Efficient Annotated Terms

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

[1] *CWI, Department of Software Engineering*
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

[1] *University of Amsterdam, Programming Research Group*
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

August 9, 1999

Abstract

How do distributed applications exchange (tree-like) data structures? To this end, we introduce the abstract data type of *Annotated Terms* (*ATerms*) and discuss their design, implementation and application. A comprehensive procedural interface enables creation and manipulation of ATerms in C. A Java version is also supported. The ATerm implementation is based on maximal subterm sharing and automatic garbage collection. A binary exchange format for the concise representation of ATerms (sharing preserved) allows the fast exchange of ATerms between applications. In a typical application—parse trees which contain quite some redundant information—less than 2 *bytes* are needed to represent a node in memory, and less than 2 *bits* are needed to represent it in binary format. It is shown that the implementation of ATerms scales up to the manipulation of ATerms in the giga-byte range.

1 Introduction

Cut and paste operations on complex data structures are standard in most desktop software environments: one can easily clip a part of a spreadsheet and paste it into a text document. The exchange of complex data is also common in distributed applications: complex queries, transaction records and more complex data are exchanged between different parts of a distributed application. Compilers and programming environments consist of tools such as editors, parsers, optimizers and code generators that exchange syntax trees, intermediate code, and the like.

How is this exchange of complex data structures between applications achieved? One solution is Microsoft's Object Linking and Embedding (OLE) [D96]. This is a platform-specific, proprietary, set of primitives to construct Windows applications. Another, language-specific, solution is to use Java's serialization interface [GJS96]. This allows writing and reading Java objects as sequential byte streams. Yet another solution is to use OMG's Interface Definition Language (part of the Common Object Broker Architecture [OMG97]) to define data structures in a language-neutral way. Specific language-bindings provide the mapping from IDL data structures to language-specific data structures.

All these solutions have their merits but do not really qualify when looking for an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of complex data structures between distributed applications. To be more specific, we are interested in a solution with the following characteristics:

Open: independent of any specific hardware or software platform.

Simple: the procedural interface should contain 10 rather than 100 functions.

Efficient: operations on data structures should be fast.

Concise: inside an application the storage of data structures should be as small as possible by using compact representations and by exploiting sharing. Between applications the transmission of data structures should be fast by using a compressed representation with fast encoding and decoding. Transmission should preserve any implicit or explicit sharing in the data structures.

Language-independent: data structures can be created and manipulated in any suitable programming language.

Annotations: applications can transparently extend the main data structures with annotations of their own to represent non-structural information.

In this paper we describe the data type of *Annotated Terms*, or just *ATerms*, that have the above characteristics. They form a solution for our implementation needs in the areas of interactive programming environments [Kli93, BKMO97] and distributed applications [BK98] but are more widely applicable. Typically, we want to exchange and process tree-like data structures such as parse trees, abstract syntax trees, parse tables, generated code, and formatted source texts. The applications involved include parsers, type checkers, compilers, formatters, syntax-directed editors, and user-interfaces written in a variety of languages. Typically, a parser may add annotations to nodes in the tree describing the coordinates of their corresponding source text and a formatter may add font or color information to be used by an editor when displaying the textual representation of the tree.

The ATerm data type has been designed to represent such tree-like data structures and it is therefore very natural to use ATerms both for the internal

representation of data inside an application and for the exchange of information between applications. Besides function applications that are needed to represent the basic tree structure, a small number of other primitives are provided to make the ATerm data type more generally applicable. These include integer constants, real number constants, binary large data objects ("blobs"), lists of ATerms, and the special placeholder subtype that is used to represent typed gaps in ATerms. Using the comprehensive set of primitives and operations on ATerms, it is possible to perform operations on an ATerm received from another application without first converting it to an application-specific representation.

First, we will give a quick overview of ATerms (Section 2). Next, we discuss implementation issues (Section 3) and give some insight in performance issues (Section 4). An overview of applications (Section 5) and an overview of related work and a discussion (Section 6) conclude this paper.

2 ATerms at a Glance

Without further fanfare we describe now the constructors of the ATerm data type (Section 2.1) and the operations defined on it (Section 2.2).

2.1 The ATerm Data Type

The data type of ATerms (`ATerm`) is defined as follows:

- `INT`: An integer constant (32-bits integer) is an ATerm.[1]
- `REAL`: A real constant (64-bits real) is an ATerm.
- `APPL`: A function application consisting of a function symbol and zero or more ATerms (arguments) is an ATerm. The number of arguments of the function is called the *arity* of the function.
- `LIST`: A list of zero or more ATerms is an ATerm.
- `PLACEHOLDER`: A placeholder term containing an ATerm representing the type of the placeholder is an ATerm.
- `BLOB`: A "blob" (Binary Large data OBject) containing a length indication and a byte array of arbitrary (possibly very large) binary data is an ATerm.
- A list of ATerm pairs may be associated with every ATerm representing a list of $(label, annotation)$ pairs.

Each of these constructs except the last one (i.e., `INT`, `REAL`, `APPL`, `LIST`, `PLACEHOLDER`, and `BLOB`) form subtypes of the data type `ATerm`. These subtypes are needed when determining the type of an arbitrary ATerm. Depending on the actual implementation language they will be represented as a constant (C, Pascal) or a subclass (C++, Java).

[1] We are currently upgrading the ATerm library to support 64-bits architectures as well.

The last construct is the *annotation construct*, which makes it possible to annotate terms with transparent information[2].

Appendix A contains a definition of the concrete syntax of ATerms. The primary reason for having a concrete syntax is to be able to exchange ATerms in a human-readable form. In Section 3 we also discuss a compact binary format for the exchange of ATerms in a format that is only suitable for processing by machine. We will now give a number of examples to show some of the features of the textual representation of ATerms.

- Integer and real constants are written as is: `1`, `3.14`, and `-0.7E34` are all valid ATerms.
- Function applications are represented by a function name followed by an open parenthesis, a list of arguments separated by commas, and a closing parenthesis. When there are no arguments, the parentheses may be omitted. Examples are: `f(a,b)` and `"test!"(1,2.1,"Hello world!")`. These examples show that double quotes can be used to delimit function names that are not identifiers.
- Lists are represented by an opening square bracket, a number of list elements separated by commas and a closing square bracket: `[1,2,"abc"]`, `[]`, and `[f,g([1,2]),x]` are examples.
- A placeholder is represented by an opening angular bracket followed by a subterm and a closing angular bracket. Examples are `<int>`, `<[3]>`, and `<f(<int>,<real>)>`.
- Blobs do not have a concrete syntax because their human-readable form depends on the actual blob content.

2.2 Operations on ATerms

The operations on ATerms fall into three categories: making and matching ATerms (Section 2.2.1), reading and writing ATerms (Section 2.2.2) and annotating ATerms (Section 2.2.3). The total of only 13 functions provide enough functionality for most users to build simple applications with ATerms. We refer to this interface as the *level one* interface of the ATerm data type.

To accommodate "power" users of ATerms we also provide a *level two* interface, which contains a more sophisticated set of data types and functions. It is typically used in generated C code that calls ATerm primitives, or in efficiency critical applications. These extensions are useful only when more control over the underlying implementation is needed or in situations where some operations that can be implemented using level one constructs, can be expressed more concisely and implemented more efficiently using level two constructs. The level

[2]Transparent in the sense that the result of most operations is independent of the annotations. This makes it easy to completely ignore annotations. Typical examples of the use of annotations include annotating parse trees with positional or typesetting information, and annotating abstract syntax trees with the results of type checking.

two interface is a strict superset of the level one interface (see Appendix B for further details).

Observe that ATerms are a purely functional data type and that no destructive updates are possible, see Section 3.2 for more details.

2.2.1 Making and Matching ATerms

The simplicity of the level one interface is achieved by the *make-and-match* paradigm:

- *make* (compose) a new ATerm by providing a pattern for it and filling in the holes in the pattern.
- *match* (decompose) an existing ATerm by comparing it with a pattern and decompose it according to this pattern.

Patterns are just ATerms containing placeholders. These placeholders determine the places where ATerms must be substituted or matched. An example of a pattern is `"and(<int>,<appl>)"`. These patterns appear as string argument of both make and match and are remotely comparable to the format strings in the `printf/scanf` functions in C. The operations are:

- `ATerm ATmake(String` p`, ATerm` a_1`, ..., ATerm` a_n`)`: Create a new term by taking the string pattern p, parsing it as an ATerm and filling the placeholders in the resulting term with values taken from a_1 through a_n. If the parse fails, a message is printed and the program is aborted. The types of the arguments depend on the specific placeholders used in *pattern*. For instance, when the placeholder `<int>` is used an integer is expected as argument and a new integer ATerm is constructed.
- `ATbool ATmatch(ATerm` t`, String` p`, ATerm *`a_1`, ..., ATerm *`a_n`)`:

  Match term t against pattern p, and bind subterms that match with placeholders in p with the result variables a_1 through a_n. Again, the type of the result variables depend on the placeholders used. If the parse of pattern p fails, a message is printed and the program is aborted. If the term itself contains placeholders these may occur in the resulting substitutions. The function returns `true` when the match succeeds, `false` otherwise.
- `Boolean ATisEqual(ATerm` t_1`, ATerm` t_2`)`: Check whether two ATerms are equal. The annotations of t_1 and t_2 must be equal as well.
- `Integer ATgetType(ATerm` t`)`: Retrieves the type of an ATerm. This operation returns one of the subtypes mentioned before in Section 2.1.

2.2.2 Reading and Writing ATerms

For reasons of efficiency and conciseness, reading and writing can take place in two forms: using ordinary text files and using special binary files. In the

first form, files are read and written in a format that is identical to the textual representation discussed earlier in Section 2.1 and Appendix A. This format is human-readable, space-inefficient[3], and any implicit sharing in terms is lost since this cannot be expressed in a pure textual representation.

In the second form, a dedicated binary format is used (Binary ATerm Format, see Section 3.5). This portable format is machine-readable, very compact, and preserves all implicit sharing. The operations are:

- `ATerm ATreadFromString(String` s`)`: Creates a new term by parsing the string s. When a parse error occurs, a message is printed, and a special error value is returned.
- `ATerm ATreadFromTextFile(File` f`)`: Creates a new term by parsing the data from file f. Again, parse errors result in a message being printed and an error value being returned.
- `ATerm ATreadFromBinaryFile(File` f`)`: Creates a new term by reading a binary representation from file f.
- `Boolean ATwriteToTextFile(ATerm` t`, File` f`)`: Write the text representation of term t to file f. Returns `true` for success and `false` for failure.
- `Boolean ATwriteToBinaryFile(ATerm` t`, File` f`)`: Write a binary representation of term t to file f. Returns `true` for success, and `false` for failure.
- `String ATwriteToString(ATerm` t`)`: Return the text representation of term t as a string.

Note that the file operations can also be used to read from or write to a network connection like a socket or pipe.

2.2.3 Annotating ATerms

Annotations are $(label, annotation)$ pairs that may be attached to an ATerm. Recall that ATerms are a completely functional data type and that no destructive updates are possible. This is evident in the following operations for manipulating annotations:

- `ATerm ATsetAnnotation(ATerm` t`, ATerm` l`, ATerm` a`)`: Return a copy of term t in which the annotation labeled with l has been changed into a. If t does not have an annotation with the specified label, it is added.

[3]The unnecessary size explosion could be avoided by extending the textual representation with a mechanism for labeling and referring to terms. Instead of `f(g(a),g(a))`, one could then write `f(1:g(a), #1)`. The first occurrence of `g(a)` is labeled with "`1`", and the second occurrence refers to this label ("`#1`").

- `ATerm ATgetAnnotation(ATerm` t`, ATerm` l`)`: Retrieve the annotation labeled with l from term t. If t does not have an annotation with the specified label, a special error value is returned.
- `ATerm ATremoveAnnotation(ATerm` t`, ATerm` l`)`: Return a copy of term t from which the annotation labeled with l has been removed. If t does not have an annotation with the specified label, it is returned unchanged.

3 Implementation

3.1 Requirements

In Section 1 we have already mentioned our main requirements: openness, simplicity, efficiency, conciseness, language-independence, and annotations. There are a number of other issues to consider that have a great impact on the implementation, and that make this a fairly unique problem:

- By providing automatic garbage collection ATerm users do not need to deallocate ATerm objects explicitly. This is safe and simple (for the user).
- The expected lifetime of terms in most applications is very short. This means that garbage collection must be fast and should touch as few memory locations as possible to improve caching and paging performance.
- The total memory requirements of an application cannot be estimated in advance. It must be possible to allocate more memory incrementally.
- Most applications exhibit a high level of redundancy in the terms being processed. Large terms often have a significant number of identical subterms. Intuitively this can be explained from the fact that most applications process terms with a fixed signature and a limited tree depth. When the amount of terms that is being processed increases, it is plausible that the similarity between terms also increases.
- In typical applications less than 0.1 percent of all terms have an arity that is higher than 5.
- Many applications will use annotations only sparingly. The implementation should not impose a penalty on applications that do not use them.
- In order to have a portable yet efficient implementation, the implementation language will be C. This poses some special requirements for the garbage collection strategy[4].

With these considerations in mind, we will now discuss maximal sharing of terms (Section 3.2), garbage collection (Section 3.3), the encoding of terms (Section 3.4), and the Binary ATerm Format (Section 3.5).

[4]We have implemented the library in Java as well. In this case, a lot of the issues we discuss in this paper are not relevant, either because we can use built-in features of Java (garbage collection), or because we just cannot express these low level concerns in Java.

3.2 Maximal Sharing

Our strategy to minimize memory usage is simple but effective: we only create terms that are *new*, i.e., that do not exist already. If a term to be constructed already exists, that term is reused thus ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be build and leads to the maximal sharing of subterms. The library functions to construct terms take care of building shared terms whenever possible. The sharing of terms is invisible.

3.2.1 The Effects of Maximal Sharing

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

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

Another consequence of our approach is less fortunate. Because terms can be shared without the creator knowing it, terms cannot be modified without creating unwanted side-effects. This means that terms effectively become *immutable* after creation. Destructive updates on maximally shared terms are not allowed. Especially in list operations, the fact that ATerms are immutable can be expensive. It is often the responsibility of the user of the library to choose algorithms that minimize the effect of this shortcoming.

3.2.2 Searching for Shared Subterms

The maximal sharing of terms can only be maintained when we check at term creation time whether a particular term already exists. The lookup must be *extremely fast*, in order to ensure efficient term creation. Using a hash function that depends on the *addresses* of the function symbol and the arguments of a function application, we can quickly search the hash table to find a function application before creating it.

Collisions One of the issues when using hash techniques is how to handle collisions in the hash table. The simplest technique is to linearly chain entries

together that hash to the same bucket. In this scenario, one pointer is needed in each object for hash chaining, which in our case means a memory overhead of about 25 percent. Other solutions for collision resolution will either increase the memory requirements (e.g. binary tree chaining), or the time needed for insertions or deletions (open addressing, see [Knu73]). We therefore use linear hash chaining in our implementation.

Direct or Indirect Hashing Another issue is whether to store all terms directly in the hash table, or to only store references in it. Storing the objects directly in the hash table has the advantage that we can save a memory access when retrieving a term. However, there are severe drawbacks to this approach:

- We cannot rehash the old terms because rehashing means that we have to physically move the objects in memory. In a setting where we use C as an implementation language, moving objects in memory is not allowed because we can only determine a conservative root set and therefore are not allowed to change the pointers to roots. This would mean that the hash table could not grow beyond its initial size, clearly an undesirable feature.
- The internal fragmentation is increased, because empty slots in the hash table are as large as the object size instead of only one machine word.
- We would need a separate hash table for each term size in order to decrease the internal fragmentation.

Because of these problems, we decided to use linear hash chaining in combination with indirect hashing. When the load of the hash table reaches a certain threshold, we allocate a larger table, rehash all the entries of the old table in the new one, and free the old hash table.

It is possible for a user to increase the initial size of the hash table in order to save resizing and rehashing operations. The ATerm library provides facilities for defining hash tables as well. This allows the implementation of a fast lookup mechanism for ATerms. The user-defined hash tables are used, for instance, to implement memo-functions in the ASF+SDF to C compiler, see Section 5.3.

3.3 Garbage Collection

3.3.1 Which Technique?

The most common strategies for automatic recycling of unused space are reference counting, mark-compact collection, and mark-sweep collection. We discuss these three strategies in turn, and explain why we base our design on mark-sweep garbage collection.

Reference Counting Reference counting in its simplest form works by keeping track of the number of references to an object. If this number drops to zero, there are no more references to an object and it can be reclaimed.

Reference counting has the advantage that unused terms can be reclaimed immediately when they become unused, while in the other strategies reclamation of space is more batch oriented. In most of the target applications this is not a major issue but it might be important in some cases, especially in real-time applications where the unpredictable pauses during execution caused by sweeping garbage collection are unacceptable. The usual problem with reference counting that circular structures are not collected does not apply because terms cannot be circular.

Unfortunately, in languages like C that do not offer much runtime support for dynamic memory management, the programmer needs to assist in updating reference counts, for instance, at function exit. Maybe an even bigger disadvantage is that reference counts need to be updated every time the number of references to an object changes. The total amount of work that has to be done for a pure reference counting approach is much higher than for the other strategies.

Another disadvantage of reference counting is the space overhead needed to store the reference count. As we shall see later, it is possible to implement the ATerm data type using only a couple of machine words[5] per object. Adding a whole word for keeping track of the reference count is therefore unacceptable.

Mark-compact Garbage Collection Mark-compact garbage collection works by periodically copying all *live* objects to a fresh empty memory space, and freeing the original space. Although this strategy has merits, it is unusable in our situation. Mark-compact garbage collection assumes that objects can be relocated to a different memory location, requiring that all references to objects can be updated. But when using C as an implementation language, we cannot positively identify *all* references to an object without complex support from the programmer. This problem can only be solved by using one more level of indirection but this solution is too expensive.

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

Mark-sweep garbage collection is very attractive, because it can be implemented in C, efficiently and without support from the programmer or compiler [BW88, Boe93]. Mark-sweep collection is more efficient, both in time and space than reference counting [JL96]. A possible drawback is increased memory fragmentation compared to for instance mark-compact collection. The typical space overhead for a mark-sweep garbage collection algorithm is only 1 bit per object, whereas a reference count field would take at least three or four bytes.

[5] We assume a word size of 4 bytes in this paper.

3.3.2 Reusing an Existing Garbage Collector

A number of excellent generic garbage collectors for C are freely available, so a legitimate question is why not reuse an existing implementation instead of implementing our own?

We have examined a number of alternatives, but none of these quite fitted our needs. The Boehm-Weiser garbage collector [BW88] came close, but we face a number of unusual circumstances that render existing garbage collectors impractical:

- The hash table always contains references to all objects. It must be possible to instruct the garbage collector not to scan this area for roots.
- After an object becomes garbage, it must also be removed from the hash table. This means that we need very low level control over the garbage collector.
- The ATerm data type has some special characteristics that can be exploited to dramatically increase performance:
  - Destructive updates are not allowed. In garbage collection terminology, this means that there are no pointers from old objects to younger objects. Although we do not exploit it in the current implementation, this characteristic makes the use of a *generational* garbage collector very attractive.
  - The overwhelming majority of objects only consists of between 8 and 16 bytes.
  - Practical experience has shown that not many root pointers are kept in static variables or on the generic C heap. Performance can be increased dramatically if we eliminate the expensive scan through the heap and the static data area for root pointers. The only downside is that we require the programmer to explicitly supply the set of roots that is located on the heap or in static variables.

These observations allow us to gain efficiency on several levels, using everything from low level system 'hacks' to high-level optimizations.

3.3.3 Implementing the Garbage Collector

Considering both performance and the maintainability of the code that uses the ATerm library, we have opted for a version of the mark-sweep garbage collector. Every object contains a single bit used by the mark-sweep algorithm to indicate 'live' (marked) objects. At the start of a garbage collection cycle, all objects are unmarked. The garbage collector tries to locate and mark all live objects by traversing all terms that are explicitly protected by the programmer (using the `ATprotect` function), and by scanning the C run-time stack looking for words that could be references to objects. When such a word is found, the object (and the transitive closure of all of the objects it refers to) are marked as 'live'.

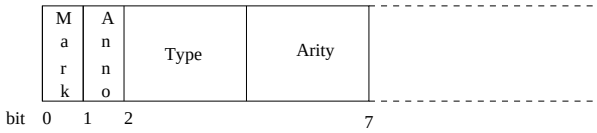

Figure 1: The header layout

This scan of the run-time stack causes all objects referenced from local variables to be protected from being garbage collected. Our garbage collector is a conservative collector in the sense that some of the words on the stack could accidentally have the same bit pattern as object references. Because there is no way to separate these 'fake' bit patterns from 'real' object references, this can cause objects to be marked as 'live' when these are actually garbage. Note that bit patterns on the stack that do not point to valid objects are not traversed at all. Only when a bit pattern represents an address that is a valid object address it is followed to mark the corresponding object.

When all live objects are marked, a single sweep through the heap is used to store all objects that are free in separate lists of free objects, one list for each object size.

As we shall see in Section 3.4, most objects consist of only a couple of machine words. By restricting the maximum arity of a function, we can also set an upper bound on the maximum size of objects. This enables us to base the memory management algorithms we use on a small number of block sizes. Allocation of objects is now simply a matter of taking the first element from the appropriate free-list, which is an extremely cheap operation. If garbage collection does not yield enough free objects, new memory blocks will be allocated to satisfy allocation requests.

3.4 Term Encoding

An important issue in the implementation of ATerms is how to represent this data type in such a way that all operations can be performed efficiently, without using more memory than is absolutely necessary.

The very concise encoding of ATerms we use is as follows. Assume that one machine word consists of four bytes, which is typical for modern architectures. Every ATerm object is stored in two or more machine words. The first byte of the first word is called the *header* of the object, and consists of four fields (see Figure 1):

- A field consisting of one bit used as a mark flag by the garbage collector.
- A field consisting of one bit indicating whether or not this term has an annotation.
- A field consisting of three bits that indicate the type of the term.

- A field consisting of three bits representing the arity (number of pointers to other terms) of this object. When this field contains the maximum value of 7, the term must be a function application and the actual arity can be found by retrieving the arity of the function symbol (see below).

Depending on the type of the node (as determined by the header byte in the first word) the remaining bytes in the first word contain either a function symbol, a length indication, or they are unused.

The *second* word is always used for hashing, and links together all terms in the same hash bucket.

The type of the node determines its exact layout and contents. Figure 2 shows the encoding of the different term types which we will now describe in more detail.

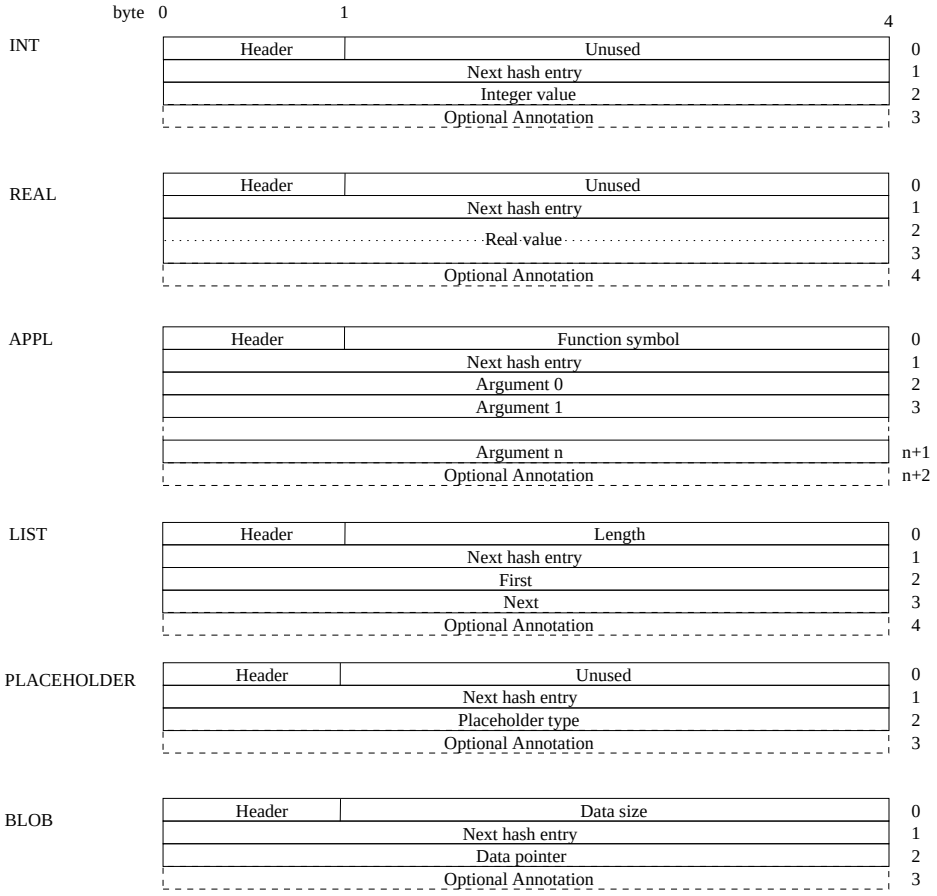

Figure 2: Encoding of the different term types

INT encoding In an integer term, the third word contains the integer value. The arity of an integer term is 0.

REAL encoding In an real term, the third and fourth word contain the real value represented by an 8 byte floating point number. The arity of a real term is 0.

APPL encoding The remaining 3 bytes following the header in the first word are used to represent the index in a table containing the function symbols. The words following the second word contain references to the function arguments. In this way, function applications can be encoded in $2 + n$ machine words, with n the arity of the function application.

LIST encoding The binary list constructor can be seen as a special function application with no function symbol and an arity of 2. The third word points to the first element in the list, this is called the `first` field, the fourth word points to the remainder of the list, and is called the `next` field. The length of the list is stored in the three bytes after the header in the first word. The empty list[6] is represented using a LIST object with empty first and next fields, and a length of 0.

After the function application, the list construct is the second most used ATerm construct. A (memory) efficient representation of lists is therefore very important. Due to the nature of the operations on ATerm lists, there are two obvious list representations: an array of term references or a linked list of term references. Experiments have shown that in typical applications quite varying list sizes are encountered. This renders the array approach inferior, because adding and deleting elements of a list would become too expensive. Consequently, we have opted for the linked list approach. Lists are constructed using binary list constructors, containing a reference to the first element in the list and to the tail of the list. Each list operation must ensure that the list is "normalized" again. This makes it very easy to perform the most commonly used operations on list, namely adding or removing the first element of a list.

Other operations are more expensive, due to the fact that we cannot allow destructive updates. Adding an element to the tail of a list for instance, requires n list creation operations, where n is the number of elements in the newly created list.

PLACEHOLDER encoding The placeholder term has an arity of 1, where the third word contains a pointer to the placeholder type.

BLOB encoding The length of the data contained in a BLOB term is stored in the three bytes after the header. This means that upto 16,777,200 bytes can be encoded in a single BLOB term. A pointer to the actual data is stored in the third word.

[6] Due to the uniqueness of terms, only one instance of the empty list is present at any time.

Annotations In all cases, annotations are represented using an extra word at the end of the term object. The single annotation bit in the header indicates whether or not an annotation is present. Only when this bit is set, an extra word is allocated that points to a term with type `LIST`, which represents the list of annotations.

3.5 The Binary ATerm Format (BAF)

As discussed in the introduction, the efficient exchange of ATerms between processes is very important. The simplest form of exchange is based on the concrete syntax presented in Appendix A. This would involve pretty printing the term on one side, and parsing it on the other. The concrete syntax is not a very efficient exchange format however, because the sharing of function symbols and subterms cannot be expressed in this way.

A better solution would be to exchange a representation in which sharing (both of function symbols and subterms) can be expressed concisely. A raw memory dump cannot be used, because addresses in the address space of one process have no meaning in the address space of another process.

In order to address these problems, we have developed BAF, the **B**inary **A**Term **F**ormat. Instead of writing addresses, we assign a unique number (index) to each subterm and each symbol occurring in a term that we want to exchange. When referring to this term, we could use its index instead of its address.

When writing a term, we first start by writing a table (in order of increasing index) of all function symbols used in this term. Each function symbol consists of the string representation of its name followed by its arity.

ATerms are written in prefix order. To write a function application, first the index of the function symbol is written. Then the indices of the arguments are written. When an argument consists of a term that has not been written yet, the index of the argument is first written itself before continuing with the next argument. In this way, every subterm is written exactly once. Every time a parent term wishes to refer to a subterm, it just uses the subterm's index.

3.5.1 Exploiting ATerm Regularities

When sending a large term containing a lot of subterms, the subterm indices can become quite large. Consequently a lot of bits are needed to represent these indices. We can considerably reduce the size of these indices when we take into account some of the regularities in the structure of terms. Empirical study shows that the set of function symbols that can actually occur at each of the argument positions of a function application with a given function symbol is often very small. A rationale for this is that although ATerm applications themselves are not typed, the data types they represent often are. In this case, function applications represent objects and the type of the object is represented by the function symbol. The type hierarchy determines which types can occur at each position in the object.

We exploit this knowledge by grouping all terms according to their top function symbol. Terms that are not function applications are grouped based on dummy function symbols, one for each term type. For each function symbol, we determine which function symbols can occur at each argument position. When writing the table of function symbols at the start of the BAF file, we write this information as well. In most cases this number of function symbol occurrences is very small compared to the number of terms that is to be written. Storing some extra information for every function symbol in order to get better compression is therefore worthwhile.

When writing the argument of a function application, we start by writing the actual symbol of the argument. Because this symbol is taken from a limited set of function symbols (only those symbols that can actually occur at this position), we can use a very small number to represent it. Following this function symbol we write the index of the argument term itself in the table of terms over this function symbol instead of the index of the argument in the total term table.

3.5.2 Example

As an example, we show how the term `mult(s(s(z)),s(z))` is represented in BAF. This term contains three function symbols: `mult` with arity two, `s` with arity one, and `z` with arity zero. When grouping the subterms by function symbol we get:

0: `mult`	1: `s`	2: `z`
`mult(s(s(z)),s(z))`	`s(s(z))` `s(z)`	`z`

When we look at the function symbols that can occur at every argument position (≥ 0) we get:

position	`mult`	`s`	`z`
0	`s`	`s`, `z`	
1	`s`		

We start by writing this symbol information to file. To do this, we have to write the following bytes[7]:

4 `"mult"` : The length (4) and ASCII representation of `mult`.
2 : The arity (2) of `mult`.
1 1 : There is only one symbol (1) that can occur at the first argument

[7] When the value of these numbers used exceeds 127, two or more bytes are used to encode them. Strings are written as strings to improve readability.

position of `mult`. This is symbol `s` with index (`1`)

`1 1` : At the second argument position, there is only (`1`) possible top symbol and that is `s` with index (`1`).

`1 "s"` : The length (`1`) and ASCII representation of `s`.

`1` : The arity (`1`) of `s`.

`2 1 2` : The single argument of `s` can be either of two (`2`) different top function symbols: `s` with index (`1`) or `z` with index (`2`).

`1 "z"` : The length (`1`) and ASCII representation of `z`.

`0` : The arity (`0`) of `z`.

Following this symbol information, the actual term `mult(s(s(z)),s(z))` can be encoded using only a handful of bits. Note that the first function symbol in the symbol table is always the top function symbol of the term (in this case: `mult`):

: No bits need to be written to identify the function symbol `s`, because it is the only possible function symbol at the first argument position of `mult`.

`0` : One bit indicates which term over the function symbol `s` is written (`s(s(z))`). Because this term has not been written yet, it is done so now.

`0` : The function symbol of the only argument of `s(s(z))` is `s`.

`1` : `s(z)` has index 1 in the term table of symbol `s`.

`1` : Symbol `z` has index 1 in the symbol table of symbol `s`.

: Because there is only one term over symbol `z`, no bits are needed to encode this term. Now we only need to encode the second argument of the input term, `s(z)`.

: No bits are needed to encode the function symbol `s`, because it is the only symbol that can occur as the second argument of mult.

`1` : `s(z)` has index 1 in the term table of symbol `s`. Because this term has already been written, we are done.

Only five bits are thus needed to encode the term `mult(s(s(z)),s(z))`. As mentioned earlier, the amount of data needed to write the table of function symbols at the start of the BAF file is in most cases negligible compared to the actual term data.

4 Performance Measurements

4.1 Benchmarks

How concise is the ATerm representation and how fast can BAF files be read and written? Since results highly depend on the actual terms being used, we will base our measurements on a collection of terms that cover most applications we have encountered so far.

4.1.1 Artificial Cases

Two artificial cases are used that have been constructed to act as borderline cases:

Random-unique: a randomly generated term over a signature of 9 fixed function symbols with arities ranging from 1 to 9 and an arbitrary number of constant symbols (functions with arity 0). The terms are generated in such a way that all constants are unique. These terms are the worst case for our implementation: there is no regularity to exploit and there are many subterms with a relatively high arity.

Random: a randomly generated term over a signature of 10 function symbols with arities ranging from 0 to 9. In these terms only a single constant can occur which will be shared, but no other regularities can be exploited and there are many subterms with a relatively high arity.

4.1.2 Real Cases

Several real-life cases are used that are based on actual applications:

COBOL Parse Table: a generated parse table for COBOL including embedded SQL and CICS. The grammar consists of 2009 productions and the generated automaton has 6699 states. The parse table contains an action-table (20947 non-empty entries) and a goto-table (76527 non-empty entries). This is an example of an abstract data type represented as ATerm.

COBOL System: a COBOL system consisting of 117 programs with a total of 247548 lines of COBOL source code. It has been parsed with the above parse table. The parse trees constructed for these COBOL programs are represented as ATerms, see Section 5.1.1 for more details.

Risla Library: a parse tree of the component library for the Risla language, a domain specific language for describing financial products [ADR95]. This component library consists of 10832 lines of code.

LPO: a linear process operator (LPO) describing the "firewire" protocol with 1 bus and 9 links [GL99, Lut99]. LPOs are the kernel of the μCRL ToolKit [DG95] which is a collection of tools for manipulation process and data descriptions in μCRL (micro Common Representation Language) [GP95]. An LPO is a structured process, where the state consists of an assignment to a sequence of typed data variables and its behaviour is described by condition, action and effect functions. These states are represented as ATerms, and are rather complex.

Casl specifications: a collection of abstract syntax trees represented as ATerms of 98 Casl files, the total number of lines of Casl code is 2506. For more details on Casl and the abstract syntax tree representation in ATerm we refer to Section 5.1.2.

Term	# nodes	# unique nodes	Sharing (%)	Memory (bytes)	Bytes/ Node
Artificial Cases					
Random-unique	1000000	1000000	0.00	15198694	15.20
Random	1000000	92246	90.81	2997120	3.00
Real Cases					
COBOL Parse Table	961070	97516	89.85	2836529	2.95
COBOL System	31332871	470872	98.50	12896609	0.41
Risla Library	708838	40073	94.35	960170	1.35
LPO	8894391	225229	97.47	3701438	0.42
Casl Specifications	34526	11699	66.12	235655	6.83
lcc Parse Forest	360829	86589	76.00	1547713	4.29
emacs Parse Forest	2288928	520473	77.26	9346680	4.08
Real Case Averages			85.65		2.90

Table 1: Memory usage of ATerms

lcc Parse Forest: a new back-end similar to the ASDL back-end [WAKS97] has been added to the lcc compiler [Han99]. This back-end maps the internal format used by the lcc compiler to ATerms. The ATerm format represents the equivalent information of an C program as the ASDL representation of that C program.

Given this back-end the C sources of the lcc compiler itself are mapped to ATerms. The lcc compiler consists of 34 source files, consisting of a total of 13588 lines of source code.

emacs Parse Forest: given the ATerm back-end of the lcc compiler the C sources of emacs have been processed as well. The emacs source consists of 169 source files, consisting of a total of 216766 lines of source code.

In the cases of the COBOL System, Casl Specifications, lcc Parse Forest, and emacs Parse Forest the set of ATerms are combined into and processed as *one* ATerm. Measurements were performed on an ULTRA SPARC-5 (270 MHz) with 256 Mb of memory. All times measured are the user CPU time for that particular job.

4.2 Measurements

In Table 1, we give results for the memory usage of our sample terms[8]. The five columns give the total number of nodes in each term, the number of unique nodes in each term, the sharing percentage, the amount of memory (in bytes) used for the storage of the term, and the average number of bytes needed per

[8]Since we consider the Random-unique and Random cases to be unrepresentative, we only present the averages for the real cases in this and the following tables.

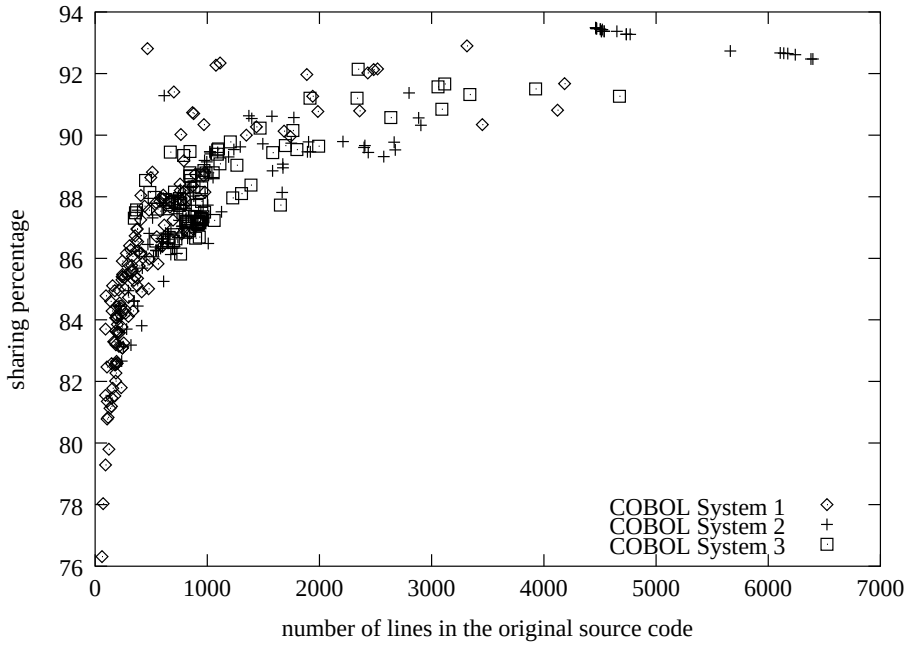

Figure 3: Sharing of a large number of COBOL parse trees

node. As can be seen in these figures, at least in our applications sharing *does* make a difference. By fully exploiting the redundancies in the input terms, we can store a node using on the average 2.9 bytes, and still perform operations on them efficiently. The worst case behaviour is 15 bytes per node. The amount of sharing is clearly less high in case of abstract syntax trees than in case of parse trees represented as AsFix terms. The AsFix terms contain a lot of redundant information which can be optimally shared. The amount of sharing in the abstract syntax trees for CASL is even lower, but this is due to the fact that the set of CASL specifications is small and each specification tests another feature of the CASL language, so not much sharing was to be expected.

Figure 3 shows the amount of sharing with respect to the size of a large number of COBOL programs. Three different sets of COBOL programs were considered. The first system consists of 151 files, the second of 116 files, and the last of 98 files. From this figure it can be concluded that the amount of sharing increases with the size of the COBOL system. In all three systems, the percentage of sharing converges to slightly over 90%. We find this high percentage in combination with the strong correlation between size and sharing very remarkable and will analyze its causes and consequences in further detail in a separate paper.

In Table 2 we give results for reading and writing our sample terms as Ascii text files. The six columns give the size of the text representation of the test term in bytes, the average number of bytes per node, the time needed to read

Term	ASCII (bytes)	Bytes/ Node	Read (s)	Read/ Node (μs)	Write (s)	Write/ Node (μs)
Artificial Cases						
Random-unique	6888889	6.89	34.76	34.76	4.06	4.06
Random	6200251	6.20	15.90	15.90	3.67	3.67
Real Cases						
COBOL Parse Table	4211366	4.38	6.33	6.95	2.30	2.29
COBOL System	135350005	4.32	199.43	6.36	65.02	2.08
Risla Library	2955964	4.17	4.25	6.00	1.40	1.98
LPO	41227481	4.64	81.90	9.21	29.16	3.28
Casl Specifications	217958	6.31	0.36	10.43	0.08	2.32
lcc Parse Forest	2132245	6.22	3.13	9.14	0.86	2.51
emacs Parse Forest	15793029	6.90	25.98	11.35	6.40	2.80
Real Case Averages		5.27		8.49		2.47

Table 2: Reading and writing ATerms as ASCII text

the text file, average time needed to read a node, the time needed to write the text file, and the average time needed to write a node. On the average, a node requires 5.3 bytes and reading and writing requires 8.5 μs and 2.5 μs, respectively.

In Table 3 we give results for reading and writing BAF files for the same set of sample terms. The columns give in order: the size of the BAF files in bytes, the average number of bytes needed per node, the time to read the BAF representation, the average read time per node, the time to write the BAF representation, and the average write time per node. Typically, we can read a node in 1.3 μs and write it in 2.4 μs.

Note that reading a BAF term is faster than writing the same term, whereas in case of ASCII the writing is faster than reading. This is caused by the fact that reading the ASCII representation of an ATerm involves quite some matching operations, whereas reading the BAF representation can be done with less matching. On the other hand, writing the BAF representation involves more calculations to encode the sharing of terms, whereas writing the ASCII representation involves a straightforward term traversal.

In Table 4 we show how the compression in BAF files compares to the compression of the standard Unix utility `gzip`. Considering the same set of examples, we give figures for a straightforward dump of each term as ASCII text (column 1), the size of the BAF version of the same term (column 2) and percentage of compression achieved (column 3). Next, we give the results of compressing the ASCII version of each term with `gzip` (column 4), and compression achieved (column 5). The compression factors are near 96% in both cases. The worst case compression of `gzip` (66%) is considerably better than

Term	BAF (bytes)	Bytes/ Node	Read (s)	Read/ Node (μs)	Write (s)	Write/ Node (μs)
Artificial Cases						
Random-unique	6073795	6.07	8.85	8.85	11.57	11.57
Random	567419	0.57	2.06	2.06	2.76	2.76
Real Cases						
COBOL Parse Table	370450	0.39	0.63	0.66	1.75	1.82
COBOL System	2279066	0.07	4.88	0.16	20.76	0.66
Risla Library	141946	0.20	0.22	0.31	0.75	1.06
LPO	1106661	0.12	1.86	0.21	9.40	1.06
Casl Specifications	32083	0.93	0.05	1.45	0.15	4.34
lcc Parse Forest	358318	0.99	0.71	1.97	1.50	4.16
emacs Parse Forest	2493457	1.09	10.25	4.48	10.43	4.56
Real Case Averages		0.54		1.32		2.38

Table 3: Reading and writing ATerms as BAF

that of BAF (12%).

4.3 Summary of Measurements

These measurements are summarized in Table 5. For in-memory storage, 2.9 byte is needed per node. In the case of BAF, only 0.54 byte (4.3 bit!) is needed to represent a node. Also observe that reading BAF is an order of magnitude faster than reading terms in textual form.

5 Applications

ATerms have already been used in applications ranging from development tools for domain specific languages [DK98] to factories for the renovation of COBOL programs [BSV97a]. The ATerm data type is also the basic data type to represent the terms manipulated by the rewrite engines generated by the ASF+SDF compiler [BKO99] and they play a central role in the development of the new ASF+SDF Meta-Environment [BKMO97].

5.1 Representing Syntax Trees: AsFix and CasFix

The ATerm data type proves to be a powerful and flexible mechanism to represent syntax trees. By defining an appropriate set of function symbols parse trees and abstract syntax trees can be represented for any language or formalism. We describe two examples: ASFIX (a parse tree format for ASF+SDF, Section 5.1.1) and CasFix (an abstract syntax tree format for Casl, Section 5.1.2).

Term	ASCII (bytes)	BAF (bytes)	Comp. (%)	gzip (bytes)	Comp. (%)
Artificial Cases					
Random-unique	6888889	6073795	11.8	2324804	66.3
Random	6199981	567419	90.9	439293	92.9
Real Cases					
COBOL Parse Table	4211366	370450	91.2	230297	94.5
COBOL System	135350005	2279066	98.3	3072774	97.7
Risla Library	2955964	141946	95.2	80009	97.3
LPO	41227481	1106661	97.3	804521	98.0
Casl Specifications	217958	32083	85.3	20767	90.5
lcc Parse Forest	2244691	358318	84.0	244502	89.1
emacs Parse Forest	15793029	2493457	84.2	1633754	89.7
Real Case Averages			90.8		93.8

Table 4: BAF *versus* gzip

	Memory	ASCII	BAF
Size per node (bytes)	2.9	5.27	0.54
Read node (μs)		8.49	1.32
Write node (μs)		2.47	2.38

Table 5: Summary of measurements (based on Real Case averages)

5.1.1 AsFix

AsFix (Asf+Sdf Fixed format) is an incarnation of ATerms for representing Asf+Sdf [HHKR92, BHK89, DHK96]. Asf+Sdf is a modular algebraic specification formalism for describing the syntax and semantics of (programming) languages. Sdf (Syntax Definition Formalism) allows the definition of the concrete and abstract syntax of a language and is comparable to (E)BNF. Asf (Algebraic Specification Formalism) allows the definition of the semantics in terms of equations, which are interpreted as rewrite rules. The development of Asf+Sdf specifications is supported by an integrated programming environment, the Asf+Sdf Meta-Environment [Kli93].

Using AsFix, each module or term is represented by its parse tree which contains both the syntax rules used and all original layout and comments. In this way, the original source text can be reconstructed from the AsFix representation, thus enabling transformation tools to access and transform comments in the source text. Since the AsFix representation is self-contained (all grammar information needed to interpret the term is also included), one can easily develop tools for processing AsFix terms which do not have to consult a common database with grammar information. Examples of such tools are a (structure)

editor or a rewrite engine.

AsFix is defined by an appropriate set of function symbols for representing common constructs in a parse tree. These function symbols include the following:

- `prod`(T) represents production rule T.
- `appl`(T_1, T_2) represents applying production rule T_1 to the arguments T_2.
- `l`(T) represents literal T.
- `sort`(T) represents sort T.
- `lex`(T_1, T_2) represents (lexical) token T_1 of sort T_2.
- `w`(T) represents white space T.
- `attr`(T) represents a single attribute.
- `attrs`(T) represents a list of attributes.
- `no-attrs` represents an empty list of attributes.

The following context-free syntax rules (in Sdf [HHKR92]) are necessary to parse the input sentence `true or false`.

```
sort Bool
context-free syntax
  true         -> Bool
  false        -> Bool
  Bool or Bool -> Bool {left}
```

The parse tree below gives the AsFix representation for the input sentence `true or false`.

```
appl(prod([sort("Bool"),l("or"),sort("Bool")],sort("Bool"),
          attrs([attr("left")])),
     [appl(prod([l("true")],sort("Bool"),no-attrs),[l("true")]),
      w(" "),l("or"),w(" "),
      appl(prod([l("false")],sort("Bool"),no-attrs),[l("false")])
     ])
```

Two observations can be made about this parse tree. First, this parse tree is an ordinary ATerm, and can be manipulated by all ATerm utilities in a completely generic way.

Second, this parse tree is completely self-contained and does not depend on a separate grammar definition. It is clear that this way of representing parse trees contains a lot of redundant information. Therefore, both maximal sharing and BAF are essential to reduce their size. In our measurements, AsFix only plays a role in the cases COBOL System and Risla Library.

The annotations provided by the ATerm data type can be used to store auxiliary information like position information derived by the parser or font and/or color information needed by a (structure) editor. This information is globally available but can be ignored by tools that are not interested in it.

5.1.2 CasFix

Casl (Common Algebraic Specification Language) is a new algebraic specification formalism [CL98] developed as part of the CoFI initiative. It is a general algebraic specification formalism incorporating common features of most existing algebraic specification languages. In addition to the language itself, a set of tools is planned for supporting the development of Casl specifications. Existing tools will be reused as much as possible.

In order to let the various tools, like parsers, editors, rewriters, and proof checkers, communicate with each other an intermediate format was needed for Casl. ATerms have been selected as intermediate format and a specialized version for representing the *abstract* syntax trees of Casl has been designed (CasFix [BKO98]). Contrast this with the approach taken for AsFix, where the more concrete *parse trees* are used as intermediate representation.

CasFix is obtained by defining an appropriate set of function symbols for representing Casl's abstract syntax [CL98] and by defining a mapping from Casl's concrete syntax to its abstract syntax. For each abstract syntax rule an equivalent CasFix construct is defined as in:

```
ALTERNATIVE ::= "total-construct" OP-NAME COMPONENTS*
```

$\Longrightarrow$

```
total-construct(<OP-NAME>,COMPONENTS*([<COMPONENTS>]))
```

In this example `"total-construct"` and `"COMPONENTS*"` are function symbols and `<OP-NAME>` and `<COMPONENTS>` represent the subtrees of the corresponding sort.

5.2 ASF+SDF Meta-Environment

The Asf+Sdf Meta-Environment [Kli93] is an interactive development environment for writing language specifications in Asf+Sdf. A new generation of this environment is being developed based on separate components connected via the ToolBus [BK98]. A description of this new architecture can be found in [BKMO97]. The new Meta-Environment provides tools for parsing, compilation, rewriting, debugging, and formatting. ATerms and AsFix play an important role in the new Meta-Environment:

- The parser generator [Vis97] produces a parse table represented as ATerm.
- The parser uses this parse table and transforms an input string into a parse tree which is represented as AsFix term.
- After parsing, the modules of an Asf+Sdf specification are stored as AsFix terms. Information concerning the specification such as the rewrite rules that must be compiled are exchanged as AsFix terms.
- The Asf+Sdf compiler (see next section) reads and writes AsFix terms.

Specification	ASF+SDF (equations)	ASF+SDF (lines)	Generated C code (lines)	ASF+SDF compiler (sec)	C compiler (sec)
ASF+SDF compiler	1876	8699	85185	216	323

Table 6: Some figures on the ASF+SDF compiler.

Application	Time (sec)	Memory (Mb)
ASF+SDF compiler (with sharing)	216	16
ASF+SDF compiler (without sharing)	661	117

Table 7: Performance with and without maximal sharing.

5.3 ASF+SDF to C compiler

The ASF+SDF to C compiler [BKO99] is a compiler for ASF+SDF. It generates ANSI-C code and depends on the ATerm library as run-time environment. All terms manipulated by the generated C code are represented as ATerms thus taking advantage of maximal subterm sharing and automatic garbage collection.

The optimized memory usage of ATerms has already been exploited in various industrial projects [BDK+96, BKV98] where memory usage is a critical success factor. This ASF+SDF compiler has, for instance, been applied successfully in projects such as the development of a domain-specific language for describing interest products (in the financial domain) [ADR95] and a renovation factory for restructuring of COBOL code [BSV97b].

The ASF+SDF compiler is an ASF+SDF specification and has been bootstrapped. Table 6 gives some figures on the size of this specification and the time needed to compile it. Table 7 gives an impression of the effect of compiling the ASF+SDF compiler with and without sharing. More information on the compiler itself and on performance issues can be found in [BKO99].

5.4 Other Applications

Other applications are still under development and include:

- A tool for protocol verification [GL99]. The ATerms are used to represent the states in the state space of the protocol. Because of the huge amount of states ($\geq 1,000,000$) it is necessary to share as many states as possible.
- A tool for the detection of code clones in legacy code.
- The Stratego compiler [VBT98].

6 Discussion

6.1 Related Work

S-expressions in LISP Many intermediate representations are derived in some form or another from the S-expressions in LISP. ATerms are no exception to this rule. The main improvements of ATerms over S-expressions are

- ATerms are language independent.
- ATerms support arbitrary binary data (Blobs, see Section 2.1).
- ATerms support annotations.
- ATerms support maximal sharing in a systematic way.
- ATerms support a concise, sharing preserving, exchange format that exploits the implicit signature of terms.
- The ATerm library provides a comprehensive collection of access functions based on the *match-and-make* paradigm.

Intermediate representations in compiler frameworks There exist numerous frameworks for compilers and programming environments that provide facilities for representing intermediate data. Examples are Centaur's VTP [BCD+89], Eli [GHL+92], Cocktail's Ast [Gro92] SUIF [WFW+94] and Montana [Kar98]. These systems either provide an explicit intermediate format (Eli, Ast, SUIF) or they provide a programmable interface to the intermediate data (VTP, Montana). Lamb's IDL [Lam87] and OMG's IDL [OMG97] are frameworks for representing intermediate data that are not tied to a specific compiler construction paradigm but have objectives similar to the systems already mentioned.

These approaches typically use a grammar-like definition of the abstract syntax (including attributes) and provide (generated) access functions as well as readers and writers for these intermediate data. In most cases support exists for accessing the intermediate data from a small collection of source languages.

The major difference between these approaches and ATerms is that they operate at different levels of abstraction. ATerms just provide the lower-level representation for terms (or more precisely directed acyclic graphs), while intermediate representations for compilers are more specialized and give a higher-level view on the intermediate data. They provide primitives for representing program constructs, symbol tables, flow graphs and other derived information. In most cases they also provide a fixed formats for representing programs at different levels of abstraction ranging from call graphs to machine-like instructions. ATerms are thus simpler and more general and they can be used to represent each of these compiler's intermediate formats.

Another difference is that most compiler frameworks use a statically typed intermediate representation. The major advantage is early error-detection. The

Term	ASCII	BAF	ASDL pickle
COBOL Parse Table	4211366	370450	5262426

Table 8: Sizes of the COBOL parse table

disadvantages are, however, less flexibility and the need to generate different access functions for each different intermediate format. In the case of ATerms, a dynamic check may be necessary on the intermediate data but only a single, generic, set of access functions is needed.

ASDL The abstract syntax definition language (ASDL) [WAKS97] is a language for describing tree data structures and is used as intermediate representation language between the various phases of a compiler [Han99]. The goals of ASDL and ATerms are in fact quite similar, exchanging syntax trees between tools, although ATerms are more general in the sense that other types of information, such as unstructured binary objects and annotations, can also be represented as an ATerm. Everything that can be represented by a grammar can be represented in ATerms as well as ASDL.

ASDL pickles and the BAF format for ATerms are comparable with respect to functionality, both are binary representations of (among others) syntax trees. The pickle and unpickle functions are generated from the ASDL description and are thus application specific (this may be more efficient) whereas the reading and writing of BAF is entirely generic (this avoids the proliferation of versions).

ASDL and ATerms can be compared at two different levels:

- *Low level:* ASDL pickle versus plain ATerms. By providing an ASDL definition of ATerms we can compare the size of the same object as ATerm (ASCII and BAF) and as ASDL pickle. This is done in Table 8 for the COBOL Parse Table. In this case, the representation in BAF is an order of magnitude smaller than the ASDL pickle.

- *High level*: compare at the level of parse trees or abstract syntax trees. ASDL is typically used to represent abstract syntax trees while ATerms can be used to represent both as we have discussed in Section 5.1. To make a meaningful comparison, we compare the abstract syntax trees generated by the lcc back-end in ATerm format (both in ASCII and BAF) and the corresponding ASDL pickles. These figures are presented in Table 9 for the abstract syntax trees generated for the lcc source files. In this case the BAF representation is 2 times smaller than the ASDL pickle. Note that the figure for the BAF representation differs from the figure in Table 3, this is caused by the fact that in Table 3 all files are combined into one BAF term whereas in Table 9 each file is a separate BAF term and their sizes are added.

Term	ASCII	BAF	ASDL pickle
lcc Parse Forest	2246436	624091	1290595

Table 9: Sizes of abstract syntax trees

XML The Extensible Markup Language [XML98] is a recently standardized format for Web documents. Unlike HTML, XML makes a strict distinction between *content* and *presentation*. XML can be *extended* by adding user-defined *tags* to parts of a document and by defining the overall structure of the document thus enabling well-formedness checks on documents. Although the original objectives are completely different, there are striking similarities between ATerms and XML: both serve the representation of hierarchically structured data and both allow arbitrary extensions (adding tags *versus* adding function symbols). There is also a straightforward translation possible between ATerms and XML.

The main difference between the two is that XML is more verbose and does not provide a simple mechanism to represent sharing. This may not be a problem for Web documents like catalogues and database records, but is does present a major obstacle in our case when we need to exchange huge terms between tools. We are currently considering whether some link between ATerms and XML may be advantageous.

Data encodings As described in Section 3.5, we use a form of data encoding to compress ATerms when they are exchanged between tools. Of course, encoding and data compression techniques are in common use in telecommunications. For instance, the ASN.1 standard gives detailed rules for data encoding [ASN95].

In an earlier project in our group, the Graph Exchange Language (GEL) [Kam94] has been developed. It is similar in goals as BAF, but BAF can only represent acyclic directed graph, while GEL can represent arbitrary (potentially cyclic) graphs. The technical approaches are different as well. GEL uses a binary-encoded postfix format to represent the nodes in the graph and introduces explicit labels to reuse previously constructed parts of the graph. BAF uses a prefix format augmented with generated symbol tables.

A final difference is in the *usage* of both approaches. GEL was used as a separate library that could be used in applications and the graph encoding was therefore visible to the programmer using it. BAF is, on the other hand, completely integrated in the ATerm implementation and is only used by the standard read and write functions for ATerms. The BAF format is therefore never visible to programmers.

6.2 History

Terms are so simple that most programmers prefer to write their own implementation rather than using (or even *looking for*) an existing implementation. This is all-right, except when this happens in a group of cooperating developers

as in our case.

A very first version of the ATerm library was developed as part of the ToolBus coordination architecture [BK98]. It was used to represent data which were transported between tools written in different languages running on different machines. Simultaneously, we were developing a formalism for representing parse trees [GB94, BKOV99]. In addition, incompatible term formats were in use in various of our compiler projects [FKW98]. Observing the similarities between all these incompatible term data types triggered the work on ATerms as described here. The benefits are twofold. First, a common term data type is used in more applications and investments in it easily pay-off. Second, the mere existence of a common data type leads to new, unanticipated, applications. For instance, we now use ATerms for representing parse tables as well.

6.3 Conclusions

As stated in the introduction, ATerms are intended to form an *open*, *simple*, *efficient*, *concise*, and *language independent* solution for the exchange of (tree-like) data structures between distributed applications.

ATerms *are* open and language independent since they do not depend on any specific hardware or software platform. ATerms *are* simple: the level one interface consists of only 13 functions. ATerms *are* efficient and concise as shown by the measurements in Section 4. In encoded form, less than 5 bits are needed per node! Last but not least, ATerms are also *useful* as shown on Section 5.

The most innovative aspects of ATerms are the simple procedural interface based on the *make-and-match* paradigm, term annotations, maximal subterm sharing, and the concise binary encoding of terms that is completely hidden behind high-level read and write operations.

Availability

The ATerm library can be obtained via

`http://www.wins.uva.nl/pub/programming-research/software/aterm/`

The current version of the library is available for Unix (including Linux) and Windows/NT. We are currently working on a 64-bits implementation of the library.

Acknowledgments

We want to thank all current users of the ATerm library. Special thanks to Jan-Friso Groote (for his input and sometimes severe requirements) and to Merijn de Jonge and Jeroen Scheerder (for detecting quite a lot of intricate bugs). Alexander van den Bergh who did the porting to Windows/NT as part of his Master thesis project. Joost Visser for providing us with useful information

on ASDL and for writing a tool to pickle ATerms. Finally, we want to thank Pierre-Etienne Moreau for the fruitful discussions on this subject during his stay in Amsterdam.

References

[ADR95] B.R.T. Arnold, A. van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE, April 1995.

[ASN95] Information Technology – Abstract Syntax Notation One (ASN.1): Encoding Rules – Packed Encoding Rules (PER). Technical report, International Telecommunication Union, 1995. ITU-T Recommendation X.691.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

[BDK+96] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and A.E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press/Addison-Wesley, 1989.

[BK98] J. A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

[BKMO97] M.G.J. van den Brand, T. Kuipers, L. Moonen, and P. Olivier. Design and implementation of a new asf+sdf meta-environment. In A. Sellink, editor, *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Workshops in Computing, Amsterdam, November 1997. Springer-Verlag.

[BKO98] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Aterms: Exchanging data between heterogeneous tools for casl. Note T-3, in [CoF98], 1998.

[BKO99] M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and Memory Management for ASF+SDF. In S. Jähnichen, editor,

Compiler Construction (CC'99), volume 1575 of *LNCS*, pages 198–213, 1999.

[BKOV99] M.G.J. van den Brand, P. Klint, P. Olivier, and E. Visser. Syntax trees for distributed environments. Technical report, University of Amsterdam, Programming Research Group, 1999. In preparation.

[BKV98] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 139–161. Elsevier Science, 1998.

[Boe93] H. Boehm. Space efficient conservative garbage collection. *PLDI*, pages 197–206, 1993.

[BSV97a] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[BSV97b] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software - Practice and Experience (SPE)*, 18(9):807–820, 1988.

[CL98] CoFI-LD. CASL – The CoFI Algebraic Specification Language – Summary, version 1.0. Documents/CASL/Summary-v1.0, in [CoF98], 1998.

[CoF98] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[9] and FTP[10], 1998.

[D96] Chappell D. *Understanding ActiveX(TM) and OLE.* MicroSoft Press, 1996.

[DG95] D. Dams and J.F. Groote. Specification and implementation of components of a μCRL toolbox. Technical Report 152, Utrecht University, 1995.

[9] http://www.brics.dk/Projects/CoFI
[10] ftp://ftp.brics.dk/Projects/CoFI

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

[DK98] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.

[FKW98] W. Fokkink, J. Kamperman, and H.R. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, 1998.

[GB94] C. Groza and M.G.J. van den Brand. The algebraic specification of annotated abstract syntax trees. Technical Report P9414, University of Amsterdam, Programming Research Group, 1994.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GL99] J.F. Groote and B. Lisser. Tutorial and reference guide for the μcrl toolset version 1.0. Technical report, CWI, Amsterdam, 1999. In preparation.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.

[Gro92] J. Grosch. Ast – a generator for abstract syntax trees. Technical Report 15, GMD Karlsruhe, 1992.

[Han99] D.R. Hanson. Early Experience with ASDL in lcc. *Software—Practice and Experience*, 29(3):417–435, 1999.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual*, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989.

[JL96] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Kam94] J. F. Th. Kamperman. Gel, a graph exchange language. Technical report, CWI, 1994.

[Kar98] M. Karasick. The architecture of Montana: an open and extensible programming environment with an incremental C++ compiler. In *Proceedings of the ACM SIGSOFT sixth International Symposium on Foundations of Software Engineering*, pages 131–142, 1998.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[Knu73] D. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[Lam87] D.A. Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9(3):297–318, 1987.

[Lut99] S.P. Luttik. Description and formal specification of the link layer protocol (sen-r9706). Technical report, CWI, Amsterdam, 1999.

[OMG97] OMG. The common object request broker: Architecture and specification, revision 2,0. Technical Report 97-02-25, Object Management Group, 1997. Available at: `http://www.omg.org`.

[VBT98] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizes with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[WAKS97] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, 1997.

[WFW+94] R.P. Wilson, R.S. French, Ch.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K.Tjiang, Shih-Wei Liao, Chau-Wen Tseng, M.W. Hall, M.S. Lamm, and J.L. Hennessy. SUIF: An infrastructure for research on parallilizeing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[XML98] Extensible markup language (XML) 1.0. Technical report, World Wide Web Consortium, 1998. Available at: `http:www.w3.org/TR/REC-xml`.

A Concrete Syntax

A formal definition of the concrete syntax of ATerms using the syntax definition formalism SDF [HHKR92] is presented here. Note that there is no concrete syntax defined for blobs, because a humanly readable representation of blobs depends on the type of data stored in the blob.

hiddens
sorts EscChar AFunChar ATerms Annotation

lexical syntax
"\"~□ → EscChar
"\"[01][0-7][0-7] → EscChar
~[\000-\040"\] → AFunChar
EscChar → AFunChar
context-free syntax
ATerm → ATerms
ATerm "," ATerms → ATerms
"{" ATerms "}" → Annotation
exports
sorts ATerm ATermList ATermAppl ATermInt ATermReal ATermPlaceholder AFun
lexical syntax
[␣\n\t] → LAYOUT

[a-zA-Z][a-zA-Z0-9_*\+\-]* → AFun
"\""AFunChar*"\"" → AFun

[0-9]+ → ATermInt
"-"ATermInt → ATermInt
ATermInt "."[0-9]+ → ATermReal
ATermInt "."[0-9]+"e"ATermInt → ATermReal
context-free syntax
AFun → ATermAppl
AFun "(" ATerms ")" → ATermAppl

"[" "]" → ATermList
"[" ATerms "]" → ATermList

"<" ATerm ">" → ATermPlaceholder

ATermAppl → ATerm
ATermAppl Annotation → ATerm
ATermList → ATerm
ATermList Annotation → ATerm
ATermInt → ATerm
ATermInt Annotation → ATerm
ATermReal → ATerm
ATermReal Annotation → ATerm
ATermPlaceholder → ATerm
ATermPlaceholder Annotation → ATerm

B Level 2 interface

The operations described in Section 2 are not sufficient for all applications. Some applications need more control over the underlying implementation, or

need operations that can be implemented using level one constructs but can be expressed more concisely and implemented more efficiently using more specialized constructs.

We have therefore designed a level 2 interface that is a strict superset of the level 1 interface described in Section 2. Some new datatypes are introduced, as well as some new operations on ATerms.

The level 2 interface introduces 7 new datatypes. Except for the auxiliary datatype AFun for representing function symbols, they are subtypes of the ATerm datatype, and implement the different term types. These subtypes allow us to introduce operations that are only valid for one specific term type, instead of the general ATerm operations described earlier.

ATermInt: This datatype represents integer terms. The operations on ATermInt are:

- `ATermInt ATmakeInt(Integer` v`)`: Construct a new integer term corresponding to the integer value v.
- `Integer ATgetInt(ATermInt` i`)`: Retrieve the value of an integer term.

ATermReal: This datatype represents real-number terms. The operations on ATermReal are:

- `ATermReal ATmakeReal(Real` v`)`: Construct a new real term.
- `Real ATgetReal(ATermReal` r`)`: Retrieve the value of a real term.

AFun: An AFun consists of a string defining the function name, an arity, and an indication whether the symbol name is quoted or not. The operations on symbols are:

- `AFun ATmakeAFun(String` nm`, Integer` ar`, Boolean` q`)`: Construct a new symbol. If a symbol with the given name nm, arity ar, and quotation q already exists, the existing symbol is returned. Otherwise a new symbol is created and returned. AFuns are also subject to garbage collection in order to avoid long running (interactive) programs from slowly running out of symbols.
- `String ATgetName(AFun` s`)`: Retrieve the name of symbol s.
- `Integer ATgetArity(AFun` s`)`: Retrieve the arity of a symbol.
- `Boolean ATisQuoted(AFun` s`)`: Check if a symbol is quoted.

ATermAppl: This datatype represents function applications consisting of a function symbol and a number of arguments. The operations on this datatype are:

- `ATermAppl ATmakeAppl`n`(AFun` f`, ATerm` a_0`, ..., ATerm` a_{n-1}`)`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to construct a new function application with the given function symbol f and arguments.
- `ATermAppl ATmakeAppl(AFun` f`, ATermList` as`)`: Construct a new function application with the given function symbol f and a list of arguments $args$
- `AFun ATgetFun(ATermAppl` ap`)`: Retrieve the function symbol of a function application.
- `ATerm ATgetArgument(ATermAppl` ap`, Integer` n`)`: Retrieve a specific argument.

ATermList: This datatype represents the binary list constructor. Element indices start at 0. Thus a list of length n has elements $0, \ldots, n-1$. The operations on ATermList are:

- `ATermList ATmakeList`n`(ATerm` e_0`, ..., ATerm` e_{n-1}`)`: This is a family of operations, one for each n between 0 and 6 (inclusive). These operations are used to quickly construct small lists of terms.
- `Integer ATgetLength(ATermList` l`)`: Retrieve the length of l.
- `ATerm ATgetFirst(ATermList` l`)`: Retrieve the first element of list l.
- `ATermList ATgetNext(ATermList` l`)`: Retrieve all but the first element of list l.
- `ATermList ATgetPrefix(ATermList` l`)`: Retrieve all but the last element of list l.
- `ATerm ATgetLast(ATermList` l`)`: Retrieve the last element from list l.
- `ATermList ATgetSlice(ATermList` l`, Integer` frm`, Integer` to`)`: Retrieve the portion of list l from position frm through $to-1$.
- `Boolean ATisEmpty(ATermList` l`)`: Check if list l contains zero elements.
- `ATermList ATinsert(ATermList` l`, ATerm` e`)`: Insert a single element e at the start of list l.
- `ATermList ATinsertAt(ATermList` l`, ATerm` e`, Integer` i`)`: Insert a single element e at position i in list l.

- `ATermList ATappend(ATermList` l`, ATerm` e`)`: Append a single element e to the end of list l.
- `ATermList ATconcat(ATermList` l_1`, ATermList` l_2`)`: Concatenate lists l_1 and l_2.
- `Integer ATindexOf(ATermList` l`, ATerm` e`, Integer` i`)`: Search for an element e in list l and return the index of the first location where e is present. Start searching at index i. If the element is not present, return -1.
- `Integer ATlastIndexOf(ATermList` l`, ATerm` e`, Integer` i`)`: Search backwards for element e in list l, and return the index of the last location where the element is present. Start searching at index i. If the element is not present, return -1.
- `ATerm ATelementAt(ATermList` l`, Integer` i`)`: Retrieve element at position i from list l.
- `ATermList ATremoveElement(ATermList` l`, ATerm` e`)`: Remove once occurrence of element e from list l.
- `ATermList ATremoveElementAt(ATermList` l`, Integer` i`)`: Remove the element at position i from list l.

ATermPlaceholder: This datatype represents placeholder terms. The operations on ATermPlaceholder are:

- `ATermPlaceholder ATmakePlaceholder(ATerm` tp`)`: Construct a new placeholder term.
- `ATerm ATgetPlaceholder(ATermPlaceholder` ph`)`: Retrieve the type of this placeholder.

ATermBlob: This datatype represents Binary Large OBject terms. The operations on ATermBlob are:

- `ATermBlob ATmakeBlob(Integer` n`, Data` d`)`: Construct a new blob term of size n and containing data d.
- `Integer ATgetBlobSize(ATermBlob` b`)`: Retrieve the size of blob b.
- `Data ATgetBlobData(ATermBlob` *blob*`)`: Retrieve the data pointer stored in blob b.

The memory management of blobs must be done explicitly by the application programmer.

Auxiliary: The level two interface provides functionality to create and manipulate user-defined hash tables.