A Type System for Usage of Software Components*

Dag Hovland

Department of Informatics, The University of Bergen,
PB 7803, N-5020 Bergen, Norway
dagh@ii.uib.no

Abstract. The aim of this article is to support component-based software engineering by modelling exclusive and inclusive usage of software components. Truong and Bezem describe in several papers abstract languages for component software with the aim to find bounds of the number of instances of components. Their language includes primitives for instantiating and deleting instances of components and operators for sequential, alternative and parallel composition and a scope mechanism. The language is here supplemented with the primitives `use`, `lock` and `free`. The main contribution is a type system which guarantees the safety of usage, in the following way: When a well-typed program executes a subexpression `use`$[x]$ or `lock`$[x]$, it is guaranteed that an instance of x is available.

Key words: Component Software, Type System, Parallel Execution, Component Usage, Process Model

1 Introduction

The idea of "Mass produced software components" was first formulated by McIlroy [1] in an attempt to encourage the production of software routines in much the same way industry manufactures ordinary, tangible products. The last two decades "component" has got the more general meaning of a highly reusable piece of software. According to Szyperski [2] (p. 3), "(...) software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system". We will model software that is constructed of such components, and assume that during the execution of such a program, instances of the components can be created, used and deleted.

Efficient component software engineering is not compatible with programmers having to acquire detailed knowledge of the internal structure of components that are being used. Components can also be constructed to use other

* S. Berardi, F. Damiani, and U. de'Liguoro (Eds.): TYPES 2008, LNCS 5497, pp. 186-202, 2009. http://www.springerlink.com/content/w182776723804g60/.

components, such that instantiating one component, could lead to several instances of other components. This lack of knowledge in combination with nested dependencies weakens the control over resource usage in the composed software.

The goal of this article is to guarantee the safe usage of components, such that one can specify that some instances must be available, possibly exclusively to the current thread of execution. In [3,4,5], Truong and Bezem describe abstract languages for component software with the aim of finding bounds of the number of instances of components existing during and remaining after execution of a component program. Their languages include primitives for instantiating and deleting instances of components and have operators for sequential, alternative and parallel composition and a scope mechanism. The first three operators are well-known, and have been treated by for example Milner [6] (where alternative composition is called *summation*). The scope mechanism works like this: Any component instantiated in a scope has a lifetime limited to the scope. Furthermore, inside a scope, only instances in the local store of the same scope can be deleted. The types count the maximum number of active component instances during execution and remaining after execution of a component program.

The languages described by Truong and Bezem lack a direct way of specifying that one or more instances of a component must exist at some point in the execution. In this paper we have added the primitives `use`, `lock` and `free` in order to study the usage of components. The first (`use`) is used for "inclusive usage", that is, when a set of instances must be available, but these instances may be shared between threads. The other form (`lock` and `free`) is used when the instances must exclusively be available for this execution thread. The difference between exclusive and inclusive usage can be seen by comparing the expressions $\texttt{new}\,x(\texttt{use}[x] \parallel \texttt{use}[x])$ and $\texttt{new}\,x(\texttt{lock}[x]\texttt{free}[x] \parallel \texttt{use}[x])$. The first expression is safe to execute, while executing the latter expression can lead to an error if x is locked, but not freed, by the left thread before it is used by the right thread. Instances of the same component cannot be distinguished, such that locking and freeing is not applied to specific instances, but to the number of instances of each component.

The type system must guarantee that the instances that are to be used are available. The system will not test whether the deletion of instances in local stores is safe, as this can be tested using the type systems in [7,3,4,5] together with an easy translation described in Section 7. Only non-recursive programs are treated, but an extension with loops and simple recursion, described in [7], can also be applied to this system.

Section 2 introduces an example using C++, which is applied to the type system in Section 6. The language of component programs is defined in Section 3, and the operational semantics is defined in Section 4. The types and the type system are explained in Section 5. Important properties of the type system are formulated in Section 7, while the main results concerning correctness are collected in Section 8. The article ends with a section on related work and a conclusion.

2 Example: Objects on the Free Store in C++

We will introduce an example with dynamically allocated memory in C++ [8]. In Section 6 we will apply the type system to the example. The example is inspired by a similar one in [7].

In the program fragment in Figure 1, so-called POSIX threads [9] are used for parallelism. After creating an instance of the class `C`, the function `pthread_create` launches a new thread calling the function which is third in the parameter list with the argument which is fourth. This function call, either `P1(C_instance)` or `P2(C_instance)`, is executed in parallel to `P3(C_instance)`, and the two threads are joined in `pthread_join` before the instance of `C` is deleted.

The dynamic data type `C` and the functions `P1, P2, P3` are left abstract. We will assume the latter three functions use the instance of `C` in some way, and that `P2` needs exclusive access to the instance.

```
void EX(int choice) {
  pthread_t pth;
  C* C_instance = new C();
  pthread_create(&pth, NULL, choice ? P1 : P2 , C_instance);
  P3(C_instance);
  pthread_join(pth, NULL);
  delete C_instance;
}
```

Fig. 1. C++ code using threads and objects on the free store.

The question in this example is whether we can guarantee that `P2` gets exclusive access to the instance of `C`. In this small example it is possible to see that this is not the case. After the grammar is explained in the next section we will model the program in the language as shown in Figure 2, and use the type system to answer the question and correct the program.

3 Syntax

The language for components is parametrized by an arbitrary set $\mathbb{C} = \{a, b, c, \ldots\}$ of *component names*. We let variables x, y, z range over $\mathbb{C}$. Bags and multisets are used frequently in this paper, and will therefore be explained here.

3.1 Bags and Multisets

Bags are like sets but allow multiple occurrences of elements. Formally, a *bag* with underlying set of elements $\mathbb{C}$ is a mapping $M : \mathbb{C} \to \mathbb{N}$. Bags are often also called multisets, but we reserve the term multiset for a concept which allows one to express a deficit of certain elements as well. Formally, a *multiset* with underlying

set of elements $\mathbb{C}$ is a mapping $M : \mathbb{C} \to \mathbb{Z}$. We shall use the operations $\cup, \cap, +, -$ defined on multisets, as well as relations $\subseteq$ and $\in$ between multisets and between an element and a multiset, respectively. We recall briefly their definitions: $(M \cup M')(x) = \mathsf{max}(M(x), M'(x))$, $(M \cap M')(x) = \mathsf{min}(M(x), M'(x))$, $(M + M')(x) = M(x) + M'(x)$, $(M - M')(x) = M(x) - M'(x)$, $M \subseteq M'$ iff $M(x) \leq M'(x)$ for all $x \in \mathbb{C}$. The operation $+$ is sometimes called *additive union*. Bags are closed under all operations above with the exception of $-$. Note that the operation $\cup$ returns a bag if at least one of its operands is a bag. For convenience, multisets with a limited number of elements are sometimes denoted as, for example, $M = [2x, -y]$, instead of $M(x) = 2,\ M(y) = -1,\ M(z) = 0$ for all $z \neq x, y$. In this notation, $[\,]$ stands for the *empty* multiset, i.e., $[\,](x) = 0$ for all $x \in \mathbb{C}$. We further abbreviate $M + [x]$ by $M + x$ and $M - [x]$ by $M - x$. Both multisets and bags will be denoted by M or N (with primes and subscripts), it will always be clear from the context when a bag is meant. For any bag, let $\mathsf{set}(M)$ denote its set of elements, that is, $M = \{x \in \mathbb{C} \mid M(x) > 0\}$. Note that a bag is also a multiset, while a multiset is also a bag only if it maps all elements to non-negative numbers.

3.2 Grammar

Table 1. Syntax

$$
\begin{array}{lll}
\mathit{Expr} & ::= & \mathit{Factor} \mid \mathit{Expr} \cdot \mathit{Expr} \\
\mathit{Factor} & ::= & \mathtt{new}\, x \mid \mathtt{del}\, x \mid \mathtt{lock}\, M \mid \mathtt{free}\, M \mid \mathtt{use}\, M \mid \mathtt{nop} \\
 & & \mid (\mathit{Expr} + \mathit{Expr}) \mid (\mathit{Expr} \parallel \mathit{Expr}) \mid \mathit{ScExp} \\
\mathit{ScExp} & ::= & \{M, \mathit{Expr}\} \\
M & ::= & \text{bag of elements from } \mathbb{C} \\
\mathit{Prog} & ::= & \mathtt{nil} \mid \mathit{Prog}, x \prec\!\!\!- \mathit{Expr}
\end{array}
$$

Component expressions are given by the syntax in Table 1. We let capital letters $A, \ldots, E$ (with primes and subscripts) range over *Expr*. A *component program* P is a comma-separated list starting with `nil` and followed by zero or more *component declarations*, which are of the form $x \prec\!\!\!- \mathit{Expr}$, with $x \in \mathbb{C}$ (`nil` will usually be omitted, except in the case of a program containing no declarations). $\mathsf{dom}(P)$ denotes the set of component names declared in P (so $\mathsf{dom}(\mathtt{nil}) = \varnothing$). Declarations of the form $x \prec\!\!\!- \mathtt{nop}$ are used for *primitive* components, i.e., components that do not use *subcomponents*.

We have two primitives `new` and `del` for creating and deleting instances of a component, three primitives `free`, `lock` and `use` for specifying usage of instances of components and four primitives for composition: sequential composition denoted by juxtaposition, $+$ for choice (also called sum), $\parallel$ for parallel and $\{\ldots\}$ for scope. Note that instances of the same component cannot be distinguished.

The effect of `lock` is therefore to decrease the number of instances available for usage, while `free` increases this number.

Executing the sum $E_1 + E_2$ means choosing either one of the expressions E_1 or E_2 and executing that one. Executing E_1 and E_2 in parallel, that is, executing $(E_1 \parallel E_2)$, means executing both expressions in some arbitrary interleaved order. Executing an expression inside a scope, $\{[\,], E\}$ means executing E, while only allowing deletion of instances inside the same scope, and after the execution of E, deleting all instances inside the scope.

The grammatical ambiguity in the rule for *Expr* is unproblematic. Like in process algebra, sequential composition can be viewed as an associative multiplication operation and products may be denoted as $E\,E'$ instead of $E \cdot E'$. The operations $+$ and $\parallel$ are also associative and we only parenthesize if necessary to prevent ambiguity. Sequential composition has the highest precedence, followed by $\parallel$ and then $+$. The primitive `nop` models zero or more operations that do not involve component instantiation or deallocation.

In the third clause of the grammar we define *scope expressions*, used to limit the lifetime of instances and the scope of deletion. A scope expression is a pair of a bag, called the local store, and an expression. Scope expressions appearing in a *component declaration* in a program are required to have an empty local store. Non-empty local stores only appear *during execution* of a program.

Definition 1. *By* $\mathsf{var}(E)$ *we denote the set of component names occurring in* E*, formally defined by* $\mathsf{var}(\mathtt{nop}) = \varnothing$, $\mathsf{var}(\mathtt{new}\,x) = \mathsf{var}(\mathtt{del}\,x) = \{x\}$,$\mathsf{var}(\mathtt{use}\,M) = \mathsf{var}(\mathtt{free}\,M) = \mathsf{var}(\mathtt{lock}\,M) = \mathsf{set}(M)$, $\mathsf{var}(E_1{+}E_2) = \mathsf{var}(E_1 \parallel E_2) = \mathsf{var}(E_1\,E_2) = \mathsf{var}(E_1) \cup \mathsf{var}(E_2)$ *and* $\mathsf{var}(\{M, E\}) = \mathsf{set}(M) \cup \mathsf{var}(E)$.

Definition 2. *The* size *of an expression* E*, denoted* $\sigma(E)$*, is defined by* $\sigma(\mathtt{new}\,x) = \sigma(\mathtt{del}\,x) = \sigma(\mathtt{use}\,N) = \sigma(\mathtt{lock}\,N) = \sigma(\mathtt{free}\,N) = \sigma(\mathtt{nop}) = 1$, $\sigma(\{M, E\}) = \sigma(E) + 1$ *and* $\sigma(A + B) = \sigma(AB) = \sigma(A||B) = \sigma(A) + \sigma(B) + 1$. *The size of a program* P*, denoted* $\sigma(P)$*, is defined by* $\sigma(P, x \prec A) = \sigma(P) + 1 + \sigma(A)$ *and* $\sigma(\mathtt{nil}) = 1$.

3.3 Examples

We assume that a program is executed by executing $\mathtt{new}\,x$, where x is the last component declared in the program, starting with empty stores of component instances. Examples of programs that will execute properly and will be well-typed are

Example 1.

$$x \prec \mathtt{nop}, y \prec \mathtt{new}\,x\,\mathtt{use}[x]\,\mathtt{lock}[x]\,\mathtt{free}[x]$$
$$x \prec \mathtt{nop}, y \prec \mathtt{new}\,x\,\mathtt{new}\,x\,\{[\,], (\mathtt{use}[x] \parallel \mathtt{lock}[x])\}\,\mathtt{free}[x]$$

Examples of programs that can, for some reason, produce an error are:

Example 2.

$$
\begin{array}{l}
x \prec \mathtt{nop}, y \prec \mathtt{new}\, x\, \mathtt{new}\, x\, \{[\,], (\mathtt{use}[x] \parallel \mathtt{lock}[x])\} \\
x \prec \mathtt{nop}, y \prec \mathtt{new}\, x\, \mathtt{lock}[x]\, \mathtt{use}[x]\, \mathtt{free}[x] \\
x \prec \mathtt{nop}, y \prec \mathtt{new}\, x\, \{[\,], (\mathtt{use}[x] \parallel \mathtt{lock}[x])\}\, \mathtt{free}[x] \\
x \prec \mathtt{nop}, y \prec \mathtt{new}\, x\, \mathtt{free}[x]\, \mathtt{lock}[x] \\
x \prec \mathtt{nop}, y \prec \mathtt{new}\, x\, \{[\,], (\mathtt{use}[x] + \mathtt{lock}[x])\}\, \mathtt{free}[x]
\end{array}
$$

The first program leaves one instance of x locked after execution. The second will get stuck as no instance of x will be available for use by the `use`-statement. The third might also get stuck. Note that there exists an error-free execution of the third program, where the left branch of $(\mathtt{use}[x] \parallel \mathtt{lock}[x])$ is executed before the right one. But as we do not wish to make any assumptions about the scheduling of the parallel execution, we consider this an error. The fourth program tries to free a component instance that is not locked. The fifth program has a run in which $\mathtt{free}[x]$ is executed, but no instance of x has been locked.

C++ Example We now describe the model of the example program in Figure 1. Functions (such as `EX`) as well as objects on the free store (such as `C_instance`) are modelled as components. We let $\mathtt{call}\, f$ abbreviate $\mathtt{new}\, f\, \mathtt{del}\, f$ and use this expression to model a function call. Note that f is deleted automatically by $\mathtt{call}\, f$, which models the (automatic) deallocation of stack objects created by f. However, the subcomponents of f are not deleted by $\mathtt{del}\, f$. We use small letters for the component names and model functions as components, where the function body is given by the right hand side of the declaration. Since `P2` needs exclusive access to an instance of `C` we add $\mathtt{lock}[c]\, \mathtt{free}[c]$ to the declaration of p_2. For p_1 and p_3 we indicate the non-exclusive usage by $\mathtt{use}[c]$. Collecting all declarations we get the program in Figure 2.

$$
\begin{array}{l}
c \prec \mathtt{nop}, \\
p_1 \prec \mathtt{use}[c], \\
p_2 \prec \mathtt{lock}[c]\, \mathtt{free}[c], \\
p_3 \prec \mathtt{use}[c], \\
ex \prec \mathtt{new}\, c\, ((\mathtt{call}\, p_1 + \mathtt{call}\, p_2) \parallel \mathtt{call}\, p_3)\, \mathtt{del}\, c
\end{array}
$$

Fig. 2. Program P, a model of the C++ program in Figure 1.

4 Operational Semantics

A *state*, or state expression, is a pair $(M_u, \{M, E\})$ consisting of a bag M_u (called the global store) with underlying set of elements $\mathbb{C}$, and a scope expression $\{M, E\}$. The store M in this scope expression is called the local store of the

expression. An *initial state* is of the form $([\,], \{[\,], \mathtt{new}\, x\})$, and a *terminal state* is of the form $(M_u, \{M, \mathtt{nop}\})$.

A state $(M_u, \{M, E\})$ expresses that we execute E with a local bag M and a global bag M_u of instances of components. The operational semantics is given in Table 2 as a state transition system in the style of structural operational semantics [10]. The inductive rules are osPar1, osPar2, osScp and osSeq. The other rules are not inductive, but osNew, osDel, osLock, osUse and osPop are conditional with the condition specified as a premiss of the rule. The transition relation with respect to a program P is denoted by $\rightsquigarrow_P$, with transitive and reflexive closure by $\rightsquigarrow_P^*$.

Table 2. Transition rules for a component program P

(osNop)
$$\frac{}{(M_u, \{M, \mathtt{nop}\, E\}) \rightsquigarrow_P (M_u, \{M, E\})}$$

(osNew)
$$\frac{x \multimap A \in P}{(M_u, \{M, \mathtt{new}\, x\}) \rightsquigarrow_P (M_u + x, \{M + x, A\})}$$

(osDel)
$$\frac{x \in (M \cap M_u)}{(M_u, \{M, \mathtt{del} x\}) \rightsquigarrow_P (M_u - x, \{M - x, \mathtt{nop}\})}$$

(osLock)
$$\frac{N \subseteq M_u}{(M_u, \{M, \mathtt{lock}\, N\}) \rightsquigarrow_P (M_u - N, \{M, \mathtt{nop}\})}$$

(osFree)
$$\frac{}{(M_u, \{M, \mathtt{free}\, N\}) \rightsquigarrow_P (M_u + N, \{M, \mathtt{nop}\})}$$

(osUse)
$$\frac{N \subseteq M_u}{(M_u, \{M, \mathtt{use}\, N\}) \rightsquigarrow_P (M_u, \{M, \mathtt{nop}\})}$$

(osScp)
$$\frac{(M_u, \{N, A\}) \rightsquigarrow_P (M'_u, \{N', A'\})}{(M_u, \{M, \{N, A\}\}) \rightsquigarrow_P (M'_u, \{M, \{N', A'\}\})}$$

(osPop)
$$\frac{N \subseteq M_u}{(M_u, \{M, \{N, \mathtt{nop}\}\}) \rightsquigarrow_P (M_u - N, \{M, \mathtt{nop}\})}$$

(osAlt$i$)
$$\frac{i \in \{1, 2\}}{(M_u, \{M, (E_1 + E_2)\}) \rightsquigarrow_P (M_u, \{M, E_i\})}$$

(osSeq)
$$\frac{(M_u, \{M, A\}) \rightsquigarrow_P (M'_u, \{M', A'\})}{(M_u, \{M, A\, E\}) \rightsquigarrow_P (M'_u, \{M', A'\, E\})}$$

(osParEnd)
$$\frac{}{(M_u, \{M, (\mathtt{nop} \parallel \mathtt{nop})\}) \rightsquigarrow_P (M_u, \{M, \mathtt{nop}\})}$$

(osPar1)
$$\frac{(M_u, \{M, E_1\}) \rightsquigarrow_P (M'_u, \{M', E'_1\})}{(M_u, \{M, (E_1 \parallel E_2)\}) \rightsquigarrow_P (M'_u, \{M', (E'_1 \parallel E_2)\})}$$

(osPar2)
$$\frac{(M_u, \{M, E_2\}) \rightsquigarrow_P (M'_u, \{M', E'_2\})}{(M_u, \{M, (E_1 \parallel E_2)\}) \rightsquigarrow_P (M'_u, \{M', (E_1 \parallel E'_2)\})}$$

4.1 Unsafe States

A *stuck state* is usually defined as a state which is not terminal, and where there is no possible next transition. We wish to use a different condition, because we want to assure that all possible runs are error-free. This means that we do not assume anything about the interleaving used in parallel executions. This is more in line with how parallelism works by default in many environments, for example with `pthreads` and C++ without mutex locking. Informally, we call a state *unsafe* if there is at least one transition which cannot be used in this state, but which would be possible with a larger global store. For example, $([\,], \{[x], \texttt{lock}\,[x] \parallel \texttt{free}\,[x]\})$ is an unsafe state, because using osPar1 is only possible with a larger global store.

Definition 3 (Unsafe states). *Given a component program P, a state $(M_u, \{M, E\})$ is called unsafe if and only if there exist bags M'_u, M and N and an expression E' such that $(M_u + N, \{M, E\}) \rightsquigarrow_P (M'_u + N, \{M', E'\})$, but not $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$*

It is also possible to characterize the unsafe states with the following inductive rules parametrized by a program P and bags M_u and M: for all x and N, where $x \notin M_u$ and $N \not\subseteq M_u$, $(M_u, \{M, \texttt{lock}\,N\})$, $(M_u, \{M, \texttt{use}\,N\})$, $(M_u, \{M, \texttt{del}\,x\})$ and $(M_u, \{M, \{N, \texttt{nop}\}\})$ are unsafe, and for all expressions E and F, if $(M_u, \{M, E\})$ is unsafe then for all bags N, also $(M_u, \{N, \{M, E\}\})$, $(M_u, \{M, EF\})$, $(M_u, \{M, E \parallel F\})$ and $(M_u, \{M, F \parallel E\})$ are unsafe. Recall that deletion of component instances in the local store is assumed to always be safe, as this can be assured by the system in [7]. A state which is not unsafe is called *safe*.

4.2 Valid States

For some state $(M_u, \{M, E\})$ in a run, M_u models all component instances available for usage. We must therefore have M_u no larger than the sum of N in all subexpressions $\{N, A\}$ of E. For example $([x], \{[\,], \texttt{nop}\})$ should not appear in a run because $M_u \supset []$. Conditions for this to be true will be stated later. However, there are transitions where the states in the transition fulfil this condition, while the derivation of the transition contains states which do not fulfil the condition. An example is the transition $([x], \{[x], \{[], \texttt{use}\,[x]\}\}) \rightsquigarrow_P ([x], \{[x], \{[], \texttt{nop}\}\})$, in which both states fulfil this condition, while it is the result of applying osScp to the premiss $([x], \{[], \texttt{use}\,[x]\}) \rightsquigarrow_P ([x], \{[], \texttt{nop}\})$, where none of the two states fulfil the condition.

To express this property more formally we need a way to sum all the local stores in an expression. In doing so, however, one counts in instances that will never coexist, such as in $\{M_1, E_1\} + \{M_2, E_2\}$ and $\{M_1, E_1\}\,\{M_2, E_2\}$. Therefore we also define the notion of a valid expression, in which irrelevant bags are empty.

Definition 4 (Sum of local stores). *For any expression E, let ΣE be the sum of all N in subexpressions $\{N, A\}$ of E. More formally: $\Sigma\{M, E\} = M + \Sigma E$ and $\Sigma(E_1 \parallel E_2) = \Sigma(E_1\,E_2) = \Sigma(E_1 + E_2) = \Sigma E_1 + \Sigma E_2$ and $\Sigma\texttt{del}\,x = \Sigma\texttt{new}\,x =$*

$\Sigma\texttt{use}N = \Sigma\texttt{lock}N = \Sigma\texttt{free}N = \Sigma\texttt{nop} = [\,]$. *An expression* E *is valid if for all subexpressions of the form* $(E_1 + E_2)$ *we have* $\Sigma(E_1 + E_2) = [\,]$*, and for all subexpressions of the form* $F\,E'$*,* F *a factor, we have* $\Sigma E' = [\,]$.

Note that an expression is valid if and only if all its subexpressions are valid. We will say that a state $(M_u, \{M, E\})$ is valid if and only if E is valid. The initial state is valid by definition. In any declaration $x \prec\!\!- E$, since only empty bags are allowed to occur in E, E is obviously valid and $\Sigma E = [\,]$.

5 Type System

5.1 Types

A *type* of a component expression is a tuple $X = \langle X^u, X^n, X^l, X^d, X^p, X^h \rangle$, where X^n, X^u and X^p are bags and X^l, X^d and X^h are multisets. We use $U, \ldots, Z$ to denote types. The properties of the different parts of the types are summarized in Table 3, and will be explained below. The bag X^u (u for "usage") contains the minimum size the global store must have for an expression to be safely executed.

Because of sequential composition, we also need a multiset X^l. To run the expression $E_1\,E_2$, we must not only know the minimum safe sizes for executing E_1 and E_2 separately, but also how much E_1 decreases or increases the global store. The multiset X^l therefore contains, for each $x \in \mathbb{C}$, the *lowest* net increase in the number of instances in the global store after the execution of the expression. (Where a decrease is negative increase.) This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, then $X^l \subseteq M'_u - M_u$.

The scope operator makes necessary the component X^d. When a scope is *popped* with the rule osPop, the remaining bag in the scope is subtracted from the global store. The difference between these two bags must therefore be controlled by X^d. In addition, concerning the two alternatives joined in a choice expression, the net effect on the difference between the global store and the local store must be equal. An example of an invalid expression excluded by this rule is $(\texttt{lock}x + \texttt{use}x)$. If the latter expression was allowed in a program, it would not

Table 3. The parts of the types

X^u:	Minimum safe size of the global store.
X^n:	Largest decrease of the global store during execution.
X^l:	Lower bound of the net effect on the global store.
X^d:	Net change in the difference between the local and the global store.
X^p:	Maximum increase, during execution, of the difference between the global store and the sum of all local stores.
X^h:	Maximum net effect on the difference between the global store and the sum of all the local stores.

be possible to give the guarantees needed for osPop to the number of instances of x locked after execution. The multiset X^d therefore contains the exact change in the difference between the local store and the global store made by execution of the expression. This difference is independent of how the expression is executed. This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, then $X^d = (M'_u - M') - (M_u - M)$.

Parallel composition necessitates the bag X^n. The minimum safe size for executing $(E_1 \parallel E_2)$ depends not only on the minimum safe size for executing each of E_1 and E_2, but also on how much each of them decreases the global store. For example, both $\texttt{use}\,x$ and $\texttt{lock}\,x\,\texttt{free}\,x$ need one instance of x, but $\texttt{use}\,x \parallel \texttt{use}\,x$ also needs only one, whereas $\texttt{lock}\,x\,\texttt{free}\,x \parallel \texttt{lock}\,x\,\texttt{free}\,x$ needs two instances of x. X^n contains, for each $x \in \mathbb{C}$, the highest *negative* net change in the number of instances in the global store during the execution of the expression. This implies that, if the type of E is X and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$, then $-X^n \subseteq M'_u - M_u$.

As seen in Example 2 in Section 3.3, there are grammatically correct programs that "free" instances that are not locked. So far, we have not distinguished between $\texttt{free}[x]\,\texttt{lock}[x]$ and $\texttt{lock}[x]\,\texttt{free}[x]$. Obviously, these expressions cannot be assigned the same type. For example, the program $x \prec \texttt{nop}, y \prec \texttt{new}\,x\,\texttt{free}[x]\,\texttt{lock}[x]$ is wrong, and should not be well-typed, while the program $x \prec \texttt{nop}, y \prec \texttt{new}\,x\,\texttt{lock}[x]\,\texttt{free}[x]$ is correct and should be well-typed. There is a need for types concerned with the difference between the number of instances in the sum of all local stores and the number of instances in the global store. If $(M_u, \{M, E\})$ is a state during the execution of a component program, then the value of $(M_u - \Sigma\{M, E\})(x)$ for a component x is negative if an instance of x is locked, but not yet freed, and positive if it has been freed without being locked. The latter is seen as an error and should not occur in the run of a well-typed program. The bag X^p and multiset X^h are used for keeping track of the set $M_u - \Sigma\{M, E\}$, and contain, the highest *positive* net change during execution and the *highest* net increase of this bag after execution. This implies that if the type of E is X, then if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ then $X^p \supseteq (M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})$, and if $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, we get $X^h \supseteq (M'_u - M') - (M_u - \Sigma\{M, E\})$. In the type of a well-typed program these parts must be empty bags.

5.2 Typing Rules

The typing rules in Table 4 and Table 5 must be understood with the above interpretation in mind. They define a ternary typing relation $\Gamma \vdash E : X$ and a binary typing relation $\vdash P : \Gamma$ in the usual inductive way. Here Γ is usually called a *basis*, mapping component names to the type of the expression in its declaration. In the relation $\vdash P : \Gamma$, Γ can be viewed as a type of P. An expression of the form $\Gamma \vdash E : X$ or $\vdash P : \Gamma$ will be called a *typing* and will also be phrased as 'expression E has type X in Γ' or 'program P has type Γ', respectively.

A basis Γ is a partial mapping of components $x \in \mathbb{C}$ to types. By $\mathsf{dom}(\Gamma)$ we denote the domain of Γ, and for any $x \in \mathsf{dom}(\Gamma)$, $\Gamma(x)$ denotes its type in Γ.

For a set $S \subseteq \mathsf{dom}(\Gamma)$, $\Gamma|_S$ is Γ restricted to the domain S. For any $x \in \mathbb{C}$ and type X, $\{x \mapsto X\}$ denotes a basis with domain $\{x\}$ and which maps x to X. An expression E is called *typable* in Γ if $\Gamma \vdash E : X$ for some type X. The latter type X will be proved to be unique and will sometimes be denoted by $\Gamma(E)$.

Table 4. Typing Rules

(AxmP)
$$\frac{}{\vdash \texttt{nil} : \varnothing}$$

(Axm)
$$\frac{}{\Gamma \vdash \texttt{nop} : \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle}$$

(Lock)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{lock}\, N : \langle N, N, -N, -N, [\,], -N \rangle}$$

(Free)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{free}\, N : \langle [\,], [\,], N, N, N, N \rangle}$$

(New)
$$\frac{\Gamma(x) = X}{\Gamma \vdash \texttt{new}\, x : \langle X^u, X^n, X^l + x, X^d, X^p, X^h \rangle}$$

(Del)
$$\frac{\Gamma(x) = X}{\Gamma \vdash \texttt{del}\, x : \langle [x], [x], [-x], [\,], [\,], [\,] \rangle}$$

(Use)
$$\frac{\mathsf{set}(N) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \texttt{use}\, N : \langle N, [\,], [\,], [\,], [\,], [\,] \rangle}$$

(Prog)
$$\frac{\Gamma \vdash E : X, \;\; \vdash P : \Gamma, \;\; x \notin \mathsf{dom}(\Gamma)}{\vdash P, x \prec E : \Gamma \cup \{x \mapsto X\}}$$

Definition 5 (Well-typed program). *A program P with at least one declaration is* well-typed *if there are Γ and X such that $\vdash P : \Gamma$, $\Gamma \vdash \texttt{new}\, x : X$ and $X^d = X^u = X^p = []$, where x is the last component declared in P.*

The condition in Definition 5 that parts X^d, X^u and X^p be empty deserves an explanation. X^d must be empty, because the global and local store must be equal in the final state, that is, no instances are still locked when the program ends. X^u is the minimum safe size of the global store, and we assume the program is executed starting with an empty global store, so X^u must be empty. X^p must be empty, because this is the only way to guarantee that, during execution, no instance is freed, unless there already is a locked instance of the same component.

Type inference in this system is similar to [7,3,4,5]. In particular, the type inference algorithm has quadratic runtime. An implementation of the type system can be downloaded from the author's website.

6 C++ Example Continued

Recall the C++ program in Figure 1 and the component program in Figure 2. Type inference gives the following results:

Table 5. Typing Rules (continued)

(Par)
$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2}{\Gamma \vdash E_1 \parallel E_2 : \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle}$$

(Alt)
$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2,\ X_1^d = X_2^d}{\Gamma \vdash E_1 + E_2 : \langle X_1^u \cup X_2^u, X_1^n \cup X_2^n, X_1^l \cap X_2^l, X_1^d, X_1^p \cup X_2^p, X_1^h \cup X_2^h \rangle}$$

(Seq)
$$\frac{\Gamma \vdash E_1 : X_1,\ \Gamma \vdash E_2 : X_2}{\Gamma \vdash E_1\ E_2 : \left\langle \begin{array}{l} X_1^u \cup (X_2^u - X_1^l), X_1^n \cup (X_2^n - X_1^l), \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p \cup (X_2^p + X_1^h), X_1^h + X_2^h \end{array} \right\rangle}$$

(Scope)
$$\frac{\Gamma \vdash E : X,\ \mathsf{set}(M) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \{M, E\} : \langle X^u \cup (M - X^d), X^n \cup (M - X^d), X^d - M, X^d - M, X^p, X^h \rangle}$$

$$\begin{array}{l} \texttt{call}p_1 : \langle [c], [], [], [], [], [] \rangle, \\ \texttt{call}p_2 : \langle [c], [c], [], [], [], [] \rangle, \\ \texttt{call}p_3 : \langle [c], [], [], [], [], [] \rangle, \\ \texttt{call}ex : \langle [c], [], [], [], [], [] \rangle \end{array}$$

This signals in the first multiset ($\cdot^u$) of the type of $\texttt{call}ex$ that one instance of c is needed before execution of $\texttt{call}ex$. This is caused by the possible choice of $\texttt{call}p_2$ instead of $\texttt{call}p_1$ by ex, whereby there could be parallel calls to p_4 and p_5. One way to fix this is to instantiate two instances of `C` instead of just one. Then one instance could be passed to `P1` or `P2` and the second to `P3`. This means that P is changed by changing ex into $ex' \prec \texttt{new}\, c\, \texttt{new}\, c\, ((\texttt{call}p_1 + \texttt{call}p_2) \parallel \texttt{call}p_3)\, \texttt{del}\, c$. The type of $\texttt{call}ex'$ is $\langle [], [], [c], [], [], [] \rangle$ which signals that the expression now can be executed starting with an empty store. But the third multiset ($\cdot^l$) signals that there is one instance of c left after execution. This can be fixed by deleting one more instance, that is, changing ex' to $ex'' \prec \texttt{new}\, c\, \texttt{new}\, c\, ((\texttt{call}p_1 + \texttt{call}p_2) \parallel \texttt{call}p_3\, \texttt{del}\, c)\, \texttt{del}\, c$. The type of $\texttt{call}ex''$ is $\langle [], [], [], [], [], [] \rangle$.

Another way of solving the original problem is to remove the parallelism from the program, such that ex is changed to $ex''' \prec \texttt{new}\, c\, (\texttt{call}p_1 + \texttt{call}p_2)\, \texttt{call}p_3\, \texttt{del}\, c$. The type of $\texttt{call}ex'''$ is also $\langle [], [], [], [], [], [] \rangle$.

7 Properties of the Type System

This section contains several basic lemmas about the type system. Proofs in this and the next section are omitted for space considerations. Contact the author for a full version including proofs.

It should be noted again that the type systems in [7,3,4,5] can be used to test whether the deletion of instances is safe, by first translating `use`, `lock` and `free` to `nop`. We can therefore regard only the programs where deletion of instances from the local store is safe.

Lemma 1 (Basics).

1. *If* $\Gamma \vdash E : X$, *then* $\mathsf{var}(E) \subseteq \mathsf{dom}(\Gamma)$.
2. *If* $\vdash P : \Gamma$ *and* $\Gamma \vdash E : X$, *then* $\mathsf{dom}(P) = \mathsf{dom}(\Gamma)$ *and* $-X^u \subseteq -X^n \subseteq X^l$ *and* $X^h \subseteq X^p$.

Lemma 2 (Associativity). *If* $\Gamma \vdash A : X$, $\Gamma \vdash B : Y$ *and* $\Gamma \vdash C : Z$, *then the two ways of typing the expression* $A\,B\,C$ *by the rule* Seq, *corresponding to the different parses* $(A\,B)\,C$ *and* $A\,(B\,C)$, *lead to the same type.*

The following lemma is necessary since the typing rules are not fully syntax-directed. If, e.g., $E_1 = A \cdot B$, then the type of $E_1 \cdot E_2$ could have been inferred by an application of the rule Seq to A and $B\,E_2$. In that case we apply the previous lemma.

Lemma 3 (Inversion).

1. *If* $\vdash P : \Gamma$ *and* $\Gamma(x) = X$, *then there exists a program* P' *and an expression* A *such that* $P', x \prec A$ *is the initial segment of* P *and* $\vdash P' : \Gamma|_{\mathsf{dom}(P')}$ *and* $\Gamma|_{\mathsf{dom}(P')} \vdash A : X$.
2. *If* $\Gamma \vdash \mathtt{new}\,x : X$, *then* $X = \langle \Gamma(x)^u, \Gamma(x)^n, \Gamma(x)^l + x, \Gamma(x)^d, \Gamma(x)^p, \Gamma(x)^h \rangle$.
3. *If* $\Gamma \vdash \mathtt{del}\,x : X$, *then* $X = \langle [x], [x], [-x], [\,], [\,], [\,] \rangle$.
4. *If* $\Gamma \vdash \mathtt{lock}\,N : X$, *then* $X = \langle N, N, -N, -N, [\,], -N \rangle$.
5. *If* $\Gamma \vdash \mathtt{free}\,N : X$, *then* $X = \langle [\,], [\,], N, N, N, N \rangle$.
6. *If* $\Gamma \vdash \mathtt{use}\,N : X$, *then* $X = \langle N, [\,], [\,], [\,], [\,], [\,] \rangle$.
7. *If* $\Gamma \vdash \mathtt{nop} : X$, *then* $X = \langle [\,], [\,], [\,], [\,], [\,], [\,] \rangle$.
8. *For* $\circ \in \{+, \|, \cdot\}$, *if* $\Gamma \vdash (E_1 \circ E_2) : X$, *then there exists* X_i *such that* $\Gamma \vdash E_i : X_i$ *for* $i = 1, 2$. *Moreover,*
   $X = \langle X_1^u \cup X_2^u, X_1^n \cup X_2^n, X_1^l \cap X_2^l, X_1^d, X_1^p \cup X_2^p, X_1^h \cup X_2^h \rangle$ *and* $X_1^d = X_2^d$ *if* $\circ = +$,
   $X = \left\langle \begin{array}{l} (X_1^u + X_2^n) \cup (X_2^u + X_1^n), X_1^n + X_2^n, \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p + X_2^p, X_1^h + X_2^h \end{array} \right\rangle$ *if* $\circ = \|$, *and*
   $X = \left\langle \begin{array}{l} X_1^u \cup (X_2^u - X_1^l), X_1^n \cup (X_2^n - X_1^l), \\ X_1^l + X_2^l, X_1^d + X_2^d, X_1^p \cup (X_2^p + X_1^h), X_1^h + X_2^h \end{array} \right\rangle$ *if* $\circ = \cdot$.
9. *If* $\Gamma \vdash \{M, A\} : X$, *then there exists a type* Y, *such that* $\Gamma \vdash A : Y$ *and* $X = \langle Y^u \cup (M - Y^d), Y^n \cup (M - Y^d), Y^d - M, Y^d - M, Y^p, Y^h \rangle$.

The last lemma in this section is concerned with three forms of uniqueness of the types inferred in the type system. This is necessary in some of the proofs, and for an algorithm for type inference.

Lemma 4 (Uniqueness of types).

1. *If* $\Gamma_1 \vdash E{:}X$, $\Gamma_2 \vdash E{:}Y$ *and* $\Gamma_1|_{\mathsf{var}(E)} = \Gamma_2|_{\mathsf{var}(E)}$, *then* $X = Y$.
2. *If* $\vdash P{:}\Gamma$ *and* $\vdash P{:}\Gamma'$, *then* $\Gamma = \Gamma'$.
3. *If* $\vdash P_1 : \Gamma_1$ *and* $\vdash P_2 : \Gamma_2$ *and* P_2 *is a reordering of* a subset of P_1, *then* $\Gamma_1|_{\mathsf{dom}(P_2)} = \Gamma_2$.

8 Correctness

This section contains lemmas and theorems connecting the type system and the operational semantics. Included are theorems comparable to what is often called preservation and progress, for example in [11]. The following lemma implies that all states in sequences representing the execution of a well-typed program are valid, as defined in Definition 4.

Lemma 5. *If* $\vdash P{:}\Gamma$, $\Gamma \vdash E{:}X$, E *is valid and* $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ *is a step in the operational semantics, then also* E' *is valid.*

The next lemma fixes several properties of two states connected by a single step in the operational semantics. This is used heavily in the main theorems below. The first part is known under the names *subject reduction* and *type preservation.* The remaining parts reflect the fact that every step reduces the set of reachable states. Hence maxima do not increase and minima do not decrease.

Lemma 6 (Invariants). *Let* P *be a component program,* E *a valid expression,* Γ *a basis and* U *a type such that* $\vdash P : \Gamma$, $\Gamma \vdash E : U$, *and* $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ *is a step in the operational semantics. Then we have for some type* V*:*

1. $\Gamma \vdash E'{:}V$.
2. $M'_u - V^u \supseteq M_u - U^u$, *i.e., the safety margin of the global store does not decrease.*
3. $M'_u - V^n \supseteq M_u - U^n$, *i.e., the lower bound on the global store in all reachable states does not decrease.*
4. $M'_u + V^l \supseteq M_u + U^l$, *i.e., the lower bound on the global store in the terminal state does not decrease.*
5. $M'_u - M' + V^d = M_u - M + U^d$, *i.e., the difference between the local and the global store in the terminal state does not change.*
6. $M'_u - \Sigma\{M', E'\} + V^p \subseteq M_u - \Sigma\{M, E\} + U^p$, *i.e., the upper bound on the difference, in any reachable state, between the global store and the sum of the local stores, does not increase.*
7. $M'_u - \Sigma\{M', E'\} + V^h \subseteq M_u - \Sigma\{M, E\} + U^h$, *i.e., the upper bound on the net effect on the difference between the global store and the sum of the local stores does not increase.*

The following Theorem 1 is a combination of several statements which in combination are often called *soundness* or *safety*. Items 1, 2 and 3 are similar to the properties often called *preservation*, *progress* and *termination*, respectively. (See for example [11]). Items 1, 4 and 5 assert that the parts of the types have the meanings given in 5.1.

Theorem 1 (Soundness). *If* $\vdash P : \Gamma$, $\Gamma \vdash E : X$, E *is valid and* $X^u \subseteq M_u$, *then the following holds:*

1. *If* $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ *and* $\Sigma\{M, E\} - M_u \supseteq X^p$, *then there is* Y *such that* $\Gamma \vdash E' : Y$, $M'_u \supseteq Y^u$ *and* $\Sigma\{M', E'\} - M'_u \supseteq Y^p$.
2. *If* E *is not* `nop`, *we have* $(M_u, \{M, E\}) \rightsquigarrow_P (M'_u, \{M', E'\})$ *for some* $(M'_u, \{M', E'\})$.
3. *All* $\rightsquigarrow_P$*-sequences starting in state* $(M_u, \{M, E\})$ *are finite.*
4. *If* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$, *then* $X^l \subseteq M'_u - M_u$, $X^d = (M'_u - M') - (M_u - M)$ *and* $X^h \supseteq (M'_u - M') - (M_u - \Sigma\{M, E\})$.
5. *If* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ *then* $-X^n \subseteq M'_u - M_u$ *and* $X^p \supseteq (M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})$.
6. *All states reachable from* $(M_u, \{M, E\})$ *are safe.*

Finally, we summarize the properties of the type system for well-typed programs, as defined in Definition 5 on page 11. The reader is referred to the paragraph following Definition 5 for an explanation of the three bags required to be empty, to Section 4.1 and Definition 3 for an explanation of safe states, and to Section 4.2 for an explanation of why it is important that $M'_u \subseteq \Sigma\{M', E'\}$.

Corollary 1. *If* $\vdash P : \Gamma$ *and* $\Gamma \vdash \texttt{new}x : X$, *where* x *is the last component declared in* P *and* $X^d = X^u = X^p = [\,]$, *then*

- *All maximal transition sequences starting with* $([\,], \{[\,], \texttt{new}x\})$ *end with* $(M, \{M, \texttt{nop}\})$ *for some bag* M.
- *All states* $(M'_u, \{M', E'\})$ *reachable from* $([\,], \{[\,], \texttt{new}x\})$ *are safe, and such that* $M'_u \subseteq \Sigma\{M', E'\}$.

The following theorem states that the types are *sharp*. Informally, this means, they are as small as they can be, while still guaranteeing safety of execution. The part X^d is not included as it is already stated in Theorem 1 to be exact. The property is formulated differently for the part X^u because of its nature — the other parts contain information about how some of the bags or the difference between them change, while X^u only states the minimum safe size of the bag M_u.

Theorem 2 (Sharpness). *Assume some program* P, *bags* M *and* M_u *and valid expression* E *such that* $\vdash P : \Gamma$ *and* $\Gamma \vdash E : X$ *and* $M_u \subseteq \Sigma\{M, E\}$

1. *If* $M_u \not\supseteq X^u$, *then an unsafe state is reachable from* $(M_u, \{M, E\})$.
2. *If* $M_u \supseteq X^u$*:*
   n For every $y \in \mathbb{C}$ *there exists a state* $(M'_u, \{M', E'\})$ *such that* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ *and* $(M'_u - M_u)(y) = -X^n(y)$.

l *For every* $y \in \mathbb{C}$ *there exists a terminal state* $(M'_u, \{M', \texttt{nop}\})$ *such that* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$ *and* $(M'_u - M_u)(y) = X^l(y)$.

p *For every* $y \in \mathbb{C}$ *there exists a state* $(M'_u, \{M', E'\})$ *such that* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', E'\})$ *and* $(M'_u - \Sigma\{M', E'\}) - (M_u - \Sigma\{M, E\})(y) = X^p(y)$.

h *For every* $y \in \mathbb{C}$ *there exists a terminal state* $(M'_u, \{M', \texttt{nop}\})$ *such that* $(M_u, \{M, E\}) \rightsquigarrow_P^* (M'_u, \{M', \texttt{nop}\})$ *and* $(M'_u - M') - (M_u - \Sigma\{M, E\})(y) = X^h(y)$.

9 Related Work and Conclusion

There is a large amount of work related to similar problems. Most approaches differ from this article by using super-polynomial algorithms, by assuming more on the runtime scheduling of parallel executions, or by treating only memory consumption. For the functional languages, see e.g. [12,13,14,15]. Popea and Chin in [16] also discuss usage in a related way. Their algorithm depends on solving constraints in Presburger arithmetic, which in the worst case uses doubly exponential time. Igarashi and Kobayashi in [17], analyse the *resource usage problem* for an extension of simply typed lambda calculus including resource usage. The algorithm extracts the set of possible traces of usage from the program, and then decides whether all these traces are allowed by the specification. This latter problem is still computationally hard to solve and undecidable in the worst case. Parallel composition is not considered. For the imperative paradigm, which is closer to the system described here, e.g. [18,19,20] treat memory usage. The problem of component usage in a parallel setting is related to prevention of deadlocks and race conditions. Boyapati et al. describe in [21] an explicitly typed system for verifying there are no deadlocks or race conditions in Java programs. In addition to the higher level of detail, the main difference from the system described in this article is the assumptions on the scheduling of parallel executions, namely the ability of a thread to wait until another thread frees/releases a lock. This scheduling has of course a cost in terms of added runtime and of complexity of the implementation.

We have defined a component language with a small-step operational semantics and a type system. The type system combined with the system in [7] or the system in [4] guarantees that the execution of a well-typed program will terminate and cannot reach an unsafe state. The language described in this article is an extension of the language first described in [5], and uses the results from [5,7]. The properties proved in the current article are new, though, and in some ways orthogonal to those shown in [5,7]. The language we introduced is inspired by CCS [6], with the atomic actions interpreted as component instantiation, deallocation and usage. The basic operators are sequential, alternative and parallel composition and a scope operator. The operational semantics is SOS-style [10], with the approach to soundness similar in spirit to [22]. We have presented a type system for this language which predicts sharp bounds of the number of instances of components necessary for safe execution. The type inference algorithm has quadratic runtime.

References

1. McIlroy, D.M.: Mass produced software components. In Naur, P., Randell, B., eds.: Software Engineering: Report of a conference sponsored by the NATO Science Committee, Scientific Affairs Division, NATO (October 1968) 79–87
2. Szyperski, C.: Component Software—Beyond Object–Oriented Programming. 2nd edn. Addison–Wesley / ACM Press (2002)
3. Bezem, M., Truong, H.: A type system for the safe instantiation of components. Electronic Notes in Theoretical Computer Science **97** (2004) 197–217
4. Truong, H.: Guaranteeing resource bounds for component software. In Steffen, M., Zavattaro, G., eds.: FMOODS. Volume 3535 of Lecture Notes in Computer Science., Springer (2005) 179–194
5. Truong, H., Bezem, M.: Finding resource bounds in the presence of explicit deallocation. In Hung, D.V., Wirsing, M., eds.: Proceedings ICTAC. Volume 3722 of Lecture Notes in Computer Science., Springer (2005) 227–241
6. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer (1980)
7. Bezem, M., Hovland, D., Truong, H.: A type system for counting instances of software components. Technical Report 363, Department of Informatics, The University of Bergen, P.O. Box 7800, N-5020 Bergen, Norway (October 2007)
8. Stroustrup, B.: The C++ Programming Language, Third Edition. Addison-Wesley (2000)
9. IEEE: The open group base specifications issue 6 (2004)
10. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (July-December 2004) 17–139
11. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
12. Crary, K., Weirich, S.: Resource bound certification. In: POPL '00: Proceedings of the 27th ACM SIGPLAN–SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2000) 184–198
13. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2003) 185–197
14. Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the π-calculus. Logical Methods in Computer Science **2**(3) (2006)
15. Unnikrishnan, L., Stoller, S.D., Liu, Y.A.: Optimized live heap bound analysis. In: VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, London, UK, Springer-Verlag (2003) 70–85
16. Popeea, C., Chin, W.N.: A type system for resource protocol verification and its correctness proof. In Heintze, N., Sestoft, P., eds.: PEPM, ACM (2004) 135–146
17. Igarashi, A., Kobayashi, N.: Resource usage analysis. ACM Trans. Program. Lang. Syst. **27**(2) (2005) 264–313
18. Braberman, V., Garbervetsky, D., Yovine, S.: A static analysis for synthesizing parametric specifications of dynamic memory consumption. Journal of Object Technology **5**(5) (June 2006) 31–58
19. Chin, W.N., Nguyen, H.H., Qin, S., Rinard, M.C.: Memory usage verification for OO programs. In Hankin, C., Siveroni, I., eds.: SAS. Volume 3672 of Lecture Notes in Computer Science., Springer (2005) 70–86

20. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis (for an object-oriented language). In Sestoft, P., ed.: Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems. Volume 3924 of LNCS., Springer (2006) 22–37
21. Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA. (2002) 211–230
22. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94