Chapter 1. ATerm SAF: A High Performance Streamable Format

Arnold Arnold Lankamp

2008-07-16 10:00:31 +0200 (Wed, 16 Jul 2008)

Table of Contents

Introduction

This document contains the technical documentation of the (Semi-) Streamable ATerm Format, otherwise known as SAF. This document is mainly intended for the developers and users of the ATerm library. If you do not understand what ATerms are you'd better stop reading now ;).

Goals

SAF is intended to be a high performance format, capable of exchanging ATerms in an fast, efficient and platform independent way. The main reason for its development emerged from the wish to transmit terms across network connections in a 'streamable' way with the option to suspend this process at any point in time to enable multiplexing, while keeping (de-)serialization memory usage constant. The currently available ATerm formats (BAF, TAF and the ASCII ATerm format) do not supply any of these functionalities.

Requirements

The most important requirements for this format and its implementation(s) are:

Format requirements

Portabilty

The format is intended to be fully portable, so it should not contain any platform of language specific elements.

Streamability

Meaning the ability to send a term in blocks of a by the user specified size, so the (de-)serialization process can be suspended at regular intervals, if required.

Compactness

Network connection speed is generally the bottleneck when sending data, thus we want the serial representation to be as compact as possible to conserve bandwidth.

Implementation requirements

Transformation / parsing speed

We want to be able to read and write terms to and from SAF as fast as possible. Current network connections have a throughput of tens or even hundreds of Megabits per second; we want to be able to utilize those to their full effect and do not want the transformation / parsing speed to limit the maximal throughput.

Low memory usage

Memory usage during the (de-)serialization process should be as low as possible and predictable; this is inline of what is expected of an implementation for use in a high performance / soft real-time environment. For implementations in a language that use a garbage collector it is highly recommended that the amount of temporarily allocated object remains as limited as possible and no 'mid-lived' objects are created; this way there are no (or just very minor) performance penalties for applications that use the implementation.

No recursive calls

We do not want to be limited by the size of the stack; the implementation should in no way impair the maximal depth of a tree.

Conclusion

The above requirements are somewhat conflicting. The streamability and portability requirements limit the sorts of compression techniques that can be used. Additionally compression, generally speaking, incurs computational overhead, so there is a trade-off between compression and transformation / parsing speed as well. Low memory usage and performance do not always go hand-in-hand either. Although all of these requirements are important, transformation / parsing speed should be favored as long as it does not cancel any of the other requirements out entirely.

Representation

The serial representation of the format is fairly simple. Every term has a header containing general information about the term. After this header the serial representation of the term itself is present; the way this serial representation is layed out is type specific.

Serialization order

The terms and annotations will be serialized in the order in which they are present in the tree (prefix order). Which is: |term|children|annotations|.

So if we have a term with two children of which the first child has three children, the order will look like this: |term|child1|child1.1|child1.2|child1.3|child2|.

This is similar to the structure of the ASCII and TAF ATerm formats. A more extensive example will be presented later on in this document.

Term header

The header contains general information about a term and can optionally contain type specific data in the two free fields.

This is what the fields in the header represent:

Bit number	1	2 + 3	4	5 + 6 + 7 + 8
Bit mask	0x80	0x40 + 0x20	0x10	0x0f
Meaning	IsShared?	Free / Type specific	HasAnnos?	Type field

The reason the type field is on the right side of the byte is for performance reasons; this way we do not have to perform any shifts before adding it to the header. There are no specific reasons for the locations of the other fields, since they are only one bit flags, it does not matter were they are located.

IsShared?

This is a boolean value that indicates if this term is shared. We will explain how this sharing works later on. Note that if this bit is set all the other data in the header is not required and can be safely ignored if present.

Bit 2 + 3

Bit 2 and 3 are free and may be used for type specific data.

HasAnnos?

This is a boolean value that indicates if the term has annotations or not.

Type

The type field contains a four bit value that represents the type id of the term. Note that bit 5 (0x08) isn't being used at the moment, since we only have seven different term types; this leaves plenty of room for extension.

Encoding of types

Every term type has a different encoding.

These are the binary representations of the content of the different term types:

ATermInt

Field	Header	Value
Size (bytes)	1	1 to 5

For more information on the encoding of integers see the compression chapter.

ATermReal

Field	Header	Value
Size (bytes)	1	8

Reals are encoded as 64 bit IEEE 754 floating point numbers.

Note that we always use 8 bytes to encode a real. This is because IEEE 754 encoded floating point numbers always occupy a couple of bits in the highest order byte, restricting us from using the same trick as with the encoding of integers. They are written in two blocks of 4 bytes, in little endian order.

ATermBlob

Field	Header	Length	Data
Size (bytes)	1	1 to 5	0 to 2^32-1 (depends on length)

ATermAppl + AFun

Field	Header	Arity	Name length	Name bytes
Size (bytes)	1	1 to 5	1 to 5	0 to 2^32-1 (depends on name length)

An ATermAppl always has a function symbol associated with it. For that reason we decided to combine them.

ATermAppl + AFun header

Bit 2 (0x40) represents IsFunShared? and bit 3 (0x20) IsQuoted?.

- IsFunShared? is a boolean value that indicates if the function symbol of this appl is shared. If this flag is set the isQuoted? flag is not required and can be safely ignored if present. We will discuss sharing [6] in more detail in the compression chapter.
- IsQuoted? is a boolean value that indicates if the function symbol associated with this ATermAppl is quoted or not.

ATermList

Field	Header	Size
Size (bytes)	1	1 to 5

ATermPlaceholder

Field	Header
Size (bytes)	1

Encoding of shared elements

A shared ATerm

Field	Header (with isShared? Flag set)	Term identifier
Size (bytes)	1	1 to 5

A shared AFun

Field	Header (with isFunShared? Flag set)	Function symbol identifier
Size (bytes)	1	1 to 5

Reading and writing

SAF is a (semi-)streamable format, so reading and writing it goes a little different then usual. It works in a block-wise way. A SAF writer can be requested for the following X bytes of the serial representation of a term, which can contain partially serialized elements. When reading SAF you will need information about how large those blocks were to be able to reconstruct the term. For this purpose we propose to emit a two byte unsigned integer value before every block, which specifies its size. This is, for example, the case for SAF file I/O; thus you will always need a buffer of 2^16 (65536) bytes when reading from a 'standard' SAF file. (Note that a 0 block length value indicates a block of 65536 bytes, since 0 length blocks can't exist). Custom ‘shielded’ I/O implementations for SAF are allowed to use their own values and / or protocol (by shielded we mean implementations that are not intended to interface with any implementations other then themselfs, since compatibility can not be guaranteed in these cases).

This block-wise writing and reading method enables us to suspend the (de-)serialization process at fixed intervals, without the need to assign a different thread to each process. This enables us to interleave the simultaineous construction of multiple trees of terms in a single threaded environment.

Splitting elements

To reduce the complexity of implementations we decided that only function symbols and BLOBs should be (de-)serializable in pieces. These are currently the only two types of terms who's serial representation can occupy more then nine bytes in this format and consequently are the only types for which it is interesting to split them. All other types of terms are undividable and must occur sequentially in the same block. For this reason a write buffer must be at least nine bytes in size (although using such a small buffer is strongly discouraged, because of the relatively large overhead this would yield in both time and space). All SAF writer implementations have to adhere to this rule to guarantee the generation of a stream that is compatible with all SAF reader implementations.

Compression

As noted before compression, generally speaking, incurs computational overhead. In this particular case computational overhead is something we want to avoid or at least restrict as much as possible.

Also the streamability and portability requirements limit our options in terms of compression techniques. For this reason we decided to stick the to sharing of 'elements'. With elements we mean terms / sub trees and function symbols. We can achieve fairly good compression rates with this, because it is a type of compression that is specifically meant for ATerms; we know what the data and composition look like and can use that knowledge to our advantage.

We use a LZW like compression technique to handle the sharing. What we do is, every time we encounter an element we have not seen before we add it to a table and assign it the next 'identifier' (which is an unsigned integer; the first identifier is 0, which represents the root of the tree). If we encounter an element that is already present in the table, we set the with the element’s type corresponding 'shared?' flag in the header and emit the associated identifier. During the deserialization process we do the exact opposite, every unique element that is encountered is added to an array in the order in which we find them in the SAF stream; when we run into a shared element, we read the associated id and replace it by the value that is present at that index in the array. We use separate tables and arrays for both shared terms / sub trees and shared function symbols.

Integer encoding / compression

We also make use of the fact that small unsigned integers are most common. We are saving some space by only using the minimal amount of bytes to represent an integer value. The last bit of every byte is used as a flag to indicate if there are more bytes coming (1) or not (0). In most cases this means we only need one or two bytes to represent an integer value. On the other hand, to represent large and negative integer values we need five instead of the regular four bytes, since we 'lose' one bit per byte. However we expect those cases to be fairly rare. Additionally, all the identifiers that are used for sharing are small unsigned integers, which occupy a large part of the serial representation of any term, especially in those with heavy sharing; this was the deciding factor for using this type of integer encoding. The encoding of the value of the integers is two's complement, because this is the standard on most (if not all) of todays personal computers. If the underlaying integer representation of the system you are writing an implementation for is different, keep in mind that you will need to encode them as two's complement yourself. The byte order is little-endian.

Integer encoding examples

Here are some examples of what certain integers would look like in the above described encoding:

Value	*Encoded representation*
0	**0**0000000
1	**0**0000001
100	**0**1100100
128	**1**0000000 **0**0000001
1000	**1**1101000 **0**0000111
1000000	**1**1000000 **1**0000100 **0**0111101
2000000000	**1**0000000 **1**0101000 **1**1010110 **1**0111001 **0**0000111
-256	**1**0000000 **1**1111110 **1**1111111 **1**1111111 **0**0001111

Compression rates

Here is an overview of the amount of compression that is achieved by the different formats:

SDF syntax (a relatively large	*ASCII*	*TAF*	*BAF*	*SAF*	*GZIP*

term with lots of sharing)					
Size (bytes)	3387103	73082	35308	45097	65279
Compression (%)	0	97.842	98.958	98.669	98.073

Pico syntax (a medium size term)	*ASCII*	*TAF*	*BAF*	*SAF*	*GZIP*
Size (bytes)	61488	28131	13653	15903	6351
Compression (%)	0	54.25	77.796	74.136	89.671

a(1) (a very small term, illustrating worst case overhead)	*ASCII*	*TAF*	*BAF*	*SAF*	*GZIP*
Size (bytes)	4	5	28	6	31
Compression (%)	0	-25	-600	-50	-775

As you can see the compression rates SAF achieves are fairly close to those of BAF, which was designed with compression as its main goal. A comparison with GZIP is a bit harder, since it uses an entirely different algorithm. Whether or not it achieves better compression rates depends on the amount of sharing in the tree. Percentage wise larger terms will have more sharing then smaller terms. The results above illustrate this behavior.

Performance

The current SAF (de-)serialization implementation in both C and Java is multiple times faster then that of any of the other ATerm formats (BAF, TAF and the ASCII ATerm format).

Here are some benchmarks that illustrate the performance difference between the current C and Java implementations of the different formats:

(The benchmarks were performed on a AMD 64 3500+ with 1 GB DDR-400 dual-channel RAM. It shows the 'best of five runs' execution time, measured inside the code (user time spend). Keep in mind that these measurements are subject to change and are merely an indication).

SDF syntax

This is a relatively large term with lots of sharing.

	C ASCII	*C TAF*	*C BAF*	*C SAF*	*Java ASCII*	*Java TAF*	*Java SAF*
Serialization x10000 (ms)	1744500	38376	100749	23677	2810000	91700	65300
Deserialization x10000 (ms)	2623500	91494	52544	22777	8150000	166500	83300

Note: in this specific benchmark the ASCII ATerm format measurements are extrapolated from a run with a hundred iterations, otherwise the test would take too long.

Pico syntax

This is a medium size term.

	C ASCII	*C TAF*	*C BAF*	*C SAF*	*Java ASCII*	*Java TAF*	*Java SAF*
Serialization x10000 (ms)	49756	16742	34024	6660	49100	31300	18830
Deserialization x10000 (ms)	46269	34234	17114	5661	144600	58700	26200

a(1)

This is a very small term. This test illustrates the worst case overhead for (de-)serializing a term.

	C ASCII	*C TAF*	*C BAF*	*C SAF*	*Java ASCII*	*Java TAF*	*Java SAF*
Serialization x1000000 (ms)	697	5468	87218	2050	18850	18630	2000
Deserialization x1000000 (ms)	539	5786	1790	2030	21530	27300	5500

Conclusion

In every benchmark the SAF (de-)serialization implementation clearly has the upper hand by a very large margin. Only the C ASCII implementation performs better then the C SAF implementation in the overhead test, since it does not have to allocate any memory. However the overhead of the SAF implementation in both C and Java is still relatively low compared to the other implementations.

Memory usage

Memory usage of the current SAF (de-)serialization implementations scale linearly with the amount of unique elements in the tree. This is because a reference to every unique element in the tree is stored in a hashtable or array during both the serialization as the deserialization process. Also the depth of the tree influences the memory usage, since that determains the size of the stack that keeps track of the parent to child relations of the terms in the tree; the size of this stack scales linearly with the depth of the tree.

The worst case memory usage can be calculated in the following way:

Serialization memory usage: (the number of unique terms in the tree * 4 * 4) + (the number of unique function symbols in the tree * 4 * 4) + (the depth of the tree * 4 * 4).

Deserialization memory usage: (the number of unique terms in the tree * 2 * 4) + (the number of unique function symbols in the tree * 4) + (the depth of the tree * 8 * 4).

Calculating the exact amount of memory usage is possible, but largely depends on the layout of the tree. The above calculations serve as a guideline to indicate the maximal memory usage for the (de-)serialization of a certain term.

Example

The following term:

line(box(rect(2), rect(5), square(4, 3)), circle(10), circle(10))

Will look like this in the binary format:

0x01 0x03 0x04 line 0x01 0x03 0x03 box 0x01 0x01 0x04 rect 0x02 0x02 0x41 0x03 0x02 0x05 0x01 0x02 0x06 square 0x02 0x04 0x02 0x03 0x01 0x01 0x06 circle 0x02 0x0a 0x80 0x06

Which in bits looks like this (the indent and lines were added to show the child-parent relationship):

```
|00000001  appl
|00000011  arity = 3
|00000100  fun-length = 4
|01101100  fun-bytes = line
|01101001
|01101110
|01100101
|---|
    |00000001  appl
    |00000011  arity = 3
    |00000011  fun-length = 3
    |01100010  fun-bytes = box
    |01101111
    |01111000
    |---|
    |   |00000001  appl
    |   |00000001  arity = 1
    |   |00000100  fun-length = 4
    |   |01110010  fun-bytes = rect
    |   |01100101
    |   |01100011
    |   |01110100
    |   |---|
    |   |   |00000010  int
    |   |   |00000010  value = 2
    |   |
    |   |01000001  appl with shared function symbol
    |   |00000011  shared function symbol identifier = 3
    |   |---|
    |   |   |00000010  int
    |   |   |00000101  value = 5
    |   |
    |   |00000001  appl
    |   |00000010  arity = 2
    |   |00000110  fun-length = 6
    |   |01110011  fun-bytes = square
    |   |01110001
    |   |01110101
    |   |01100001
    |   |01110010
    |   |01100101
    |   |---|
    |       |00000010  int
    |       |00000100  value = 4
    |       |
    |       |00000010  int
    |       |00000011  value = 3
    |
    |00000001  appl
    |00000001  arity = 1
```

```
    |00000110  fun-length = 6
    |01100011  fun-bytes = circle
    |01101001
    |01110010
    |01100011
    |01101100
    |01100101
    |---|
    |   |00000010  int
    |   |00001010  value = 10
    |
    |10000000  shared term
    |00000110  shared term identifier = 6
```