The binary aterm file/stream format

Part of [1], written by van den Brand, de Jong, Klint and Olivier

1 The Binary ATerm Format (BAF)

This text is largely literally copied from [1]. Efficient storing and exchange of ATerms between processes is very important. The simplest form of employs concrete syntax. This would involve pretty printing and parsing the term. The concrete syntax is not a very efficient exchange format because sharing of function symbols and subterms cannot be employed.

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 process have no meaning in the address space of another process. In order to address these problems we have developed BAF, the Binary Aterm Format. 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.

1.1 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.

1.2 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:

4 `"mult"` : The length (4) and the ASCII representation of `mult`.
1 : The number of unique subterms that has `mult` as its top symbol.
2 : The arity (2) of `mult`.
1 1 : There is only one symbol (1) that can occur at the first argument 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`.
2 : The number of unique subterms that has `s` as its top symbol.
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`.
1 : The number of unique subterms that has `z` as its top symbol.
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 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.

References

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