An algorithm to find a representant for sorts in the context of sort aliases and recursive sorts

Jan Friso Groote

May 4, 2010

In mCRL2 it is possible to define sort aliases, which have the form $A = B$. This means that sort A and sort B are considered the same, and are completely exchangeable.

Typical examples of sort aliases are

sort $Time = \mathbb{N}$;
$F = C \to D$;
$L = List(C)$;
$Complex = Bag(A \to Set(B))$;

It is also possible to define structured sorts that can be recursive (contrary to function sorts, lists, sets, and bags above, which cannot be recursive).

A structured sort has the shape:

$$\begin{array}{ll}\textbf{sort} & A = \textbf{struct}\ c_1(pr_{1,1} : A_{1,1},\ \ldots, pr_{1,k_1} : A_{1,k_1})?isC_1 \\ & \quad \mid c_2(pr_{2,1} : A_{2,1},\ \ldots, pr_{2,k_2} : A_{2,k_2})?isC_2 \\ & \qquad \vdots \\ & \quad \mid c_n(pr_{n,1} : A_{n,1}, \ldots, pr_{n,k_n} : A_{n,k_n})?isC_n\end{array}$$

This declares sort A to consists of n constructors c_i, projection functions $pr_{i,j}$ and recognisers isC_i. All the $A_{i,j}$ are sorts. The $A_{i,j}$ can be equal to A, in which case A is a recursive structured sort.

A very well known example is that of a tree data structure in which natural numbers can be stored.

sort $Tree = \textbf{struct}\ node(left : Tree, right : Tree) \mid leave(\mathbb{N})?is\_leave$;

By combining aliases and structured sorts, it is possible to have very different looking sort expressions that denote the same sorts. Two such expressions are equal if by folding and unfolding the definitions in sort aliases and in structured sorts, the sort expressions can be rewritten to each other. When manipulating terms it is inconvenient to be forced to perform folding and unfolding to determine equivalences of sorts. Therefore, it is useful to replace all equivalent sorts by a single unique representation, reducing the check for equivalency of sorts to checking whether the sorts are syntactically equal. This process is called *sort normalisation*. Note that normalisation depends on a data specification. Adding one sort alias or one structured sort can change the outcome of the normalisation procedure.

Below we give an algorithm to perform normalisation which is used in the mCRL2 tool suite. The essential idea is that all the definitions of structured sorts are interpreted from right to left, whereas all other rules are interpreted as rewrite rules from left to right. So, in the example above, $Time$ is rewritten to $\mathbb{N}$, F is rewritten to $C \to D$, etc. Because, ordinary sort aliases rules cannot be recursive, and structured sorts shrink with every rewrite step, this rewrite system is terminating.

But as the rewrite system is not confluent, unique normal forms are not guaranteed. The following example shows the problem.

sort $A = \textbf{struct}\ f(\mathbb{N})$;
$C = \textbf{struct}\ f(\mathbb{N})$;

A sort of the shape $\mathbf{struct}\ f(\mathbb{N})$ can be normalised to sort A and sort B. In order to deal with this problem, we apply Knuth-Bendix completion, to guarantee that all normal forms are unique.

The algorithm is performed in three steps. First, the set of aliases is checked for recursive definitions in all sorts except the structured sorts. If such a loop in the sort aliases is detected, an exception is thrown. The algorithm consists of a simple depth first search.

Second, we have two sets of rewrite rules. We have two auxiliary multimaps that map types to their respective right hand side. The multimap $resulting\_normalized\_sort\_aliases$ contains all definitive type rewrite rules, except that in case of multiple entries, with the same lhs only one will end up in the definitive set of type rewrite rules. The multimap $sort\_aliases\_to\_be\_investigated$ contains those type rewrite rules that must be investigated for critical pairs to determine whether they lead to extra type rewrite rules.

All aliases $B = type\_expression$ are directly added to $resulting\_normalized\_sort\_aliases$ as a rewrite rule $B \to type\_expression$ if $type\_expression$ is not a structured type. Otherwise they are added as a rewrite rule of the shape $type\_expression \to B$ to $sort\_aliases\_to\_be\_investigated$.

As a third step the sort aliases are taken as rewrite rules, and a form of Knuth-Bendix completion is applied to them, to transform them into a confluent term rewriting system, guaranteeing unique representations. Only the rules in $sort\_aliases\_to\_be\_investigated$ need to be investigated as those in $resulting\_normalized\_sort\_aliases$ cannot give rise to critical pairs.

So, if there are two overlapping left hand sides in the rewrite system, this means that one term is a subterm of the other. So, we have a rule $C(g(t)) \to u_1$ and a rule $g(t) \to u_2$ where C represents a possibly empty context. So, the term $C(g(t))$ can rewrite to both u_1 and $C(u_2)$. In this case we add a rewrite rule $t \to u_1$ where t is the normal form of $C(u_2)$ for the rewrite rules in $resulting\_normalized\_sort\_aliases$.

An important observation is that the rules always have one of the following shapes:

$$\begin{array}{rcl} \mathbf{struct}\ldots & \to & A, \\ B & \to & \mathit{Exp} \end{array}$$

where A and B are basic sorts and *Exp* is a sort expression which can be a basic sort, but can also contain all other type forming constructs. There are the following invariants on the rules. For each basic sort B there is at most one rule of the form $B \to \cdots$. Furthermore, a basic sort A occurring at the right of a struct rule can never occur as the left hand side of a rewrite rule also.

So, when one left hand side of a rule overlaps with another left hand side, one of the rules must have the shape $\mathbf{struct}\ldots \to \ldots$, whereas the other can contain a struct or a basic term at the left hand side. As the rule with a struct rewrites to a basic sort A, the newly added rewrite rule has A at its right hand side.

The number of newly added rules in this way is bounded. When both left hand sides contain structs, the newly added rule has a strictly smaller number of structs in its right hand side than one of its originals. Moreover, no new basic sort is introduced that can act as the lhs of a new rule. When a rule of the shape $A \to \mathit{Exp}$ contains overlap, a rule is obtained where an occurrence of A is replaced by an occurrence of Exp. But as these rules are acyclic, this can only be performed a finite number of times.

In more detail, we have two sets of rewrite rules. One that is definitive $m\_normalised\_sort\_aliases$ and $sort\_aliases\_to\_be\_investigated$ that contains sort rewrite rules still to be investigated. Initially, all rules are in $sort\_aliases\_to\_be\_investigated$. Each rewrite rule $t_1 \to u_1$ in $sort\_aliases\_to\_be\_investigated$ is checked with each rule $t_2 \to u_2$ in $m\_normalised\_sort\_aliases$. If t_1 is a subterm of t_2 (i.e. $t_2 = C(t_1)$) and u_2 and $C(u_1)$ do not have the same normal forms, then a rule $C(u_1) \to u_2$ is added to $sort\_aliases\_to\_be\_investigated$. If t_2 is a subterm of t_1 a symmetric sequence of steps is done. After all rewrite rules $t_1 \to u_1$ in $m\_normalised\_sort\_aliases$ have been investigated, $t_2 \to u_2$ is added to $m\_normalised\_sort\_aliases$.

The resulting rewrite system is terminating, provided that the original rewrite system was terminating. Each new rule that is added has the shape $C = a$, where C is a basic or complex type, and a is a basic sort, which is a normal form in the rewrite system. The only way that there is non termination, is when there is an infinite sequence of basic sorts $a_1, a_2, \ldots$, such that a_i rewrites to a_{i+1}. This loop came into existence by adding some rewrite rule $a = a'$ at some moment in time, where a' was not a normal form. But this cannot happen, because by construction a' is a normal form.

After constructing the normal forms, the content of $m\_normalised\_sort\_aliases$ is copied into *m_normalised_aliases*, where every right hand side is normalised, to speed up rewriting when applied to concrete sorts.

Normalisation of concrete sorts is now very simple. Every sort which equals a left hand side of a sort alias is replaced by the right hand side. This is repeated until no such substitution can be applied. This can be done using a simple innermost rewriting procedure. This rewriter has been implemented in `normalize_sorts_function`.

Acknowledgements Thanks go to Aleksi Peltonen for identifying that in the algorithm up to spring 2018 the newly added type rewrite rules were not normalised, leading to the addition of an exponential type rewrite rules, slowing type rewriting down. His example was

sort $A_t = Nat; B_t = Nat; C_t = Nat; D_t = Nat; E_t = Nat; F_t = Nat; G_t = Nat;$
$S_t = structs(A : A_t, B : B_t, C : C_t, D : D_t, E : E_t, F : F_t, G : G_t);$
init $\delta;$

This would lead to 2^n type rewrite rules for all n arguments of the function s.