Annotated Terms for Efficient Data Exchange

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

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

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

Abstract

How do distributed applications exchange tree-like data structures? 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 or Java. 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 considerable 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. The implementation of ATerms scales up to the manipulation of ATerms in the giga-byte range.

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) [Cha96]. 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 approach 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. Currently XML [XML98] is a very popular format to exchange information between applications. More and more tool vendors are switching to XML. One of the biggest drawbacks of XML is that it only allows a restricted form of sharing, and the fact that XML terms can only be exchanged in a verbose plain text format. The latter is quite often solved by using compression tools like `(g)zip`.

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 sharing of in-memory representation 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 placeholders 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.

In this paper, we give an overview of the ATerm data type and its operations and discuss implementation issues. We briefly sketch some applications and discuss related approaches. For a full account of ATerms, including detailed performance measurements, we refer the reader to [BJKO00].

ATerms at a Glance

We now describe the constructors of the ATerm data type and the operations defined on it.

The ATerm Data Type

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

- `INT`: An integer constant is an ATerm.
- `REAL`: A real constant 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 the type is represented as a constant (`C`) or a subclass (`Java`, `C#`).

The last construct is the *annotation construct*, which makes it possible to annotate terms with transparent information[1].

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 conventionally: `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.

[1]Transparent in the sense that the result of most operations is independent of the annotations. This makes it easy to completely ignore annotations. 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.

Operations on ATerms

The operations on ATerms fall into three categories: making and matching ATerms, reading and writing ATerms, and annotating ATerms. The total of only 13 functions provides 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 two interface is a strict superset of the level one interface.

Observe that ATerms are a purely functional data type and that no destructive updates are possible, see the section on maximal sharing for more details.

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 for making and matching ATerms 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 depends on the place holders 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.

- `ATbool 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 above.

Reading and Writing ATerms

For reasons of efficiency and conciseness, reading and writing can take place in two forms: text and binary. The text format uses the textual representation discussed earlier. This format is human-readable, space-inefficient[2], and any sharing of the in-memory representation of terms is lost.

The binary format (Binary ATerm Format) is portable, machine-readable, very compact, and preserves all in-memory sharing. The operations for reading and writing ATerms 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.

- `String ATwriteToString(ATerm` t`)`: Return the text representation of term t as a string.

- `ATbool ATwriteToTextFile(ATerm` t`, File` f`)`: Write the text representation of term t to file f. Returns `true` for success and `false` for failure.

- `ATbool ATwriteToBinaryFile(ATerm` t`, File` f`)`: Write a binary representation of term t to file f. Returns `true` for success, and `false` for failure.

Either format (textual or binary) can be used on any linear stream, including files, sockets, pipes, etc.

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.

- `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.

[2]We also support a textual format in which the unnecessary size explosion is avoided using a mechanism for implicit labeling and referring to terms. Instead of `f(g(a),g(a))`, one could then write `f(g(a),#A)`. The first occurrence of `g(a)` is implicitly labeled with "`A`", and the second occurrence refers to this label ("`#A`").

- `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.

Implementation

Requirements

In the introduction we have already mentioned our main requirements: openness, simplicity, efficiency, conciseness, language-independence, and capable of dealing with 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 a minimal amount of memory locations 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 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 on the garbage collection strategy[3].

With these considerations in mind, we will now discuss maximal (in-memory) sharing of terms, garbage collection, the encoding of terms, and the Binary ATerm Format.

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, ensuring maximal sharing. This strategy fully exploits the redundancy that is typically present in the terms to be built and leads to maximal sharing of subterms. The library functions that construct terms make sure that shared terms are returned whenever possible. The sharing of terms is thus invisible to the library user.

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 be 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. All 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 the table and ensures that all entries in the table are of equal size. Hence the (modest but not negligible) cost at term creation time is one hash table lookup.

Fortunately, we get two returns on this investment. First, the considerably reduced memory usage also leads to reduced execution time. Second, we gain substantially as the equality check on

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

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 their 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.

Collisions One issue in hash techniques is handling collisions. The simplest technique is linear chaining [Knu73]. This requires one pointer in each object for hash chaining, which in our implementation implies a memory overhead of about 25 percent. Other solutions for collision resolution will either increase the memory requirements, or the time needed for insertions or deletions (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 only references. Storing the objects directly in the hash table saves a memory access when retrieving a term as well as the space needed to store the reference. However, there are severe drawbacks to this approach:

- We cannot rehash old terms because rehashing means that we have to move the objects in memory. When using 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.
- Internal fragmentation is increased, because empty slots in the hash table are as large as the object instead of only one machine word.

Because of these problems, we use linear hash chaining combined with indirect hashing. When the load of the hash table reaches a certain threshold, we rehash into a larger table.

The user can increase the initial size of the hash table to save on resizing and rehashing operations.

The ATerm library provides facilities for defining application specific hash tables as well. This allows the implementation of a fast lookup mechanism for ATerms.

Garbage Collection

Which Technique?

The most common strategies for automatic recycling of unused space are reference counting, mark-compact collection, and mark-sweep collection. In our case, reference counting is not a valid alternative, because it takes too much time and space and is very hard to implement in C. Mark compact garbage collection is also unattractive because it assumes that objects can be relocated. This is not the case in C where we cannot identify *all* references to an object. We can only determine the root set conservatively which is good enough for mark-sweep collection discussed below, but not for mark-compact collection.

Mark-sweep Garbage Collection Mark-sweep garbage collection works using three 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 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 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.

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'.

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 they 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 the next section, 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.

Term Encoding

An important issue in the implementation of ATerms is how to represent this data type so that all operations can be performed efficiently in time and space.

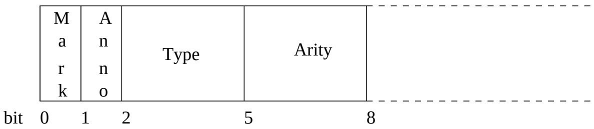

Figure 1: The header layout

The very concise encoding of ATerms we use is as follows. Assume that one machine word consists of four bytes. 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. We will now describe the encoding of the different term types in more detail.

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 IEEE 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 is represented using a LIST object with empty first and next fields, and a length of 0. The arity of the empty list is 0.

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 up to 16,777,200 bytes can be encoded in a single BLOB term. A pointer to the actual data is stored in the third word.

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.

ATerm Exchange: the Binary ATerm Format and Shared Textual ATerm Format

The efficient exchange of ATerms between tools is very important. The simplest form of exchange is based on the concrete syntax. This would involve 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.

Binary ATerm Format 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 begin by writing a table (in order of increasing indices) 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.

Shared Textual ATerm Format In addition to the binary aterm format there is also a textual aterm format which supports maximal sharing but uses a much less complex algorithm than the one used to encode and decode BAF files. This results in files that are somewhat larger than their baf counterparts, but are often (if the terms contain redundancy) significantly smaller than their unparsed form.

The format uses abbreviations to refer to previously written terms. An abbreviation consists of a hash character ('#') followed by a number in encoded using the Base64 Alphabet.

Each term whose unparsed representation would take up more bytes than the textual representation of the next available abbreviation is assigned such an abbreviation *after* it has been written. Subsequent occurrences of this term are then written by emitting the abbreviation instead of the term itself.

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 [BSV97]. 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 [BDH+01].

We will not give an extensive description of the various applications but we will mention a few striking ones:

- Components of the ASF+SDF Meta-Environment:
  - Parse table generator [Vis97]: parse table is an ATerm.
  - Parser: parse table is an ATerm and the parse tree is represented as an ATerm.
  - ASF+SDF compiler [BKO99] uses the ATerm library as run-time environment.
- 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 much data as possible.
- A tool for the detection of code clones in legacy code.
- The Stratego compiler [VBT98].
- CasFix [BS00], an abstract syntax tree representation for the algebraic specification language Casl [CL98].
- Efix [vdBR01], an abstract syntax tree representation for the specification language ELAN [BKK+98].

Discussion

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 support arbitrary binary data (Blobs).
- 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], ASDL [WAKS97], 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, ASDL). 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.

XML The Extensible Markup Language [XML98] is a 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, whereas ATerms provide the BAF format. This may not be a problem for Web documents like catalogs and database records, but it 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.

Another main difference is the use of DTD (document type definition) in XML, which allows the definition of meta-structure of the represented data. This can be compared with an (abstract) syntax definition when representing syntax trees. The content of the XML document can be validated using the corresponding DTD. The ATerm format does not provide an explicit DTD mechanism. This could be a useful extension in the future.

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. Last but not least, ATerms are also *useful*.

The ATerm format is supported by a binary exchange format (BAF) which provides a mechanism to exchange ATerms in a concise way. This BAF format maintains the in-memory sharing of terms and uses a minimal amount of bits to represent the nodes, in case of AsFix terms only 2 bits are needed per node.

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.

For a more detailed account of ATerms, including performance measurements, we refer the reader to [BJKO00].

Availability

The ATerm library is available on the web at `http://www.cwi.nl/projects/MetaEnv/aterm`.

References

[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).

[BDH+01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[BJKO00] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30:259–291, 2000.

[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.

[BKK+98] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and Ch. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Science Publishers, 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.

[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.

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

[BS00] M.G.J. van den Brand and J. Scheerder. Development of Parsing Tools for CASL using Generic Language Technology. In D. Bert and C. Choppy, editors, *Workshop on Algebraic Development Techniques (WADT'99)*, volume 1827 of *LNCS*. Springer-Verlag, 2000.

[BSV97] 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.

[Cha96] D. Chappell. *Understanding ActiveX(TM) and OLE*. MicroSoft Press, 1996.

[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[4] and FTP[5], 1998.

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

[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.

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

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

[4] http://www.brics.dk/Projects/CoFI
[5] ftp://ftp.brics.dk/Projects/CoFI

[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.

[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 optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, 1998.

[vdBR01] M.G.J. van den Brand and C. Ringeissen. Asf+sdf parsing tools applied to elan. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers, 2001.

[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`.

About the Authors

Mark van den Brand completed his PhD on the generation of incremental programming environments at the University of Nijmegen. He is a permanent staff member at CWI and his interests include generic language technology, compilation techniques and parser generation.

Hayco de Jong holds a MSc in Computer Science from the University of Amsterdam and is currently working on his PhD at CWI in Amsterdam. His interests include component technologies and the automatic generation of component interfaces and adapters.

Paul Klint completed his PhD on string processing languages at the Technical University of Eindhoven. He is themeleader at CWI and professor in computer science at the University of Amsterdam. His research interests include generic language technology, domain-specific languages, software renovation, component-based development and coordination architectures.

Pieter Olivier completed his PhD on debugging of heterogeneous applications at the University of Amsterdam. He is currently working as post-doc at CWI in Amsterdam. His interests include component technologies, debugging, and online multiplayer games.