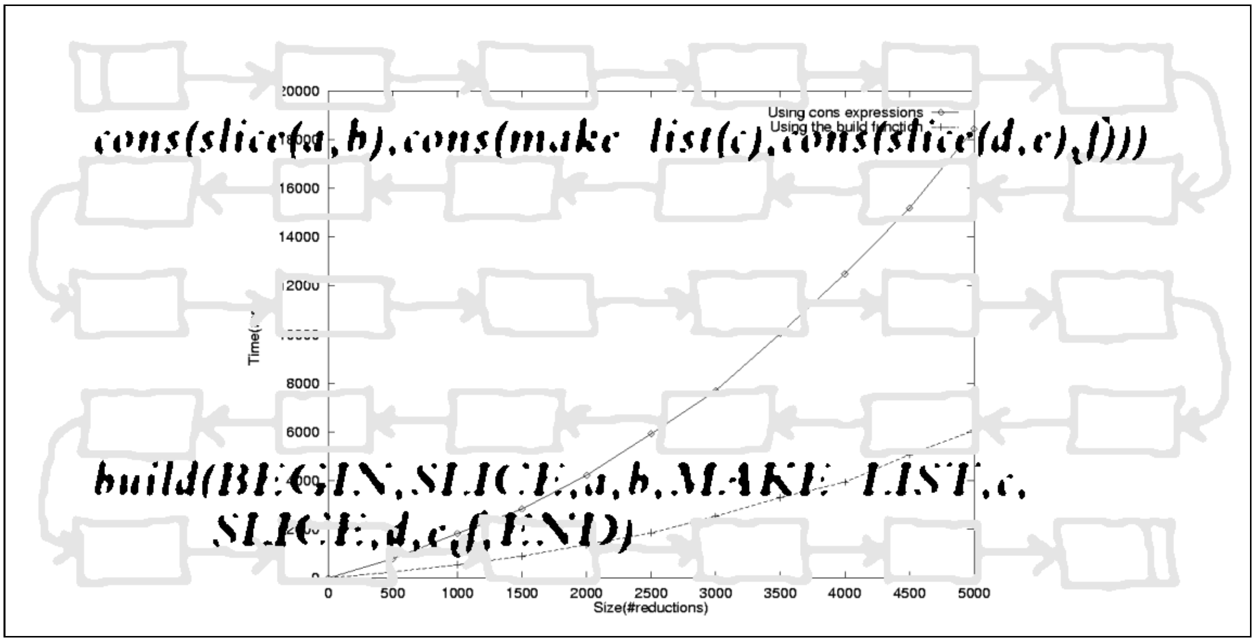
University of Amsterdam
Programming Research Group

Optimizations of List Matching in the ASF+SDF Compiler



Jurgen J. Vinju

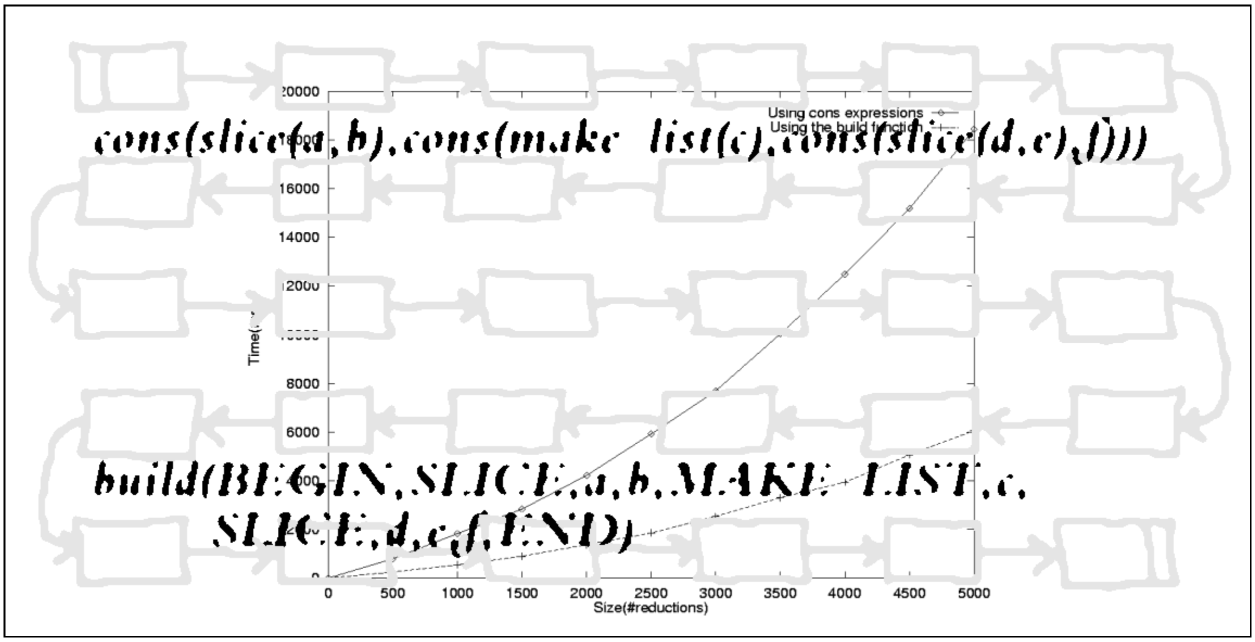
Master's thesis November 1999

University of Amsterdam
Programming Research Group

Supervisors: Prof. dr. P. Klint and dr. M.G.J. van den Brand

University of Amsterdam
Programming Research Group

Optimizations of List Matching in the ASF+SDF Compiler



Supervisors: Prof. dr. P. Klint and dr. M.G.J. van den Brand

Jurgen J. Vinju

Master's thesis November 1999

Acknowledgments

I would like to thank Mark van den Brand for the initial subject of this thesis and his continuous scientific and personal support. For numerous discussions and well advise, I thank Pieter Olivier and Coen Visser. I want to mention Chris Verhoef and Alex Sellink because they are great sources of inspiration.

I honor my parents, Annelies and Fred Vinju, and my sister Krista for always supporting and loving me.

Contents

Chapter 1

Introduction

The subject of this masters thesis is the improvement of the run-time performance of the code generated by the ASF+SDF compiler [4, 9]. The ultimate goal is to improve the run-time performance of *lists* in the generated C code. See [7] for an analysis of the current behavior of the compiler. Since lists are a frequently used feature of ASF+SDF it is interesting to know if their implementation can be optimized. From practical experience with minor and major ASF+SDF specifications we have learned that lists are sometimes a real performance hazard.

Algebraic specification is a rather high level programming paradigm. The lists in ASF+SDF bring it to an even higher level, they provide the programmer with implicit construction of lists and with abstraction from list traversal. It is a challenge to implement these high level features as efficiently as possible.

These are the two separate subjects of interest when optimizing list reduction: construction of lists and list matching. Some ideas for optimizing them are discussed in this thesis. The construction of lists in the generated code is based on a general purpose library. Possible optimizations might be extensions of this library that use information that is specifically available in our context. The search for optimizations in list matching will be more fundamental. We can search for specific classes of problems that have a more efficient rewriting strategy than the general strategy.

This thesis is part of a longterm project to improve the ASF+SDF compiler. It is a general exploration of the possibilities of optimizing list matching. We hope to discover techniques that possibly improve the run-time behavior of compiled specifications.

1.1 Overview

The following sections of the introduction describe ASF+SDF, the ASF+SDF compiler and the ATerm library. This context information is needed because we will refer to it later on in this thesis. It will give the reader a view on the internals of the compiler and the specifics concerning lists. Ideas for optimizations are presented in the next chapters, followed by a chapter that summarizes related work on compilation of list reduction. We conclude with a summary in the final chapter.

1.2 ASF+SDF

ASF+SDF is a general algebraic specification formalism. The SDF part stands for *Syntax Definition Formalism* [15]. This is a formalism for the definition of the syntax of context-free languages. The formalism provides means for defining lexical and context-free grammar rules, associativity, and priorities. Figure 1.1 shows an

```
imports Integers
exports
  sorts Element Set
  context-free syntax
    Int                    → Element
    "[" Element∗ "]"       → Set
hiddens
  variables
    E "*"[0-9]∗ → Element∗
    E [0-9]∗    → Element
equations
```

$[0]\ [E_1^* \ E \ E_2^* \ E \ E_3^*] = [E_1^* \ E \ E_2^* \ E_3^*]$

Figure 1.1: The Set specification in ASF+SDF.

example of an SDF specification. Notice the definition of a *list sort* and the definition of special *list variables*, which are important in this thesis.

The ASF part stands for *Algebraic Specification Formalism*. It can be used for the definition of many sorted algebras. But in its executable form a specification is actually a definition of a rewrite system. The rules of the algebra are interpreted as rewrite rules from left to right. They are called *equations*. Any term that can be matched on a left-hand side of an equation can be rewritten as the right-hand side. Although not adding any computational power, rules in ASF can have a list of conditions. These conditions serve to facilitate programming in ASF+SDF. A rule is only applicable if all conditions are true. There is no priority between normal rules, but for each function a default rule can be defined, which is only applicable if all other rules fail. Equations can be non left-linear.

A special feature of ASF+SDF is *list matching*. ASF has special list sorts. The variables of these sorts can match zero or more terms in a list of terms. The equations section in Figure 1.1 shows how list matching could be used to remove multiple occurrences of a term from a list. This example demonstrates that list matching allows for very elegant specifications. Using list variables all elements of a list are equally accessible without the specification of explicit traversal. This is what is called associative or flat lists[1]. Note that associative lists do not add computational power to ASF. A proof is given in [21], where associative list matching is reduced to term matching. This reduction to term matching is not wanted in the compiler because that would undo the efficiency benefit of a builtin list construct.

The connection between SDF and ASF is made by using the non-terminals in SDF as the sorts in ASF, grammar rules are functions in ASF and any sentence in the language(s) defined in the SDF part is a term of a sort in ASF. The result of combining ASF with SDF is that a specification writer can easily manipulate the abstract syntax representation of parsed input using a rewrite system. Note that not only the syntax of the input is user-defined, but also the syntax of the functions on the input are defined in SDF. Another important feature of ASF+-SDF is modularization. ASF+SDF specifications are divided into modules. Modules import each others functionality using an easy-to-use import mechanism without parameterizability or renaming capabilities.

[1]Compare this to lists in functional languages, for example, where the head of the list is the only accessible element. The tail is the only accessible sublist.

ASF+SDF has been successfully used for the complete and automatic implementation of programming languages[2] and automatic software renovation projects[3]. Also, ASF+SDF is used for the implementation of the ASF+SDF to C compiler and other major in-house projects. Currently, there is a programming environment known as *the* ASF+SDF *Meta-Environment* consisting of syntax-directed editors, a parser generator and term evaluators. The compiler that generates C code from ASF+SDF specifications is part of the new and improved meta-environment that is currently being developed at CWI and UvA. This compiler is the subject of this masters thesis.

Semantics of ASF

We provide the reader with an indication of the semantics of ASF because they form the starting point of the compiler. Of course, any optimization of the compiler should be conservative with respect to the semantics of ASF. A more in depth discussion of the semantics of ASF can be found in [3]. The semantics of ASF are described at the level of μASF. μASF is the abstract syntax representation of ASF+SDF. μASF is actually a single sorted algebraic specification formalism. But μASF specifications have conditions, default rules and list matching just like ASF specifications.

The idea behind the semantics of μASF is to have a deterministic reduction strategy for ASF+SDF specifications. The key points of the semantics are:

- An innermost reduction strategy.
- Conditions are normalized from left to right.
- All conditions must be satisfied before a rule can be applied.
- Default rules are only tried if all other rules fail.
- The ordering of the rules is arbitrary.
- A term is in normal form if no rule can be applied to it.

Apart from these general reduction issues, there is list reduction. A rule containing list variables is called a *list pattern*. List variables in a pattern can match zero or more terms in a list. The result is that an instance of a list can match a pattern in several ways. Some of the matches may not satisfy all conditions, but others might. Therefor we need backtracking over the possible matches within a rule to find a match that does satisfy all conditions. In order for the backtracking to be deterministic, an ordering must be defined on the possible matches. This ordering is defined by:

- Let $list(\vec{X})$ be a list pattern.
- Let $X_1, \ldots, X_k$ be the sequence of list variables in $list(\vec{X})$ in order of appearance.
- A match is a function $\rho : X_i \rightarrow SORT*$ that assigns a sublist to each $X_i \in \vec{X}$.
- Let $|X_i|$ be the length of the list assigned to X_i by ρ.
- $L_i = |X_1|, \ldots, |X_k|$ is a sequence of lengths for a specific match i.

[2]For example: parsers, pretty-printers, type-checkers and interpreters.
[3]For example, a COBOL renovation factory.

```
module Set
signature
  list(_);
  set(_);
  conc(_,_);

rules
set(list(conc(*E1, conc(E, conc(*E2, conc(E, conc(*E3))))))) =
    set(list(conc(*E1, conc(E1, conc(*E2, *E3)))));
```

Figure 1.2: The Set specification in μASF.

Ordering the sequences L_i lexicographically induces an ordering on all possible matches. The reduction strategy of list patterns is defined as reducing the lexicographically first match that meets all conditions. The result is deterministic and finite backtracking within a single rewrite rule.

Having an idea of the semantics of ASF+SDF, we are now ready to discuss the ASF+SDF to C compiler. We now have an indication that lists in ASF+SDF possibly introduce a performance hazard. Namely, they introduce the need for backtracking.

1.3 The ASF+SDF to C compiler

The compilation of ASF+SDF specifications to C is described in [7]. The implementation of the compiler is written as an ASF+SDF specification [9]. The first problem is how to represent ASF specifications as parse trees, since their syntax is defined differently for each specification. This is solved by the introduction of a notation for ASF+SDF specifications called ASFIX[4] [8]. The conversion from ASF+SDF to ASFIX removes the user-defined syntax by writing all functions in prefix notation. Parsed ASF+SDF specifications in ASFIX are like any other language with a fixed syntax and much easier to compile.

The abstract syntax representation of ASFIX inside the compiler is μASF. The μASF specifications are not translated to C code in a single complicated step. A specification is changed gradually as the more complicated features of μASF are resolved by the compiler. The μASF code is only converted to C at the point where the translation from a μASF function to a C function seems rather natural.

Figure 1.4 shows an overview of the phases in the compilation process. The compilation starts with the parsing of ASF+SDF to ASFIX by a separate parser. Then the modules of a specification are re-shuffled, such that each file contains the rules of a single function. Then, roughly, each of these functions runs through the following stages in the compiler:

1. Preprocessing μASF:
   (a) ASFIX is translated to μASF.
   (b) Complex matching conditions are translated to assignment conditions.
   (c) Non-left linear rules are translated to left-linear rules.
   (d) List matching in rules with only one list variable is removed (*).
   (e) Different kinds of conditions are normalized to a single format.
2. Translating μASF to C:

[4] ASFIX is an acronym for ASF+SDF Fixed format

```
ATerm Set(ATerm arg0) {
  if(check_sym(arg0, listsym)) {
    ATerm tmp0 = arg_0(arg0);
    ATerm tmp1[2];
    tmp1[0] = tmp0;
    tmp1[1] = tmp1;

    while(not_empty_list(tmp0)) {
      ATerm tmp3 = list_head(tmp0);
      ATerm tmp2[2];
      tmp0 = list_tail(tmp0);
      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_nf(setsym, arg0);
}
```

Figure 1.3: The Set specification in C. This is generated code, which is slightly moderated in favor of readability.

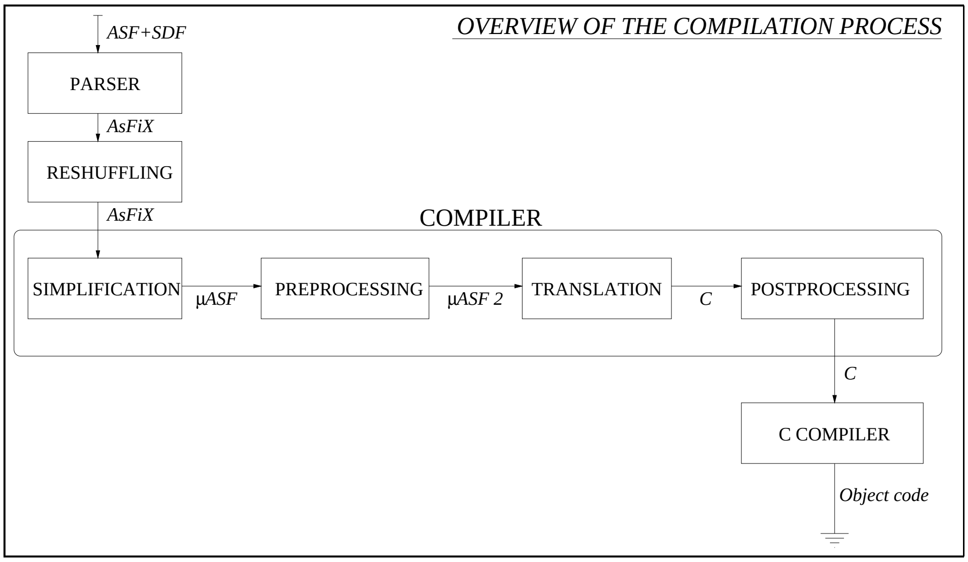


Figure 1.4: Overview of the compilation process.

   (a) Construction of constructor functions.
   (b) Construction of C functions from μASF rules.

3. Post-processing C:
   (a) Tail recursive calls are replaced by goto statements (*).
   (b) The usage of constants is detected and they are reused (*).

Stages marked with (*) are optimizing steps, the others are essential stages in the translation process. The C functions are build from the preprocessed μASF by translating the left-hand sides of the rules to a matching automaton. Also, matching conditions are merged into the matching automaton. The right-hand sides of the rules are translated to function calls. After all, the function symbols on the right-hand side of a rule are translated to C functions themselves. Assignment conditions of the rule are directly translated to C assignments. Notice how the innermost rewriting paradigm smoothly translates to C code because each μASF function is translated to a separate C function.

The generated C code is dependent on a *support* library, which is discussed in the next section. This library provides the compiled specifications with builtin primitives for matching and construction of terms. This library also takes care of garbage collecting, which keeps the generated C code slim and limited to the essence of matching and rewriting.

Some specifics about the translation of list matching need to be discussed. Firstly, the associative lists in AsFix are translated to *cons* notation in μASF(Figure 1.2). This cons notation translates immediately to the *conc* builtin of Table 1.1. The other list construction builtins are introduced only at the translation stage. There, list variables are translated to calls to the *slice* builtin and normal variables of list patterns are translated to calls to the *make_list* builtin.

Secondly, the lexicographically ordered matches are traversed by while loops. Multiple list variables in a pattern correspond to nested while loops. The conditions of a rule are checked in the innermost loop. When all conditions are met, the function returns with a call to the translated constructor functions of the reduct. Figure 1.3 shows the Set specification translated to C.

1.4 The ATerm library

The generated C code uses an abstract data-type called ATerm [16]. The ATerm library provides functionality for the creation and manipulation of terms (in their abstract representation). The ATerm library also provides the user with singly linked lists. Since rewriting is replacing terms, the efficiency of the generated code is very dependent on the implementation of the ATerm library. A lot of effort was invested to make the library both time and memory efficient [5]. The use of the ATerm library is not limited to the compiled ASF+SDF specifications. It is a generic tool used in many applications.

Maybe the most important feature of the ATerm library is *maximal sharing* of terms. This means that only one instance of a specific term is in memory at a specific time. Terms are checked for existence using an efficient hashing algorithm. This technique has proven to be very memory efficient as well as time efficient. For example, due to maximal sharing the equivalence test on terms is reduced to a single pointer comparison. A negative consequence of maximal sharing is that destructive updates are not supported; a completely new instance will be constructed if a term is changed.

The lists in the ATerm library are of most interest in this thesis naturally, because the generated code uses ATerm lists. They are represented by a singly linked list of nodes. Each node contains information on the length, a pointer to the ATerm it holds and a pointer to the next node. List nodes are also fully shared. Please note that this does not imply that every sublist can be shared among lists; only tails can be shared among lists due to the fact that the identity of a list node is partly defined by its reference to the next node. We immediately recognize a performance issue here. For example, if the last element of a list is removed, the entire prefix has to be copied.

The ATerm library is wrapped by the *support* library to make the implementation of generated C code independent of the implementation of the ATerm library. Also the support library introduces some extra functionality specific to the rewriting process and some bookkeeping procedures. Because the ATerm library is used by numerous other projects, it cannot be changed significantly to improve the run-time performance of compiled ASF+SDF specifications. Unless the semantics and other important design properties of the ATerm library remain constant, any optimization considering ATerms must be implemented in the support layer.

Remember how the right-hand sides of rules are translated into function calls. Since lists are a builtin construct, there are library functions needed for building new lists from the matched variables in the left-hand side. See Table 1.1 for a simplified view of their functionality and complexity. These builtins are effectively wrappers of the ATerm library. Writing more specialized builtins for the rewriting process might be beneficial. Notice that *conc*, a very frequently used builtin, is linear in the length of its first argument. This is because the second argument can be reused as a tail. The other builtins need no further specific explanation.

Declaration		Description	Complexity
ATermList	singleton(ATerm t)	Creates a singleton list	$O(1)$
ATerm	list_head(ATermList l)	Returns the head	$O(1)$
ATermList	list_tail(ATermList l)	Returns the tail	$O(1)$
ATermList	conc(ATermList l1, ATermList l2)	Concatenates l1 before l2	$O(\|l1\|)$
Boolean	not_empty_list(ATermList l)	Determines emptiness	$O(1)$
Boolean	is_single_element(ATermList l)	Determines if l is a singleton	$O(1)$
ATermList	slice(ATermList l1, ATermListl2)	Returns the elements between l1 and l2.	$O(\|l2\| - \|l1\|)$
ATermList	make_list(ATerm l)	Creates a singleton if l is not already a ATermList, otherwise it just returns l	$O(1)$

Table 1.1: Interface of the support library for lists.

Chapter 2

Optimizing list construction

The representation of lists in the ATerm library is a singly linked list of ATerms. This is sufficient for finding the lexicografically first match because we can search the list from left to right. The *traversal* primitives of the ATerm library are very fast. But the natural use of the library may prohibit a more efficient implementation of the *construction* of lists in the context of rewriting. We will search for more efficient algorithms or data-structures for the list construction builtins of the support library.

After studying some generated C code, we had an idea that makes the actual creation of slices unnecessary. This is idea is discussed in Section 2.1. Then, in Section 2.2 we investigate the use of destructive lists as opposed to maximally shared lists. Destructive lists might be a solution to the problem that a single deletion of a list element can result in the copying of the entire list.

2.1 Linearization

2.1.1 Motivation

When we take a look at the C code generated from the Set example in Figure 1.3, we see that the translation of the right hand side of the rule is an expression containing the builtin list functions from Table 1.1. A list is a linear construct, while this right-hand side is more like a tree. Each function in this expression tree returns a list that is constructed and kept in memory. The idea is to replace this cons expression tree by a single function, containing all the arguments of the original expression. This *build* function will have all necessary information to build the reduct without the need for intermediate lists[1]. We will have *linearized* the cons expression to a single argument list. This will probably save time as well as space[2]. For example, the result of the linearization of the Set example is given in Figure 2.1.

If maximal sharing does have such a negative effect on list editing operations, it is imperative to find a fast implementation of list construction. And, it is likely that any optimization in this matter will have a significant effect. We will try the idea of a build function in a pilot implementation.

2.1.2 Pilot implementation

Firstly, the support library was extended with the build function. This build function contains all arguments of the cons expression from left to right. Notice that the number of arguments of this build function is not constant. We have used a C

[1]In the context of non-strict lazy functional languages, a similar idea is presented in [12].

[2]The intermediate slices of the cons expression are most likely only needed for the building of this reduct, but they will occupy space on the heap.

```
ATerm Set(ATerm arg0) {
  if(check_sym(arg0, listsym)) {
    ATerm tmp0 = arg_0(arg0);
    ATerm tmp1[2];
    tmp1[0] = tmp0;
    tmp1[1] = tmp1;

    while(not_empty_list(tmp0)) {
      ATerm tmp3 = list_head(tmp0);
      ATerm tmp2[2];
      tmp0 = list_tail(tmp0);
      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(build(BEGIN,
                                CONCAT,
                                SLICE, tmp1[0],tmp1[1],
                                CONCAT, tmp3,
                                CONCAT,
                                SLICE, tmp2[0], tmp2[1],
                                tmp0,
                                END)));
        }

        tmp2[1] = list_tail(tmp2[1]);
        tmp0 = tmp2[1];
      }

      tmp1[1] = list_tail(tmp1[1]);
      tmp0 = tmp1[1];
    }
  }

  return make_nf(setsym, arg0);
}
```

Figure 2.1: The generated C code from the Set specification, with the *build* function.

function with a variable argument list to cover this[3]. The proper operation on each of the arguments is expressed by some extra arguments (tags):

- BEGIN indicates the beginning of a list.
- CONCAT is a separator. This separator is actually not needed, but it is there for the sake of readability.
- SLICE means that all the elements between the next two argument nodes are inserted.
- MAKE_LIST inserts the next ATerm as an element or, if it is a list, it inserts all elements of this list.
- END indicates the end of a list. This is needed, for there is no general way of knowing how many arguments a function has in C.

These integer labels can be distinguished from all possible pointer arguments because they are all odd valued. Odd values are never pointers in C in most modern implementations. The build function collects all elements of the list into a buffer and creates an ATerm list from this buffer. Because the cons expression reuses the tail, care is taken such that the build function does the same thing; a list is only inserted into the buffer after it is clear that more elements need to be appended behind it. If it was the last argument, then the result is created by inserting the elements in the buffer in front of this list. The code of the build function can be found in Appendix A.

To test the build function the generated C code of three minor specifications was changed by hand: Set (Figure 1.1), Symbol-Table (Figure 2.2) and Bubble (Figure 2.3). The needed adaptations were rather simple and mechanical: first the cons expressions were wrapped by the build function. Then every cons function and its brackets were replaced by a CONCAT tag, every slice function by a SLICE tag, etc. Notice that the order in which the arguments of a cons expression appear does not change due to these editing operations.

2.1.3 Measurements

To measure the behavior of the build function we used profiling information[4]. The time spent in a rewrite rule depending on the number of redices in a test term was measured. The reason for using ASF+SDF specifications to measure is that we need real motivation for introducing the build function into the code generation process. A theoretical chance of a significant effect only is not reason enough for adapting the compiler.

Set

The Set equation removes multiple occurrences of an element from a list. For terms we used lists with a fixed prefix of different symbols followed by a linearly increasing number of equal elements. The results are in Figure 2.4. This figure shows a significant speedup.

---

[3]There is an upper-bound on the number of arguments in a variable argument list in C. Introducing the build function induces an upper-bound on the size of list patterns. This upper-bound is sufficiently large to expect that nobody will ever reach it.

[4]The C compiler and a program called *gprof* [11] provide functionality for profiling C programs.

imports Layout
exports
  sorts Pair Label Symbol-Table
  lexical syntax
    [a-z]+ → Label
  context-free syntax
    “(” Label “,” Label∗ “)” → Pair
    “[” Pair∗ “]” → Symbol-Table

    Symbol-Table “++” Symbol-Table → Symbol-Table {**right**}
hiddens
  variables
    L [*0-9*]∗ → Label
    L “∗”[*0-9*]∗ → Label∗
    P [*0-9*]∗ → Pair
    P “∗”[*0-9*]∗ → Pair∗
    S [*0-9′*]∗ → Symbol-Table
equations

[0] $[\,] +\!\!+ S = S$

[1] $[(L,\, L_0^*)\ P_0^*] +\!\!+ [P_1^*\ (L,\, L_1^*)\ P_2^*] = [P_0^*] +\!\!+ [P_1^*\ (L,\, L_0^*\ L_1^*)\ P_2^*]$

[default] $[(L,\, L^*)\ P_0^*] +\!\!+ [P_1^*] = [P_0^*] +\!\!+ [(L,\, L^*)\ P_1^*]$

Figure 2.2: The Symbol-Table specification.

imports Integers
exports
  sorts List
  context-free syntax
    “[” Int∗ “]” → List
hiddens
  variables
    Int [*0-9*]∗ → Int
    Int “∗”[*0-9*]∗ → Int∗
equations

[0] $$\frac{Int_0 > Int_1 \ = \ \mathrm{true}}{[Int_0^*\ Int_0\ Int_1\ Int_1^*] \ = \ [Int_0^*\ Int_1\ Int_0\ Int_1^*]}$$

Figure 2.3: The Bubble specification.

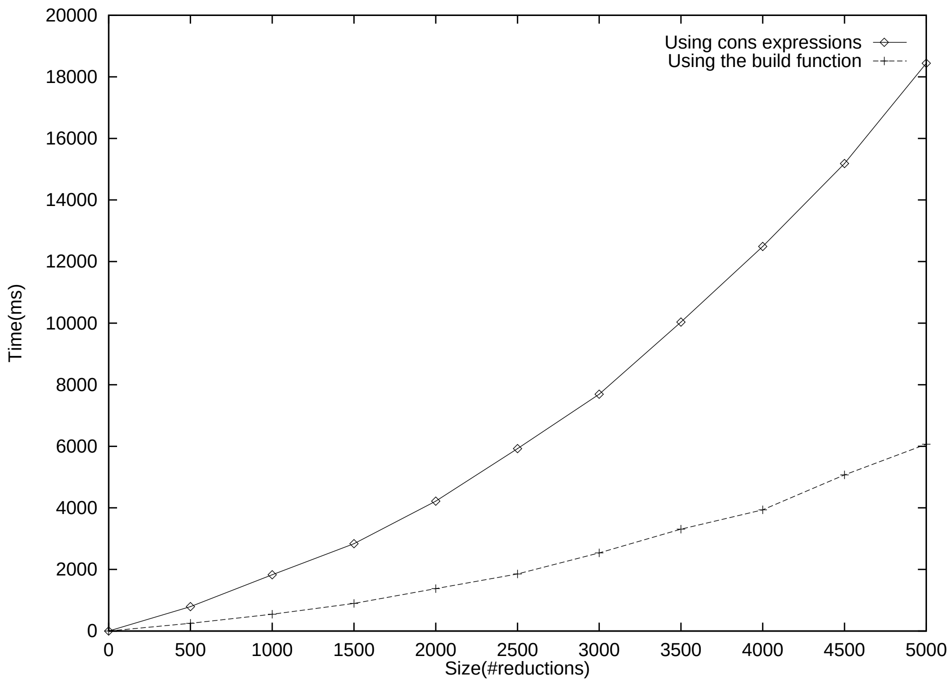


Figure 2.4: The time spent in the Set equation against the number of equal elements.

Symbol-Table

The Symbol-Table specification merges two lists of tuples. Symbol-Table is slightly different from the other examples because the merge function has two list arguments. Again, we use lists of linearly increasing size to measure the gain. The results in Figure 2.5 show a significant speedup. We also notice local maxima in both graphs. The version using the build function reaches the local maximum at significantly larger input size.

To explain these local maxima, we need some insight in the behavior of the garbage collector of the ATerm library. We profiled:

- The garbage collector by counting the number of garbage collections.
- The number of block allocations. A new block is allocated when the heuristics of the garbage collector decide that space becomes too limited.
- The number of hash-table resizes. The hash-table is resized when it becomes to small to hold all the currently used ATerms.

The results of this profiling in Figure 2.6 show a drop in the number of garbage collections exactly when an extra block is allocated.

Bubble

The Bubble specification implements the Bubblesort algorithm on lists of naturals. The terms we have used here are growing lists of naturals in completely reversed order. Figure 2.7 shows the results. We notice a slight gain in performance.

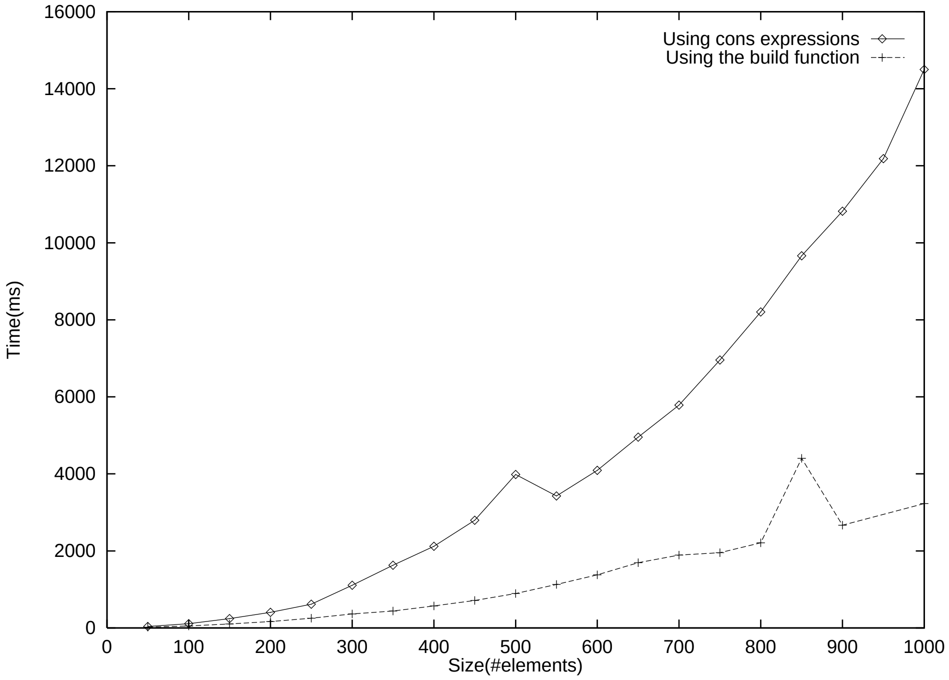


Figure 2.5: The time spent in the Symbol-Table equation against the number of elements.

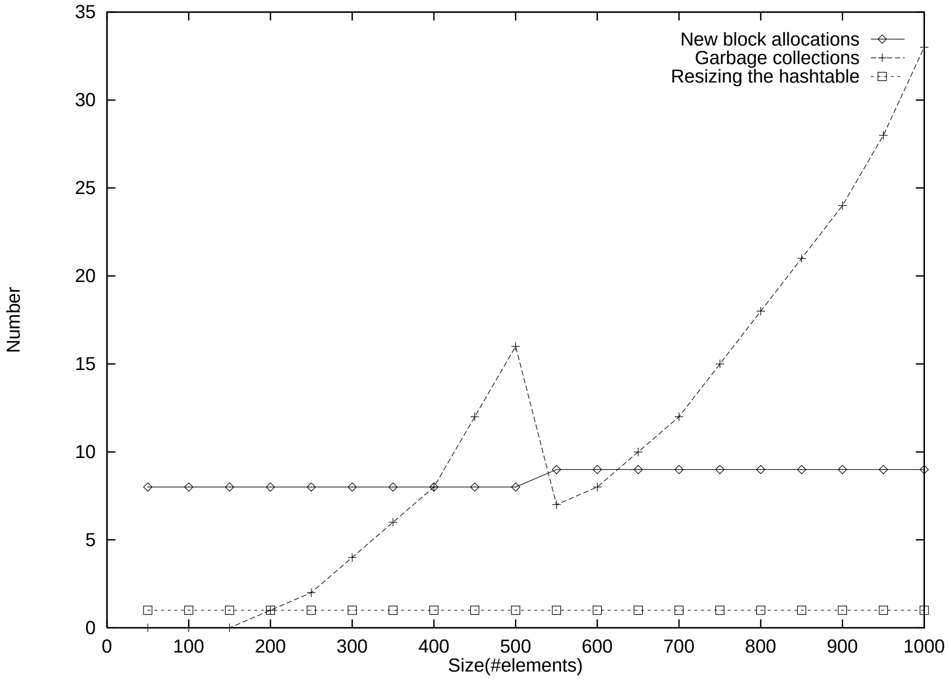


Figure 2.6: Profile of the garbage collector in the Symbol-Table equation in Figure 2.5 (using cons expressions).

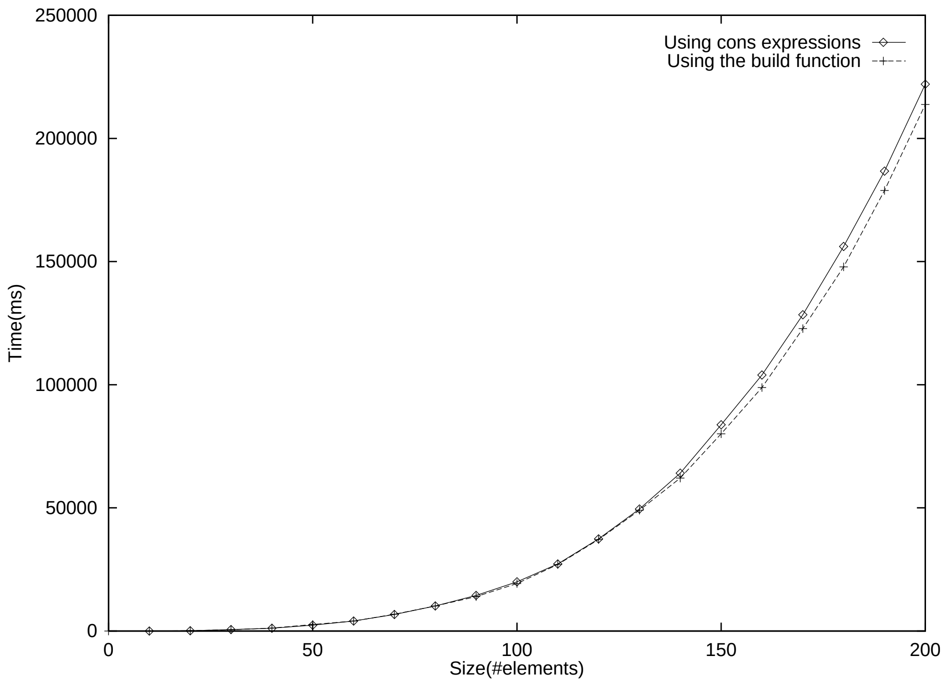


Figure 2.7: The time spent in the Bubble equation against the number of elements.

2.1.4 Analysis

The results of the measurements show a significant gain for Set and Symbol-Table. But the gain for Bubble is less noticeable. To explain these results we need a small model of the situation. We have measured the *total* running time of a rule. So we will seek an expression for the time spent reducing an entire recursive rule. Our model will distinguish between the time spent to build a reduct, and the rest of the work, which includes evaluating all conditions:

> For any recursive rewrite rule: Let f_i be the time spent to find the ith redex. Let l_i be the time spent to build the ith list reduct. Let n_i be the length of the ith list. Let t_i be the length of the reused tail of the ith list. Let $s_i = n_i - t_i$. Let R be the number of recursive calls, or redices. Any execution of a recursive rewrite rule consists of finding redices, which includes calculating conditions, and building reducts. Thus, we model the execution time of a recursive rewrite rule by:
>
> $$T = \sum_{i=1}^{R}(f_i + l_i) \tag{2.1}$$
>
> Notice that l_i will always be in $O(n_i)$, but f_i can be much harder. The time that is needed for building a list linearly depends on the number of elements that have to be inserted into (possibly intermediate) lists. For the building lists using cons expressions we write:
>
> $$l_i = 2s_i \tag{2.2}$$
>
> Each element in the cons expression will be inserted twice: First into a slice or singleton and then into the resulting list. The tail elements

are reused, so they have no effect on the equation. All list builtins are linear in the size of the input.

The build function does not insert the elements in slices and singletons, therefor we write for the build function:

$$l_i = s_i \tag{2.3}$$

From this model we learn that the speedup of the build function not only depends on the specific rewrite rule and the size of the input, but also on the specific location of the redices in the list. A large reusable tail will result in a small gain. The model also shows that building the reduct (l_i) can be insignificant compared to the rest of the work (f_i). This effect is amplified by recursive behavior.

Set

For the Set equation and the terms we used for testing it, finding a redex is in linear time. Because we used a prefix of unequal elements in each list s_i is rather large. So we have a considerable speedup.

Symbol-Table

The Symbol-Table equation finds its redices in linear time. So the gain of the build function is noticeable. The local maxima seen in Figure 2.5 do not fit into our model. These maxima occur at very specific input sizes.

From Figure 2.6 we learn that the maximum is caused by a suddenly decreasing number of garbage collections. The need for garbage collections is gone, because another block of memory is allocated according to internal heuristics of the ATerm garbage collector. The tradeoff between time efficiency and memory efficiency in the ATerm library is made very visible in this example.

From Figure 2.6 we also conclude that the optimized version uses less memory, because it allocates an extra block at 900 elements, while the unoptimized version already needs it at 550 elements.

Bubble

The Bubble equation has more trouble finding a redex. Firstly, an integer comparison is not done in constant time. And we have a worst case of n_i^2 integer comparisons to do for finding each redex. We notice that finding a redex is considerably harder than building the reduct. Therefor, the difference between the unoptimized and the optimized version is not very large.

2.1.5 General implementation

The above analysis motivates the implementation of this optimization into the code generation process. The build function seems to have a positive effect on time and memory use. The support library has already been extended with the build function. For a general implementation all we need to do is extend the compiler with a transformation of cons expressions into build function calls.

Remember how lists in μASF are already represented by cons expressions. But the slice and make_list functions are introduced at the final stage in the compilation path. This is the reason for doing this transformation on the generated C code. The compiler does some other transformations on the C code[5]. Care is taken such that the build function does not interfere with these; the build is introduced after all other C level transformations.

[5] For example, constant elimination [9].

cons(Expression1, Expression2)	→	CONCAT, Expression1, Expression2
slice(Expression1, Expression2)	→	SLICE, Expression1, Expression2
make_list(Expression)	→	MAKE_LIST, Expression

Figure 2.8: TRS for translating cons expressions to arguments lists of the build function.

The transformation traverses the C grammar and finds each cons expression. It wraps the expression by the build function. Then it translates the cons expression using the TRS in Figure 2.8 to an argument list for the build function.

2.1.6 Testing

Now that we have extended the compiler and the support library with the build function, it is time to test this optimization on a less trivial application. The specification we tested was the generic pretty-printer of the new Meta-Environment [6]. This specification makes frequent use of lists.

After profiling PP we found that the gain of using the build function is minimal. In some cases, we even noticed a minimal drop in performance. When measuring the number of list insert operations as an indication of the amount of work, we found that the optimized version saves thousands of insert operations compared to the normal version. But this is insignificant compared to the millions of insertions in the entire specification.

The small overhead of the build function, due to the tags in the argument list and the need for a larger temporary buffer to store all elements of the result, explains the performance drops. The generic pretty printer has more trouble with matching than with construction of list. We conclude that this specification does not benefit of the use of the build function.

2.1.7 Conclusion

The build function has a positive effect on the time needed for building lists. Also, it saves a considerable amount of memory. The gain of the build function is dependent on the specification at hand. Specifications that have few elements in slices do not benefit specifically from the build function. And specifications that have a hard time finding a redex will also notice little advantage from the build function.

The implementation of this optimization consists of an extension of the support library and a modular extension of the compiler. Both do not interfere with any existing code.

2.2 Destructive lists

2.2.1 Motivation

The ATerm library does not do any destructive updates on terms. The consequence for rewriting is that redices (terms) that are not in memory are build from scratch. When rewriting a recursive function, a lot of intermediate results are calculated before a normal form is reached. When these intermediate results are lists they usually do not differ a lot between recursive calls. It seems like a waste of resources to build each intermediate result from scratch. Especially when large lists are involved, the use of a destructive data-structure might improve the runtime performance of many recursive rewrite rules. If we use the pieces of an existing list to build a redex, the complexity of building list redices might even be brought down from $O(\#\text{elements})$ to $O(\#\text{variables})$.

As said in the introduction, the ATerm library cannot be subject to any changes. But maybe we can do destructive updates within the controlled environment of a single rewrite rule, using an new extension of the support library. We could transform ATerm lists to a destructive data-structure at the beginning of a recursive rewrite rule and convert it back to an ATerm when a local normal form is found. Because of the two conversion steps, this idea can only be beneficial for recursive rewrite rules. The destructive representation of a list can be kept between recursive calls.

2.2.2 Pilot implementation

Apparently, this optimization is far more complex than the previous one. For example, the generated code must work with a completely different data-structure. On the other hand, it is imperative to find an elegant solution, because we do not want to change the entire compiler to perform this pretest. Read this section as a feasibility study for the application of a destructive list data-structure in the current ASF+SDF compiler. The pilot design falls into three major parts: the design of the data-structure, the memory allocation and the list construction algorithms.

Data-structure

To test the idea of destructive lists, we need a destructive data-structure first. Here are the requirements on such a data-structure:

- The operations on lists, which are mainly concatenation and slicing, must be faster than linear in the size of their arguments. We are not interested in a constant factor and the current implementation already performs in linear time.
- Conversion from ATerm list to destructive list and vice versa should be as cheap as possible.
- Copying and creating a destructive lists should be relatively fast. The needed terms are not always available in memory.
- The garbage collection of destructive lists must be clean and easy. We do not want to introduce any memory leaks.
- The representation of destructive lists must be memory efficient.
- The destructive lists must be compatible with the current rewriting strategy. We can change some details of the implementation, but the general strategy must remain the same. This is for the sake of simplicity.

Different data-structures can be considered. The first one that comes to mind is a C array that represents the nodes of a list. C arrays allow for fast creation, destruction, copying and traversal. Also, this can be a very memory efficient representation. But concatenation of slices using C arrays is in linear time. This contradicts one of the above requirements. Any other acceptable data-structure in C would be some kind of linked list. An advantage of linked lists is that most operations can be done in place. Next, we need to decide on the information stored in a node:

- A reference to the actual element of the list. This is an ATerm pointer.
- A reference to the next node. This is for left to right traversal.

- A reference to the previous node. This is to facilitate slicing. The current list matching algorithm finds matches slices using a left inclusive and a right *exclusive* bounds. The right *inclusive* bounds can be found using the previous pointer in constant time.

A linked list with the above specifications is easily implemented. But the compatibility with the existing generated C code is not tackled yet. When we compare the normal ATerm lists with the above specifications, they almost comply. Normal ATerm lists have a reference to the actual element, and a reference to the next node. Furthermore, they have a header containing some additional information and an extra pointer that is used to implement maximal sharing of ATerms.

The idea is to use normal ATerm lists as a destructive list data-structure. The extra pointer for maximal sharing can be used as a previous pointer because we do not need maximal sharing. We can fill the header of this ATerm list with enough information to trick the existing generated code into believing that it is a normal ATerm list. This takes care of the compatibility problem; we hardly have to change anything in the generated code. One disadvantage of using the ATerm list data-structure is that it uses more memory than required for this application[6]. But in this context, simplicity of implementation is slightly more important than saving memory cells.

Finally, we need to be able to distinguish among normal ATerm lists and destructive lists. The header seems to be the ideal tool for this purpose. Normal ATerm lists keep a record of their length in the header. We cannot do this for destructive lists, because that would make every operation of at least linear complexity. By setting the length to zero of every destructive list we effectively distinguish them from normal ATerm lists.

Memory allocation and freeing

Using the ATerm list nodes does not mean we can use the ATerm garbage collector. We do not want maximal sharing on these nodes, so we will have allocate and free them ourselves. Firstly, what are the requirements on memory allocation and freeing of destructive lists? They need to be quick and simple. An extra garbage collection scheme next to the ATerm garbage collector would not only be too complicated in the context of a pilot implementation, it would probably also create a drop in performance.

If we do not introduce destructive lists globally, then let us assume that recursive rules will translate normal ATerm lists to destructive lists and the translation back to ATerm lists is done when a normal form is encountered. If we do not do the transformation back to ATerm lists in the recursive rule, a separate mechanism must be designed to do this. This contradicts the requirement of a simple garbage collection scheme. So, the usage of our destructive data-structure is limited to a single function. And memory allocation and garbage collection will be done within this rewrite rule.

These choices imply that it will not be beneficial for just any recursive rule to use destructive lists. Only in case of tail recursion, which is replaced by a goto statement by the compiler, the destructive lists can be maintained between recursive calls. Although we now have a limited set of rules that comply, we are able to test the positive effect of destructive lists.

Firstly, there is a choice between allocating each node separately on the heap, or allocating a contiguous block of memory to hold all nodes within the function. The first alternative helps to ensure that no more memory is allocated than is needed. But this solution introduces a problem with garbage collecting; during the

[6]The header information seems to be unnecessary.

reductions of a recursive rule a lot of nodes can become oblivious. These nodes are no longer referenced to by other nodes. Thus, extra bookkeeping of these nodes is required to prevent memory leaks.

The easier solution is to allocate a larger block of memory for each rule. All builtin list construction primitives can make use of this buffer to create new nodes. The garbage collecting at the end of a function is now limited to freeing this single block of memory. A different approach could be to share a global buffer among all rules that needs to be allocated only once during a calculation. But due to conditional rules, multiple recursive C functions can be on the stack at the same time. This means that no function can clear the entire buffer when it returns. This calls for more sophisticated garbage collection, which contradicts the simplicity requirement.

In short, the following scheme was chosen: New destructive nodes are allocated at the end of a *local* heap. Unused nodes are not reclaimed during the execution of a function; the heap is freed when a recursive function returns. A function always returns normal ATerm lists.

List construction

The above decisions on the data-structure and memory allocation provide the framework for the algorithms of list construction. The interface of the *build function* of the previous optimization is the starting point of this design. Since the build function contains all the arguments of the resulting list, we have all possible information at hand. As opposed to cons expressions, where the information is distributed among the different list construction builtins. If all nodes in the arguments can be reused, concatenation of slices can be done in constant time[7] .

The arguments of the new build function can be destructive lists as well as normal ATerm lists. The build function returns destructive lists, so it should convert any normal arguments to destructive lists. As a side-effect, this postpones the conversion from ATerm lists to destructive lists to the time that it is actually needed; when a reduct is formed. Ergo, we let the build function take care of converting from ATerms to destructive lists.

The build function will concatenate singletons and slices in constant time. But if only the beginning of a (destructive) list is given, it needs to be traversed to the end. Traversal to the end of a list is postponed until another element, slice or list needs to be concatenated. This is analogous to the way tails are reused in the non-destructive build function.

Then we have the problem of non-linearity. Consider a pattern that has two instances of a list variable in the right-hand side of a list pattern. The nodes of this slice need to be copied once to construct a complete new destructive list. This problem was not tackled in the pilot implementation, because the test specifications Set, Symbol-Table and Bubble are all linear. But in a possible general implementation, a solution for this problem must be found.

2.2.3 Measurements

We have seen that the normal build function has positive effects on list construction. So we are interested in the effect of destructive lists compared to the performance of the build function. The same specifications were adapted to use destructive lists: Set, Symbol-Table and Bubble. And the exact same terms were used to measure the performance.

The adaptation of the generated code of these specifications to use destructive lists was easily done due to the simplicity of the design. A local heap variable was

[7] *Constant time* means not depending on the number of elements in the reduct.

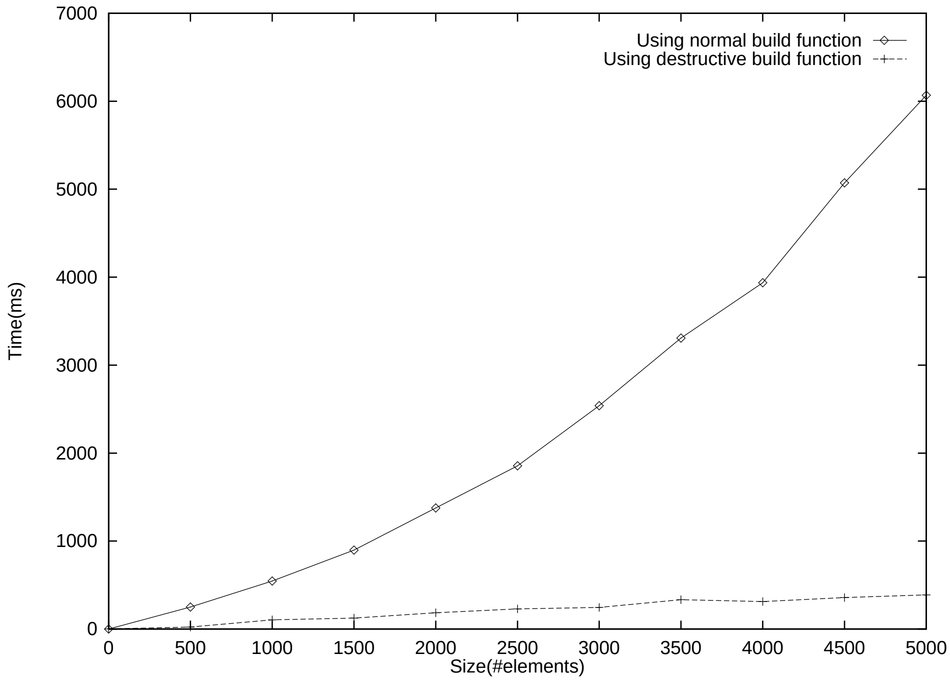


Figure 2.9: The time spent in the Set equation against the number of elements.

added to hold a pointer to a heap containing destructive list nodes. The build function was renamed to the name of the destructive build function. And the arguments of normal forms were wrapped by a function that converts destructive lists back to ATerm lists.

Set

The results of measuring the Set specification are in Figure 2.9. The graph shows a linear increase in time for the destructive version. And a significant speedup compared to the normal build function.

Symbol-Table

The graph of the Symbol-Table specification is in Figure 2.10. This figure shows a less impressive difference between each implementation. We notice that the local maximum returned to approximately the same size as the implementation using cons expressions.

Bubble

Even the Bubble specification seems to benefit significantly from destructive lists. In Figure 2.11 we see that the graphs diverge for large lists.

2.2.4 Analysis

We extend our model of the previous analysis with an equation that approximates the behavior of the destructive build function:

$$l_i = 1 \tag{2.4}$$

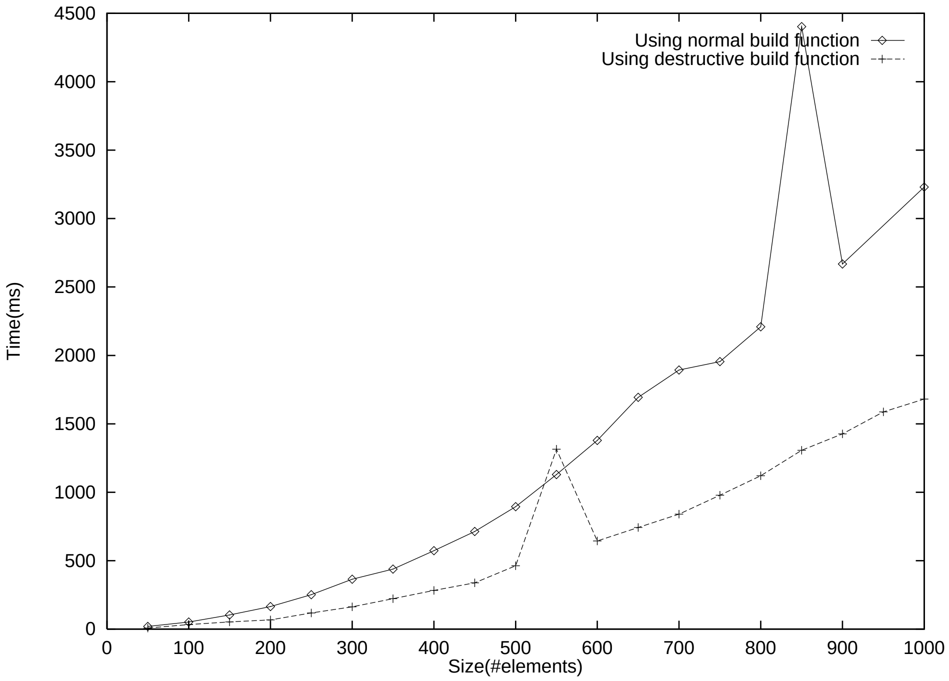


Figure 2.10: The time spent in the Symbol-Table equation against the number of elements.

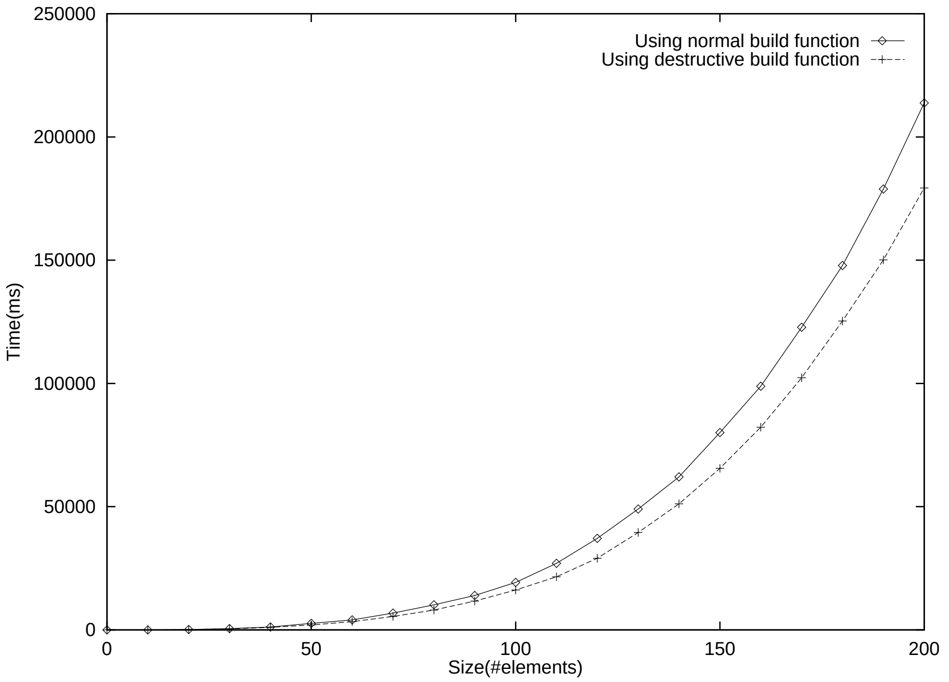


Figure 2.11: The time spent in the Bubble equation against the number of elements.

```
module Foo
signature
  list(_);
  foo(_);
  conc(_,_);

rules
foo(list(conc(*E1, conc(E, conc(E, *E2))))) =
    foo(list(conc(*E2, *E2)))
```

Figure 2.12: An example of a non right-linear pattern in μASF.

Here the constant *one* models the constant amount of list parts that have to be concatenated. In our test specifications there is never the need for traversal when building the list. So, in the context of our test specification, this model suffices.

The model predicts enormous gain when f_i is in linear time. As seen in the Set example, where execution time drops to almost independent of the input size. But when finding the redices is harder, such as in the Bubble specification, we notice a smaller gain factor.

2.2.5 General implementation

The results from the pilot implementation seem promising enough to try and find a more general implementation of destructive lists in the compiler. But already we have restricted ourselves by ensuring that the existence of a destructive list is bounded by a single rule (C function). This restriction not only prohibits a more general implementation of destructive lists, it also has a negative on their possible gain. The main issue is that the restriction causes the constant conversion from and to ATerm lists. This conversion is in many ways an obstacle as we have seen in the previous section.

In any case, the goal of this so called general implementation is to show that destructive lists is, or is not a possible solution to the performance issues of lists in the ASF+SDF compiler. If we have a slightly more general implementation in the compiler of destructive lists, we can compile a non-trivial specification and draw some conclusions. The three test cases missed some features that can complicate the implementation of destructive lists severely:

- Non right-linear patterns. List variables can occur more than once in the right-hand side of a list pattern.

- Passing destructive lists to conditions.

Non-linear patterns

Figure 2.12 shows a non-linear set pattern in μASF. If we build the right-hand side from the matched variables we obviously need to copy the values of the double variables. This problem can be solved either run-time or compile-time. A run-time solution would be to adapt the destructive build function to keep track of the used variables. This solution inherently comes with a significant overhead. So we choose for a compile-time solution.

A straightforward solution is to detect multiple occurrences of a variable and to change the *tags* in front of these variables to indicate that copying is needed. Note that this solution does imply a more complicated build function, since we need to distinguish among more different tags than before. Figure 2.13 we show the result

```
arg0 = list(dbuild(BEGIN, CONCAT, MAKE_LIST, tmp[0], tmp[0], END));
 goto label_foo;
                                ⇓

arg0 = list(dbuild(BEGIN, CONCAT, COPY_MAKE_LIST, tmp[0], tmp[0],
END));
 goto label_foo;
```

Figure 2.13: The right-hand-side of example of Figure 2.12 in C code. Multiple occurring variables are resolved by introducing COPY tags. Notice that tail recursion is resolved by a goto statement.

of transforming the argument list of the build function to take care of multiple occurrences of variables. This transformation is added to the compiler[8].

Passing destructive lists to conditions

The evaluation of conditions introduces some problems. The arguments of some of the conditions in a recursive rewrite rule might be destructive lists. If we convert these lists back normal shared lists before we pass them on to the conditions, chances are that the performance drops significantly; we will introduce the effect of translating lists back and forth in each recursive call.

Take the compiled specification in Figure 2.14 for example. A slice (list variable) is passed to an assignment condition. After that, the slice is used in the build function to construct the right-hand-side. If the condition translates the list to a normal ATerm list, the build function will translate it to a destructive list. In the next recursive step, the condition will translate the list again, etc.

On the other hand, if we pass a *destructive* list to a condition, its consistency is in danger. The assignment condition in Figure 2.14 for example, might very well contain a rewrite function that uses a slice of the list to check something. Without a detailed data-flow analysis of the entire ASF+SDF specification, we can never be sure if we can use the list after we have turned it over to a condition. Such a data-flow analysis is beyond the scope of this masters thesis, but it may be an idea for future work.

We need to make sure that conditions never change destructive lists. To accomplish this a trick is used: remember how we give each recursive rule its own heap of destructive nodes. We can check if a destructive list has its nodes on the local heap by comparing memory addresses[9]. If its nodes are on a different heap, we need to copy before we can change. If a condition only *investigates* a destructive list, then there is no unnecessary converting. Still, if we use this solution, there is a chance of converting list from and to destructive lists in each recursive call. Which is the exact opposite of what we are trying to accomplish. Also notice that the strategy of always returning normal ATerm lists prohibits the reuse of the results of conditions.

There is another consequence of passing destructive lists to conditions. A list can be become part of a larger term, without being changed. This way a destructive list can appear inside of a normal form, which evidently goes wrong when the calling function returns and frees its local heap. One way to ensure that a destructive list is never part of a normal form is to adapt the builtins that create normal forms[10] to traverse all terms and convert destructive lists. We tried this approach and we noticed a large drop in performance.

[8]The rewrite function to do this transformation is very similar to the Set equation.
[9]Each local heap is a consecutive block of memory.
[10]These builtins are part of the support library.

```
ATerm Bar(ATerm arg0) {

 label_Bar:

  if(check_sym(arg0, listsym)) {
    ATerm tmp0 = arg_0(arg0);
    ATerm tmp1[2];
    tmp1[0] = tmp0;
    tmp1[1] = tmp1;

    while(not_empty_list(tmp0)) {
      ATerm tmp3 = list_head(tmp0);
      ATerm tmp2[2];
      tmp0 = list_tail(tmp0);
      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)) {
          ATerm tmp5 = mycondition(list(tmp0));

          if(check_sym(tmp5, listsym)) {
            arg0 = list(dbuild(BEGIN, CONCAT, SLICE, tmp1[0],
                               tmp1[1], CONCAT, tmp3, CONCAT,
                               SLICE, tmp2[0], tmp2[1],
                               MAKE_LIST, tmp0, END));
            goto label_Bar;
          }

        tmp2[1] = list_tail(tmp2[1]);
        tmp0 = tmp2[1];
      }

      tmp1[1] = list_tail(tmp1[1]);
      tmp0 = tmp1[1];
    }
  }

  return make_nf(Barsym, arg0);
}
```

Figure 2.14: A compiled specification with list variables in conditions.

If checking all normal forms for destructive lists is inefficient, we need to restrict the search for destructive lists. Unfortunately, there is no easy way of doing this. The conditions in ASF+SDF rules are the cause of this. They make it possible to "bury" a destructive list deep inside of a normal form. Possibly with a specification wide data-flow analysis, we could find out which terms need converting.

2.2.6 Conclusion

Using the pilot implementation of destructive lists we have shown that destructive lists can result in a serious gain in performance. Problems that were tackled are the choice of data-structure and the list building algorithm.

But our design severely limited a possible general implementation. The conversion from and to destructive lists is a major bottleneck, especially when lists are passed to conditions. A more general approach would certainly benefit the possible gain of destructive lists. If destructive lists are wanted in the generated code, we recommend a global introduction of non shared list nodes. This would relief us from the burden of conversion. A global garbage collecting scheme for destructive nodes would have to be designed coexisting with the garbage collection of fully shared terms.

Consistency problems when passing destructive lists to conditions were solved here by a memory comparison depending on the local heap allocation. In a possible general application of destructive lists, applying the same trick might not be possible. Possible solutions could use a compile-time data-flow analysis of the entire specification, run-time reference counting or a combination of these techniques.

We were not able to test a non-trivial specification due to the above problems. There is an obvious opportunity for future work.

Chapter 3

Optimizing list matching

As said in the previous chapter, the ATerm library performs well at list traversal. The general approach to extensively search a list for a pattern works well, but in the case of direct recursive rewrite rules there is more information on matches. For example, in the Set specification in Figure 1.1 elements that are matched in the first list variable certainly do not occur in the rest of the list. We know that for sure, because we just searched the list for them and did not find anything. So, the next recursive call does not have to check them again.

Recursive list patterns are prone to be inefficient. In the current implementation, each recursive call does an extensive search on a possible match. Recursive patterns are a frequently used programming style in ASF+SDF. The combination of recursion and list matching results in very concise and clear specifications. For example, this programming style is used extensively in the generic pretty printer of the Meta-Environment [6]. Almost the entire functionality of this specification is encoded using list matching rules and most of them are recursive. The result is a highly readable specification but after compilation we are left with a rather inefficient executable. If the generic pretty printer was written using explicit list traversal[1] we would have obtained a rather obscure specification, but with a better performance. It would be preferable to have both readable and efficient specifications, so we will try to find a solution for this problem.

This chapter describes a search for a general approach to exploit extra information due to *recursive* rules. If a general solution cannot be found, there is still the possibility of handling specific recursive patterns. In both cases, there are two problems to be tackled. Firstly, how do we detect such patterns? Secondly, what optimizing strategy do we use? The goal is to create a compiler that can optimize specifications without help of the specification writer[2].

One of the reduction strategies mentioned in [1] is the *multiple reduction strategy* for regular (orthogonal) rewrite systems. The strategy first identifies all possible matches in a term and then simultaneously contracts them. We would like something similar for list patterns, although they are by definition not regular. First identifying matches would relief us from searching from the beginning in each recursive call.

One of the issues here is not to change the normalization properties of a rule. The specification writer expects a lexicographical ordering on the matches. Some patterns might depend on that, others might be executed using a different and possibly faster strategy to get the same resulting normal form. List patterns that are guaranteed to reach a unique normal form using any reduction strategy are called

[1] I.o.w. without list matching

[2] Some formal languages, like STRATEGO [22], take another approach and let the user guide the rewriting process to obtain an efficient implementation.

confluent. The Set equation in Figure 1.1 is an obvious example. The following example equation is a trivial non confluent pattern:

$$[X^*, a, a, Y^*] = [X^*, b, a, Y^*] \tag{3.1}$$

In [20] several properties of ASF+SDF specifications are calculated automatically using ASF+SDF. Unfortunately, none of these methods apply to our setting. In general, it is not always easy to prove confluence of non-regular rewrite systems[3].

Besides the discussion on confluence, it is interesting to know if there are reduction strategies that perform faster than the straightforward lexicographical ordering. Firstly, we will present some ideas for reduction strategies on recursive rules. Secondly, we will address the problem of detection.

3.1 Continuous list patterns

For instance, consider the class of left-linear list patterns that start with a list variable and end with a list variable. The second constraint is that these two list variables do not occur in the conditions. The third constraint is that these list variables are part of the reduct too. This is what such patterns look like:

$$f(X^*, <\text{pattern}>, Y^*) = f(X^*, <\text{pattern}>', Y^*) \tag{3.2}$$

We name this the class of *continuous* list patterns. The key functionality of the rule is coded in <pattern>. The list variables X^* and Y^* only serve to indicate that this <pattern> may occur anywhere in the list.

We observe that the right-hand sides of such patterns contain the matched variable X^*. The recursive call to f will try to find <pattern> in X^*, which is possibly superfluous; the current instance of f has already decided that X^* can not be matched with <pattern>. Especially when a lot of conditions have to be evaluated for each match, not searching this prefix again will save time.

The right-hand side of the rule might introduce a redex that contains matches on a smaller X^*. Therefor, we need another check of the list before we can be sure it is in normal form. An idea for an optimization is to change all continuous patterns to this:

$$f(<\text{pattern}>', Y^*) = f(Z^*) \implies f(X^*, <\text{pattern}>, Y^*) = f(X^*, Z^*) \tag{3.3}$$

This rule recursively calls f on the list without the X^* prefix and then recursively calls f again on the entire list. The new rule ensures that the X^* is not searched immediately to find the next redex. But eventually, the prefix will be searched by the recursive call. Having added a new recursive call to the rule, we actually do more work than we did before. So this idea fails due to the possibility of introducing new redices matching a smaller X^* that need to be searched for. We try to fix this problem in the next section.

3.2 Guarded list variables

Consider the following subclass of the continuous patterns:

$$f(X^*, V, <\text{pattern}>, Y^*) = f(X^*, V, <\text{pattern}>', Y^*) \tag{3.4}$$

Where V is called a *guard* for X^* and is allowed to be in the conditions. X^* is called a *guarded prefix*. The lexicographical ordering of matches guarantees that

[3]Confluence is an undecidable property of general rewrite systems [13].

X^* does not contain <pattern>. But in the general case, X^* should be searched again in case a new redex is introduced that matches with <pattern> on a part of the old X^*. In this case the guard V next to X^* prevents this and searching the list again becomes superfluous.

3.2.1 Possible implementation techniques

After the first match, X^* can be left out of the computation completely. But after the computation of the recursive rule has finished, X^* must be concatenated in front of the result. We can introduce this optimization either at the μASF level, or at the C code level. The problem of *detecting* such a pattern is tackled in Section 3.4.

At the μASF level, we would require introducing an extra *concatenation* function for concatenating the X with the result of f. But the resulting term of the concatenation would be a call to f again, and the list would be checked for redices again. So, we also need a new primitive in the support library to indicate that a term is guaranteed in normal form. Adapting the compiler to add a concatenation function, and to accommodate a normal form attribute is not a trivial task.

At the C code level, the concatenation can be done without introducing an extra function. And the C code can be adapted to return a normal form after concatenating. On the other hand, the transformation of C code in the compiler is much more complicated due to the more complicated syntax of C.

3.3 Special list patterns

For some often used patterns there might be a specific smart implementation. The designer of the compiler uses his intelligence to find a smart and equivalent implementation of a certain pattern and the compiler is only sophisticated enough to detect the pattern and replace the rule.

So we have two issues here: firstly we need a collection of frequently occurring special patterns with smart implementations. Secondly, we need a uniform manner to detect them in the compiler. There is currently only one frequently used list pattern that has a known smart implementation; the famous Set pattern.

3.3.1 The Set pattern

This is the Set pattern in μASF, with cons notation replaced by associative notation for the sake of readability:

$$V = W \Longrightarrow f(X^*, V, Y^*, W, Z^*) = f(X^*, V, Y^*, Z^*) \tag{3.5}$$

In the conventional generated code, this rule will result in a lot of redundant searching. Consider rewriting this rule to:

$$\begin{aligned} V =& W \;\; \& \\ f(V, Z^*) =& f(V, Z_2^*) \;\; \& \\ f(Y^*, Z_2^*) =& f(Z_3^*) \Longrightarrow f(X^*, V, Y^*, W, Z^*) = f(X^*, V, Z_3^*) \end{aligned} \tag{3.6}$$

Now, it seems to be guaranteed that each element is compared with all others only once. Which is exactly what the author of such a pattern implies. It is imperative that the extra conditions are added below the original conditions to ensure that the recursive calls will only be made in case a redex is found. This transformation is an extended case of the guarded list patterns. We notice the unnecessary searching due to the recursive call in the right-hand side: the conditions remove all doubly

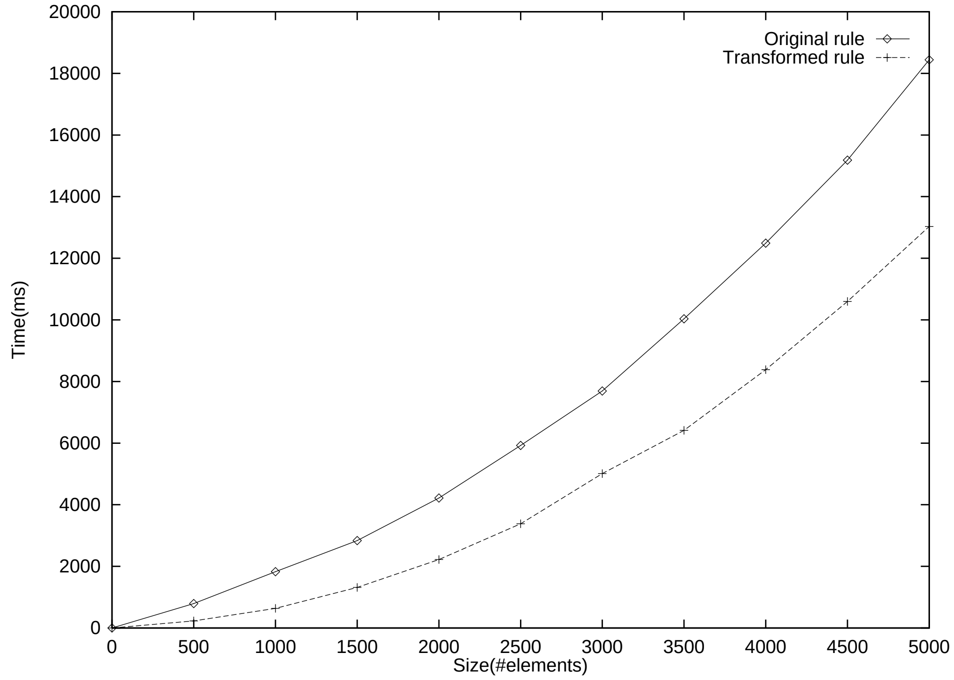


Figure 3.1: The time spent in the Set equation after transforming it.

occurring elements from the list, so the recursive call on the right-hand side is completely superfluous.

Figure 3.1 shows the results after applying this transformation to the Set equation by hand. There seems to be some gain. We also measured the same transformed rule after adapting the C code to return a normal form directly at the right-hand side. There was no significant difference measured there.

3.4 Detecting patterns

Whatever optimizations of patterns we come up with, special or more general, we need a way of detecting them in the compiler. A uniform approach for all patterns would be preferable. It would be very nice to to be able to add an optimizable pattern without changing a lot of extisting code. It seems natural to try and detect list patterns using list patterns. Which means, in the current compiler, that these patterns need to be detected *before* the simplification to μASF. After all, the associative list notation is translated to cons notation in that phase (Section 1.4).

But this location for detecting is impractical, the detection of a pattern and the actual transformation would be as far apart as possible. We can either adapt μASF to keep the associative lists, or we can awkwardly translate cons notation back to associative notation to solve this problem[4].

Apart from these practical considerations, we need a fundamental idea of pattern[5] equality. A starting point is to take syntactical equality of a μASF rule modulo a renaming relation α. We recognize the following complications:

- Function symbols different from the outermost symbol in the left in the right-hand side of a rule must be checked recursively for equality.

[4]Note that keeping the associative notation would also benefit the use of the *build* function.
[5]Remember that a list pattern is a synonym for a rule containing list variables.

- The same goes for foreign symbols in the conditions.
- Equality for single rules is not enough, we need to check all rules with the same outermost symbol in the left-hand side.

These are necessary requirements for equality of μASF functions. A unification algorithm over α will provide the equality test for a single rule. This syntactical matching algorithm is too strict for detecting more abstract classes of patterns. We need *meta variables* to provide us with *abstract rules* or *abstract functions*.

These meta variables range over parts of μASF rules. The most natural way is to let the meta variables range over the nonterminals of the μASF grammar. Since the compiler is implemented in ASF+SDF, we can use normal ASF+SDF variables as meta variables. And we can conveniently use ASF+SDF matching as the unification algorithm.

One of the hazards of optimizing abstract classes is that the designer of a class of patterns can under-specify a class, creating false positives. This might result in serious compilation errors. Another pitfall is that two optimizable classes can have a non-empty intersection. One pattern instance might benefit from the first optimization, but the next might benefit from the other. Finding a smart ordering for the optimizable classes will benefit the resulting code (if that is possible).

Some optimizations can be done at the μASF level. These can be applied at the time of detection. But other optimizations might be more subtle, and need appropriate C level transformations. These μASF functions should be *annotated* with this information, such that the compiler can do the appropriate transformations at the right time.

3.5 Discussion

This work is not finished. We have located a bottleneck in the execution of ASF+-SDF specifications: recursive list patterns. Some starting points for attacking this problem have been proposed: Firstly, we propose to change μASF to have associative lists instead of cons lists to improve matching capabilities. Secondly, we have started to partition the recursive patterns, introducing the continuous patterns and the guarded patterns. Thirdly, we presented some thoughts on the detection of classes (abstract patterns).

To implement a quick pilot test of any of the ideas presented in this chapter is not feasible. The current design of the compiler does not easily allow for such adaptations.

Chapter 4

Related work

In this chapter we discuss research related to optimizing matching and rewriting of lists. We will study the possible application of ideas to compilation of lists in ASF+SDF specifications. Other implementations of formal languages include: ELAN [2], CLEAN [19] and OPAL [18]. But we start with a discussion on an optimizing technique for lazy functional languages: deforestation.

4.1 Deforestation

In [23] a technique is introduced that transforms programs in non-strict lazy functional languages to eliminate intermediate term construction: deforestation. Especially when using lists, which is an often stimulated functional programming style, deforestation can result in significant run-time speedups.

Deforestation of functional programs is described by first unfolding and inlining compositions of functions and then rewriting specific patterns to eliminate intermediate results. Let us consider how we could apply some kind of deforestation algorithm to the compilation of ASF+SDF specifications.

Firstly, we have to solve inlining of compositions. At compile-time, we want to substitute function compositions like $f(g(t))$ for a specialized function $(fg)(t)$. Such a composite function then contains more information about the entire computation of $f(g(t))$ locally and might allow for specific optimizing steps[1]. We immediately notice that the combination of innermost reduction and the evaluation of conditions is a deadly combination. After all, the composition (fg) must check the conditions for both f and g. But the evaluation of the conditions of f depend on the right-hand side of g due to innermost reduction, so we have to evaluate g and construct its normal form anyway. This example also translates to the case where g has no conditions, but f does.

If, by chance, no matching is done on the arguments of f, then we can safely inline g into f. But since there is no matching, we cannot access any intermediate lists. So this type of inlining will probably not allow for any optimizations, except for saving a single reduction step.

Restricting ourselves to compositions $f(g(t))$ where both f and g have no conditions might help. We can now substitute $g(t)$ by its right-hand side at compile-time, effectively saving a single run-time reduction. But does this allow for any conservative deforesting transformations on (fg)? We suspect that any transformation on (fg), where (fg) is defined by substituting the right-hand side of g in the defini-

[1] A more real example: some μASF rule could first sort a list and then eliminate doubly occurring elements by specifying $\text{List}' = \text{set}(\text{sort}(\text{List}))$ in the conditions. We could change this condition using an optimizable composition to: $\text{List}' = \text{setsort}(\text{List})$.

tion of f, will be incompatible with innermost reduction. In the general case, the outermost symbol just does not contain enough information to predict its outcome when its arguments are not yet in normal form, which is the case at compile time.

Once more, we need more information on the specification before we can apply such smart optimizations: data-flow analysis. Or we restrict ourselves even more: take the functions g which right-hand sides only consist of constructors and normal forms. We now have all information at compile-time to construct an (fg) that, for example, eliminates the construction of the intermediate result of g. Consider the following example of a rewriting system. The function f concatenates its three arguments using a function g that concatenates its two arguments:

$$\begin{aligned} f(a,b,c) &= g(g(a,b),c) \\ g(a,b) &= \text{cons}(a,b) \end{aligned} \tag{4.1}$$

We have a function composition on the right-hand-side of f: $g(g(a,b),c)$. We create a special function for this composition, named (gg), by substituting the right-hand side of g into the first argument of g itself. Then we can substitute all occurrences of $g(g(t))$ by $(gg)(t)$:

$$\begin{aligned} f(a,b,c) &= (gg)(a,b,c) \\ g(a,b) &= \text{cons}(a,b) \\ (gg)(a,b,c) &= \text{cons}(\text{cons}(a,b),c) \end{aligned} \tag{4.2}$$

The (gg) function might now allow for an optimization. An obvious optimization would be to transform (gg) and eliminate the construction of $\text{cons}(a,b)$ by applying the build function:

$$(gg)'(a,b,c) = \text{build}(\text{BEGIN},a,b,c,\text{END})) \tag{4.3}$$

We have eliminated the construction of $\text{cons}(a,b)$. This was only possible after the inlining of g.

It has become clear that some kind of deforestation technique is possible for a specific subset of ASF+SDF functions. If this subset is large enough to have an effect on non-trivial applications is unknown. Furthermore, we need more research in the actual application of these ideas into the compiler.

4.2 ELAN

ELAN is a specification language similar to ASF+SDF, but with some distinct features. It does not have list matching, but it has AC matching [17]. In ELAN the user has the possibility to declare *associative* and *commutative* functions. AC matching backtracks over a term modulo associativity and commutativity.

The following example extracted from [17] demonstrates to use of AC matching. Consider the polynomials with integer coefficients $a_m * X^m + \cdots + a_1 * X + a_0$. If we want to simplify polynomials by deleting terms with a null coefficient, we simply write the rule: $P + 0 * P \rightarrow P$ in ELAN. The use AC matching leads to concise specifications in the context of associative and commutative operators. The same example using ASF+SDF list matching would be: $P1 + 0 * P2 + P3 = P1 + P3$[2], which is slightly larger due to the absence of commutativity.

There are some striking similarities between AC matching in ELAN and list matching in ASF+SDF. Most importantly, both introduce the need for local backtracking over a function. Secondly, they both handle associativity. In a certain way,

[2] $P1$ and $P3$ are list variables that much zero or more summands.

imports SDF
exports
  sorts Restriction Restrictions Lookaheads
  context-free syntax
    "(" Symbols "−/−" Lookaheads ")" → Restriction
    "[" Restriction∗ "]" → Restrictions
    {Lookahead ","}∗ → Lookaheads
hiddens
  variables
    R [*0-9*]∗ → Restriction
    R "∗"[*0-9*]∗ → Restriction∗
    L [*0-9*]∗ → Lookahead
    L "∗"[*0-9*]∗ → {Lookahead ","}∗
    [*ST*][*0-9*]∗ → Symbol
    [*ST*]"∗"[*0-9*]∗ → Symbol∗
    [*ST*]"+"[*0-9*] → Symbol+
equations

[0] $[R^*_1\ (S^*_1\ S\ S^*_2\ -/-\ L^*_1)\ R^*_2\ (T^*_1\ S\ T^*_2\ -/-\ L^*_2)\ R^*_3] = [\ ...\ ]$

Figure 4.1: A nested list pattern in the ASF+SDF implementation of the parser generator of the Meta-Environment. Brackets were added and the right-hand-side of the rule was omitted to improve readability.

AC matching is more general than list matching and therefor more complex. Note that the current ELAN compiler does not handle all occurrences of AC matching yet [2].

The matching algorithm for AC matching is far more complicated than the list matching algorithm in our compiler. Although in [17] an optimization of a subset of the AC patterns is proposed, even this matching algorithm remains too complicated for simple list matching in ASF+SDF. But, there is a class of list matching patterns that are almost as complex as AC matching problems: non-linear nested list patterns. Nested list pattern are not frequently used in ASF+SDF specifications. But this might be due to their inefficiency problems. A real world example of a listed nest pattern is found in the parser generator of the new Meta-Environment (Figure 4.1). This pattern contains *three* list types in two layers. Such a pattern results in very inefficient backtracking behavior of the compiled code. This class of patterns is not directly analogous to some kind of AC patterns, but applying some of the ideas of AC matching to this class of problems might be beneficial. For example, the use of constraint propagation to reduce the search space.

Considering the complexity of such algorithms and the scope of this thesis, we postpone further research in this subject.

4.3 CLEAN

CLEAN is a lazy functional programming language with a very fast implementation. The semantics and implementation of CLEAN are not based on the usual β-reduction scheme, but on general graph rewriting. By applying a functional reduction strategy on the graphs, a functional execution scheme is obtained.

One of the features of CLEAN that are of interest to us is the *uniqueness typing* *system*. This typing system calculates at compile-time which terms are used by only one single function, allowing this function to do destructive updates. CLEAN features indexed arrays. Especially when *indexed arrays* are updated destructively, a serious gain in performance is noted [14]. This supports our conclusions in Section 2.2 on the application of destructive updates on ASF+SDF lists.

But, uniqueness typing does depend on type annotations made by the CLEAN programmer. And uniqueness of objects is inferred in the context of a functional reduction strategy. So, we cannot borrow the uniqueness type inference system directly.

4.4 OPAL

OPAL is a specification language that combines algebraic specification with functional programming. Data-structures (types) are defined using algebraic specification. But the execution of OPAL specifications fits into the functional programming paradigm. OPAL programs are extended with a first-order property language to express certain properties of functions.

By adding algebraic properties to functional programs, more information is available for an optimizing compiler [10]. We notice that once more optimizations are dependent on the intelligence of the programmer[3]. The approach of adding specific information the guide the rewriting process is also found in other languages[4]. It remains to be seen if an optimizing compiler for a formal language can be constructed that does not need such methods.

In OPAL, there is a garbage collecting scheme that sometimes allows for destructive updates: the *selective update/reuse analysis*. It is based on static and *dynamic* reference counting.

[3]Compare this to uniqueness typing in CLEAN.
[4]For example, STRATEGO [22].

Chapter 5

Conclusion

We have considered two aspects of list rewriting separately: list construction and list matching. The main focus was on list construction. In this chapter we will briefly summarize the conclusions and give directions for future work.

List construction

Linearization of expressions that build lists proved to be a feasible optimization. It saves time and memory usage by applying this algorithm for list construction. Linearized list construction can easily be added to the current compiler.

We noticed that for most non trivial specifications the construction of lists is not the major bottleneck. The gain of linearization can even dissappear in the significance of matching. As an aside, we noticed the trade-off between time and memory efficiency in the garbage collector very clearly.

Related work and the study of compiled ASF+SDF code motivated our investigation into the use of *destructive lists*. We have shown in a pilot implementation that they can provide us with even faster execution of list matching rules. Problems that were tackled for this pilot implementation can be helpfull in a future implementation: the data-structure and the list construction algorithm.

The above conclusions motivated implementing destructive lists in a more general way. The restrictions we imposed on ourselves were not to change the current run-time environment of compiled ASF+SDF specifications (the ATerm library) and not to change the current compiler too much. We encountered a number of issues:

- Allocation and destruction of list nodes.
- Conversion to and from destructive lists.
- Evaluation of conditions containing lists.

We have solved each of these problems separately, but their combination proved to be either inefficient, or not leading to a correct algorithm. The conclusion is that our initial restrictions prohibit the general use of destructive lists in the ASF+SDF compiler.

In the light of future work, the most natural way seems to extend the ATerm library with destructive list nodes and to extend the compiler with either or both reference counting and some compile-time data-flow analysis to cover consistency issues.

List matching

We have argued that recursive list patterns are a bottleneck in the execution of ASF+SDF specifications. Then we investigated some preliminary ideas for optimizing them, introducing continuous patterns and guarded patterns.

The main utility needed for optimizing such patterns is detection of specific patterns and classes of patterns. We have briefly investigated the possible algorithm for detection, proposing some techniques and identifying some problems.

Our conclusions on this subject are indefinite, there is more work to be done.

Other directions of future work

The subject of optimizing formal languages is alive. We have found and discussed interesting techniques applied in the domain of functional languages. Their application in the domain of innermost rewriting is not yet totally clear, but we have indicated a possible way of applying a technique called *deforestation*.

Reducing the search space for a matching algorithm is done in the ELAN system. Parts of its algorithmic techniques might be applicable to reduce the work of *nested* list patterns, which are too inefficient to be used in the current ASF+SDF implementation.

Appendix A

The build function

The following C function is the implementation of the *build* function which is described in Section 2.1. This function can be added to either the ATerm library, or the support library. The (trivial) utility functions for memory allocation of temporary buffers are left out for the sake of brevity.

```
#include <aterm1.h>
#include <aterm2.h>

⋮

typedef enum { END = 0, CONCAT = 1, SLICE = 3, MAKE_LIST = 5,
               BEGIN = 7 } buildOperation;

static ATerm vbuild(buildOperation first, va_list args);

/* A temporary buffer for holding list elements */
static ATerm* buffer;

/* Name: build
 * Args: buildOperations and ATermList pointers
 * Pre : first = BEGIN, the last argument must be END,
 *       each SLICE is followed by 2 ATermLists,
 *       each MAKE_LIST is followed by 1 ATermList or 1 ATerm.
 * Post: A list containing all elements specified by the arglist
 *       in order of appearance is returned.
 */
ATerm build(buildOperation first, ...)
{
  va_list args;
  ATerm result;

  va_start(args, first);
  result = vbuild(first, args);
  va_end(args);

  return result;
}
```

```
/* Name: vbuild [internal function]
 *
 * See function: build for specific pre and postconditions,
 *               they are identical for vbuild. Only the
 *               argument list is given as a va_list in this
 *               function.
 */
static ATerm vbuild(buildOperation first, va_list args) {
  ATermList result = ATempty;
  int arg;
  int len = 0;

  /* We process the argument list, from left to right,
   * using the END tag as a stopcondition.
   *
   * A temporary buffer is used to keep the elements that
   * have to appear in the resulting list. But a list is only
   * added to the buffer when we see that another element needs
   * to be appended behind it. This way we can reuse the final argument
   * as the tail for the result by inserting the elements in
   * the buffer in front of it.
   */

  for(arg = first; arg != END; arg = va_arg(args, buildOperation)){
    switch(arg) {
    case BEGIN:
    case CONCAT:
    case MAKE_LIST:
      break;
    case SLICE:
      {
        /* Get the next two args and interpret them as ATermLists */
        ATermList list1 = va_arg(args, ATermList);
        ATermList list2 = va_arg(args, ATermList);

        if(!ATisEmpty(list1) && !ATisEqual(list1, list2)) {
          /* resize the buffer to be able to contain the previous tail
           * and the slice
           */
          RESIZE_BUFFER(len + ATgetLength(result) + ATgetLength(list1) - ATgetLength(list2));

          /* First insert the temporary tail into the buffer */
          for( ;!ATisEmpty(result); result = ATgetNext(result), len++)
            buffer[len] = ATgetFirst(result);

          /* Add the elements of the slice to the buffer */
          for( ;!ATisEqual(list1,list2); list1 = ATgetNext(list1), len++)
            buffer[len] = ATgetFirst(list1);
        }
      }
      break;
```

```
    default:
      {
        /* the argument is interpreted as an ATerm */
        ATerm term = (ATerm) arg;

        /* Create a new temporary tail from the argument, if arg is a list */
        if(ATgetType(term) == AT_LIST) {
          if(!ATisEmpty((ATermList) term)) {
            /* resize the buffer to be able to contain the previous tail */
            RESIZE_BUFFER(len + ATgetLength(result));

            /* First insert the previous temporary tail into the buffer */
            for( ;!ATisEmpty(result); result = ATgetNext(result), len++)
              buffer[len] = ATgetFirst(result);

            /* Set the temporary tail to the last list we found */
            result = (ATermList) term;
          }
        }
        /* Otherwise a singleton is added to the buffer */
        else {
          /* resize the buffer to be able to contain the previous tail
           * and the new element
           */
          RESIZE_BUFFER(len + ATgetLength(result) + 1);

          /* First insert the previous temporary tail into the buffer */
          for( ;!ATisEmpty(result); result = ATgetNext(result), len++)
            buffer[len] = ATgetFirst(result);

          /* Then add the new element to the buffer */
          buffer[len++] = term;
        }
      }
      break;
    }
  }

  /* Finally, reuse the temporary tail by appending the buffer
   * in front of it, creating the result list
   */
  while(--len >= 0)
    result = ATinsert(result, buffer[len]);

  return (ATerm) result;
}
```

References

[1] J. A. Bergstra, J.W. Klop, and A. Middeldorp. *Termherschrijfsystemen*. Kluwer programmatuurkunde, 1989.

[2] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P. Moreau, C. Ringeissen, and M. Vittek. *ELAN user manual*. Université de Nancy, 1998. V3.3.

[3] M. G. J. van den Brand and J. A. Bergstra. Syntax and Semantics of a High-Level Intermediate Representation for ASF+SDF. Technical report, University of Amsterdam, Programming Research Group, 1998.

[4] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: The ASF+SDF compiler. Technical report, Programming Research Group, University of Amsterdam, 1999. In preparation.

[5] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Acecpted for publication in Software – Practice & Experience*, 1999.

[6] M. G. J. van den Brand and M. de Jonge. Pretty-Printing within the ASF+SDF Meta-Environment: a Generic Approach. Technical report, CWI, Department of Software Engineering, 1999.

[7] M. G. J. van den Brand, P. Klint, and P. A. Olivier. Compilation and memory management for ASF+SDF. In *Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 198–213. Springer-Verlag, 1999.

[8] M. G. J. van den Brand, P. Klint, P. A. Olivier, and E. Visser. Syntax trees for distributed environments. Technical report, University of Amsterdam, Programming Research Group, 1997. In preparation.

[9] M. G. J. van den Brand and P. A. Olivier. *An AsFix to C compiler based on muASF*, 1998.

[10] K. Didrich, A. Fett, C. Gerke, W. G., and P. Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer, 1994.

[11] J. Fenlason and R. Stallman. *GNU gprof - The GNU Profiler*, 1997. `http://www.freesoft.org/gprof/gprof_toc.html`.

[12] A.J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, 1996.

[13] B. Gramlich. *Termination and confluence properties of structured rewrite systems*. PhD thesis, Fachbereich Informatik der Universität Kaiserlautern, 1996.

[14] J.H.G. van Groningen. The implementation and efficiency of arrays in clean 1.1. In *Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 105–124. Springer-Verlag, 1997.

[15] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The Syntax Definition Formalism SDF, Reference Manual*. University of Amsterdam, Programming Research Group, 1992.

[16] H. A. de Jong and P. A. Olivier. *ATerm Library User Manual*, 1998.

[17] P. Moreau and H. Kirchner. A compiler for rewrite programs in associative-commutative theories. In *Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 230–249. Springer-Verlag, 1998.

[18] P. Pepper. *The programming language Opal*. Fachbereich Informatik der Technische Universität Berlin, fifth edition, 1997.

[19] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[20] J. Romijn. Automatic analysis of term rewriting systems. Master's thesis, Programming Research Group, University of Amsterdam, 1995.

[21] E. Visser. Combinatory algebraic specification & compilation of list matching. Master's thesis, Programming Research Group, University of Amsterdam, 1993.

[22] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 1999.

[23] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.