

At compile time, we can take various measures to avoid redundant term creation (only the last two have been implemented in the ASF+SDF compiler):

– Postponing term construction. Only the (sub)terms of the normal form must be constructed, *all* other (sub)terms are only needed to direct the rewriting process. By transforming the specification and extending it with rewrite rules that reflect the steering effect of the intermediate terms, the amount of term construction can be reduced. In the context of functional languages this technique is known as *deforestation* [27]. Its benefits for term rewriting are not yet clear.
– Local sharing of terms, only those terms are shared that result from non-linear right-hand sides, e.g., `f(X) = g(X,X)`. Only those terms will be shared

```
void register_xor() {
   xorsym = "prod(Bool xor Bool -> Bool {left})";
   register_prod("prod(Bool xor Bool -> Bool {left})",
                 xor, xorsym);
}
void resolve_xor() {
   true = lookup_func("prod(true -> Bool)");
   truesym = lookup_sym("prod(true -> Bool)");
   false = lookup_func("prod(false -> Bool)");
   falsesym = lookup_sym("prod(false -> Bool)");
   not = lookup_func("prod(not Bool -> Bool)");
   notsym = lookup_sym("prod(not Bool -> Bool)");
}
ATerm xor(ATerm arg0, ATerm arg1) {
   if (check_sym(arg0, truesym))
     return (*not)(arg1);
   if (check_sym(arg0, falsesym))
     return arg1;
   return make_nf2(xorsym,arg0,arg1);
}
```

Fig. 8. Generated C code for the `xor` function of the Booleans.

of which the sharing can be established at compile-time; the amount of sharing will thus be limited. This technique is also applied in ELAN [8].
– Local reuse of terms, i.e., common subterms are only reduced once and their normal form is reused several times. Here again, the common subterm has to be determined at compile-time.

At run-time, there are various other mechanisms to reduce the amount of work:

– Storage of all original terms to be rewritten and their resulting normal forms, so that if the same term must be rewritten again its normal form is immediately available. The most obvious way of storing this information is by means of pairs consisting of the original term and the calculated normal form. However, even for small specifications and terms an explosion of pairs may occur. The amount of data to be manipulated makes this technique useless.
  A more feasible solution is to store only the results of functions that have been explicitly annotated by the user as "memo-function" (see Section 5).
– Dynamic sharing of (sub)terms. This is the primary technique we use and it is discussed in the next subsection.

3.1 Maximal Sharing of Subterms

Our strategy to minimize memory usage during rewriting is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term is reused thus ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be build during rewriting. The library functions to construct normal forms take care of building shared terms whenever possible. The sharing of terms is invisible, so no extra precautions are necessary in the code generated by the compiler.

Maximal sharing of terms can only be maintained when we check at every term creation whether a particular term already exists or not. This check implies a search through all existing terms but must nonetheless be executed *extremely fast* in order not to impose an unacceptable penalty on term creation. Using a hash function that depends on the internal code of the function symbol and the addresses of its arguments, we can quickly search for a function application before creating it. The (modest but not negligible) costs at term creation time are hence one hash table lookup.

Fortunately, we get two returns on this investment. First, the considerably reduced memory usage also leads to reduced (real-time) execution time. Second, we gain substantially since the equality check on terms (`term_equal`) becomes very cheap: it reduces from an operation that is linear in the number of subterms to be compared to a constant operation (pointer equality). Note that the compiler generates calls to `term_equal` in the translation of patterns and conditions.

The idea of subterm sharing is known in the LISP community as *hash consing* and will be discussed below.

3.2 Shared Terms versus Destructive Updates

Terms can be shared in a number of places at the same time, therefore they cannot be modified without causing unpredictable side-effects. This means that all operations on terms should be *functional* and that terms should effectively be *immutable* after creation.

During rewriting of terms by the generated code this restriction causes no problems since terms are created in a fully functional way. Normal forms are constructed bottom-up and there is no need to perform destructive updates on a term once it has been constructed. When normalizing an input term, this term is not modified, the normal form is constructed independent of the input term. If we would modify the input term we would get graph rewriting instead of (innermost) term rewriting. The term library is very general and is *not* only used for rewriting; destructive updates would therefore also cause unwanted side effects in other components based on this term library.

However, destructive operations on lists, like list concatenation and list slicing, become expensive. For instance, the most efficient way to concatenate two lists is to physically replace one of the lists by the concatenation result. In our case, this effect can only be achieved by taking the second list, prepending the elements of the first list to it, and return the new list as result.

In LISP, the success of hash consing [1] has been limited by the existence of the functions `rplaca` and `rplacd` that can destructively modify a list structure. To support destructive updates, one has to support two kinds of list structures "mono copy" lists with maximal sharing and "multi copy" lists without maximal sharing. Before destructively changing a mono copy list, it has to be converted to a multi copy list. In the 1970's, E. Goto has experimented with a Lisp dialect (HLisp) supporting hash consing and list types as just sketched. See [25] for a recent overview of this work and its applications.

In the case of the ASF+SDF compiler, we *generate* the code that creates and manipulates terms and we can selectively generate code that copies subterms in cases where the effect of a destructive update is needed (as sketched above). This explains why we can apply the technique of subterm sharing with more success.

3.3 Reclaiming Unused Terms

During rewriting, a large number of terms is created, most of which will not appear in the end result. These terms are used as intermediate results to guide the rewriting process. This means that terms that are no longer used have to be reclaimed in some way.

After experimentation with various alternatives (reference counting, mark-and-compact garbage collection) we have finally opted for a mark–and-sweep garbage collection algorithm to reclaim unused terms. Mark-and-sweep collection is more efficient, both in time and space than reference counting [20]. The typical space overhead for a mark-sweep garbage collection algorithm is only 1 bit per object.

Mark-and-sweep garbage collection works using three (sometimes two) phases. In the first phase, all the objects on the heap are marked as 'dead'. In the second phase, all objects reachable from the known set of root objects are marked as 'live'. In the third phase, all 'dead' objects are swept into a list of free objects.

Mark-and-sweep garbage collection is also attractive, because it can be implemented efficiently in C and can work without support from the programmer or compiler [7]. We have implemented a specialized version of Boehm's conservative garbage collector [6] that exploits the fact that we are managing ATerms.

4 Benchmarks

Does maximal sharing of subterms lead to reductions in memory usage? How does it affect execution speed? Does the combination of techniques presented in this paper indeed lead to an implementation of term rewriting that scales-up to industrial applications?

To answer these questions, we present in Section 4.1 three relatively simple benchmarks to compare our work with that of other efficient functional and algebraic language implementations. In Section 4.2 we give measurements for some larger ASF+SDF specifications.

Compiler	Time (sec)
Clean (strict)	32.3
SML	32.9
Clean (lazy)	36.9
ASF+SDF (with sharing)	37.7
Haskell	42.4
Opal	75.7
ASF+SDF (without sharing)	190.4
Elan	287.0

Table 1. The execution times for the evaluation of 2^{23}.

4.1 Three Small Benchmarks

All three benchmarks are based on symbolic evaluation of expressions 2^n **mod** 17, with $17 \leq n \leq 23$. A nice aspect of these expressions is that there are many ways to calculate their value, giving ample opportunity to validate the programs in the benchmark [2].

Note that these benchmarks were primarily designed to evaluate *specific* implementation aspects such as the effect of sharing, lazy evaluation, and the like. They cannot (yet) be used to give an overall comparison between the various systems. Also note that some systems failed to compute results for the complete range $17 \leq n \leq 23$ in some benchmarks. In those cases, the corresponding graph also ends prematurely. Measurements were performed on an ULTRA SPARC-5 (270 MHz) with 512 Mb of memory. So far we have used the following implementations in our benchmarks:

– The ASF+SDF compiler as discussed in this paper: we give results *with* and *without* maximal sharing.
– The Clean compiler developed at the University of Nijmegen [24]: we give results for standard (*lazy*) versions and for versions optimized with strictness annotations (*strict*).
– The ELAN compiler developed at INRIA, Nancy [8].
– The Opal compiler developed at the Technische Universität Berlin [16].
– The Glasgow Haskell compiler [21].
– The Standard ML compiler [3].

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. This benchmark is a worst case for our implementation, because little can be gained by maximal sharing. The results are shown in Table 1. The differences between the various systems are indeed small. Although, ASF+SDF (with sharing) cannot benefit from maximal sharing, it does not loose much either.

[2] The actual source can be obtained at
`http://adam.wins.uva.nl/~olivierp/benchmark/index.html`

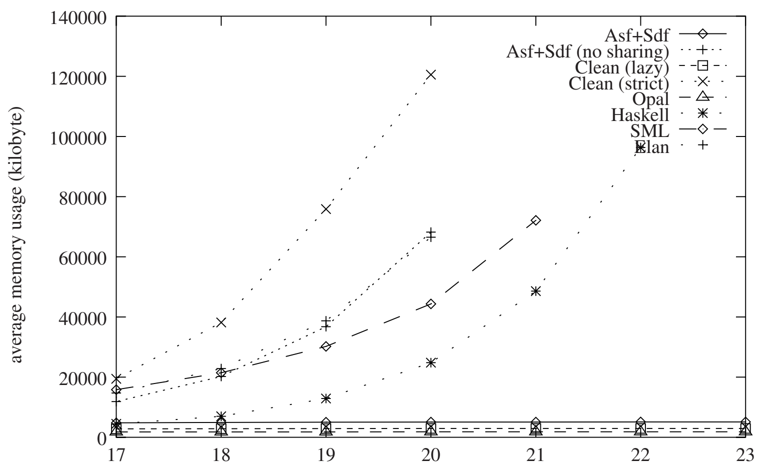

Fig. 9. Memory usage for the `evalexp` benchmark

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

Memory usage is shown in Figure 9. Clearly, normal strict implementations cannot cope with the excessive memory requirements of this benchmark. Interestingly, ASF+SDF (with sharing) has no problems whatsoever due to the use of maximal sharing, although it is also based on strict evaluation

Execution times are plotted in Figure 10. Only Clean (lazy) is faster than ASF+SDF (with sharing) but the differences are small.

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 eager implementations. Figure 11 shows that neither the lazy nor the strict implementations can cope with the memory requirements of this benchmark. Only ASF+SDF (with sharing) can keep the memory requirements at an acceptable level due to its maximal sharing. The execution times plotted in Figure 12 show that only ASF+SDF scales-up for $n > 20$.

4.2 Compilation Times of Larger ASF+SDF Specifications

Table 2 gives an overview of the compilation times of four non-trivial ASF+SDF specifications and their sizes in number of equations, lines of ASF+SDF specification, and generated C code. The ASF+SDF compiler is the specification of the ASF+SDF to C compiler discussed in this paper. The parser generator is an ASF+SDF specification which generates a parse table for an GLR-parser [26]. The COBOL formatter is a pretty-printer for COBOL, this formatter is used

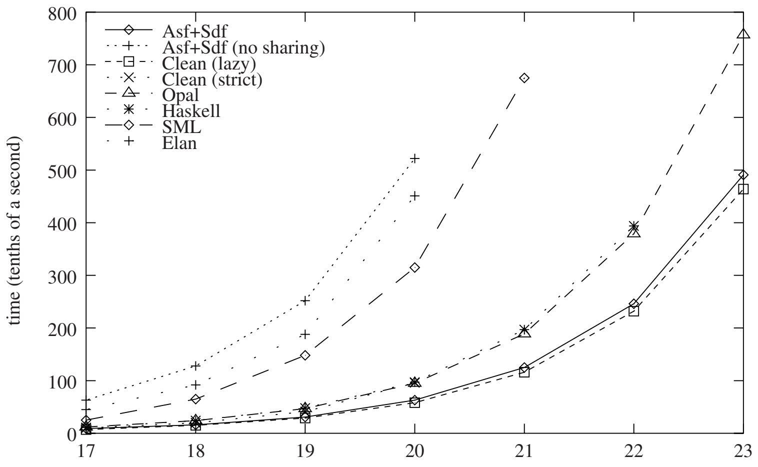

Fig. 10. Execution times for the `evalexp` benchmark

Specification	ASF+SDF (equations)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF compiler (sec)	C compiler (sec)
ASF+SDF compiler	1876	8699	85185	216	323
Parser generator	1388	4722	47662	106	192
COBOL formatter	2037	9205	85976	208	374
Risla expander	1082	7169	46787	168	531

Table 2. Measurements of the ASF+SDF compiler.

within a renovation factory for COBOL [12]. The Risla expander is an ASF+SDF specification of a domain-specific language for interest products, it expands modular Risla specifications into "flat" Risla specifications [4]. These flat Risla specifications are later compiled into COBOL code by an auxiliary tool. The compilation times in the column "ASF+SDF compiler" give the time needed to compile each ASF+SDF specification to C code. Note that the ASF+SDF compiler has been fully bootstrapped and is itself a compiled ASF+SDF specification. Therefore the times in this column give a general idea of the execution times that can be achieved with compiled ASF+SDF specifications. The compilation times in the last column are produced by a native C compiler (SUN's `cc`) with maximal optimizations.

Table 3 gives an impression of the effect of maximal sharing on execution time and memory usage of compiled ASF+SDF specifications. We show the results (with and without sharing) for the compilation of the ASF+SDF to C compiler itself and for the expansion of a non-trivial Risla specification.

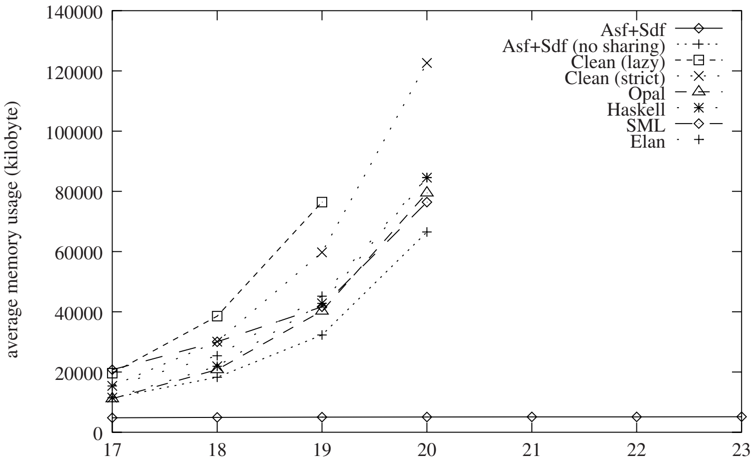

Fig. 11. Memory usage for the `evaltree` benchmark

Application	Time (sec)	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 3. Performance with and without maximal sharing.

5 Concluding Remarks

We have presented the techniques for the compilation of Asf+Sdf to C, with emphasis on memory management issues. We conclude that compiled Asf+Sdf specifications run with speeds comparable to that of other systems, while memory usage is in some cases an order of magnitude smaller. We have mostly used and adjusted existing techniques but their combination in the Asf+Sdf compiler turns out to be very effective.

It is striking that our benchmarks show results that seem to contradict previous observations in the context of SML [2] where sharing resulted in slightly increased execution speed and only marginal space savings. On closer inspection, we come to the conclusion that both methods for term sharing are different and can not be compared easily. We share terms immediately when they are created: the costs are a table lookup and the storage needed for the table while the benefits are space savings due to sharing and a fast equality test (one pointer comparison). In [2] sharing of subterms is *only* determined during garbage collection in order to minimize the overhead of a table lookup at term creation. This implies that local terms that have not yet survived one garbage collection are not yet shared thus loosing most of the benefits (space savings and fast equality

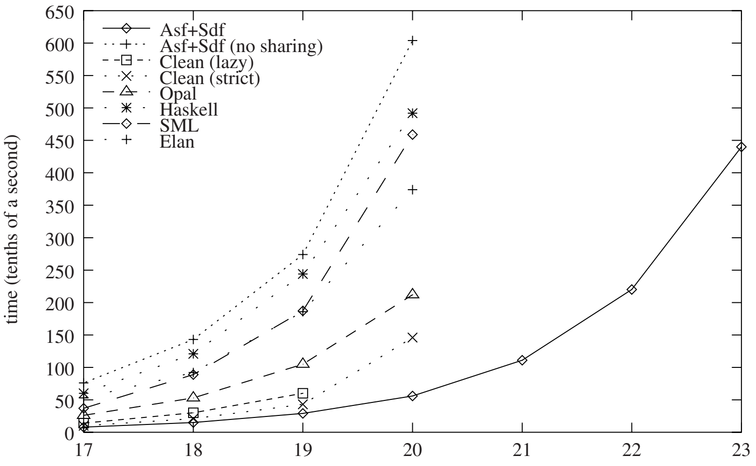

Fig. 12. Execution times for the `evaltree` benchmark

test) as well. The different usage patterns of terms in SML and ASF+SDF may also contribute to these seemingly contradicting observations.

There are several topics that need further exploration. First, we want to study the potential of compile-time analysis for reducing the amount of garbage that is generated at run-time. Second, we have just started exploring the implementation of *memo-functions*. Although the idea of memo-functions is rather old, they have not be used very much in practice due to their considerable memory requirements. We believe that our setting of maximally shared subterms will provide a new perspective on the implementation of memo-functions. Finally, our ongoing concern is to achieve an even further scale-up of prototyping based on term rewriting.

Acknowledgments

The discussions with Jan Heering on ASF+SDF compilation are much appreciated. The idea for the benchmarks in Section 4.1 originates from Jan Bergstra. Reference [2] was pointed out to us by one of the referees.

References

1. J.R. Allen. *Anatomy of LISP*. McGraw-Hill, 1978. 207
2. A.W. Appel and M.J.R. Goncalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993. 212, 213
3. A.W. Appel and D. MacQueen. A standard ML compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, LNCS, pages 301–324, 1987. 209

4. B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995. 198, 211
5. J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press/Addison-Wesley, 1989. 198, 199
6. H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 28, 6, pages 197–206, June 1993. 208
7. H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience (SPE)*, 18(9):807–820, 1988. 208
8. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1996. 206, 209
9. M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and A.E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996. 198
10. M.G.J. van den Brand, J. Heering, P. Klint, and P.A. Olivier. Compiling rewrite systems: The asf+sdf compiler. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1999. In preparation. 203
11. M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 139–161. Elsevier Science, 1998. 198
12. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. 198, 211
13. M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996. 198
14. J.J. Brunekreef. A transformation tool for pure Prolog programs. In J.P. Gallagher, editor, *Logic Program Synthesis and Transformation. Proceedings of the 6th International Workshop, LOPSTR'96*, volume 1207 of *LNCS*, pages 130–145. Springer-Verlag, 1996. 198
15. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996. 198, 199
16. K. Didrich, A. Fett, C. Gerke, W. Grieskamp, and P. Pepper. OPAL: Design and implementation of an algebraic programming language. In J. Gutknecht, editor, *International Conference on Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer-Verlag, 1994. 209
17. C.H.S. Dik. A fast implementation of the Algebraic Specification Formalism. Master's thesis, University of Amsterdam, Programming Research Group, 1989. 203
18. P.H. Hartel et al. Benchmarking implementations of functional languages with 'pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6:621–655, 1996. 199

19. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Most recent version available at URL: http://www.cwi.nl/∼gipe/. 198, 199
20. R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996. 208
21. S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler. The glasgow haskell compiler: a technical overview. *Proc. Joint Framework for Information Technology (JFIT) Technical Conference*, pages 249–257, 1993. 209
22. S. Kaplan. A compiler for conditional term rewriting systems. In P. Lescanne, editor, *Proceedings of the First International Conference on Rewriting Techniques*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 1987. 203
23. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993. 200
24. M.J. Plasmeijer and M.C.J.D. van Eekelen. *Concurrent Clean - version 1.0 - Language Reference Manual, draft version*. Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands, 1994. 209
25. M. Terashima and Y. Kanada. HLisp—its concept, implementation and applications. *Journal of Information Processing*, 13(3):265–275, 1990. 208
26. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997. 198, 210
27. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 22 June 1990. 205