REPORTS
IN
INFORMATICS

ISSN 0333-3590

# A Type System for Counting Instances of Software Components

Marc Bezem     Dag Hovland     Hoang Truong

REPORT NO 363     October 2007

*Department of Informatics*
**UNIVERSITY OF BERGEN**
*Bergen, Norway*

This report has URL
`http://www.ii.uib.no/publikasjoner/texrap/pdf/2007-363.pdf`

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at `http://www.ii.uib.no/publikasjoner/texrap/`.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

# A Type System for Counting Instances of Software Components

Marc Bezem Dag Hovland Hoang Truong

**Abstract**

We identify an abstract language for component software based on primitives for instantiating components and for deleting instances of components, as well as for sequential, alternative and parallel composition. We define an operational semantics for our language and give a type system in which the types express quantitative information on the components involved in the execution of the expressions of the language. Included in this information is for each component the maximum number of instances that are simultaneously active during the execution of the expression. The main contribution is a type inference algorithm which runs in time quadratic in the size of the input, whereas executing the expression according to the operational semantics can result in exponentially many runs of exponential length. We consider extensions of the language with loops and tail recursion, and with a scope mechanism. We illustrate the approach with some examples, one on UML diagram refinement and one on counting objects on the free store in C++.

## 1 Introduction

Component software is computer software which has been assembled from standardized, reusable programs called *components*. The fact that components may be manufactured by different third parties adds up to the difficulties one has to ensure basic safety properties, in particular those connected to resources. For example, how to ensure that only one driver of each serial device is used, and only one password generator? How to know that there is enough memory space for all instances of components?

Most of the current approaches to this problem are dynamic in the sense that the running system is programmed to protect itself. For example, a server will deny service to new clients when its workload becomes too high. Another example is the singleton pattern, see [12], which allows at most one object of the class in question to be created.

In this paper we develop techniques for the static analysis (that is, compile time or design time) of component software. As many safety properties actually are undecidable, the abstraction level of our techniques is quite high. They are not meant as a substitute for dynamic techniques, but aim at complementing them.

The static technique we use is based on type theory [3], [19], and our language for component software is based on process theory [18] with interpreted atomic actions for component instantiation and deallocation. For example, a declaration like $x \multimap ((\mathtt{new}\,a + \mathtt{new}\,b) \parallel \mathtt{new}\,c) \cdot \mathtt{del}\,a$ means that the instantiation $\mathtt{new}\,x$ deploys $x$ in a way described by the expression after the $\multimap$-symbol. That is, either $a$ or $b$ is instantiated in parallel to the instantiation of $c$, after which $a$ is deallocated. Clearly, the expression $\mathtt{new}\,x$ is only safe to execute if $b$ and/or $c$ instantiates $a$, and this should follow by inspection of their respective declarations. Even without recursion, such component declarations are non-trivial to analyse on safety issues like: will there always be an instance of $a$ when a deallocation takes place, how many instances of $b$ are simultaneously active during the execution, et cetera.

Since the aim is to count instances we have abstracted from all behaviour of components that doesn't affect component instantiation and deallocation. Also, we do not specify which particular instance is deallocated, we have abstracted from the different identities of instances of component $a$ and are only able to see the number of such instances. (This abstraction is alleviated by a scope mechanism. Writing $\{[\,], E\}$ limits the lifetime of all instances created by $E$ to the execution of $E$ and enforces deallocations by $E$ to apply to these instances only.) We shall show that on this abstraction level estimating the number of instances of components involved in the execution is both feasible and non-trivial.

The operations for component composition we consider are: sequential composition $\cdot$, alternative composition $+$ (also called choice), and parallel composition $\|$. These are well-known process theoretic operators. The primitives for component instantiation/deallocation are `new` and `del` as used in the example above. This basic system will be defined in Section 2. The operations and primitives will get a precise meaning by their operational semantics, given in Section 3. Types will be introduced in Section 4, and their basic properties, including quadratic-time type inference, will be proved in Section 5. In Section 6 we prove soundness of the type system with respect to the operational semantics. In Section 7 we consider extensions of the basic system with loops and tail recursion (7.1), to deal with memory usage (7.3), and with a scope operator (7.4). Elaborated examples can be found in Section 4.3 and 7.2. We conclude in Section 8, after a short review of related work.

# 2 Basic System

## 2.1 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 $\mathbb{C}$. *Component expressions* are given by the following syntax. We let capital letters $A, \ldots, E$ (with primes and subscripts) range over *Expr*. The 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 $\|$ are also associative and we only parenthesize to prevent ambiguity. Sequential composition has the highest precedence, followed by $\|$ 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 *state expressions*, to be used in the operational semantics in the next section. A state expression is a pair of a bag (see Section 3.1) and an expression where the latter may be `nop`, in which case the state is *terminal*.

Table 1: Syntax

$$\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Factor} \mid \mathit{Expr} \cdot \mathit{Expr} \\
\mathit{Factor} & ::= & \texttt{new}\,x \mid \texttt{del}\,x \mid (\mathit{Expr} + \mathit{Expr}) \mid (\mathit{Expr} \parallel \mathit{Expr}) \mid \texttt{nop} \\
\mathit{StExp} & ::= & \{M, \mathit{Expr}\} \quad \text{(for any bag $M$ of elements from $\mathbb{C}$)} \\
\mathit{Prog} & ::= & \texttt{nil} \mid \mathit{Prog}, x \mathrel{-\!\!\prec} \mathit{Expr}
\end{array}$$

By $\mathsf{var}(E)$ we denote the set of component names occurring in $E$, formally defined by $\mathsf{var}(\texttt{nop}) = \varnothing$, $\mathsf{var}(\texttt{new}\,x) = \mathsf{var}(\texttt{del}\,x) = \{x\}$, $\mathsf{var}(E_1 + E_2) = \mathsf{var}(E_1 \parallel E_2) = \mathsf{var}(E_1\,E_2) = \mathsf{var}(E_1) \cup \mathsf{var}(E_2)$. The *size* of an expression $E$, denoted $\sigma(E)$, is defined by $\sigma(\texttt{new}\,x) = \sigma(\texttt{del}\,x) = \sigma(\texttt{nop}) = 1$ and $\sigma(A + B) = \sigma(AB) = \sigma(A||B) = \sigma(A) + \sigma(B) + 1$.

A *component program* $P$ is a comma-separated list starting with `nil` and followed by zero or more *component declarations* of the form $x \mathrel{-\!\!\prec} \mathit{Expr}$, 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 $x \multimap \mathtt{nop}$ are used for *primitive* components, i.e., components that do not use *subcomponents*. The size of a program $P$, denoted $\sigma(P)$, is defined by $\sigma(P, x \multimap A) = \sigma(P) + 1 + \sigma(A)$ and $\sigma(\mathtt{nil}) = 1$.

## 2.2 Small examples

Examples of component programs that will be well-typed (see Section 4) are:

$a \multimap \mathtt{nop}, b \multimap \mathtt{new}\,a \cdot \mathtt{del}\,a$ ($b$ creates an instance of primitive component $a$ and then deletes an instance of $a$);

$a \multimap \mathtt{nop}, b \multimap \mathtt{nop}, c \multimap (\mathtt{new}\,a \parallel \mathtt{new}\,b) \cdot (\mathtt{del}\,a + \mathtt{del}\,b)$ ($c$ creates in parallel an instance of $a$ and one of $b$, and then deletes an instance of either $a$ or $b$);

$a \multimap \mathtt{nop}, b \multimap \mathtt{new}\,a \cdot \mathtt{new}\,a, c \multimap \mathtt{new}\,b \cdot \mathtt{new}\,b$.

We adopt the convention that a component program $P$ not equal to $\mathtt{nil}$ is executed by executing $\mathtt{new}\,x$, where $x$ is the last component declared in $P$. In the last example, $\mathtt{new}\,c$ creates two instances of $b$ which each create two instances of $a$, so four in total. This shows that the execution of a component program (see Section 3.2) can be exponential in the size of the program, even for programs without $+, \parallel$. Programs with $\cdot$ and $+$, or with $\cdot$ and $\parallel$, can be executed in exponentially many different ways, and each of these may have exponential length. This means that it is in general not an option to run the program and see what happens. We have to prove, however, that the static analysis we propose can be done in reasonable time.

Examples of component programs that when executed either will not terminate or might lead to errors:

$a \multimap \mathtt{new}\,a$ (circular);

$b \multimap \mathtt{new}\,a$ ($a$ not declared);

$a \multimap \mathtt{nop}, b \multimap \mathtt{nop}, b \multimap \mathtt{new}\,a$ ($b$ declared twice);

$a \multimap \mathtt{nop}, b \multimap \mathtt{del}\,a$ ($b$ deletes non-existing instance of $a$);

$a \multimap \mathtt{nop}, b \multimap (\mathtt{new}\,a + \mathtt{del}\,a)$ ($b$ deletes non-existing instance of $a$ in one branch);

$a \multimap \mathtt{nop}, b \multimap (\mathtt{new}\,a \parallel \mathtt{del}\,a)$ ($b$ deletes non-existing instance of $a$ if $\mathtt{del}\,a$ is executed before $\mathtt{new}\,a$).

# 3 Operational Semantics

## 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: $(M \cup M')(x) = \max(M(x), M'(x))$, $(M \cap M')(x) = \min(M(x), M'(x))$, $(M + M')(x) = M(x) + M'(x)$, $(M - M')(x) = M(x) - M'(x)$, $M \subseteq M'$ 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 $M$ (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, $\mathsf{set}(M) = \{x \in \mathbb{C} \mid M(x) > 0\}$.

### 3.2 Operational Semantics of Basic System

A *state* is a pair $\{M, E\}$ consisting of a bag $M$ with underlying set of elements $\mathbb{C}$, and an expression $E$. The expression may be $\mathtt{nop}$, in which case $\{M, \mathtt{nop}\}$ is called a *terminal state*. An *initial state* is of the form $\{[\,], \mathtt{new}\,x\}$. A state expresses that we execute $E$ with a bag $M$ of instances of components. The operational semantics is given as a state transition system in the style of structured operational semantics [20]. In Table 2 we list the transition rules. The inductive rules are $\mathsf{osPar1}, 2$ and $\mathsf{osSeq}$. The other rules are not inductive, but $\mathsf{osNew}$ and $\mathsf{osDel}$ are conditional with the condition specified as a premiss of the rule.

Table 2: Transition rules for a component program $P$

(osNew)
$$\frac{x \mathrel{-\!\!\prec} A \in P}{\{M, \mathtt{new}\,x\} \rightsquigarrow_P \{M + x, A\}}$$

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

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

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

(osAlt1)
$$\frac{}{\{M, (E_1 + E_2)\} \rightsquigarrow_P \{M, E_1\}}$$

(osPar1)
$$\frac{\{M, E_1\} \rightsquigarrow_P \{M', E_1'\}}{\{M, (E_1 \parallel E_2)\} \rightsquigarrow_P \{M', (E_1' \parallel E_2)\}}$$

(osAlt2)
$$\frac{}{\{M, (E_1 + E_2)\} \rightsquigarrow_P \{M, E_2\}}$$

(osPar2)
$$\frac{\{M, E_2\} \rightsquigarrow_P \{M', E_2'\}}{\{M, (E_1 \parallel E_2)\} \rightsquigarrow_P \{M', (E_1 \parallel E_2')\}}$$

Terminal state is $\{M, \mathtt{nop}\}$

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

# 4 Type System

## 4.1 Types

Since we are interested in the maximum number of simultaneously active instances during the execution of an expression $E$, it is natural to use a bag with underlying set $\mathbb{C}$ as the type of $E$. However, we want typing to be compositional, that is, the type of $E\,E'$ should be expressed in the types of the subexpressions $E$ and $E'$. Due to $\mathtt{del}$, the highest number of simultaneously active instances *during* execution of $E$ can be greater than the highest number of instances which are still allocated *after* execution of $E$. Consider a sequence of transitions $\{[\,], E\,E'\} \rightsquigarrow_P^* \{M, E'\}$ for some program $P$. The highest number of simultaneously active instances during this particular execution of $E\,E'$ depends on $M$ and $M$ cannot be predicted without extra information in the type of $E$.

The solution is to include also the highest net increase in number of instances after execution in the type of an expression. As the latter number may be negative, this should be a multiset in the sense of Section 3.1.

Dually, since we are interested in safe deallocation, we need to know for each component the highest *negative* net change, that is, the maximum decrease, of instances during the execution of an expression, for which we use another bag in the type. For maintaining compositionality we then also have to include a multiset for the lowest increase in instances after execution in the type. This multiset is of interest also since it can signal a memory leak. The minimum and the maximum after execution can be different because of the choice operator.

A *type* of a component expression is a quadruple $X = \langle X^n, X^p, X^l, X^h \rangle$, where $X^n$ and $X^p$ are bags and $X^l$ and $X^h$ are multisets. The bags $X^n$ and $X^p$ contain, for each $x \in \mathbb{C}$, the highest *negative* and highest *positive* net change, respectively, in the number of instances during the execution of the expression. This implies that, if the type of $E$ is $X$ and if $\{M, E\} \rightsquigarrow_P^* \{M', E'\}$, then $-X^n \subseteq M' - M \subseteq X^p$.

The multisets $X^l$ and $X^h$ contain, for each $x \in \mathbb{C}$, the *lowest* and the *highest* net increase in the number of instances, respectively, after the execution of the expression. This implies that, if the type of $E$ is $X$ and if $\{M, E\} \rightsquigarrow_P^* \{M', \texttt{nop}\}$, then $X^l \subseteq M' - M \subseteq X^h$. For example, with $a$ a primitive component, the type of $(\texttt{new}\,a\,\texttt{del}\,a) + \texttt{del}\,a$ is $\langle [a], [a], [-a], [\,] \rangle$.

We use $U, \ldots, Z$ to denote types. We extend $+$ from multisets to types, such as done in the rule $\mathsf{Par}$ in Table 3: $X_1 + X_2$ is the type $\langle X_1^n + X_2^n, X_1^p + X_2^p, X_1^l + X_2^l, X_1^h + X_2^h \rangle$. For a type $X$, $\mathsf{var}(X) = \mathsf{set}(X^n) \cup \mathsf{set}(X^p)$.

## 4.2 Typing rules

With the above interpretation in mind the typing rules in Table 3 are easily understood. They define a ternary typing relation $\Gamma \vdash E : X$ and a binary typing relation $\vdash P : \Gamma$ in the usual inductive way. Here $\Gamma$ is usually called a *basis*, mapping variables to types. In the relation $\vdash P : \Gamma$, $\Gamma$ can be viewed as the type of $P$. In this way we avoid duplicating type derivations for variables occurring more than once in expressions. 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 $\Gamma$' or 'program $P$ has type $\Gamma$', respectively.

A basis $\Gamma$ is a partial mapping of components $x \in \mathbb{C}$ to types. By $\mathsf{dom}(\Gamma)$ we denote the domain of $\Gamma$, and for any $x \in \mathsf{dom}(\Gamma)$, $\Gamma(x)$ denotes its type in $\Gamma$. For a set $S \subseteq \mathsf{dom}(\Gamma)$, $\Gamma|_S$ is $\Gamma$ restricted to the domain $S$. For any $x \in \mathbb{C}$ and type $X$, $\{x \mapsto X\}$ denotes a basis with domain $\{x\}$ mapping $x$ to $X$.

An expression $E$ is called *typable* in $\Gamma$ 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 $\Gamma(E)$.

As an easy example, recall the last program in Section 2, $P = \texttt{nil}, a \multimap \texttt{nop}, b \multimap (\texttt{new}\,a \parallel \texttt{del}\,a)$. One easily infers $\vdash P : \Gamma$ with $\Gamma(a) = \langle [\,], [\,], [\,], [\,] \rangle$, $\Gamma(b) = \langle [a], [a], [\,], [\,] \rangle$, and $\Gamma \vdash \texttt{new}\,b : \langle [a], [a, b], [b], [b] \rangle$. The type of $\texttt{new}\,b$ signals a possible deficit by the bag $[a]$ (arises when $\texttt{del}\,a$ is scheduled before $\texttt{new}\,a$), the fact that $a$ and $b$ can be simultaneously active (the bag $[a, b]$), as well as a memory leak by the bags $[b]$ (caused by not deleting $b$). More examples of typings can be found in the next subsection.

## 4.3 Example: refining UML activity diagrams

In this section we show how UML activity diagrams can be analysed/refined with our techniques. We abbreviate $\texttt{new}\,f\,\texttt{del}\,f$ by $\texttt{call}\,f$ and use this expression to model a function

Table 3: Typing rules

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

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

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

(Par)
$$\frac{\Gamma \vdash E_1\!:\!X_1,\ \Gamma \vdash E_2\!:\!X_2}{\Gamma \vdash E_1 \parallel E_2\ :\ X_1 + X_2}$$

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

(Seq)
$$\frac{\Gamma \vdash E_1\!:\!X_1,\ \Gamma \vdash E_2\!:\!X_2}{\Gamma \vdash E_1\,E_2\ :\ \langle X_1^n \cup (X_2^n - X_1^l), X_1^p \cup (X_2^p + X_1^h), X_1^l + X_2^l, X_1^h + X_2^h\rangle}$$

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

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

call. Note that $f$ is deleted automatically by $\mathtt{call}\,f$, but not the subcomponents that $f$ possibly instantiates.

Following [21, Ch. 8], we think ourselves a theatre box with members and non-members. Members can order seats that will be paid by charging an account that comes with membership, and members will then earn some bonuspoints. Non-members pay by credit card and do not get bonuspoints. The design of this ordering procedure can be described by the simple UML-diagram in Figure 1. Arrows indicate control flow. Rounded boxes are used for actions and rhombic boxes delimit branching. Actions can be refined, that is, described by other UML diagrams. Since there is no specific order between the actions in the left branch, they are assumed parallel, which is expressed by the horizontal fat bars. In the right branch the actions are serialized. The expression corresponding to this set-up is $(\mathtt{call}\,a \parallel \mathtt{call}\,b \parallel \mathtt{call}\,d) + \mathtt{call}\,c\,\mathtt{call}\,a$.

Let us assume that the actions $a, b, d$ involve one and the same database, whereas $c$ is primitive. Then we can refine: $a -\!\!\!\prec \mathtt{new}\,db\,\mathtt{call}\,a'\,\mathtt{del}\,db$, $b -\!\!\!\prec \mathtt{new}\,db\,\mathtt{call}\,b'\,\mathtt{del}\,db$, $d -\!\!\!\prec \mathtt{new}\,db\,\mathtt{call}\,d'\,\mathtt{del}\,db$. Here $db$ is a primitive component for accessing the database, and $a', b', d'$ are the database transactions for assigning seats, awarding the bonus and debiting the account, respectively. If we then add $o -\!\!\!\prec (\mathtt{call}\,a \parallel \mathtt{call}\,b \parallel \mathtt{call}\,d) + \mathtt{call}\,c\,\mathtt{call}\,a$ we get the following typing: $\mathtt{call}\,o\!:\!\langle[\,],[a,a',b,b',c,d,d',3db,o],[\,],[\,]\rangle$. We see from the type that there is a possibility of three parallel database connections. If this is undesirable, the parallel composition should be changed into a sequential one:
$o' -\!\!\!\prec \mathtt{call}\,a\,\mathtt{call}\,b\,\mathtt{call}\,d + \mathtt{call}\,c\,\mathtt{call}\,a$, with the following typing:
$\mathtt{call}\,o'\!:\!\langle[\,],[a,a',b,b',c,d,d',db,o'],[\,],[\,]\rangle$.

Now assume connecting to the database is an expensive operation. How often the database is opened can be analysed by changing $db$ from a primitive component into $db -\!\!\!\prec \mathtt{new}\,db'$ with $db'$ a primitive component that represents opening the database. Note that we have not written $\mathtt{call}\,db'$ but $\mathtt{new}\,db'$, which means that instances of $db'$ are not deleted. The maximum number of times the database is opened is now accounted for in the new typing $\mathtt{call}\,o'\!:\!\langle[\,],[a,a',b,b',c,d,d',db,3db',o'],[db'],[3db']\rangle$. Assume opening the database many times should be avoided. One could then consider redefining $a, b, d$ in the

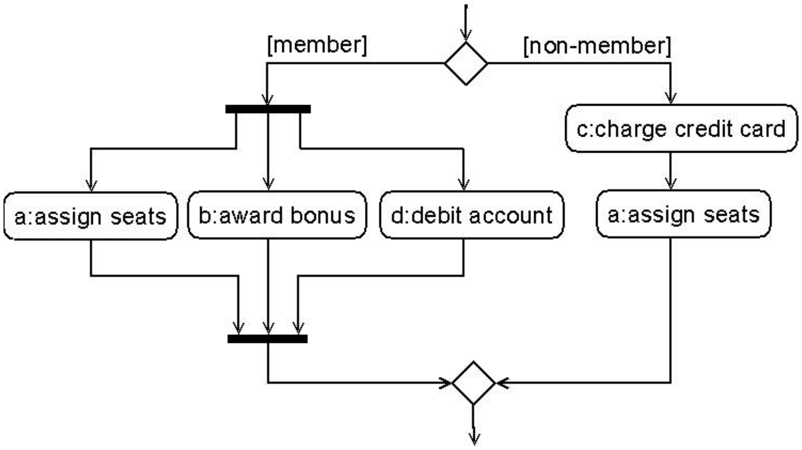

Figure 1: A UML activity diagram for ordering seats

following way: $a \mathrel{-\!\!\prec} \texttt{new}\, db\, \texttt{call}\, a'$, $b \mathrel{-\!\!\prec} \texttt{call}\, b'$, $d \mathrel{-\!\!\prec} \texttt{call}\, d'\, \texttt{del}\, db$. Although this is fine for the left branch, it is wrong for the right branch. The new typing signals what is wrong: $\texttt{call}\, o' : \langle [\,], [a, a', b, b', c, d, d', db, db', o'], [db'], [db, db'] \rangle$. The occurrence of $db$ in the last multiset is caused by the fact that $db$ is not deleted in the right branch. We finish with the refinement which is probably the most economical: $\texttt{nil}$, $a' \mathrel{-\!\!\prec} \texttt{nop}$, $b' \mathrel{-\!\!\prec} \texttt{nop}$, $c \mathrel{-\!\!\prec} \texttt{nop}$, $d' \mathrel{-\!\!\prec} \texttt{nop}$, $db \mathrel{-\!\!\prec} \texttt{nop}$, $o'' \mathrel{-\!\!\prec} \texttt{new}\, db\, \texttt{call}\, a'\, \texttt{call}\, b'\, \texttt{call}\, d'\, \texttt{del}\, db + \texttt{call}\, c\, \texttt{new}\, db\, \texttt{call}\, a'\, \texttt{del}\, db$, with typing $\texttt{call}\, o'' : \langle [\,], [a', b', c, d', db, o''], [\,], [\,] \rangle$.

# 5 Basic Properties of the typing system

## 5.1 Uniqueness of types

In this section we will prove several useful lemmas leading to the uniqueness of types. The following lemma will be used frequently without explicit mentioning.

**Lemma 5.1 (Basics)**

1. *An expression $E$ is typable in a basis $\Gamma$ if and only if* $\mathsf{var}(E) \subseteq \mathsf{dom}(\Gamma)$.
2. *If* $\vdash P : \Gamma$ *and* $\Gamma \vdash E : X$, *then* $\mathsf{dom}(P) = \mathsf{dom}(\Gamma)$, $-X^n \subseteq X^l \subseteq X^h \subseteq X^p$ *and* $\mathsf{var}(E) \subseteq \mathsf{var}(X)$.

PROOF:

1. By two easy inductions, one on the size of $E$ (the if-part) and one on the derivation of $\Gamma \vdash E : X$ (the only-if-part).
2. By induction on $\vdash P : \Gamma$ one proves $\mathsf{dom}(P) = \mathsf{dom}(\Gamma)$. The second part requires a double induction, the primary induction on the length of $\Gamma$ and a secondary induction on the derivation $\Gamma \vdash E : X$. The primary base case, $\Gamma = \emptyset$ and $E = \texttt{nop}$ is trivial. Now let $\vdash P : \Gamma$, $\Gamma \vdash E : X$ for some non-empty $\Gamma$ and assume the result has been

proven for all shorter bases. We prove $-X^n \subseteq X^l \subseteq X^h \subseteq X^p$ by a secondary induction on the derivation of $\Gamma \vdash E : X$. The secondary base cases $E = \texttt{nop}$ and $E = \texttt{del}x$ are trivial. Consider the base case $E = \texttt{new}x$ with $\Gamma(x) = X'$ for some $X'$. Then $\Gamma' \vdash E' : X'$ for some $\Gamma' \subset \Gamma$ with $x \multimap E' \in P$. Now we can apply the primary induction hypothesis to $\Gamma'$ and the result for $X$ follows from that of $X'$. The secondary induction steps require many easy calculations. We do two and leave the others to the reader. Assume $-X_1^n \subseteq X_1^l$, $-X_2^n \subseteq X_2^l$. Then $-(X_1^n \cup X_2^n) = (-X_1^n) \cap (-X_2^n) \subseteq X_1^l \cap X_2^l$ and $-(X_1^n \cup (X_2^n - X_1^l)) = (-X_1^n) \cap (-X_2^n + X_1^l) \subseteq X_1^l + X_2^l$. Finally, one proves $\mathsf{var}(E) \subseteq \mathsf{var}(X)$ by induction on $\Gamma \vdash E : X$.

□

**Lemma 5.2 (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 $\mathsf{Seq}$, corresponding to the different parses $(A\,B)\,C$ and $A\,(B\,C)$, lead to the same type.*

PROOF: By applying $\mathsf{Seq}$ to $\Gamma \vdash A : X$ and $\Gamma \vdash B : Y$ we get

$$\Gamma \vdash A\,B : \langle X^n \cup (Y^n - X^l), X^p \cup (Y^p + X^h), X^l + Y^l, X^h + Y^h \rangle$$

and combining this with $\Gamma \vdash C : Z$ we get $\Gamma \vdash A\,B\,C : \langle (X^n \cup (Y^n - X^l)) \cup (Z^n - (X^l + Y^l)), (X^p \cup (Y^p + X^h)) \cup (Z^p + (X^h + Y^h)), (X^l + Y^l) + Z^l, (X^h + Y^h) + Z^h \rangle$. By applying $\mathsf{Seq}$ to $\Gamma \vdash B : Y$ and $\Gamma \vdash C : Z$ we get

$$\Gamma \vdash B\,C : \langle Y^n \cup (Z^n - Y^l), Y^p \cup (Z^p + Y^h), Y^l + Z^l, Y^h + Z^h \rangle$$

and combining this with $\Gamma \vdash A : X$ we get $\Gamma \vdash A\,B\,C : \langle X^n \cup ((Y^n \cup (Z^n - Y^l)) - X^l), X^p \cup ((Y^p \cup (Z^p + Y^h)) + X^h), X^l + (Y^l + Z^l), X^h + (Y^h + Z^h) \rangle$. It remains to prove that the two types resulting from the combination are equal. For the last two parts of the quadruples this trivially follows from the associativity of $+$ for multisets. For the first parts of the types this follows from the associativity of $\cup$ and the distributivity of $+$ and $-$ over $\cup$. □

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

**Lemma 5.3 (Inversion)**

1. *If $\vdash P : \Gamma$ and $\Gamma(x) = X$, then there exists a program $P'$ such that $P', x \multimap A$ is an initial segment of $P$ and $\vdash P' : \Gamma|_{\mathsf{dom}(P')}$ and $\Gamma|_{\mathsf{dom}(P')} \vdash A : X$.*

2. *If $\Gamma \vdash \texttt{new}x : X$, then $X = \langle \Gamma(x)^n, \Gamma(x)^p + x, \Gamma(x)^l + x, \Gamma(x)^h + x \rangle$.*

3. *If $\Gamma \vdash \texttt{del}x : X$, then $X = \langle [x], [\,], [-x], [-x] \rangle$.*

4. *If $\Gamma \vdash \texttt{nop} : X$, then $X = \langle [\,], [\,], [\,], [\,] \rangle$.*

5. *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,*
   *$X = \langle X_1^n \cup X_2^n, X_1^p \cup X_2^p, X_1^l \cap X_2^l, X_1^h \cup X_2^h \rangle$ if $\circ = +$,*
   *$X = X_1 + X_2$ if $\circ = \|$, and*
   *$X = \langle X_1^n \cup (X_2^n - X_1^l), X_1^p \cup (X_2^p + X_1^h), X_1^l + X_2^l, X_1^h + X_2^h \rangle$ if $\circ = \cdot$.*

PROOF: We first prove the first part by an easy induction on $\vdash P : \Gamma$. The base case $\mathsf{AxmP}$ is trivial, and in the induction case we have the following application of the rule $\mathsf{Prog}$:

$$\frac{\Gamma' \vdash A : Z,\ \vdash P' : \Gamma',\ y \notin \mathsf{dom}(\Gamma')}{\vdash P', y \prec A : \Gamma' \cup \{y \mapsto Z\}}$$

If $x = y$ we have the result from the rule application. Otherwise we can apply the induction hypothesis to the premiss $\vdash P' : \Gamma'$.

The other parts are proved by structural induction on the derivation of $\Gamma \vdash E : X$. The base cases $\mathsf{Axm}$, $\mathsf{Del}$ and $\mathsf{New}$ and the induction cases $\mathsf{Alt}$ and $\mathsf{Par}$ are obvious (no need for the induction hypothesis). The only interesting case is the rule $\mathsf{Seq}$, which has three subcases. Consider the conclusion $\Gamma \vdash E_1\, E_2 : X$. If this has been inferred by an application of $\mathsf{Seq}$ with premises $\Gamma \vdash E_i : X_i$ for $i = 1, 2$ we are done (no need for the induction hypothesis). However, it is possible that $E_1 = A\, B$ and that $\mathsf{Seq}$ is applied to $A$ and $B\, E_2$. The third case, $E_2 = B\, C$ and $\mathsf{Seq}$ applied to $E_1\, B$ and $C$, follows by symmetry. So let $E_1 = A\, B$ and consider the following application of the rule $\mathsf{Seq}$.

$$\frac{\Gamma \vdash A : Y_1, \Gamma \vdash B\, E_2 : Y_2}{\Gamma \vdash E_1\, E_2\ :\ \langle Y_1^n \cup (Y_2^n - Y_1^l), Y_1^p \cup (Y_2^p + Y_1^h), Y_1^l + Y_2^l, Y_1^h + Y_2^h \rangle}$$

The type in the conclusion is the type $X$ for which we have to find types $X_i$ such that $\Gamma \vdash E_i : X_i$ for $i = 1, 2$, and

$$X = \langle X_1^n \cup (X_2^n - X_1^l), X_1^p \cup (X_2^p + X_1^h), X_1^l + X_2^l, X_1^h + X_2^h \rangle$$

By the induction hypothesis applied to $\Gamma \vdash B\, E_2 : Y_2$ we get types $Z$ and $X_2$ such that $\Gamma \vdash B : Z$ and $\Gamma \vdash E_2 : X_2$. By applying $\mathsf{Seq}$ to $\Gamma \vdash A : Y_1$ and $\Gamma \vdash B : Z$ we get a type $X_1$ such that $\Gamma \vdash E_1 : X_1$. It follows by Lemma 5.2 that

$$\begin{aligned}&\langle X_1^n \cup (X_2^n - X_1^l), X_1^p \cup (X_2^p + X_1^h), X_1^l + X_2^l, X_1^h + X_2^h \rangle = \\ &\langle Y_1^n \cup (Y_2^n - Y_1^l), Y_1^p \cup (Y_2^p + Y_1^h), Y_1^l + Y_2^l, Y_1^h + Y_2^h \rangle.\end{aligned}$$

— □

**Lemma 5.4 (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* $\Gamma = \Gamma'$.

PROOF:

1. By structural induction on the derivation of $\Gamma_1 \vdash E : X$. In the case of the rule $\mathsf{Axm}$, $\mathsf{Del}$ and $\mathsf{New}$ we have that $E = \mathtt{nop}$, $E = \mathtt{del}x$, and $E = \mathtt{new}x$, respectively. In all three cases $X = Y$ follows by applying the Inversion Lemma 5.3 to $\Gamma_2 \vdash E : Y$.

   Assume $\Gamma_1 \vdash E : X$ is inferred by the following application of the rule $\mathsf{Par}$:

   $$\frac{\Gamma_1 \vdash E_1 : X_1,\ \Gamma_1 \vdash E_2 : X_2}{\Gamma_1 \vdash E_1 \parallel E_2\ :\ X_1 + X_2}$$

   Applying the Inversion Lemma to $\Gamma_2 \vdash E_1 \parallel E_2\ :\ Y$ gives types $Y_i$ such that $\Gamma_2 \vdash E_i : Y_i$ and $Y = Y_1 + Y_2$. Now we apply the induction hypothesis to the premises $\Gamma_2 \vdash E_i : X_i$ and get $X_i = Y_i$ for $i = 1, 2$. It follows that $X = Y$.

   The cases of the rules $\mathsf{Alt}$ and $\mathsf{Seq}$ are analogous to the case of $\mathsf{Par}$.

2. The base case $\mathsf{AxmP}$ is trivial. Assume therefore $\vdash P : \Gamma$ is inferred by the following application of the rule $\mathsf{Prog}$:

$$\frac{\Gamma_1 \vdash E : X,\ \vdash P_1 : \Gamma_1,\ x \notin \mathsf{dom}(\Gamma_1)}{\vdash P_1, x \multimap E : \Gamma_1 \cup \{x \mapsto X\}}$$

By applying Lemma 5.3 to $\vdash P : \Gamma'$ and then the induction hypothesis we get $\Gamma_1 = \Gamma'|_{\mathsf{dom}(P_1)}$. Hence we get by part 1 that $\Gamma(x) = \Gamma(E) = \Gamma'(E) = \Gamma'(x)$, so $\Gamma = \Gamma'$.

□

## 5.2 Type inference

Type inference means to compute, for a given component program $P$ and expression $E$, types $\Gamma$ and $X$ such that $\vdash P : \Gamma$ and $\Gamma \vdash E : X$ if there are such types, and to report failure otherwise. This may require reordering $P$, a task that should not burden the programmer. We should then prove that the type, if it exists, is independent of the specific reordering used. We prepare reordering with a lemma.

**Lemma 5.5** *For any program $P$, the following are equivalent:*

1. $\vdash P : \Gamma$ *for some* $\Gamma$*;*
2. *Every $x$ is declared at most once in $P$ and for every initial segment $P', x \multimap A$ of $P$ we have that* $\mathsf{var}(A) \subseteq \mathsf{dom}(P')$.

PROOF: For proving that 1 implies 2, assume 1 and let $P', x \multimap A$ be an initial segment of $P$. At some point in the derivation of $\vdash P : \Gamma$, $P'$ is extended to $P', x \multimap A$ by the following application of $\mathsf{Prog}$:

$$\frac{\vdash P' : \Gamma',\ \Gamma' \vdash A : X,\ x \notin \mathsf{dom}(\Gamma')}{\vdash P', x \multimap A : \Gamma' \cup \{x \mapsto X\}}$$

for some $\Gamma'$ and $X$. By the premiss and Lemma 5.1 it follows that $\mathsf{var}(A) \subseteq \mathsf{dom}(P')$ and that $x \notin \mathsf{dom}(P')$.

It remains to prove that 2 implies 1. This will be done by induction on the length of $P$. The base case `nil` is typed by $\mathsf{AxmP}$. Assume $P = P', x \multimap A$ satisfies 2. Then also $P'$ satisfies these conditions, so by the induction hypothesis $\vdash P' : \Gamma'$ for some $\Gamma'$. Since $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma')$ we can by Lemma 5.1 infer $\Gamma' \vdash A : X$ for some type $X$. By $\mathsf{Prog}$ we conclude that $\vdash P', x \multimap A : \Gamma' \cup \{x \mapsto X\}$. □

Part 2 of the above lemma partially specifies the ordering in $P$. For example, if $P$ is $\mathtt{nil}, x \multimap \mathtt{new}\, z, y \multimap \mathtt{new}\, z, z \multimap \mathtt{nop}$ then both $P_1 = \mathtt{nil}, z \multimap \mathtt{nop}, x \multimap \mathtt{new}\, z, y \multimap \mathtt{new}\, z$ and $P_2 = \mathtt{nil}, z \multimap \mathtt{nop}, y \multimap \mathtt{new}\, z, x \multimap \mathtt{new}\, z$ satisfy 2. The following strengthening of Lemma 5.4 proves that in general types do not depend on the ordering chosen.

**Lemma 5.6 (Strong Uniqueness)** *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* $\Gamma_1|_{\mathsf{dom}(P_2)} = \Gamma_2$.

PROOF: Let conditions be as above. We use induction on the derivation of $\vdash P_2 : \Gamma_2$. The base case is $P_2 = \mathtt{nil}$, in which case $\Gamma_2 = \Gamma_1|_\varnothing = \varnothing$. For the induction case, assume $\vdash P_2 : \Gamma_2$ has been inferred by the following application of the rule $\mathsf{Prog}$:

$$\frac{\vdash P_2' : \Gamma_2',\ \Gamma_2' \vdash E : \Gamma_2(x),\ x \notin \mathsf{dom}(\Gamma_2')}{\vdash P_2', x \multimap E : \Gamma_2' \cup \{x \mapsto \Gamma_2(x)\}}$$

Since $x \in \mathsf{dom}(\Gamma_1)$ we get by the Inversion Lemma 5.3 that there is $P_1'$ such that $P_1', x \multimap E$ is an initial segment of $P_1$ and for $\Gamma_1' = \Gamma_1|_{\mathsf{dom}(P_1')}$ that $\vdash P_1' : \Gamma_1'$ and $\Gamma_1' \vdash E :$

$\Gamma_1(x)$. Since $\Gamma_2' \vdash E : \Gamma_2(x)$ the Basics Lemma 5.1 implies $\mathsf{var}(E) \subseteq \mathsf{dom}(P_2')$. Since $\mathsf{dom}(P_1) \supset \mathsf{dom}(P_2')$ we have from the Basics Lemma 5.1 and the Uniqueness Lemma 5.4 that $\Gamma_1|_{\mathsf{dom}(P_2')} \vdash E : \Gamma_1(x)$. Since $P_2'$ is a reordering of a subset of $P_1$ and $\vdash P_2' : \Gamma_2'$, the induction hypothesis gives us $\Gamma_1|_{\mathsf{dom}(P_2')} = \Gamma_2'$, so again from the Uniqueness Lemma 5.4 we get $\Gamma_1(x) = \Gamma_2(x)$. This yields $\Gamma_1|_{\mathsf{dom}(P_2)} = \Gamma_2$. □

**Theorem 5.7 (Type inference)** *There exists an algorithm that, given a component program $P$ and an expression $E$, does the following:*

1. *First program $P$ is reordered to satisfy part 2 in Lemma 5.5. If $P$ cannot be reordered in such a way, or if $\mathsf{var}(E) \not\subseteq \mathsf{dom}(P)$, the algorithm reports a failure.*
2. *In the second phase, assuming that $P$ has successfully been reordered and that $\mathsf{var}(E) \not\subseteq \mathsf{dom}(P)$, a basis $\Gamma$ and a type $X$ are computed such that $\vdash P : \Gamma$ and $\Gamma \vdash E : X$.*

*The algorithm works in time $O(\sigma(P)^2 + \sigma(E)^2)$. The types $X$ and $\Gamma$ in phase 2 are unique if they exist.*

PROOF: After assuring there is at most one declaration of each component, phase 1 can easily be done by a topological sort [17] of the directed graph with nodes $\mathsf{dom}(P)$ and edges from $y$ to $x$ if and only if there exists a declaration $x \multimap A$ in $P$ such that $y$ occurs in $A$.

For phase 2, $\vdash P : \Gamma$ and then $\Gamma \vdash E : X$ can be inferred in the type system with inference trees linear in the size of $E$ and $P$, respectively. As the multiset operations are in linear time the whole phase takes quadratic time.

The algorithm reports failure if $P$ cannot be reordered or if $\mathsf{var}(E) \not\subseteq \mathsf{dom}(\Gamma)$. $\Gamma$ and $X$ are independent of the particular reordering of $P$ by Lemma 5.6. □

# 6 Correctness properties

The following lemma is instrumental for analysing the effect that applying the rules osNew and osDel has on the type of the expression. Note that parts 1 and 2 are dual, 3 and 4 are dual and 5 is self-dual using the duality: $Z^l \leftrightarrow Z^h$, $\pm Z^n \leftrightarrow \mp Z^p$, $+x \leftrightarrow -x$ and $\subseteq \leftrightarrow \supseteq$.

**Lemma 6.1** *Let typings $A : X$, $B : Y$, $E : Z$, $A\,E : U$, $B\,E : V$ be inferred in $\Gamma$. Then the following facts hold:*

1. *If $Y^n \subseteq X^n + x$ and $X^l = Y^l + x$, then $V^n \subseteq U^n + x$.*
2. *If $Y^p \subseteq X^p + x$ and $X^h = Y^h - x$, then $V^p \subseteq U^p + x$.*
3. *If $X^n = Y^n + x$ and $X^l = Y^l - x$, then $U^n = V^n + x$.*
4. *If $X^p = Y^p + x$ and $X^h = Y^h + x$, then $U^p = V^p + x$.*
5. *$X^\diamond - Y^\diamond = U^\diamond - V^\diamond$, for $\diamond \in \{l, h\}$.*

PROOF: By easy calculations based on the typing rule Seq. We do one and leave the others to the reader. Let conditions be as stated in part 1 of the lemma. Then $V^n = Y^n \cup (Z^n - Y^l) \subseteq (X^n + x) \cup (Z^n - (X^l - x)) = (X^n \cup (Z^n - X^l)) + x = U^n + x$. □

The following lemma captures some essential invariants of the operational semantics. 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 6.2** *Let $\vdash P : \Gamma$, $\Gamma \vdash E : U$ and let $\{M, E\} \leadsto_P \{M', E'\}$ be a step in the operational semantics. Then we have:*

1. *$\Gamma \vdash E' : V$ for some type $V$.*
2. *$M' - V^n \supseteq M - U^n$, i.e., the minimum safety margin doesn't decrease.*
3. *$M' + V^p \subseteq M + U^p$, i.e., the maximum resource use doesn't increase.*
4. *$M' + V^l \supseteq M + U^l$, i.e., the minimum net effect doesn't decrease.*
5. *$M' + V^h \subseteq M + U^h$, i.e., the maximum net effect doesn't increase.*

Proof: All parts are proved by simultaneous induction on the definition of $\leadsto_P$. Part 1 uses the Inversion Lemma 5.3 to break down the typing $\Gamma \vdash E : U$. Thereafter a type for $E'$ can be inferred.

For the base case osNew, let $\Gamma \vdash \texttt{new}x : U$ and consider a step $\{M, \texttt{new}x\} \leadsto_P \{M + x, A\}$. By applying the Inversion Lemma 5.3 and the Uniqueness Lemma 5.4 we get that $U = \langle V^n, V^p + x, V^l + x, V^h + x\rangle$, where $V = \Gamma(A)$. Parts 3, 4 and 5 become equalities while part 2 follows from $M' - V^n = M - U^n + x$.

For the base case osDel, let $\Gamma \vdash \texttt{del}x : U$ and consider a step $\{M, \texttt{del}x\} \leadsto_P \{M - x, \texttt{nop}\}$. By applying the Inversion Lemma 5.3 we get $U = \langle [x], [\,], [-x], [-x]\rangle$ and $V = \langle [\,], [\,], [\,], [\,]\rangle$. This makes parts 2, 4 and 5 equalities, while part 3 follows from $U^p = [\,] \supseteq V^p - x$.

For the base case osNop, let $\Gamma \vdash \texttt{nop}\, E' : U$ and consider a step $\{M, \texttt{nop}\, E'\} \leadsto_P \{M, E'\}$. By applying the Inversion Lemma 5.3 we get a type $V$ such that $\Gamma \vdash E' : V$ and $V = U$. Parts 2 to 5 become equalities. The base case osParEnd is similar.

For osAlt1, 2, let $\Gamma \vdash (E_1 + E_2) : U$, and consider $\{M, (E_1 + E_2)\} \leadsto_P \{M, E_i\}$. From the Inversion Lemma 5.3 we have $X_1$ and $X_2$ such that the typings $E_1 : X_1$ and $E_2 : X_2$ hold in $\Gamma$. We have $U = \langle X_1^n \cup X_2^n, X_1^p \cup X_2^p, X_1^l \cap X_2^l, X_1^h \cup X_2^h\rangle$ and $V = X_i$. The calculations for parts 2 to 5 of the lemma are done by using $U^\diamond \supseteq X_i^\diamond$ for $\diamond \in \{n, p, h\}$ and $U^l \subseteq X_i^l$ using mono/antitonicity properties of $\cup$ and $\cap$.

For the induction case osPar1, let $\Gamma \vdash (E_1 \parallel E_2) : U$ and consider the step $\{M, (E_1 \parallel E_2)\} \leadsto_P \{M', (E_1' \parallel E_2)\}$, inferred from the step $\{M, E_1\} \leadsto_P \{M', E_1'\}$. From the Inversion Lemma we have types $X$ and $X_2$ such that typings $E_1 : X$ and $E_2 : X_2$ hold in $\Gamma$ where $U = X + X_2$. We get from the induction hypothesis a type $Y$ such that $\Gamma \vdash E_1' : Y$ and all parts of the lemma hold with $X$ for $U$ and $Y$ for $V$. We get $V = Y + X_2$ by applying the typing rule Par. All parts carry over from the induction hypothesis for $X$, $Y$ via $X + X_2$, $Y + X_2$. The case of osPar2 follows by symmetry.

For the induction case osSeq, let $\Gamma \vdash A\, E'' : U$ and consider a step $\{M, A\, E''\} \leadsto_P \{M', B\, E''\}$ inferred from a step $\{M, A\} \leadsto_P \{M', B\}$. By applying the Inversion Lemma 5.3 and the induction hypothesis we get types $X, Y, Z$ such that the typings $A : X$, $B : Y$ and $E'' : Z$ hold in $\Gamma$, such that

$$\begin{aligned} U &= \langle X^n \cup (Z^n - X^l), X^p \cup (Z^p + X^h), X^l + Z^l, X^h + Z^h\rangle \\ V &= \langle Y^n \cup (Z^n - Y^l), Y^p \cup (Z^p + Y^h), Y^l + Z^l, Y^h + Z^h\rangle \end{aligned}$$

Parts 2 to 5 of the lemma carry over from the induction hypothesis for $X, Y$ by Lemma 6.1. □

**Definition 6.3** *A state $\{M, E\}$ is* safely typed *by a basis $\Gamma$, denoted by $\Gamma \vdash \{M, E\}$, if $\Gamma \vdash E : X$ for some $X$ such that $X^n \subseteq M$.*

The next theorem characterizes several correctness properties of the system. The inferred types are also proved to be *sharp* (part 6 below). An alternative intuition for sharpness is that if $E$ has type $X$, then there is no type $Y$ that enjoys all of these correcteness properties in relation to $E$ and at the same time improves one or more constituents of $X$, that is, $Y^n \subset X^n \vee Y^p \subset X^p \vee Y^l \supset X^l \vee Y^h \subset X^h$.

**Theorem 6.4** *If $\vdash P\!:\!\Gamma$, $\Gamma \vdash E\!:\!X$ and $X^n \subseteq M$, then the following holds:*

1. *If $\{M, E\} \rightsquigarrow_P \{M', E'\}$ then $\Gamma \vdash \{M', E'\}$.*
2. *If $E$ is not* `nop`*, we have $\{M, E\} \rightsquigarrow_P \{M', E'\}$ for some $\{M', E'\}$.*
3. *All $\rightsquigarrow_P$-sequences starting in state $\{M, E\}$ are finite.*
4. *If $\{M, E\} \rightsquigarrow_P^* \{M', \texttt{nop}\}$, then $X^l \subseteq M' - M \subseteq X^h$.*
5. *If $\{M, E\} \rightsquigarrow_P^* \{M', E'\}$ then $-X^n \subseteq M' - M \subseteq X^p$.*
6. *The type $X$ is sharp in the following sense: for every $y \in \mathbb{C}$ and any $\diamond \in \{l, h\}$ there exists a terminal state $\{M', \texttt{nop}\}$ such that $\{M, E\} \rightsquigarrow_P^* \{M', \texttt{nop}\}$ and $(M' - M)(y) = X^\diamond(y)$; for every $y \in \mathbb{C}$ and any $\diamond \in \{n, p\}$ there exists a state $\{M', E'\}$ such that $\{M, E\} \rightsquigarrow_P^* \{M', E'\}$ and $(M' - M)(y) = X^\diamond(y)$ if $\diamond = p$ and $(M' - M)(y) = -X^\diamond(y)$ if $\diamond = n$.*

PROOF: Let $\vdash P\!:\!\Gamma$, $\Gamma \vdash E\!:\!X$ and $X^n \subseteq M$.

1. Assume $\{M, E\} \rightsquigarrow_P \{M', E'\}$. $E'$ is typable in $\Gamma$ by Lemma 6.2, part 1. Moreover, $\Gamma(E')^n \subseteq M'$ follows immediately from part 2 of the same lemma.
2. By induction on the size of $E$. Any $E$ can be written in one of the following forms: $\texttt{new}\,x$, $\texttt{del}\,x$, `nop`, $E_1\,E_2$, $(E_1 + E_2)$, $(E_1 \parallel E_2)$. For each of these forms we check that part 2 of the lemma holds. In case $\texttt{new}\,x$ we have a declaration for $x$ in $P$ by Lemma 5.1 so that we can apply osNew. In case $\texttt{del}\,x$ we have $x \in M$ by $[x] = X^n \subseteq M$ so that we can apply osDel. The case `nop` holds trivially. In case $E_1\,E_2$, if $E_1 = \texttt{nop}$ we can apply rule osNop, otherwise we have from the Inversion Lemma 5.3 a type $X_1$ such that $\Gamma \vdash E_1 : X_1$ and $X_1^n \subseteq X^n$. We can then apply the induction hypothesis for the smaller $E_1$ and use this step as premiss for an application of osSeq. In case $(E_1 \parallel E_2)$, if $E_1 = E_2 = \texttt{nop}$ we can apply osParEnd. Otherwise we can use the induction hypothesis for at least one of the smaller $E_1$ or $E_2$ so that we can apply osPar1 or osPar2. In case $(E_1 + E_2)$ we can always apply one of osAlt1 or osAlt2.
3. Assume $\vdash P : \Gamma$, and let $\mathbb{E}_P$ be the set of terms that can be typed in $\Gamma$. For every $E \in \mathbb{E}_P$, define $|E|$ in the following recursive way: $|\texttt{del}\,x| = |\texttt{nop}| = 1$, $|\texttt{new}\,x| = 1 + |A|$ if $x \prec A \in P$, $|(E_1 + E_2)| = 1 + \max(|E_1|, |E_2|)$ and $|E_1 \cdot E_2| = |(E_1 \parallel E_2)| = |E_1| + |E_2|$. By structural induction on the derivation of $\Gamma \vdash E : X$ one easily sees that $|E|$ is well-defined and gives an upper bound to the number of steps in the operational semantics.
4. By induction on the number of steps, using Lemma 6.2. The base case $\{M, E\} = \{M', \texttt{nop}\}$ is trivial. For the induction step, consider $\{M, E\} \rightsquigarrow_P \{M_1, E'\} \rightsquigarrow_P^* \{M', \texttt{nop}\}$ and assume $\Gamma(E) = X$, $\Gamma(E') = Y$ and $Y^l \subseteq M' - M_1 \subseteq Y^h$. By Lemma 6.2, part 4, we have $X^l + M \subseteq Y^l + M_1$, so we get $X^l \subseteq Y^l + M_1 - M \subseteq M' - M$. By part 5 of the same lemma we have $Y^h + M_1 \subseteq X^h + M$, so we get $M' - M \subseteq X^h$.
5. By induction on the number of steps, using Lemma 6.2. The base case (zero steps) is trivial. For the induction step, consider $\{M, E\} \rightsquigarrow_P \{M_1, E_1\} \rightsquigarrow_P^* \{M', E'\}$ and assume $\Gamma(E) = X$, $\Gamma(E_1) = Y$ and $-Y^n \subseteq M' - M_1 \subseteq Y^p$. By Lemma 6.2, part 2, we have $M - X^n \subseteq M_1 - Y^n$ so we get $-X^n \subseteq M' - M$. From part 3 we have $Y^p + M_1 \subseteq X^p + M$ so we get $M' - M \subseteq X^p$.

6. By primary induction on the length of $P$ and secondary induction on the derivation of $\Gamma \vdash E : X$. If the length of $P$ is zero the result is trivial. Otherwise, $\vdash P : \Gamma$ has been proved by the following application of the rule $\mathsf{Prog}$:

$$\frac{\vdash P' : \Gamma',\ \Gamma' \vdash A : Y,\ x \notin \mathsf{dom}(\Gamma')}{\vdash P', x \multimap A : \Gamma' \cup \{x \mapsto Y\}}$$

Assume the result has been proved for all programs shorter than $P$. We now prove that $X$ is sharp whenever $\Gamma \vdash E : X$ by (secondary) induction on the derivation of the latter. Let $y \in \mathbb{C}$. Note first that, for $\diamond \in \{n, p\}$, if some $X^\diamond(y) = 0$, one can take $\{M', E'\} = \{M, E\}$ to get the desired result for $X^\diamond(y)$. With this in mind the base cases $\mathsf{Axm}$ and $\mathsf{Del}$ are easy.

The base case $\mathsf{New}$ is more interesting since it uses the primary induction hypothesis. Assume $\Gamma \vdash \mathtt{new}\, x : X$ is inferred by the following application of the rule $\mathsf{New}$:

$$\frac{\Gamma(x) = Y}{\Gamma \vdash \mathtt{new}\, x : \langle Y^n, Y^p + x, Y^l + x, Y^h + x\rangle}$$

If $x$ is not the last variable declared in $P$, then the sharpness of $X = \langle Y^n, Y^p + x, Y^l + x, Y^h + x\rangle$ follows from the primary induction hypothesis (in combination with Strong Uniqueness). Otherwise, we have that $P$ is $P', x \multimap A$ as in the application of $\mathsf{Prog}$ above, with $\vdash P' : \Gamma',\ \Gamma' \vdash A : Y$. For any $y \in \mathbb{C}$ different from $x$ we can use the primary induction hypothesis for $A, Y$ and prefix the sequences obtained by a step using the rule $\mathsf{osNew}$ as we have $M(y) = (M + x)(y)$ and $Y^\diamond(y) = (Y^\diamond + x)(y)$. For $x$, note that $Y^n(x) = 0$ and take $\{M', E'\} = \{M, E\}$ to get $-X^n(x) = Y^n(x) = 0 = (M' - M)(x)$. For $\diamond \in \{p, l, h\}$ we have $Y^\diamond(x) = 1$ and we can take $\{M', E'\} = \{M + x, A\}$.

The induction cases of the rules $\mathsf{Alt}$ and $\mathsf{Par}$ are simple. In both cases the secondary induction hypothesis can be applied to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. In case of $\mathsf{Alt}$, if $X^\diamond(y) = X_i^\diamond(y)$ for given $y$ and $\diamond$, then one uses the induction hypothesis for $E_i$ ($i$ may vary with $y, \diamond$). In case of $\mathsf{Par}$, we apply the secondary induction hypothesis to both premises and concatenate both sequences using the inductive rule $\mathsf{osPar}$. By additivity this gives the desired results. (Any interleaving of the two sequences would amount to the same.)

In the case of the rule $\mathsf{Seq}$ we also apply the secondary induction hypothesis to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. Concerning $X^\diamond(y)$ for $\diamond \in \{l, h\}$ we can concatenate the two sequences. For $X^n(y)$ we distinguish between $X^n(y) = X_1^n(y)$ and $X^n(y) = X_2^n(y) - X_1^l(y)$. In the first case we take the sequence for $\{M, E_1\}$ and postfix all expressions with $E_2$ to obtain a sequence for $\{M, E_1\, E_2\}$ with the desired property. In the second case we take the sequence $\{M, E_1\} \rightsquigarrow_P^* \{M', \mathtt{nop}\}$ with $(M' - M)(y) = X_1^l(y)$, postfix its expressions with $E_2$, and then proceed with the sequence $\{M', E_2\} \rightsquigarrow_P^* \{M'', E_2'\}$ with $(M'' - M')(y) = -X_2^n(y)$. The total sequence $\{M, E_1\, E_2\} \rightsquigarrow_P^* \{M', E_2\} \rightsquigarrow_P^* \{M'', E_2'\}$ enjoys $(M'' - M)(y) = X_1^l(y) - X_2^n(y) = -X^n(y)$. The last case, $X^p(y)$, can be dealt with in a way very similar to $X^n(y)$.

Note that sharpness has been obtained by runs that may depend on the component $y$ and on the part of the type for which sharpness is desired. □

**Corollary 6.5** *If* $\vdash P : \Gamma$ *and* $\Gamma \vdash \{[\,], E\}$*, then* $\{[\,], E\} \rightsquigarrow_P^* \{M, \mathtt{nop}\}$*;*
*if* $\{[\,], E\} \rightsquigarrow_P^* \{M', E'\}$*, then* $M' \subseteq \Gamma(E)^p$*.*

# 7 Extensions

In this section we extend the basic system with loops and tail recursion, followed by an example on counting objects on the free store in C++. Thereafter we give two more extensions, one for dealing with memory usage and one introducing a scope operator which supports the implicit deallocation of resources.

## 7.1 Loops and Tail Recursion

It will not come as a surprise that recursion and unbounded loops are difficult to deal with in static analysis, since many properties become undecidable. Finite loops can be dealt with by iterated sequential composition. Under rather strict conditions, basically that the body of the loop has no net effect on the bag of instances of components, we can also deal with unbounded loops. These are modelled by the special form of recursion known as *tail recursion*. For loops we extend the syntax with $\mathit{Factor} \;\; ::= \;\; \mathtt{loop}(n, \mathit{Expr}),\ n > 0$. For tail recursion no new syntax is needed.

The intuition behind a finite loop $\mathtt{loop}(n, E)$ is the $n$-fold sequential composition of $E$ with itself. If $\Gamma \vdash E : X$, then some easy calculations based on the typing rule $\mathsf{Seq}$ give $\Gamma \vdash E \cdots E : Y$ ($n$ times $E$) with $Y^l = n * X^l$ and $Y^h = n * X^h$ (all multiplicities multiplied by $n$). Furthermore, we can calculate $Y^n = X^n \cup (X^n - X^l) \cup \cdots \cup (X^n - (n-1) * X^l)$. This expression can be simplified. If $X^l(x) \geq 0$, then $Y^n(x) = X^n(x)$. If $X^l(x) < 0$, then $Y^n(x) = X^n(x) - (n-1) * X^l(x)$. We can abbreviate the case distinction by $X^l(x)\,?\,X^n(x) : X^n(x) - (n-1) * X^l(x)$. Abstracting from the variable $x$ we state $Y^n = X^l\,?\,X^n : X^n - (n-1) * X^l$, where $A\,?\,B : C$ is the multiset defined by $(A\,?\,B : C)(x) = B(x)$ if $A(x) \geq 0$, and $C(x)$ otherwise. Similarly we find $Y^p = X^h\,?\,X^p + (n-1) * X^h : X^p$. With this in mind the rules concerning loops in Table 4 are easily understood.

Table 4: Rules for loops and tail recursion

(osLoop)

$$\frac{n > 1}{\{M, \mathtt{loop}(n, E)\} \rightsquigarrow_P \{M, E\,\mathtt{loop}(n-1, E)\}}$$

(osLoop1)

$$\frac{}{\{M, \mathtt{loop}(1, E)\} \rightsquigarrow_P \{M, E\}}$$

(Loop)

$$\frac{\Gamma \vdash E : X, \;\; n > 0}{\begin{aligned}\Gamma \vdash \mathtt{loop}(n, E) : \;\; & \langle X^l\,?\,X^n : X^n - (n-1) * X^l, \\ & X^h\,?\,X^p + (n-1) * X^h : X^p, n * X^l, n * X^h\rangle\end{aligned}}$$

(Rec)

$$\frac{\vdash P : \Gamma,\ \Gamma \vdash E : X,\ \Gamma \vdash A : Y,\ x \notin \mathsf{dom}(\Gamma),\ X^l = X^h = [\,]}{\vdash P, x \multimap E\,\mathtt{del}\,x\,\mathtt{new}\,x + A : \Gamma \cup \{x \mapsto \langle X^n \cup Y^n, (X^p \cup Y^p), Y^l, Y^h\rangle\}}$$

For tail recursion we add the typing rule $\mathsf{Rec}$ in Table 4. We do not need an extra rule in the operational semantics. The intuition behind $E\,\mathtt{del}\,x\,\mathtt{new}\,x$ is that, after the body $E$ has been executed, the frame of the tail recursive call (on top of the call stack) is popped before a new frame is pushed. Note that we do not have to deal with arbitrarily high multiplicities of $x$.

The most important modification of the theory is the requirement of fairness in the execution of recursive components such as $x \multimap E\,\mathtt{del}\,x\,\mathtt{new}\,x + A$. This means that eventually, after zero or more times choosing $E\,\texttt{del}\,x\,\texttt{new}\,x$, the base case $A$ is chosen. Fairness is necessary for termination. Also for type inference some care must be taken. First, in Lemma 5.5, part 2, one has to allow declarations of the form $x \prec E\,\texttt{del}\,x\,\texttt{new}\,x + A$ and require $\mathsf{var}(E\,A) \subseteq \mathsf{dom}(P')$, $\vdash P' : \Gamma'$, and $\Gamma'(E)^l = \Gamma'(E)^h = []$. The proof of this lemma can then easily be extended. For the dependency graph we consider a tail recursive $x$ to depend on variables in $\mathsf{var}(E\,A)$ only. With these precautions we get quadratic type inference also for the system with loops and tail recursion.

## 7.2 Example: counting objects on the free store in C++

In this section we show how to apply our techniques to the analysis of dynamically allocated memory in C++ [22]. In the example below, functions (such as `P1, U`) as well as objects on the free store (such as `C_instance`) are modelled as components. Again we let $\texttt{call}\,f$ abbreviate $\texttt{new}\,f\,\texttt{del}\,f$ and use this expression to model a function call. Note that $f$ is deleted automatically by $\texttt{call}\,f$, which models the (automatic) deallocation of stack objects created by $f$. However, the subcomponents of $f$ are not deleted by $\texttt{del}\,f$. In languages like C++, it is the programmer's responsibility to deallocate objects on the free store created by a function.

In the program fragment in Figure 7.2, so-called POSIX threads [1] are used for parallelism. 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 `P5()`, and the two threads are joined in `pthread_join`. The functions `P3(), P4(), P5()` are left abstract, and so is the dynamic data type `C`. Every function has been annotated with an expression in our language between `/* ... */`, where `P3, P4, P5, C` are assumed to be primitive components. We trust that all the rest is self-explaining. In the sequel,

```
void* P1(void* x) /* P1 -< call P3 del C  */ {
   P3(); delete (C*) x; return NULL;
}
void* P2(void* x)   /* P2 -< call P4 */ {
   P4(); /* position 1 */ return NULL;
}
/* U -< new C ((call P1 + call P2) || call P5) */
void U(int choice) {
   pthread_t pth;
   C* C_instance = new C();
   pthread_create(&pth, NULL, choice ? P1 : P2 , C_instance);
   P5(); /* position 2 */
   pthread_join(pth, NULL) /* position 3 */;
}
void UU(int choices[]) /* UU -< loop(10, call U) */ {
  for(int i=0; i<10; i++) U(choices[i]);
}
```

Figure 2: C++ code using threads and objects on the free store.

we will also discuss some variations of the above example. The central question in all examples is: is the deallocation of the objects `C_instance` on the free store correct?

Collecting all declarations in the above example we get the program $P$:

$$\begin{aligned}
&p_3 \mathrel{-\!\!\!\prec} \mathtt{nop},\\
&p_4 \mathrel{-\!\!\!\prec} \mathtt{nop},\\
&p_5 \mathrel{-\!\!\!\prec} \mathtt{nop},\\
&c \ \mathrel{-\!\!\!\prec} \mathtt{nop},\\
&p_1 \mathrel{-\!\!\!\prec} \mathtt{call}\,p_3\,\mathtt{del}\,c,\\
&p_2 \mathrel{-\!\!\!\prec} \mathtt{call}\,p_4,\\
&u \ \mathrel{-\!\!\!\prec} \mathtt{new}\,c\,((\mathtt{call}\,p_1 + \mathtt{call}\,p_2) \parallel \mathtt{call}\,p_5),\\
&uu \mathrel{-\!\!\!\prec} \mathtt{loop}(10, \mathtt{call}\,u)
\end{aligned}$$

Type inference gives the following results:

$$\begin{aligned}
&\mathtt{call}\,p_1 : \langle [c], [p_1, p_3], [-c], [-c] \rangle,\\
&\mathtt{call}\,p_2 : \langle [], [p_2, p_4], [], [] \rangle,\\
&\mathtt{call}\,u : \langle [], [c, p_1, p_2, p_3, p_4, p_5, u], [], [c] \rangle,\\
&\mathtt{call}\,uu : \langle [], [10c, p_1, p_2, p_3, p_4, p_5, u, uu], [], [10c] \rangle
\end{aligned}$$

This signals in the last multiset ($\cdot^h$) of the type of $uu$ a memory leak of $[10c]$ (in the worst-case). Obviously, this is caused by the possible choice of $\mathtt{call}\,p_2$ instead of $\mathtt{call}\,p_1$ by $u$, whereby created instances of $c$ are not deleted.

Let us discuss a few ways to improve the program. The most probable source of the error is that the programmer simply forgot to delete $c$ in $p_2$ (at position 1 in the example). Changing the declaration of $p_2$ into $p_2 \mathrel{-\!\!\!\prec} \mathtt{call}\,p_4\,\mathtt{del}\,c$ fixes the memory leak: for the new $p_2$ we have $\mathtt{call}\,p_2 : \langle [c], [p_2, p_4], [-c], [-c] \rangle$ and as a consequence the last multiset of the new types of $\mathtt{call}\,u$ and $\mathtt{call}\,uu$ becomes empty.

Let us go back to the program $P$ and consider another attempt to fix the memory leak. One idea is to insert `delete C_instance;` at position 2 in the function `U`. This means that $P$ is changed by changing the declaration of $u$ into: $u' \mathrel{-\!\!\!\prec} \mathtt{new}\,c\,((\mathtt{call}\,p_1 + \mathtt{call}\,p_2) \parallel \mathtt{call}\,p_5\,\mathtt{del}\,c)$. Type inference now signals that we actually delete $c$ too many times: $\mathtt{call}\,u' : \langle [c], [c, p_1, p_2, p_3, p_4, p_5, u], [-c], [] \rangle$. The reason is of course the possible choice of $\mathtt{call}\,p_1$ instead of $\mathtt{call}\,p_2$ by $u'$, whereby the instance of $c$ is deleted twice. Therefore $u'$ should be combined with removing $\mathtt{del}\,c$ from the declaration of $p_1$. Even then $u'$ contains a hidden error: depending on the particular scheduling used in the parallel composition, the instance of $c$ may be deleted too early if in use by $p_3$ or $p_4$. We can simulate the use of $c$ by $p_3$ by adding a primitive component $c_u$, changing the declaration of $p_3$ to $p_3 \mathrel{-\!\!\!\prec} \mathtt{del}\,c_u\,\mathtt{new}\,c_u$ and inserting $\mathtt{del}\,c_u$ and $\mathtt{new}\,c_u$ in front of $\mathtt{del}\,c$ and $\mathtt{new}\,c$, respectively. Then the type of $u'$ would have signalled a deficit in the first bag ($\cdot^n$).

Another idea might be to insert `delete C_instance;` at position 3 in the function `U`, in combination with removing `delete x;` from the function `P1`. The resulting program in our formalism reads: $p_3 \mathrel{-\!\!\!\prec} \mathtt{nop}, p_4 \mathrel{-\!\!\!\prec} \mathtt{nop}, p_5 \mathrel{-\!\!\!\prec} \mathtt{nop}, c \mathrel{-\!\!\!\prec} \mathtt{nop}, p_1' \mathrel{-\!\!\!\prec} \mathtt{call}\,p_3$, $p_2 \mathrel{-\!\!\!\prec} \mathtt{call}\,p_4, u'' \mathrel{-\!\!\!\prec} \mathtt{new}\,c\,((\mathtt{call}\,p_1' + \mathtt{call}\,p_2) \parallel \mathtt{call}\,p_5)\,\mathtt{del}\,c, uu' \mathrel{-\!\!\!\prec} \mathtt{loop}(10, \mathtt{call}\,u'')$, with type $\mathtt{call}\,uu' : \langle [], [c, p_1', p_2, p_3, p_4, p_5, u'', uu'], [], [] \rangle$. This would be the preferred solution, with $\mathtt{new}\,c$ and $\mathtt{del}\,c$ in the same declaration. This solution would still be valid if $p_5$ would use $c$ (with `P5(C_instance)` instead of `P5()` in function `U`).

## 7.3 Memory

This extension illustrates how the techniques can be used for quantifying memory usage. Conceptually the easiest way to deal with memory is to introduce a primitive component $u$ representing one unit of memory and using in each declaration the right number of $\mathtt{new}\,u$ expressions. For example, consider $a \mathrel{-\!\!\!\prec} \mathtt{new}\,b\,(\mathtt{new}\,c + \mathtt{new}\,d)$ and let $a$ use two units of memory at deployment, and later on one more in the branch $\mathtt{new}\,d$. This could be expressed in the following way:

$$a \mathrel{-\!\!\!\prec} \mathtt{new}\,u\,\mathtt{new}\,u\,\mathtt{new}\,b\,(\mathtt{new}\,c + \mathtt{new}\,u\,\mathtt{new}\,d)$$

Though attractive by its simplicity, this method has one major drawback: it contains in essence a unary representation of numbers, and this makes that the expressions can become exponentially long. Therefore we opt for a different approach, which keeps expressions polynomial.

Let $m : \mathbb{C} \to \mathbb{N}$ be a function specifying the memory use for each component. More precisely, for every $x \in \mathbb{C}$ the amount of memory used by an instance of $x$, *not including the memory used by the subcomponents of* $x$, is given by $m(x)$. Given $\Gamma \vdash E : X$, applying the function $m$ to the elements of $X^p$ and adding the results certainly gives an upper bound for the memory use of $E$. However, as the maxima for different components need not be attained in the same run, these upper bounds are not sharp, as illustrated by the following example.

Consider the following typing with basis $a \multimap \mathtt{nop}, b \multimap \mathtt{nop}$:

$$\mathtt{new}\, a\, \mathtt{new}\, a\, \mathtt{new}\, b + \mathtt{new}\, a\, \mathtt{new}\, b\, \mathtt{new}\, b : \langle [\,], [2a, 2b], [a, b], [2a, 2b] \rangle$$

Obviously, $2m(a) + 2m(b)$ is an unsharp upper bound for the total memory use. A sharp bound is $\max(2m(a) + m(b), m(a) + 2m(b))$.

Sharp bounds can be obtained by an elegant change of the typing rules in Section 4. Let $m : \mathbb{C} \to \mathbb{N}$ be as above and let $u$ be a new (unused) component name. Let $\mathit{mem}(x)$ be the bag consisting of $m(x)$ copies of $u$. The allocation (deallocation) of memory used by an instance of $x$ is modelled by adding (subtracting) the bag $\mathit{mem}(x)$. Recall the operational semantics and the type system with rules as given in Table 2 and Table 3, respectively. Replace systematically $[x]$ by $[x] + \mathit{mem}(x)$, $[-x]$ by $[-x] - \mathit{mem}(x)$, $M + x$ by $M + x + \mathit{mem}(x)$ and $M - x$ by $M - x - \mathit{mem}(x)$ and so on. (This replacement also holds for the first bag in the type of $\mathtt{new}\, x$ in New in Table 3 by the following reasoning. Note that $x$ doesn't occur in $X^n$. So in fact $X^n = [\,] \cup (X^n - x)$ in New. The replacement now gives $[\,] \cup (X^n - x - \mathit{mem}(x)) = [\,] \cup (X^n - \mathit{mem}(x))$ in New'.) The rules that change are shown in Table 5, all other rules stay the same.

Table 5: Modified rules for memory use

(osNew')

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

(osDel')

$$\frac{x \in M}{\{M, \mathtt{del}\, x\} \rightsquigarrow_P \{M - x - \mathit{mem}(x), \mathtt{nop}\}}$$

(Del')

$$\frac{\Gamma(x) = X}{\Gamma \vdash \mathtt{del}\, x : \langle [x] + \mathit{mem}(x), [\,], [-x] - \mathit{mem}(x), [-x] - \mathit{mem}(x) \rangle}$$

(New')

$$\frac{\Gamma(x) = X}{\begin{array}{ll}\Gamma \vdash \mathtt{new}\, x : & \langle [\,] \cup (X^n - \mathit{mem}(x)), X^p + x + \mathit{mem}(x), \\ & X^l + x + \mathit{mem}(x), X^h + x + \mathit{mem}(x) \rangle\end{array}}$$

With only minor modifications, the theory developed in Section 4–6 holds for the modified system. For example, in Lemma 5.1 one has to exclude $x = u$, in the Inversion Lemma 5.3, the first two parts, one has to apply similar substitutions for $x$ as above. In the end one obtains a similar result as Corollary 6.5. Here one has to interpret $N(u)$, for any bag $N$, as an amount of memory.

**Corollary 7.1** *With the rules modified as in Table 5 we have: if* $\vdash P : \Gamma$, $\Gamma \vdash \{[\,], E\}$ *and* $\{[\,], E\} \rightsquigarrow_P^* \{M, E'\}$*, then* $M(u) \leq \Gamma(E)^p(u)$.

This result can be proved to be sharp and can be extended to the system with loops and tail recursion.

## 7.4 A Scope Operator

In this section we will introduce a scope operator. The primary goal is to have a convenient mechanism for deallocating components. Consider the following example: $a \mathrel{-\!\!\prec} \mathtt{nop}, b \mathrel{-\!\!\prec} \mathtt{new}\,a, c \mathrel{-\!\!\prec} (\mathtt{new}\,a + \mathtt{new}\,b)\,\mathtt{nop}$ Assume we want to deallocate all instances created by executing $\mathtt{new}\,c$. This would mean a $c$ and either an $a$ or an $a$ and a $b$, depending on which alternative has been chosen. Note that $\mathtt{nop}$ abstracts from possible use of $a$ or $a, b$. Using right-distributivity and including proper deallocation in the various alternatives would lead to: $c' \mathrel{-\!\!\prec} \mathtt{new}\,a\,\mathtt{nop}\,\mathtt{del}\,a + \mathtt{new}\,b\,\mathtt{nop}\,\mathtt{del}\,b\,\mathtt{del}\,a$. Now $\mathtt{new}\,c'\ \mathtt{del}\,c'$ would correctly deallocate all instances, but this way of explicit memory management is very unattractive, and would moreover lead to exponentially long expressions. It is better to have a mechanism which automatically deallocates all instances of components created by an expression, in the same way as local variables disappear when a block is left. This is achieved by the scope mechanism proposed in this section. This also alleviates to some extent the abstraction from the identities of instances, since we can view the bag of all instances as partitioned into smaller, local ones.

To demarcate a scope we use a matching pair of curly brackets. Writing $\{M, E\}$ limits the lifetime of all instances in $M$ and those created by $E$ to the execution of $E$ and enforces deallocations by $E$ to apply to these instances only. The use of the same brackets as in states is deliberate, since a state in Section 3 is nothing more than an outermost scope.

As can be expected, the scope mechanism involves an extension of the syntax, extra rules for the operational semantics, and two new typing rules. In Table 6 we specify the extensions.

Table 6: Additions for the scope operator

Syntax, extending Table 1:
$Factor \ ::= \ StExp$

Operational semantics, extending Table 2:

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

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

Typing rule, extending Table 3:

(Scp)
$$\frac{\Gamma \vdash E : X,\ X^n \subseteq M,\ \mathsf{set}(M) \subseteq \mathsf{dom}(\Gamma)}{\Gamma \vdash \{M, E\} : \langle [\,], M + X^p, [\,], [\,] \rangle}$$

First, the syntax for expressions in Table 1 is extended with an extra rule for $Factor$. Note that this also extends the state expressions themselves, since they depend on expressions. In a declaration $x \mathrel{-\!\!\prec} E$, where $E$ is an expression in the extended syntax, only empty bags may occur in subexpressions of the form $\{[\,], E'\}$. As before, a program consists of a list of declarations. Thus expressions occurring in a program form a subset of those occurring in states. State expressions may have subexpressions of the form $\{M, E\}$

with arbitrary bags $M$, where $M$ represents the store of component instances that are local to this occurrence of $E$. For example, $\{M, (\{M', A\,\{[\,], B\}\} \parallel C)\,(D + E)\}$ corresponds to a state in which the expression is executed with an outermost store $M$ of instances and with local stores $M'$ for $A$ and $[\,]$ for $B$. Expression $C$ uses $M$, and so do $D$ and $E$ if the leading factor of the expression has terminated. $A$ does not affect $M$ and $B$, who starts after $A$ has terminated, does not affect $M$ nor $M'$.

Table 6 gives two extra rules for the operational semantics which are easy to understand. Furthermore, the typing rule $\mathsf{Scp}$ requires that, for $\{M, E\}$ to be well typed, it has to be safe to execute $E$ using $M$. If so, the type of $\{M, E\}$ reflects that there will be no deficit underway, and no instances left over after executing $\{M, E\}$. The maximum resource use involved equals $M$ plus the maximum involved in executing $E$. Note that the new typing rule types state expressions and not only expressions occurring in programs.

The lemmas in Section 5 will not be repeated, since their formulation as well as their proofs are very similar. From the Basics Lemma 5.1, part 1, we must weaken the if-part, since the rule $\mathsf{Scp}$ imposes an additional condition on typability. (This condition is fulfilled in places where the if-part is used.) For the Inversion Lemma 5.3, we must add the following clause related to the new typing rule.

**Lemma 7.2 (Extensions to the Inversion Lemma)**

6. *If $\Gamma \vdash \{M, A\} : X$, then there exists a type $Y$, such that $\Gamma \vdash A : Y$ with $Y^n \subseteq M$ and $X = \langle [\,], M + Y^p, [\,], [\,] \rangle$.*

PROOF: The inductive proof of Lemma 5.3 can easily be extended to the new case above. The new typing rule $\mathsf{Scp}$ does not complicate the other steps. □

In this extended system, the total number of component instances in a state $\{M, E\}$ should take into account not only $M$ but also all bags possibly occurring in $E$. This is done by the following definition of the total sum $\Sigma$. In doing so, however, one counts in instances that will never coexist, such as in $\{M, E_1\} + \{M, E_2\}$ and $\{M, E_1\}\,\{M, E_2\}$. Therefore we also define the notion of a valid expression, in which irrelevant bags are empty.

**Definition 7.3 (Sum and valid expression)** *For any expression $E$, let $\Sigma E$ 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 $\Sigma\mathtt{del}\,x = \Sigma\mathtt{new}\,x = \Sigma\mathtt{nop} = [\,]$. 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. In any declaration $x \multimap E$, since only empty bags are allowed to occur in $E$, $E$ is obviously valid and $\Sigma E = [\,]$. As defined earlier, the initial state when executing a program is $\{[\,], \mathtt{new}\,x\}$, where $x$ is the last component declared in the program. The initial state is valid by definition. The following lemma implies that all states in sequences representing the execution of a program are valid.

**Lemma 7.4** *If $\vdash P : \Gamma$, $\Gamma \vdash E : X$, $E$ is valid and $\{M, E\} \rightsquigarrow_P \{M', E'\}$ is a step in the operational semantics, then also $E'$ is valid.*

PROOF: By induction on the definition of $\rightsquigarrow_P$, using that an expression is valid whenever all its subexpressions are. Assume $E$ is valid. In the cases of $\mathsf{osDel}$, $\mathsf{osPop}$ and $\mathsf{osParEnd}$, $E' = \mathtt{nop}$ and hence valid. In the cases of $\mathsf{osNop}$, $E'$ is a subexpression of $E$ and hence valid. In the case $\mathsf{osNew}$, note that $\Sigma A = [\,]$ for any declaration $x \multimap A$. In the cases of $\mathsf{osAlt1}, 2$, $E = (E_1 + E_2)$ and hence $\Sigma E = [\,]$, so also $\Sigma E' = [\,]$. In the cases of $\mathsf{osPar1}, 2$, $\mathsf{osSeq}$ and $\mathsf{osScp}$ we use the induction hypothesis. □

Note that for $E = \{[x], \mathtt{nop}\} + \{[x], \mathtt{nop}\}$ we have $\vdash E : \langle [\,], [x], [\,], [\,] \rangle$, but $\Sigma E = [2x]$. The following lemma states that $\Gamma(E)^p$ is an upper bound for $\Sigma E$ provided $E$ is valid.

**Lemma 7.5** *If $E$ is a valid expression and $\Gamma \vdash E : X$, then $\Sigma E \subseteq X^p$.*

PROOF: Proof by induction on $E$ (no induction on derivations needed). Let $E$ be a valid expression. The cases $E = \texttt{nop}$, $\texttt{new}x$, $\texttt{del}x$, $(E_1 + E_2)$ are trivial since then $\Sigma E = [\,]$ by Definition 7.3. In the cases $E = (E_1 \parallel E_2)$, $\{M, E'\}$ the induction hypothesis can be applied. For the last case, $E = E_1\, E_2$, assume $\Gamma(E) = X$. From the Inversion Lemma 5.3/7.2 we get $\Gamma(E_1) = X_1$ and $\Gamma(E_2) = X_2$ such that $X^p = X_1^p \cup (X_2^p + X_1^h)$. Since $E$ is valid we get $\Sigma E = \Sigma E_1$ and $E_1$ valid, so $\Sigma E \subseteq X_1^p \subseteq X^p$. □

Lemma 6.1 holds for the system extended with scope since it only concerns the typing rule Seq which did not change. Lemma 6.2 also holds in the extended system, but here it is important to stress that the denotations like $\rightsquigarrow_P$, $\vdash$ and $\{M, E\}$ are to be understood in the extended system. This means that the proof by induction on $\rightsquigarrow_P$ gets two extra cases for the new rules osScp, osPop. For convenience we rephrase the lemma in a short form.

**Lemma 7.6** *Let $\vdash P : \Gamma$, $\Gamma \vdash E : U$ and $\{M, E\} \rightsquigarrow_P \{M', E'\}$ be a step in the operational semantics. Then we have: 1. $\Gamma \vdash E' : V$ for some type $V$; 2. $M' - V^n \supseteq M - U^n$; 3. $M' + V^p \subseteq M + U^p$; 4. $M' + V^l \supseteq M + U^l$; 5. $M' + V^h \subseteq M + U^h$.*

PROOF: It suffices to extend the proof of Lemma 6.2 with the following two cases.

For the base case osPop, let $\Gamma \vdash \{N, \texttt{nop}\} : U$ and consider $\{M, \{N, \texttt{nop}\}\} \rightsquigarrow_P \{M, \texttt{nop}\}$. By the Inversion Lemma 5.3/7.2 $U = \langle [\,], N, [\,], [\,] \rangle$ and further $\Gamma \vdash \texttt{nop} : V$ where $V = \langle [\,], [\,], [\,], [\,] \rangle$. This proves part 1 and parts 2, 4 and 5 become equalities since $U^\diamond = V^\diamond$, for $\diamond \in \{n, l, h\}$. Part 3 follows from $V^p = [\,] \subseteq N$.

For the inductive case osScp, let $\Gamma \vdash \{N, A\} : U$ and consider a step $\{M, \{N, A\}\} \rightsquigarrow_P \{M, \{N', A'\}\}$, inferred from a step $\{N, A\} \rightsquigarrow_P \{N', A'\}$. By the Inversion Lemma 5.3/7.2 we have $X$ such that $\Gamma \vdash A : X$, where $X^n \subseteq N$, and $U = \langle [\,], N + X^p, [\,], [\,] \rangle$. We get from the induction hypothesis, parts 1 and 2, that $\Gamma \vdash A' : Y$ and $N' - Y^n \supseteq N - X^n$, so $[\,] \subseteq N' - Y^n$ or $Y^n \subseteq N'$. Hence we can apply the typing rule Scp to get $\Gamma \vdash \{N', A'\} : V$, where $V = \langle [\,], N' + Y^p, [\,], [\,] \rangle$. All parts follow directly from this type and from the induction hypothesis, part 3, $Y^p + N' \subseteq X^p + N$. □

The notion of a safely typed state extends the one in Section 6 but the formulation can be simplified since we can type state expressions in the system with scope. Thereafter follows the Soundness Theorem in which only the last two parts differ from Theorem 6.4.

**Definition 7.7** *A state $\{M, E\}$ is safely typed by a basis $\Gamma$, denoted by $\Gamma \vdash \{M, E\}$, if $\Gamma \vdash \{M, E\} : X$ for some $X$.*

**Theorem 7.8** *If $\vdash P : \Gamma$, $\Gamma \vdash E : X$ and $X^n \subseteq M$, then parts 1–4 in Theorem 6.4 also hold for the system extended with scope. Moreover, the parts 5 and 6 hold in the following formulation under the extra assumption that $E$ is valid:*

5. *If $\{M, E\} \rightsquigarrow_P^* \{M', E'\}$, then $-X^n \subseteq M' - M$ and $\Sigma\{M', E'\} \subseteq M + X^p$.*
6. *The type $X$ is sharp in the following sense: for every $y \in \mathbb{C}$ and any $\diamond \in \{l, h\}$ there exists a terminal state $\{M', \texttt{nop}\}$ such that $\{M, E\} \rightsquigarrow_P^* \{M', \texttt{nop}\}$ and $(M' - M)(y) = X^\diamond(y)$; for every $y \in \mathbb{C}$ and any $\diamond \in \{n, p\}$ there exists a state $\{M', E'\}$ such that $\{M, E\} \rightsquigarrow_P^* \{M', E'\}$ and $(\Sigma\{M', E'\} - M)(y) = X^\diamond(y)$ if $\diamond = p$ and $(M' - M)(y) = -X^\diamond(y)$ if $\diamond = n$.*

PROOF: Let $\Gamma \vdash E : X$ and $X^n \subseteq M$. Parts 1-4 are proved for the system extended with scope with some minor extensions with respect to the proof of Theorem 6.4.

1. By Lemma 7.6, part 1 and 2.

2. By induction on the size of $E$. The only new form is: $\{N, E_1\}$. If $E_1 = \mathtt{nop}$ we can apply osPop. Otherwise we have from the Inversion Lemma that $\Gamma \vdash E_1 : Y$ for some $Y$ and $Y^n \subseteq N$, so we can apply the induction hypothesis for the smaller $E_1$ and use osScp on this step.

3. Extend the definition of $|\,|$ by $|\{M, E\}| = 1 + |E|$ and use the same argument.

4. By the same argument, using Lemma 7.6 instead of Lemma 6.2.

5. Let $E$ be valid. Again we use induction on the number of steps, now using Lemma 7.6. The base case (zero steps) follows by Lemma 7.5, $\Sigma E \subseteq X^p$, since $E$ is valid. For the induction step, consider $\{M, E\} \leadsto_P \{M_1, E_1\} \leadsto_P^* \{M', E'\}$ and assume $\Gamma(E) = X$, $\Gamma(E_1) = Y$ and $-Y^n \subseteq M' - M_1$ and $M' + \Sigma E' \subseteq M_1 + Y^p$. From Lemma 7.6, part 2 we get $M_1 - Y^n \supseteq M - X^n$ so we get $-X^n \subseteq M' - M$. From part 3 we have $Y^p + M_1 \subseteq X^p + M$ so we get $M' + \Sigma E' \subseteq M + X^p$.

6. Let $E$ be valid. Recall that the proof is by primary induction on the length of $P$ and secondary induction on the derivation of $\Gamma \vdash E : X$. The primary induction follows the proof of Theorem 6.4, using the rules AxmP and Prog. In the secondary induction, the only interesting cases are the bag $X^p$ and the new rule Scp.

   For $X^p$ the proof is slightly different, since we must prove $(\Sigma\{M', E'\} - M)(y) = X^p(y)$ instead of $(M' - M)(y) = X^p(y)$. Note first that, if $X^p(y) = 0$, then by Lemma 7.5 also $\Sigma E(y) = 0$ and one can take $\{M', E'\} = \{M, E\}$ to get the desired result. The cases Axm, New, Del, Alt and Par follow the argument of Theorem 6.4. For the rule Seq, let $E = E_1\, E_2$. Note that $\Sigma E_2 = [\,]$ since $E$ is valid. Similarly to the proof in Theorem 6.4, we apply the secondary induction hypothesis to the premises $\Gamma \vdash E_i : X_i$ $(i = 1, 2)$. Note that postfixing the expressions in the sequence for $\{M, E_1\}$ with $E_2$ is valid since $\Sigma E_2 = [\,]$. If $X^p(y) = X_1^p(y)$, then we can take the sequence $\{M, E_1\} \leadsto_P^* \{M', E_1'\}$ with $(\Sigma\{M', E_1'\} - M)(y) = X_1^p(y)$, postfix its expressions with $E_2$ and obtain $(\Sigma\{M', E_1'\, E_2\} - M)(y) = X^p(y)$. If $X^p(y) = X_2^p(y) + X_1^h(y)$, then we can take the sequence $\{M, E_1\} \leadsto_P^* \{M', \mathtt{nop}\}$ with (by the secondary induction hypothesis) $(M' - M)(y) = X_1^h(y)$, postfix its expressions with $E_2$ and then proceed with the sequence $\{M', E_2\} \leadsto_P^* \{M'', E_2'\}$ with $(\Sigma\{M'', E_2'\} - M')(y) = X_2^p(y)$. The total sequence enjoys $(\Sigma\{M'', E_2'\} - M)(y) = X^p(y)$.

   For the rule Scp, let $E = \{N, A\}$. From the Inversion Lemma we have $Y$ such that $\Gamma \vdash A : Y$ , $X^p = N + Y^p$ and $X^\diamond = [\,]$ for $\diamond \in \{n, l, h\}$. For $l$ and $h$ we take any terminating sequence $\{N, A\} \leadsto_P^* \{N', \mathtt{nop}\}$ and observe that this leads to a sequence $\{M, E\} \leadsto_P^* \{M, \mathtt{nop}\}$ by the rules osScp and osPop. For $X^n(y) = 0$ we take zero steps. For $X^p(y)$ we use the secondary induction hypothesis to get $\{N, A\} \leadsto_P^* \{N', A'\}$ where $\Sigma\{N', A'\}(y) = N(y) + Y^p(y)$. By the rule osScp this leads to a sequence $\{M, \{N, A\}\} \leadsto_P^* \{M, \{N', A'\}\}$ with $(M + \Sigma\{N', A'\} - M)(y) = N(y) + Y^p(y) = X^p(y)$.

□

**Corollary 7.9** *If $E$ is valid, $\vdash P : \Gamma$, $\Gamma \vdash \{[\,], E\}$ and $\{[\,], E\} \leadsto_P^* \{M, E'\}$, then $\Sigma\{M, E'\} \subseteq \Gamma(E)^p$.*

# 8 Related work and concluding remarks

## 8.1 Related work

Static analysis is a well-established subject, actually too broad to be reviewed as a whole here. Recall that our main objective is counting instances of software components, and that we are able to detect, to some extent, unsafe deallocation and memory leaks. We are not aware of related work having the same main objective as ours, and we will restrict attention to related work on memory usage and on safety of deallocation.

The most important approaches to static analysis of memory usage seem to be [15, 11, 13, 26] for functional languages, [8, 14, 6] for the imperative paradigm, and [4, 2] aiming at the bytecode (assembly) level. They aim at predicting upper bounds for memory usage (our $X^p(u)$). All are targeted at low-level models late in the development process, when many details are known and controlled.

Safety of deallocation (our $X^n$) and memory leaks (our $X^h$) have also been treated statically by others, for example, [23, 10, 5, 29, 28, 7]. Most of this work is at a lower abstraction level than ours.

It seems fair to say that these approaches are able to obtain more detailed results, but of less generality. For example, some approaches require program annotations. None of them treat parallel composition. We think that our techniques are more suitable in the first phase of the design of a software system, and less in the later, more concrete phases, where the other approaches might be more in place. In other words, the UML example 4.3 is presumably more representative than the C++ example 7.2. In C++ programs one will very soon encounter constructions that are hard to model in our abstract language (the same seems to be true for the other approaches).

In the following subsection we give some more details about the main related works. In summary, our approach has the following four salient features of which all other approaches lack at least two: parallel composition, full compositionality, automatic type inference (in quadratic time), sharpness of all bounds.

### Functional languages

Hughes and Pareto [15] introduce a strict, first-order functional language with explicit regions and give a type system with space-effects that guarantees that well-typed programs use at most the space specified by the programmer. A region is a temporary heap that programmers can create, put values on and, finally, can discard explicitly. The work combines the technique of sized types [16] and region-based memory management for a functional language [23], whereas we address de/allocation of instances of components in a possibly imperative setting. Moreover, their approach requires program annotations and automatic type inference is not explicitly dealt with.

Crary and Weirich [11] treat time as a resource. Their system certifies a time limit for a complete functional program, by using program annotations of time limits for each individual function. They claim the work can be generalized to stack space and even heap space when it is combined with the region-based memory management from [23].

In [13], Hofmann and Jost use a linear type system to compute linear bounds on heap space for a first-order functional language. One significant contribution of this work is an inference mechanism through a linear programming technique. The work is later extended in [14] to an object-oriented language, see below.

Unnikrishnan et al. [26] deal with a first-order, call-by-value, garbage-collected functional language. They create a space-bound function that takes a set of inputs of the original program and returns an upper bound on the memory consumption of the program with that input data. Their approach is based on program analysis and model checking, while ours is type-based. A limitation of this work is that the space-bound function may not give results for some inputs.

### Imperative, object-oriented languages

In [8], Wei-Ngan Chin et al. treat explicit memory management in a core object-oriented language MemJ. Their work uses alias annotations to insert explicit deallocation statements where appropriate. Programmers have to annotate the memory usage and size relations for methods. Their types have a bag for the maximum amount of memory that the method consumes, which is similar to $X^p$ in our types, and a bag for the minimum amount of memory that the method will recover at the end of the method invocation, which can be computed from our types by a simple operation $X^p - X^h$.

In [14], Hofmann and Jost apply their work [13] to an object-oriented language with explicit deallocation. The work combines amortised analysis, linear programming and functional programming to calculate the heap space bound as a function of the input. However, feasibility of type inference is not clear and, as the authors concede, their bounds can be over-approximated.

Braberman et al. [6] deal with imperative, object-oriented programs. Their approach is not type-based. The authors present an algorithm to statically compute memory consumption of a method as a non-linear function of the method's parameters. Even though their experimental results are good, the bounds are not sharp. Besides, their language does not include explicit deallocation.

### Bytecode languages

In [4], Barthe et al. use program logics to give a precise analysis of the memory consumption of sequential bytecode programs annotated with pre- and post-conditions on resource usage. Explicit deallocation and parallel composition are mentioned as future work.

Albert et al. [2] compute memory consumption of a program as a function of its input data. They also refine the program's functions by using escape analysis [9] to collect objects that do not escape their scopes. The bytecode language has no explicit deallocation.

### Safety of deallocation and memory leak detection

In [23], Tofte and Talpin introduce a static type system for a functional language based on ML that allows programmers to explicitly allocate and deallocate regions of memory while ensuring the safety of region-deallocation. A region contains a group of objects and deallocating a region frees all objects in the region. Regions are organized in a LIFO stack similar to our scope mechanism. Their type system ensures the safety of deallocation. The technique was later extended to other object-oriented languages, see e.g. [5].

Crary et al. [10] apply region-based memory management to a statically-typed intermediate language with explicit allocation and deallocation of regions called Capability Calculus. Regions need not be strictly organized as a LIFO stack (such as in [23]) and deallocation is guaranteed safe by unforgeable keys or 'capabilities'.

The papers [29], [28] and [7] are about statically detecting memory leaks. They do not use a type system. If we change our operational semantics of scope such that it does not delete the remaining instances in the local scope and programmers must explicitly deallocate them, then we could prevent 'instance leaks' by adding the side condition $X^l = X^h = []$ in the rule $\mathsf{Scp}$. Still, the difference with the above approaches would be considerable.

## 8.2 Concluding remarks and future research

The current paper is based on [25], but has been rewritten completely. The treatment of the scope operator is new, memory, loops and tail recursion have been added, as well as the examples in Section 4.3 and Section 7.2.

The language we introduced is clearly inspired by CCS [18], with the atomic actions interpreted as component instantiation and deallocation. The basic operators are sequential, alternative and parallel composition, later extended with loops, tail recursion and a scope operator.

We have presented a type system for this language that predicts sharp bounds on the number of instances of components, and allows automatic type inference in quadratic time.

The operational semantics is SOS-style [20], with the approach to soundness similar in spirit to [27]. The type system is given close to the traditional style of [3]. However, there are some significant differences with the usual type systems for functional languages [19]. First, there are no function types since there is no lambda abstraction. Variables (component names) are bound to expressions in the declarations. Second, the types contain quantitative rather than qualitative information. We see no connection of our types with linear types or linear logic.

A number of important aspects of processes have not been treated in this paper. One of them is communication between processes, but in fact every operator from process theory can be considered as a candidate for extension of our language. One can also extend the language with other primitives. One candidate is an atomic action which reuses an instance of $x$ if there is one, and creates a new instance otherwise. (See [24] for a system with a primitive $\texttt{reu}\,x$.)

Another interesting topic for future research is a primitive $\texttt{use}\,x$, which doesn't change the store, but can only be executed when there exists an instance of $x$ in the store. In systems without scope, $\texttt{use}\,x$ can be encoded as explained in the C++ example 7.2, by $\texttt{del}\,x_u\,\texttt{new}\,x_u$ for some new primitive component $x_u$, in combination with inserting $\texttt{new}\,x_u$ and $\texttt{del}\,x_u$ in front of every occurrence of $\texttt{new}\,x$ and $\texttt{del}\,x$, respectively. This clearly models the exclusive use of an instance of $x$, whereas joint use would also be interesting. (The difference can be seen in the expression $\texttt{new}\,x\,(\texttt{use}\,x \parallel \texttt{use}\,x)$.) The current type rule for scope in combination with the above encoding of $\texttt{use}\,x$ doesn't model the use of instances that exist outside the scope. Although this is a natural restriction for the deallocation of instances, it is not natural for just using them.

# References

[1] The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2004 edition. 7.2

[2] Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, 2007. ACM. 8.1, 8.1

[3] Hendrik P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992. 1, 8.2

[4] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 86–95, Washington, DC, USA, 2005. IEEE Computer Society. 8.1, 8.1

[5] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM. 8.1, 8.1

[6] Víctor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, June 2006. 8.1, 8.1

[7] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 480–491, New York, NY, USA, 2007. ACM. 8.1, 8.1

[8] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. Memory usage verification for OO programs. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2005. 8.1, 8.1

[9] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999. 8.1

[10] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 262–275, New York, NY, USA, 1999. ACM Press. 8.1, 8.1

[11] Karl Crary and Stephanie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN–SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM Press. 8.1, 8.1

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley Publishing Company, New York, NY, 1995. 1

[13] Martin Hofmann and Steffen Jost. 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*, pages 185–197, New York, NY, USA, 2003. ACM. 8.1, 8.1, 8.1

[14] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis (for an object-oriented language). In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP), Programming Languages and Systems*, volume 3924 of *LNCS*, pages 22–37. Springer, 2006. 8.1, 8.1, 8.1

[15] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. *SIGPLAN Not.*, 34(9):70–81, 1999. 8.1, 8.1

[16] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 410–423, New York, NY, USA, 1996. ACM. 8.1

[17] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, Third Edition*. Addison-Wesley, 1997. 5.2

[18] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 1, 8.2

[19] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. 1, 8.2

[20] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981. 3.2, 8.2

[21] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2005. 4.3

[22] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 2000. 7.2

[23] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. 8.1, 8.1, 8.1

[24] Hoang Truong. Guaranteeing resource bounds for component software. In Martin Steffen and Gianluigi Zavattaro, editors, *FMOODS*, volume 3535 of *Lecture Notes in Computer Science*, pages 179–194. Springer–Verlag, 2005. 8.2

[25] Hoang Truong and Marc Bezem. Finding resource bounds in the presence of explicit deallocation. In Dang Van Hung and Martin Wirsing, editors, *Proceedings ICTAC*, volume 3722 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2005. 8.2

[26] Leena Unnikrishnan, Scott D. Stoller, and Yanhong A. Liu. Optimized live heap bound analysis. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–85, London, UK, 2003. Springer-Verlag. 8.1, 8.1

[27] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. 8.2

[28] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. *ACM SIGSOFT Software Engineering Notes*, 30(5):115–125, 2005. 8.1, 8.1

[29] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. *ACM SIGSOFT Software Engineering Notes*, 28(5):307–316, 2003. 8.1, 8.1