mCRL2 Term Library

Maurice Laveaux

April 26, 2024

Abstract

This document contains a description of the term library as implemented in the mCRL2 toolset. Its contents are listed below.

Contents

1 Preliminaries

Let $\mathcal{F} = \biguplus_{i \in \mathbb{N}} \mathcal{F}_i$ be a *ranked* alphabet of *function symbols*. A function symbol $f \in \mathcal{F}_i$ is said to have a rank or *arity*, given by $\alpha(f)$, equal to i. The set of terms $\mathcal{T}$ is the smallest set such that for $t_0, \ldots, t_n \in \mathcal{T}$ and $f \in \mathcal{F}_n$ it holds that $f(t_0, \ldots, t_n) \in \mathcal{T}$, as such $\mathcal{F}_0 \subseteq \mathcal{T}$. For each term $f(t_0, \ldots, t_n)$ we say that f is its *head*, denoted by $\mathsf{head}(f(t_0, \ldots, t_n))$, and we define a function $\mathsf{args}(f(t_0, \ldots, t_n))$ to be equal to a sequence of terms $[t_0, \ldots, t_n]$.

There are several useful classes of terms that can be identified. One example is the class of *list* terms. Let $+\!\!+ \in \mathcal{F}_2$ and $[\,] \in \mathcal{F}_0$ be function symbols to define list concatenation and the empty list. The set of *list* terms $\mathcal{L} \subseteq \mathcal{T}$ is the smallest set such that $[\,] \in \mathcal{L}$ and for terms $t \in \mathcal{T}$ and $l \in \mathcal{L}$ the *concatenation* $+\!\!+(t, l) \in \mathcal{L}$. There are also other classes, such as *term string* where a string is stored as part of the function symbol name and *binary tree* where every left and right sub-tree is stored as arguments. These can all be represented in our term library.

2 Term Library

The term library provides the storage (in memory and on disk) of terms and function symbols. Formally, the term library stores a finite subset of terms $\mathcal{T}' \subseteq \mathcal{T}$. The library has been designed with the following goals in mind:

1. Minimize the amount of memory used to store the set of terms.
2. Fast creation (and deletion) of terms.
3. Fast comparison, in particular equality, between terms.

2.1 Application Programming Interface

The library is implemented in the `C++`11 compliant subset of the `C++` language. We assume some familiarity with the concepts of `C++`, for now the distinction between classes and objects is important and the role of constructors and member functions. A constructor is a language construct that instantiates an object of a specific class, and member functions operate on that instance.

The application programming interface (API) consists of several constructors that allow the user to create function symbols and terms. Function symbols are constructed from a sequence of characters to represent their name and a natural number for its arity. The relation between the mathematical elements and these constructors are listed in Figure 1.

Definition	API
$\mathtt{name} \in \mathcal{F}_n$	function_symbol($\mathtt{name}, n$)
$f(t_0, \ldots, t_n) \in \mathcal{T}$	term_appl($f, t_0, \ldots, t_n$)

Figure 1: The relation between the definition and the API

For the already identified classes of terms there are specific constructors defined as shown in Figure 2.

Definition	API
$[\,] \in \mathcal{F}_0$	`function_symbol(<empty_list>,0)`
$+\!\!+ \in \mathcal{F}_2$	`function_symbol(<list_constructor>,2)`
$+\!\!+(t, l) \in \mathcal{L}$	`term_list(t, l)`

Figure 2: The relation between the list class and the API

There are also additional member functions to check equality (or inequality) between terms. Note that there is not really a semantic inequality between terms defined, but from an implementation viewpoint this is useful to have.

There are also several member functions to access the information carried by a term. There is a `function` function maps a term to its head function symbol. The `arg` function takes a term and an index as input and returns the argument term at the specified index.

Finally, we also provide the ability to write a term to *stream* and subsequently retrieve the term from this stream. In particular, the ability to write multiple individual terms to a stream is provided. For this we provide the function $\mathsf{write}(s, t)$ to write term t to a stream s, which is a sequence of bits or characters, and symmetrically read a term from a stream by means of $\mathsf{read}(s)$, which returns this term.

2.2 Architecture

The main architectural choice for this library is that terms and function symbols are maximally shared, by a technique called *hash conscing.* Here, every function symbol and term has a unique identifier and these identifiers are used by a term as references to its head function symbol and arguments. This sharing facilitates the goal to minimize memory usage by the stored terms and also allows for constant time comparison, because the identifiers can be compared in constant time. The inequality between terms is determined by the natural order of these identifiers, which are natural numbers.

This functionality is implemented by an underlying function symbol *pool* class that stores the finite subset of function symbols $\mathcal{F}'$ and a term pool to store the finite subset of terms $\mathcal{T}'$. To facilitate ease of programming on a high-level this sharing is completely transparent to the programmer, which instantiates an object that internally stores an identifier to look up the information about the underlying function symbol (or term).

For the purpose of deleting shared elements there is a `shared_reference` class that internally counts the number of references to each term or function symbol. Here, a reference means that another term has that identifier as head symbol or argument. The `term` and `function_symbol` classes internally use this class to keep track of the number of references.

3 Function Symbol Pool

Now, we are going to dive into specific implementation details and need to be a little more concrete. An important concept of `C++` are its (simplified) memory model, where there is a *heap* that maps positions, also called *pointers* or *references*, to an array of bits, or in general data are now important. We use a (*name*, *arity*) pair to indicate the information stored on the heap about the function symbol. Where two function symbols with the same name and arity are equivalent. The function symbol pool has a member function named `create` to obtain the identifier to the function symbol with the given name and arity. This identifier is the reference to the data stored on the heap.

- $\mathsf{create}(\text{name} : \text{String}, \text{arity} : \text{Nat}) : \mathbb{N}$

The implementation of the createfunction is given Algorithm 1. The set of identifiers $\mathcal{F}'$, and later $\mathcal{T}'$, is a set with polymorphic lookup where the identifier can be found using the (*name*, *arity*). We introduce the notation $\mathcal{F}'[(name, arity)]$ to indicate the partial function mapping the arguments to the identifier, which yields $\perp$ when the element is not in the set. Note that although $\mathcal{F}'$ is a set of identifiers on to elements on the heap it also directly stores the finite set of function symbols. The allocate function reserves space on the heap and stores the *name* and *arity* pair in the reserved space. In the actual implementation this is a separate constructor for a pair-like object.

Algorithm 1 Creation of function symbols

```
1: procedure CREATE($name$, $arity$)
2:     if $\mathcal{F}'[(name, arity)] \neq \perp$ then
3:         return $\mathcal{F}'[(name, arity)]$
4:     end if
5:     $f := \mathsf{allocate}(name, arity)$
6:     $\mathcal{F}' := \mathcal{F}' \cup \{f\}$
7:     return $\mathsf{ref}(f)$
8: end procedure
```

The resulting identifier is then used by function_symbol to look up the underlying information about the function symbol.

To clean up, *i.e.*, removing the underlying data from the heap, objects with a reference count of zero there are two different approaches. The first method immediately cleans up the function symbol from $\mathcal{F}'$ and the heap whenever its reference count becomes zero in its destructor. This approach will be referred to *direct-destruction*. Another approach is to periodically remove all function symbols with a reference count of zero. This approach will be referred to as *garbage collection*.

For function symbols the *direct-destruction* method is used as the number of function symbols that is destroyed is typically very low. This destructor that is called when f goes out of scope is implemented by the `destroy` function defined below.

Algorithm 2 Destruction of function symbols

```
1: procedure DESTROY($f$)
2:     $deallocate(f)$
3:     $\mathcal{F}' := \mathcal{F}' \setminus \{f\}$
4: end procedure
```

The *deallocate* function frees the identifier to be used again by a different function symbol.

4 Term pool

The term pool is the class that stores the set of terms $\mathcal{T}'$ and is very similar to the function symbol pool. The *create_appl* can be used to create new terms, we refer to terms with arguments as *function applications*.

The constructor of a function application calls the following function to obtain a reference to an existing or new term.

- create_appl(f : $\mathcal{F}$, $t_0, \ldots, t_n : \mathcal{T}$) : $\mathbb{N}$

The implementation of this function can be seen in Algorithm 3. The function symbol f and terms t_0 to t_n should be elements of the current subset of stored function symbols $\mathcal{F}'$ and terms $\mathcal{T}'$ respectively, which means that they are identifiers. Note that on the heap a tuple $(f, c, t_0, \ldots, t_n)$ is stored that encodes the actual term with $c \in \mathbb{N}$ be a reference counter.

For terms we use the garbage collection approach, where garbage collection is triggered based on some heuristics. The advantage of garbage collection is that terms with a reference count of zero can be reused when they are recreated before being destroyed. This means that only its reference counter must be increased. In practice, this occurs quite often as the volume of terms being created and destroyed during term rewriting is quite large.

Algorithm 3 Creation of term applications

```
1: procedure CREATE_APPL($f, t_0, \ldots, t_n$)
2:     if $\mathcal{T}'[f, t_0, \ldots, t_n] \neq \bot$ then
3:         return $\mathcal{T}'[f, t_0, \ldots, t_n]$
4:     end if
5:     $t :=$ construct($f, t_0, \ldots, t_n$)
6:     if should-collect-garbage() then
7:         collect()
8:     end if
9:     $\mathcal{T}' := \mathcal{T}' \cup \{t\}$
10:    return $t$
11: end procedure
```

The boolean function *should-garbage-collect* is used to decide when garbage collection should be performed. The garbage collection itself is implemented by the *collect* function. Currently the *should-*

garbage-collect function is implemented by using a counter that is decremented on every call to *should-garbage-collect*. Whenever this counter reaches zero the function returns true. During garbage collection this counter is then set to the number of terms in $\mathcal{T}'$, which is equal to $|\mathcal{T}'|$.

4.1 Garbage Collection

The garbage collection is implemented by calling destroy on every term with a reference count of zero, as shown in the following pseudocode:

Algorithm 4 Garbage collection of terms

```
1: procedure Collect
2:     for $t \in \mathcal{T}'$ do
3:         if reference-count($t$) = 0 then
4:             Destroy(t)
5:         end if
6:     end for
7:     collect-countdown := $|\mathcal{T}'|$
8: end procedure
```

The *destroy* method recursively destroys all arguments which have a single reference, because that reference is the current function application.

Algorithm 5 Destroying individual terms

```
1: procedure Destroy($t \in \mathcal{T}'$)
2:     for $p \in$ args($t$) do
3:         if reference-count($p$) = 1 then
4:             Destroy($p$)
5:         end if
6:     end for
7:     $\mathcal{T}' := \mathcal{T}' \setminus \{t\}$
8:     deallocate($t$)
9: end procedure
```

The *deallocate* function frees the heap memory used by this term. Note that this function prematurely destroys arguments with a single reference count. The reason for this is that the function `arg` is no longer defined for t after it has been deallocated.

5 Binary Input/Output

We provide two different streaming procedures to share terms between tools of the toolset. The first method uses the concrete syntax and operators on streams of characters. A function symbol $f(t_0, \ldots, t_n)$ is simply written to the character stream as the name of f written as "f", followed by recursively printing all its arguments t_0 to t_n separated by commas and a closing bracket at end. For convenience of reading, list terms are printed as a sequence of their elements surrounded by square brackets. The reading procedure simply parses the characters of the input stream and reconstructs the corresponding terms.

Although this is useful for debugging it is not a very efficient exchange format as sharing of function symbols of terms cannot be employed and characters themselves are very verbose. Therefore, we have developed the streamable binary (a)term format (SBAF). A single term can be written to and read from a *binary* streams in SBAF one at the time and terms in the same stream share their arguments. We focus on writing terms first and then show how the resulting stream can be used to reconstruct the written terms during reading.

Similar to the in-memory storage, a (different) unique identifier is assigned to each term and function symbol that is used to identify arguments during writing. The output is going to consist of *packets* that either represent a function symbol, a term or a subterm indicated by two bits in the header of a packet.

We use one, two and three respectively to represent these headers. The difference between a term and a subterm is that a term is returned from $\mathsf{write}(s)$, where s is a stream, and the subterms are (as their name suggest) subterms of the retrieved term.

The unique identifier for each written element is stored by a mapping from function symbol and terms to a natural number. This is done in a data structure called an *indexed set* that increments a consecutive index whenever an element is inserted and assigns it to the new element. Note that we are never going to remove elements from this set so the indices are always compact.

Binary streams are not directly facilitated by the `C++` library so we have implemented a `bitstream` class that is able to write (and read) individual bits from and to a stream. It also provides several convenience functions to write character sequences and integers to this stream, where integers are stored using a *variable width* encoding, where we use the overloads $\mathsf{write}(s, \text{“string”})$ and $\mathsf{write}(s, n)$, with $n \in \mathbb{N}$. Furthermore, we use the notation n_i for some natural number n to indicate that only i bits of the binary encoding of this number are considered when writing it, instead of relying on the variable width encoding.

Let I be an indexed set of function symbols. Algorithm 6 shows how an individual function symbol is written to a stream s. First, we check whether the function symbol was already written, which means that it has an index n. Otherwise, there are only three distinct packets so only two bits are needed to encode it. After which the name and arity of the function symbol are written and a new index n is created for this function symbol.

Algorithm 6 Writing a function symbol to an output stream s

```
1:  procedure WRITE-FUNCTION-SYMBOL($s, f \in \mathcal{F}'$)
2:      if $(f, n) \in I$ then
3:          return $n$
4:      else
5:          $\mathsf{write}(s, 0_2)$
6:          $\mathsf{write}(s, \mathit{name}(f))$
7:          $\mathsf{write}(s, \mathit{arity}(f))$
8:          $I := I \cup \{f\}$
9:          return $n$ such that $(f, n) \in I$
10:     end if
11: end procedure
```

Let J be an indexed set of terms. Algorithm 7 shows the implementation to write a term to the stream s. We write terms such that the arguments (and head symbol) of a term are written to the stream before the term itself by traversing the term bottom up. This can achieved by maintaining stack of terms where each term has a value $\{\top, \bot\}$ associated to it to indicate whether its arguments have already been processed. The bottom up traversal is implemented by first changing the flag for t' from $\bot$ to $\top$ and then inserting all arguments that are not already in I (as these have already been written) to the stack. We must also maintain that the term taken from the stack has not been inserted to I in the meantime, which can happen whenever the same subterm occurs multiple times.

The case where the term is actually written to the stream, indicated by $b = \top$, starts by writing obtaining the function symbol index and possibly writing a function symbol packet to the stream. Then, the packet identifier (again using 2 bits) is written, which is an output term whenever the current term is equal to t and a subterm otherwise. Then the indices of the function symbol and each argument (which should already be included in I) are written to the stream. Note that we only need $\mathsf{ceil}(\log(|I|) + 1)$ bits to uniquely identify previously written subterms and function symbols. During reading we also know the number of terms that have been written and as such the number of bits to read can be inferred. Finally, the written term is stored in the indexed set I.

Table 1: Shows the steps needed to write mult(s(s(z)), s(z)) to a stream, indicating the output written to the stream, the number of bits needed and the values of I and J.

Term	Function Symbol	Output	Size (b)	I	J
	z	0_2 1_8 "z" 0_8	26	$\{(z,0)\}$	
z		1_2 0_1	3	$\{(z,0)\}$	$\{(z,0)\}$
	s	0_2 1_8 "s" 1_8	26	$\{(z,0),(s,1)\}$	$\{(z,0)\}$
s(z)		1_2 1_1 0_1	4	$\{(z,0),(s,1)\}$	$\{(z,0),(s(z),1)\}$
s(s(z))		1_2 1_1 1_1	4	$\{(z,0),(s,1)\}$	$\{(z,0),(s(z),1),(s(s(z)),2)\}$
	mult	0_2 4_8 "mult" 2_8	50	$\{(z,0),(s,1),(mult,2)\}$	$\{(z,0),(s(z),1),(s(s(z)),2)\}$
mult(s(s(z)), s(z))		2_2 2_2 2_2 1_2	8	-	-

Algorithm 7 Writing a term to an output stream s

```
1: procedure WRITE($s \in \mathbb{B}^*, t \in \mathcal{T}'$)
2:     $J := \emptyset$
3:     $Q := \{(s, \bot)\}$
4:     while $\neg$empty($Q$) do
5:         $(t', b) :=$ pop($Q$)
6:         if $\neg\exists n \in \mathbb{N} : (t', n) \in J$ then
7:             if $b = \top$ then
8:                 $n :=$ write-function-symbol(head($t'$))
9:                 if $t' = t$ then
10:                    write($s, 1_2$)
11:                else
12:                    write($s, 2_2$)
13:                end if
14:                write($s, n_{\mathsf{ceil}(\log(|I|)+1))}$)
15:                for $p \in$ args($t'$) do
16:                    $i$ such that $(i, i) \in I$
17:                    write($s, i_{\mathsf{ceil}(\log(|J|)+1))}$)
18:                end for
19:                $I := I \cup \{t'\}$
20:            else
21:                push($Q, (t', \top)$)
22:                for $p \in$ args($t'$) do
23:                    if $\neg\exists n \in \mathbb{N} : (p, n) \in J$ then
24:                        push($Q, (p, \bot)$)
25:                    end if
26:                end for
27:            end if
28:        end if
29:    end while
30:    return $s$
31: end procedure
```

We present a small example to illustrate this algorithm.

Example 5.1. Let us consider writing the term mult(s(s(z)), s(z)) to a stream, which is the example presented in [1]. Again, we use the subscript to indicate the number of bits of that number written to the stream. To write a string we first write its length, followed by n characters using 8 bits per character. For numbers smaller than 128 we need eight bits in our variable-width encoding. The resulting writes and the values for I and J are presented in the following table. Table 1 shows the steps needed to write mult(s(s(z)), s(z)) to a stream, indicating the values written to the stream, the number of bits needed and the values of I and J.
So writing this term requires 121 bits in total. Typically the ratio of terms to function symbols is quite high, so the small number of bits needed to encode the terms is the most important.

There are two separate indexed sets for function symbols and terms such that the resulting indices are kept smaller. Furthermore, in practice we also allow a function over terms to be used during writing (and reading) that is applied to each subterm before the term is written to a stream. This can be useful to remove some implementation details from the terms that should not shared between tools.

The symmetric reading function is implemented in Algorithm 8. Upon initialization of the stream I and J are set to $\emptyset$ and these indexed sets should be maintained in between calls to reading terms. Reading from a stream has additional auxiliary functions $\mathsf{read}_{bits}(s, n)$, $\mathsf{read}_{\mathbb{N}}(s)$ and read_{string} to reads n bits, numbers and characters from the stream s respectively. Again, we use an indexed set I. However, in the implementation we can use an array to efficiently map indices to elements in this set as the indices grow strictly increasing and the mapping is injective.

Algorithm 8 Writing a term to an output stream s

```
procedure READ($s \in \mathbb{B}^*, t \in \mathcal{T}', I, J$)
    $p := \mathsf{read}_{bits}(s, 2)$
    if $p = 0$ then
        $name := \mathsf{read}_{string}(s)$
        $arity := \mathsf{read}_{\mathbb{N}}(s)$
        $I := I \cup \{(name, arity)\}$
    else
        $f$ such that $(f, \mathsf{read}_{bits}(s, \mathsf{ceil}(\log(|J|) + 1))) \in I$
        $Q := [t_0, \ldots, t_{arity(f)}]$
        for $i \in Q$ do
            $t_i$ assigned such that $(t_i, \mathsf{read}_{bits}(s, \mathsf{ceil}(\log(|J|) + 1))) \in I$
        end for
        $t := f(t_0, \ldots, t_{arity(f)})$
        if $p = 2$ then
            return $t$
        else
            $J := J \cup \{t\}$
        end if
    end if
    return $s$
end procedure
```

In the implementation a special function symbol exists to mark the end of the stream.

Previously an alternative format, called binary aterm format, described in [1] was used to write terms into streams. This previous format allowed better compression by counting the occurrences of all terms at all argument positions beforehand and minimizing the amount of bits needed to encode this information. Although this method allowed presumably optimal compression it could only write a single term (with sharing) to the stream and required a lot of book keeping while traversing the term twice. However, for labelled transition systems, which are typical output terms, the bulk of information is the list of transitions. To encode this list of transitions as a single term the previously mentioned class of list terms was used. This meant that first this enormous term had to be constructed, then traversed depth-first twice (incurring quite some memory overhead), followed by writing this term to the stream, where all transitions could occur at the first position of the list concatenation term.

The new method avoids the construction of this list term as transitions can be written to the stream individually. Furthermore, the width of indices assigned to transition terms simply scale logarithmically in the number of transitions, which yielded a better (both in space and time) term format. Writing the example term mult(s(s(z)), s(z)) in the old format required a table of 140 bits and then 5 bits to write the term itself. Even though the new format uses more bits to write this term it is much more straightforward to process and in practice it performs (much) better.

6 Data Structures

In this section the data structures in the underlying implementation are described in more detail.

6.1 Reference Counting

The shared reference is implemented by a class named `shared_reference` which consists of a reference and a reference counter. The invariant states that the reference counter is always equal to the number of instances that point to the same referred term. In `C++` there are a number of operators to construct, move and copy classes. A move is an operator where an object is constructed from another object, but the other object can be left in an undefined state. The reference count invariant is satisfied by implementing these operations in the following way:

1. When a `shared_reference` instance is constructed from a reference its reference count is incremented by one.
2. When a `shared_reference` instance is copy-constructed its reference count is incremented by one.
3. When a `shared_reference` instance is move-constructed the reference count is kept the same, but the`shared_reference` instance that was moved from will become a null reference.
4. When a `shared_reference` instance is assigned to its current reference count is decremented by one. The reference count of the assigned reference is incremented by one.
5. When a `shared_reference` instance is move-assigned its current reference count is decremented by one. The `shared_reference` instance that was moved from will become a null reference.
6. When a `shared_reference` is destructed the reference count will be decremented by one.

For terms and function symbols all shared references are constructed with a default term or function symbol respectively. Whenever the reference count for a reference is equal to zero the referred to term can be cleaned up, except for the default that always maintained a reference count of at least one. In the case of function symbols this immediately triggers the destroy function.

6.2 Hash table

The references to terms and functions symbols are both stored in a set which provides constant time insertion $\cup$, deletion $\setminus$ and contains $\in$ functions. These sets store pointers to the elements allocated on the heap. This can be efficiently implemented by using a hash table. As typical for a hash table it requires a hash function and an equivalence check for its elements to implements it operations.

Our hash function, which is an operator to natural numbers, for terms is determined by a suitable combination of the underlying references as shown in Algorithm 9. For terms the hash function is defined by combining the values of all the references. For performance reasons it is furthermore desirable to be able to compute the hash function and equivalence check without (temporarily) instantiate elements of that type.

Algorithm 9 Hashing terms

```
1: procedure HASH($f, t_0, \ldots, t_n$)
2:     $hnr :=$ HASH($f$)
3:     for $t \in \{t_0, \ldots, t_n\}$ do
4:         $hnr :=$ COMBINE($hnr, t$)
5:     end for
6:     return $hnr$
7: end procedure
```

In practice it is quite subtle what a good (and fast) COMBINE function is that ensures a fairly random distribution of hash values. For function symbols the hash is obtained from hashing the name string and its arity and returning its combination. Note that f is a reference to a function symbol and as such it does not require this expensive computation, but the hash is just derived from the given reference (or pointer). It is straightforward to also derive functions that can be applied to terms and function symbols directly, which is required for rehashing. The equivalence between terms can be determined by comparing the references their head symbols and pairwise comparing the references for its arguments. For function symbols this is done by checking name and arity equivalence.

As the equivalence checks for terms is quite it is essential that the number of equivalence checks upon searching is minimized. For this purpose we have chosen for an open hash table with single linked-list buckets as the probing used by closed hash table resulted in a performance reduction of ten to fifteen percent. On the other hand, closed hash table have a reduced memory footprint so there is trade off.

7 Optimizations

There are several optimizations that have been done on the previously described implementation.

7.1 Term Pool per Arity

Most terms only have a small number of arguments. Let k be a constant such that a term is *small* whenever its head symbol has an arity tht is less than or equal to k. We will use $\mathcal{T}^i$, for any natural number i, to denote a subset of $\mathcal{T}$ with function symbols that have an arity equal to i.

The decision procedure has constant complexity as well. Now, instead of having one pool to store the finite subset $\mathcal{T}' \subseteq \mathcal{T}$, there will be a number of term pools:

- For small terms, k different pools to store $\mathcal{T}^0 \cup \cdots \cup \mathcal{T}^k \subseteq \mathcal{T}'$.
- A pool to store the set of terms $\mathcal{T}'' \subset \mathcal{T}'$ that do no occur in the other pools, *i.e.*, $\mathcal{T}'' = \mathcal{T}' \setminus (\mathcal{T}^0 \cup \cdots \cup \mathcal{T}^k)$.

Note that all pools are disjoint, *i.e.*, $\mathcal{T}^0 \cap \cdots \cap \mathcal{T}^k \cap \mathcal{T}'' = \emptyset$ and all pools combined store the whole subset, as such $\mathcal{T}^0 \cup \cdots \cup \mathcal{T}^k \cup \mathcal{T}'' = \mathcal{T}'$. The interface of the term pool remains unchanged. Internally this term pool uses the arity of a function symbol to decide which underlying pool will be used to create the term. For example calls to `create_appl(f)` will always go to the pool storing $\mathcal{T}^0$.

The advantage is that the complexity for all loops over the arguments of function symbols with an arity up to and including k will become constant. This enables compiler optimizations such as loop unrolling in practice, which can have quite an effect on run time performance. This optimization reduces the number of branch misses which increases the effective run-time performance as well.

7.2 Weak Reference Counting

Here, we introduce a way to relax the reference counting for arguments of terms to reduce the number of reference count changes. For this purpose we introduce the notion of a *weak reference*. A weak reference is a reference that does not change the reference counter of the referred to term, but might still refer other objects. We are going to change the references to arguments of a term to be weak references, thus reducing the number of reference count changes. However, this means that garbage collection has to be adapted, because a reference count of zero does not necessarily mean that the term is no longer referred to.

The garbage collection that is be described here is often referred to as a *tracing garbage collection*. The basic algorithm is a two-phase where first the reachable terms are *marked*, followed by a *sweep* that cleans up the unmarked terms, as these are unreachable.

The marking procedure is implemented as follows. Consider the terms as a graph where the set of vertices are given by $\mathcal{T}'$ and the edges are given by the weak references between terms and their arguments. Every term with a reference count above zero is reachable by definition, these terms form the *root* set. This has been implemented as indicated in Algorithm 10.

Algorithm 10 Marking the root set

```
 1: procedure RootMark
 2:     for $t \in S$ do
 3:         if reference-count($t$) $> 0$ then
 4:             Mark(t)
 5:         end if
 6:     end for
 7: end procedure
```

Starting from elements in this root set, we perform a search to mark the elements that can be reached by the edges in this graph. This step is implemented by the *mark* function described in Algorithm 11. A term can be marked by the function *set-mark* and the mark can be removed by using *remove-mark*. The function *is-marked* returns true if and only if that term has been marked. Note that marking stops whenever it find that the term has already been marked. This is correct, because marked terms either belong to the root term or the mark function has been previously applied to them, ensuring that its arguments have already been marked. This ensures that the subterms of shared terms are not explored again.

Algorithm 11 Marking reachable terms

```
 1: procedure Mark($t : \mathcal{T}'$)
 2:     if $\neg$is-marked($t$) then
 3:         set-mark($t$)
 4:         for $p \in args(t)$ do
 5:             Mark(p)
 6:         end for
 7:     end if
 8: end procedure
```

In the implementation this exploration is implemented depth-first and with an explicit stack to avoid excessive memory usage and stack overflow issues.

The marking of terms itself can be implemented efficiently by observing that all terms outside of the root set have a reference count of zero. This reference count can be set to a special value (in this case max value) to indicate that it has been marked and removing the mark resets it back to zero. Furthermore, in

After these steps we can conclude that all terms that have been marked are not reachable by the root set can be deallocated and removed from the set $\mathcal{T}'$ We should also remove the mark of terms such that the next garbage collection can be performed again. This is implemented by the *sweep* function that is described in Algorithm 12.

Algorithm 12 Sweeping terms that are no reachable

```
 1: procedure Sweep
 2:     for $t \in S$ do
 3:         if $\neg$is-marked($t$) then
 4:             $S := S \setminus \{t\}$
 5:             deallocate($t$)
 6:         else
 7:             remove-mark($t$)
 8:         end if
 9:     end for
10: end procedure
```

After this the heap memory used by terms that are not reachable has been freed.

7.3 Null Term

Instead of introducing a default term that is used whenever the `shared_reference` is default constructed we can introduce an actual *null* reference, as in the literal defined in the language. This breaks the invariant that any term always has a valid (possibly default) shared reference. Therefore, the `shared_reference` must first check whether it points to a valid term before trying to adapt its reference counter. For this purpose a function named *defined* is introduced that returns true iff the shared reference is not equal to *null*.

The advantage of this alternative is a removal of reference count adaptations to all instances of the default term. In the 1394 protocol state space generation with the option `--cached` this optimization reduces the number of reference count adaptations by ten percent. The run-time performance was almost unaffected. However, in the case of atomic reference counters this had about 11 percent run time reduction for the state space exploration.

7.4 Block Allocator

The allocate and deallocate functions are part of the allocator interface as defined in the STL. Note that we have combined the allocate and construct function where allocate also directly assigns the data on the heap, which is normally performed by the constructor. Although no specific implementation is required for these functions they typically call the system interface to obtain memory from the operating system for each call. In the Linux kernel a slab allocator is used that has no internal fragmentation for powers of two and a page size is typically 4KB. Compared to that, a single term of arity three only requires 40 bytes. This is a very small amount and also not a power of two, which is typically not optimal.

As these terms are fundamental to the operation of the tool set the number of allocated terms is typically very high. Therefore, a useful approach to reduce the number of system calls and memory overhead is to allocate larger blocks of memory and return references into these blocks. This is done by our *block allocator*.

The block allocator has a single linked list of memory blocks called blocks. Each block can store ElementsPerBlock number of elements. We refer to these elements as *slots* in the block that can either be free or contain an element. The allocator stores a currentIndex of the first slot in the block that has never been used. To keep track of slots that have been freed in the mean time a *free list* is used.

The `allocate` function is described in Algorithm 13. First, if the free list is not empty then we return the next element in that list and remove it from the list, this is done by `pop_front`. If we have used all indices at least once, and the freelist is empty, then a new block is allocated and added the the *blocks* list. Otherwise, we take the first index in the first block that has not been used yet and update the `currentIndex` accordingly.

Algorithm 13 Allocate

```
1:  procedure ALLOCATE
2:      if ¬freeList.empty() then
3:          return freeList.pop_front()
4:      end if
5:      if currentIndex ≥ ElementsPerBlock then
6:          blocks := blocks.push_front()
7:          currentIndex := 0
8:      end if
9:      firstBlock := blocks.front()
10:     slot := firstBlock[currentIndex]
11:     currentIndex := currentIndex + 1
12:     return slot
13: end procedure
```

Deallocating a term is straightforward as it just has to be added to the freelist, as shown in Algorithm 14

Algorithm 14 Deallocate

```
1: procedure DEALLOCATE(Reference r)
2:     freeList.push_front(r)
3: end procedure
```

One thing to note here is that the freelist can be efficiently implemented by storing the reference to the next element in the list in place of the free slots of the blocks. The first slot in this free list is pointed to some variable named *firstFreeSlot*. The *freeList* can then be stored by storing the *next* reference, which contains the reference to the next element in the list, in place of the slots. We define the invariant that all slots that are reachable by following next references after `firstFreeSlot` are part of the freelist. This means that the *freeList* is empty when the *firstFreeSlot* points to null. The *push_front* operation can be achieved by letting the *firstFreeSlot* point to the reference that was pushed into the *freeList* and setting the next reference to the head of the *freeList*. Iteration over the *freeList* can be achieved by following the *next* reference until it is null. This means that the freelist can be updated without performing any allocations by itself.

Finally, in case that many elements are deallocated it is useful to erase blocks that do not store any elements, equivalently where all slots are free. For this purpose we introduce the *consolidate* function as shown in Algorithm 15. In the first part of the algorithm all elements of the free list are marked by a special value $\top$, which is a value that should not occur in any slot before consolidate is called, in lines 2 to 6. For any block that only contains these special values it can be removed as shown in Line 6. The next part is to reconstruct the *freeList* from the free elements in the block. In practice these two looks are combined into one, maintaining whether the block contains elements that are not $\top$ and the *freeList* during iteration over a block. At the end of this iteration, the block is removed when the first condition holds and all entries of this block are removed from the *freeList*, which can be done in constant time by remembering the starting element when entering this loop.

Algorithm 15 Consolidate

```
1: procedure CONSOLIDATE
2:     for slot ∈ freeList do
3:         slot := $\top$
4:     end for
5:     for block ∈ blocks do
6:         if $\forall slot \in blockslot = \top$ then
7:             blocks.erase(block)
8:         else
9:             for slot ∈ block do
10:                if slot = $\top$ then
11:                    freeList.push_back(slot)
12:                end if
13:            end for
14:        end if
15:    end for
16: end procedure
```

Although it might seem that it is very unlikely that blocks become completely empty this optimization had quite a large effect on the larger examples (in the order of ten to twenty percent).

7.5 Alignment

In a typical processor the accesses to main memory are cached through a number of increasingly larger, but slower caches. A *cache line* is a block of consecutive memory that a processor fetches from main memory and stores in the cache at once. Whenever the processor loads or stores an address from main memory it actually fetches the whole block that contains this address. Furthermore, these blocks are disjoint and so-called *aligned* to multiples of the cache line *width*.

In a modern processor the typical cache line has a width of 64 bytes. This means that every memory access will fetch 64 consecutive bytes if it is not already in the cache. One optimization idea was to store the terms in the memory such that fetching its arguments does not cross cache line boundaries (and thus only require one fetch). However, benchmarks indicated that this did not have a good impact on performance and it does carry a potential memory increase, which is also undesirable.

7.6 Number Terms

In some cases it is useful to store a numerical value as an argument to a term directly. For example to gain better performance it might be useful to index terms in some data structure and store this index as an argument in the term. Therefore, we extend the definition of terms such that $\mathbb{N} \subseteq \mathcal{T}$. The API is extended in the following way:

$n \in \mathbb{N}$	create_int(n : $\mathbb{N}$)

Figure 3: Extension to the API with number terms

The `value` function is defined that maps constant terms to the natural number value of which it was constructed. In the implementation this numerical value is stored where normally the reference to the first argument is stored and a new function symbol `aterm_int` $\in \mathcal{F}_0$ is defined that is used to indicate such a special term.

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.