Guaranteeing Resource Bounds for Component Software

Hoang Truong

Department of Informatics, University of Bergen,
PB.7800, N-5020 Bergen, Norway
hoang@ii.uib.no

Abstract. Component software is software that has been assembled from various pieces of standardized, reusable computer programs, so-called components. Executing component software creates instances of these components. For several reasons, for example, limited resources and/or application requirements, it can be important to have control over the number of such instances.

In the previous work [3], we have given an abstract component language and a type system which ensures that the number of simultaneously active instances of any component never exceeds a sharp bound expressed in the type. The language featured instantiation and reuse of components, as well as sequential composition, choice and scope.

This work extends the previous one to include a parallel composition. Moreover, we improve on the operational semantics by using a small-step reduction relation. As a result, we can prove the soundness property of our static type system using the technique of Wright and Felleisen.

1 Introduction

Component software is built from various components, possibly developed by third-parties [15], [20], [17], [8]. These components may in turn use other components. Upon execution instances of these components are created. For example, when we launch a web browser application it may create an instance of a dial-up network connection, an instance of a menubar and several instances of a toolbar, among others. Each toolbar may in turn create its own control instances such as buttons, addressbars, bookmarks, and so on.

The process of creating an instance of a component x does not only mean the allocation of memory space for x's code and data structures, the creation of instances of x's subcomponents (and so on), but possibly also the binding of other system and hardware resources. Usually, these resources are limited and components are required to have only a certain number of simultaneously active instances. In the above example, there should be only one instance of a menubar and one instance of a modem for network connection. Other examples come from the singleton pattern and its extensions (multitons), which have been widely discussed in literature [10], [9]. These patterns limit the number of objects of a certain class dynamically, at runtime.

When building large component software it can easily happen that different instances of the same component are created. Creating more active instances than allowed can lead to errors or even a system crash, when there are not enough resources for them. An example is resource-exhaustion DoS (Denial of Service) attacks which cause a temporary loss of services. There are several ways to meet this challenge, ranging from testing, runtime checking [9], to static analysis.

Type systems are a branch of static analysis. Type systems have traditionally been used for compile-time error-checking, cf. [4]. Recently, there are several works on using type systems for certifying important security properties, such as performance safety, memory safety, control-flow safety [14], [6], [5]. In component software, typing has been studied in relation to integrating components such as type-safe composition [19] or type-safe evolution [13]. In this paper we explore the possibility of a type system which allows one to detect *statically* whether or not the number of simultaneously active instances of specific components exceeds the allowed number. Note that here we only control resources by the number of instances. However, we can extend to more specific resources, such as memory, by adding annotations to components using such resources.

For this purpose we have designed a component language where we have abstracted away many aspects of components and have kept only those that are relevant to instantiation and composition. In the previous work [3], the main features are instantiation and reuse, sequential composition, choice and scope. In this work we add a parallel composition, which allows two expressions running independently at the same time. At the first look, the parallel composition seems adding only a small difficulty to the type system. However, we have found that we have to make substantial changes to the type system to obtain sharp upper bounds as in [3]. As before, reusing a component means to use an existing instance of the component if there is already one, and to create a new instance only if there exists none. Though abstract, the strength of the primitives for composition is considerable. Choice allows us to model both conditionals and non-determinism (due to, e.g., user input). It can also be used when a component have several compatible versions and the system can choose one of them at runtime. Scope is a mechanism to deallocate instances but it can also be used to model method calls. Sequential composition is associative.

The operational semantics in this work has also been improved as compared to the previous one. Instead of using a big-step operational semantics, here we use a small-step reduction relation and as a result, we can prove the soundness of our type system using the technique of Wright and Felleisen [18].

The type inference algorithm for this system is almost the same as in [3]. We still have a polynomial time type inference algorithm but we leave it out here for the sake of brevity.

The paper is organized as follows. Section 2 introduces the component language and a small-step operational semantics. In Section 3 we define types and the typing relation. Properties of the type system and the operational semantics are presented in Section 4. Last, we outline some future directions.

2 A Component Language

2.1 Terms

Component programs, declarations and expressions are defined in Table 1. In the definition we use extended Backus-Naur Form with the following meta-symbols: infix | for choice and overlining for Kleene closure (zero or more iterations).

$Prog$	$::= \overline{Decls}; E$		Program
$Decls$	$::= \overline{x \prec E}$		Declarations
$A, .., E$	$::=$		Expressions
	ϵ		Empty expression
	$\mid$ $\mathtt{new}\, x$		New instantiation
	$\mid$ $\mathtt{reu}\, x$		Reuse instantiation
	$\mid$ $E\ E$		Sequential composition
	$\mid$ $(E + E)$		Choice composition
	$\mid$ $(E \parallel E)$		Parallel composition
	$\mid$ $\{E\}$		Scope

Table 1. Syntax

We use $a, b, .., z$ for component names and $A, .., E$ for expressions. We collect all component names in a set $\mathbb{C}$.

We have two primitives (`new` and `reu`) for creating and (if possible) reusing an instance of a component, and four primitives for composition (sequential composition denoted by juxtaposition, $+$ for choice, $\parallel$ for parallel, and $\{\ldots\}$ for scope). Together with the empty expression ϵ these generate so-called *component expressions*. A *declaration* $x \prec E$ states how the component x depends on subcomponents as expressed in the component expression E. If x has no subcomponents then E is ϵ and we call x a *primitive component*. Upon instantiation or reuse of x the expression E is executed. A *component program* consist of declarations and ends with a *main expression* which sparks off the execution, see Section 2.2.

The following example is a well-formed component program:

$$d \prec \epsilon \quad e \prec \epsilon \quad a \prec (\mathtt{new}\, d \parallel \{\mathtt{reu}\, d\}\, \mathtt{reu}\, e)$$
$$b \prec (\mathtt{reu}\, d\{\mathtt{new}\, a\} + \mathtt{new}\, e\, \mathtt{new}\, a)\, \mathtt{reu}\, d; \quad \mathtt{reu}\, b$$

In this example, d and e are primitive components. Component a is the parallel composition of $\mathtt{new}\, d$ and $\{\mathtt{reu}\, d\}\, \mathtt{reu}\, e$. Component b has a choice expression before reuse of an instance of d. The first subexpression of the choice expression is $\mathtt{reu}\, d\{\mathtt{new}\, a\}$.

We can view $\{\mathtt{new}\, a\}$ in this expression as a function call $f()$ (in traditional programming languages). Function f then has body $\mathtt{new}\, a$, which means $f()$ needs a new instance of a to carry out its task. We abstract from the details of this job, the only relevant aspect here is that it involves a new instance of a which will be deallocated upon exiting f.

The example is simple, but as we will see in the next section, there are many possible runs of the program, resulting in difference numbers of instances for each component during and after each run.

2.2 Operational Semantics

The operational semantics is based on a reduction relation and a structural congruence. The reduction relation is a set of small-step reduction rules between *configurations*. The structural congruence, essentially commutativity of $+$ and $\|$, allows us to rearrange the structure of a configuration so that reduction rules may be applied. In the sequel we assume that we are working with a program $Prog = Decls; E$ and $x \prec A \in Decls$ denotes that $x \prec A$ is a declaration in $Decls$.

Before going into the details of congruence and reduction rules, we define our notion of configuration and its relevant components. A configuration is a binary tree $\mathbb{T}$ of threads. A thread is a stack ST of pairs of a local store and a expression (M, E), where M is a multiset over component names $\mathbb{C}$, and E is an expression as defined in Table 1. A thread is *active* if it is a leaf thread. Reduction always occurs at one of the leaf/active threads. A configuration is *terminal* if it has only one thread of the form (M, ϵ). Stacks and configurations are defined as follows:

ST ::=	$(M_1, E_1) \circ ... \circ (M_n, E_n)$	Stack
$\mathbb{T}, \mathbb{S}$ ::=		Configurations
	$\mathsf{Lf}(ST)$	Leaf
	$\mid \mathsf{Nd}(ST, \mathbb{T})$	Node with one branch
	$\mid \mathsf{Nd}(ST, \mathbb{T}, \mathbb{T})$	Node with two branches

Multisets are denoted by $[\ldots]$, where sets are denoted, as usual, by $\{\ldots\}$. $M(x)$ is the multiplicity of element x in multiset M and $M(x) = 0$ if $x \notin M$. The operation $\cup$ is union of multisets: $(M \cup N)(x) = max(M(x), N(x))$. The operation $\uplus$ is additive union of multisets: $(M \uplus N)(x) = M(x) + N(x)$. We write $M + x$ for $M \uplus [x]$ and when $x \in M$ we write $M - x$ for $M - [x]$.

We assign to each node in our tree a *location*. Let α, β range over locations. A location is a sequence over $\{l, r\}$. The root is assigned the empty sequence. The locations of two direct nodes from the root are l and r. The locations of the two direct child nodes of l are ll and lr, and so on. In general, αl and αr are the locations of the direct children of α. We write $\alpha \in \mathbb{T}$ when α is a valid location in tree $\mathbb{T}$. Whenever a new node is created, a location is assigned to it and this location will not be changed by rule conBranch.

Since the location of a parent node is a subsequence of the location of its children (direct and indirect), we define the following binary prefix ordering relation $\leq$ over locations. For location $\alpha = s_0 s_1 .. s_n$ where $s_i \in \{l, r\}$, $\alpha' \leq \alpha$ if $\alpha' = s_0 s_1 .. s_m$, $0 \leq m \leq n$. The set of all locations in a tree and this binary relation form a partially ordered set [7]. A maximal element of this partially ordered set is the location of a leaf. We denote by $\mathsf{leaves}(\mathbb{T})$ the set of locations of all the leaves of $\mathbb{T}$.

We denote by $\mathbb{T}(\alpha)$ the stack at location α in $\mathbb{T}$. We write $ST = (M_1, E_1) \circ .. \circ (M_n, E_n)$ for a stack of n elements where (M_1, E_1) is the bottom and (M_n, E_n) is the top of the stack. '$\circ$' is the stack separator. We call $\alpha.k$ the *position* of the kth element (from the bottom) of the stack $\mathbb{T}(\alpha)$. Again the set of all positions $\alpha.k$ in tree $\mathbb{T}$ is a partially ordered set with the following binary relation. $\alpha_1.k_1 \leq \alpha_2.k_2$ if either $\alpha_1 = \alpha_2$ and $k_1 \leq k_2$, or $\alpha_1 < \alpha_2$. We denote by $\mathsf{hi}(ST)$ the height of the stack and $ST|_k$ is the stack of from bottom to the kth element: $ST|_k = (M_1, E_1) \circ .. \circ (M_k, E_k)$. By $[ST|_k]$ we denote the multiset of active instances in $ST|_k$, so $[ST|_k] = M_1 \uplus .. \uplus M_k$. We simply write $[ST]$ when $k = \mathsf{hi}(ST)$. We denote by $[\mathbb{T}]$ the multiset of all active instances in $\mathbb{T}$: $[\mathbb{T}] = \biguplus_{\alpha \in \mathbb{T}} [\mathbb{T}(\alpha)]$

(osNew) $\quad x \prec A \in \mathit{Decls}$
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, \mathtt{new}\, x E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M + x, AE))]\!]_\alpha$

(osReu1) $\quad x \prec A \in \mathit{Decls} \quad x \notin \mathsf{reuLf}_{\mathbb{T}}(\alpha.\mathsf{hi}(\mathbb{T}(\alpha)))$
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, \mathtt{reu}\, x E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M + x, AE))]\!]_\alpha$

(osReu2) $\quad x \prec A \in \mathit{Decls} \quad x \in \mathsf{reuLf}_{\mathbb{T}}(\alpha.\mathsf{hi}(\mathbb{T}(\alpha)))$
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, \mathtt{reu}\, x E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M, AE))]\!]_\alpha$

(osChoice)
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, (A + B)E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M, AE))]\!]_\alpha$

(osPush)
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, \{A\}E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M, E) \circ ([], A))]\!]_\alpha$

(osPop)
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, E) \circ (M', \epsilon))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M, E))]\!]_\alpha$

(osParIntr)
$\mathbb{T}[\![\mathsf{Lf}(ST \circ (M, (A \parallel B)E))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}(([], A)), \mathsf{Lf}(([], B)))]\!]_\alpha$

(osParElim1)
$\mathbb{T}[\![\mathsf{Nd}(ST \circ (M, E), \mathbb{S}, \mathsf{Lf}((M', \epsilon)))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Nd}(ST \circ (M \uplus M', E), \mathbb{S})]\!]_\alpha$

(osParElim2)
$\mathbb{T}[\![\mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}((M', \epsilon)))]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathsf{Lf}(ST \circ (M \uplus M', E))]\!]_\alpha$

(osCong) $\quad \mathbb{S} \equiv \mathbb{S}'$
$\mathbb{T}[\![\mathbb{S}]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathbb{S}']\!]_\alpha$

Table 2. Reduction rules

The next notion is that of *reusable instances* because the primitive `reu` depends on the state of the configuration. In our model, the instantiation always occurs at the top of a leaf stack, for the moment we only need the concept of reusable instances for an expression at a leaf node. Later, we will extend the notion of reusable instances to non-leaf nodes. The multiset of reuseable

(conChoice)
$\mathsf{Lf}(ST \circ (M, (A+B)E)) \;\equiv\; \mathsf{Lf}(ST \circ (M, (B+A))E)$
(conBranch)
$\mathsf{Nd}(ST, \mathsf{Lf}(ST), \mathbb{T}) \;\equiv\; \mathsf{Nd}(ST, \mathbb{T}, \mathsf{Lf}(ST))$

Table 3. Structural congruence: basic axioms

instances at level k of the leaf stack α is the collection of all existing instances in all the predecessor nodes $\beta < \alpha$ and all the existing instances from the bottom of stack $\mathbb{T}(\alpha)$ up to k (inclusion).

$$\mathsf{reuLf}_{\mathbb{T}}(\alpha.k) = \biguplus_{\beta < \alpha} [\mathbb{T}(\beta)] \;\;\uplus\;\; [\mathbb{T}(\alpha)|_k]$$

The reduction relation is defined in terms of a rewriting system [16]. By $\mathbb{T}[\![]\!]_\alpha$ we denote a tree with a hole at the leaf location α. Filling this hole with a (sub)tree $\mathbb{T}'$ will be denoted by $\mathbb{T}[\![\mathbb{T}']\!]_\alpha$.

Table 2 defines the reduction rules. Each reduction rule has two lines. The first line contains a rule name followed by a list of conditions. The second line has the form $\mathbb{T} \longrightarrow \mathbb{T}'$, which states that if the configuration has the form $\mathbb{T}$ and the condition in the first line holds, then we can move to configuration $\mathbb{T}'$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$. One step reduction is defined first by choosing an arbitrary active thread. Then depending on the pattern of the expression at the top of the chosen thread and the state of the configuration, the appropriate rewrite rule is selected. If necessary the configuration is rearranged using the congruence rules. By the rules osNew, osReu1, osReu2, and osChoice we only rewrite the element at the top of the stack. The rule osPush adds an element to the top of the leaf stack. The rule osPop only removes the element at the top of the stack when the stack has at least two elements. That means no stack in any configuration is empty. By the rule osParIntr, a leaf is replaced by a branch of a node and two leaves. In contrast, by the rules osParElim1, osParElim2, a leaf is removed from the tree and its parent node may be promoted to be a leaf if it is the case (osParElim2). The rule osCong allows the configuration to be rearranged so that reduction rule can be applied.

The structural congruence relation $\equiv$ is defined in Table 3. By the congruence rules, we can replace the left hand side of $\equiv$ by the right hand side in the reduction rule osCong.

The example at the end of Section 2.1 is used to illustrate the operational semantics. There are many possible runs of the program due to the choice composition and when a configuration has more than one leaf thread, the number of possible runs can be exponential as active threads have the same priority. Here we only show one of the possible runs. To make it easier to follow, we represent the trees graphically instead of using the formal syntax; '$\llcorner$' and '$\langle$' denote branches with one and two child nodes, respectively. At the starting point, the configuration has one leaf $\mathsf{Lf}([], \mathtt{reu}\, b)$. After the first step, there are two possibilities because we can apply the congruence rule conChoice before the rule osChoice.

$$
\begin{array}{rl}
(\mathsf{Start}) & ([], \texttt{reu}\, b) \\
(\mathsf{osNew}) \longrightarrow & ([b], (\, \texttt{reu}\, d\{\, \texttt{new}\, a\} + \texttt{new}\, e\, \texttt{new}\, a)\, \texttt{reu}\, d) \\
(\mathsf{osChoice}) \longrightarrow & ([b],\, \texttt{reu}\, d\{\, \texttt{new}\, a\}\, \texttt{reu}\, d) \\
 & \qquad\qquad\qquad (\text{or } ([b],\, \texttt{new}\, e\, \texttt{new}\, a\, \texttt{reu}\, d))
\end{array}
$$

Now we continue with the first possibility. When there are two or more leaves, we draw a box around the leaf which is to be executed in the next step.

$$
\begin{array}{rl}
 & ([b],\, \texttt{reu}\, d\{\, \texttt{new}\, a\}\, \texttt{reu}\, d) \\
(\mathsf{osReu1}) \longrightarrow & ([b,d], \{\, \texttt{new}\, a\}\, \texttt{reu}\, d) \\
(\mathsf{osPush}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([],\, \texttt{new}\, a) \\
(\mathsf{osNew}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a], (\, \texttt{new}\, d \parallel \{\, \texttt{reu}\, d\}\, \texttt{reu}\, e)) \\
(\mathsf{osParIntr}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a], \epsilon) \left\langle \begin{array}{l} ([],\, \texttt{new}\, d) \\ \boxed{([], \{\, \texttt{reu}\, d\}\, \texttt{reu}\, e)} \end{array} \right. \\
(\mathsf{osPush}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a], \epsilon) \left\langle \begin{array}{l} \boxed{([],\, \texttt{new}\, d)} \\ ([],\, \texttt{reu}\, e) \circ ([],\, \texttt{reu}\, d) \end{array} \right. \\
(\mathsf{osNew}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a], \epsilon) \left\langle \begin{array}{l} ([d], \epsilon) \\ \boxed{([],\, \texttt{reu}\, e) \circ ([],\, \texttt{reu}\, d)} \end{array} \right. \\
(\mathsf{osReu1}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a], \epsilon) \left\langle \begin{array}{l} \boxed{([d], \epsilon)} \\ ([],\, \texttt{reu}\, e) \circ ([], \epsilon) \end{array} \right. \\
(\mathsf{osElim1}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a,d], \epsilon) \leftharpoonup ([],\, \texttt{reu}\, e) \circ ([], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a,d], \epsilon) \leftharpoonup ([],\, \texttt{reu}\, e) \\
(\mathsf{osReu}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a,d], \epsilon) \leftharpoonup ([e], \epsilon) \\
(\mathsf{osParElim2}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \circ ([a,d,e], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b,d],\, \texttt{reu}\, d) \\
(\mathsf{osReu2}) \longrightarrow & ([b,d], \epsilon) \quad (\text{terminal})
\end{array}
$$

Last, we should note that we could model our operational semantics slightly simpler by using only *complete binary trees*. A complete binary tree is a binary tree with the additional property that every node must have exactly two children if an internal node, and zero children if a leaf node. Then we have only one rule for truncating the tree:

(osParElim)

$$\mathbb{T}[\mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathsf{Lf}((M'', \epsilon)))]_\alpha \longrightarrow \mathbb{T}[\mathsf{Lf}(ST \circ (M \uplus M' \uplus M'', E))]_\alpha$$

However, doing in this way reduces the reuse capability because two sibling threads cannot reuse instances of each other, after one has terminated before the other. In our model this is possible as a leaf can return its instances to its parent and the other sibling branch can reuse the instances from its parent.

3 Type System

We start this section by describing informally types and gives some intuitive examples. Then we will define and explain the typing rules in more details.

Definition 1 (Types). *Types of component expressions are tuples*

$$X = \langle X^i, X^o, X^j, X^p, X^l \rangle$$

where X^i, X^o, X^j, X^p *and* X^l *are finite multisets over* $\mathbb{C}$*. We let* $U, V, \ldots, Z$ *range over types.*

Let us first explain informally why multisets, which multisets and why five. The aim is to have an upper bound of the number of simultaneously active instances of any component during the execution of the expression (X^i). Multisets are the right data structure to collect and count such instances.

In addition, we want compositionality of typing, that is, we want the types to be computable from types of subexpressions. Since subexpressions may be scoped, it is necessary to have an upper bound of the number of instances that are still active *after* the execution of an expression (X^o). Pairs $\langle X^i, X^o \rangle$ sufficed for the purpose of the paper [2]. Here we consider also reusing instances of components and this depends on whether there is already such an instance or not. More concretely, in a sequential composition of A and B, the behaviour of `reu`'s in B depends on the instances that are active *after* the execution of A, which would violate the compositionality. In order to save the compositionality, we have to add three more multisets to the types, denoted by X^j, X^p and X^l. The first two multisets X^j, X^p express the same bounds as X^i, X^o, but with respect to executing the expression in a state where every component has already one active instance.

Without the parallel composition, these four multisets $\langle X^i, X^o, X^j, X^p \rangle$ sufficed for the purpose of [3] since the difference between $X^i(x)$ and $X^j(x)$ as well as between $X^o(x)$ and $X^p(x)$ is at most one for every x. With the new parallel composition, these differences may be greater than one, and note that due to the non-determinism of the choice composition, the surviving instances after executing A is also non-deterministic. In order to obtain a sharp bound for x, we need to know whether B can always reuse x after executing A or not. Because if it is the case, the maximum number of additional instances of x generated by B is only $Y^j(x)$, where Y is the type of B. Therefore, we need the last component X^l in the type expression. X^l is the set of instances which always active after executing A. Although X^l can be a set, we let X^l be a multiset so that the multiset operations in the later sessions can be applied without any conversion.

Based on the above intuitions, the following typings are easy:
$\mathtt{new}\,d : \langle [d], [d], [d], [d], [d] \rangle$, $\{\,\mathtt{new}\,d\} : \langle [d], [], [d], [], [] \rangle$, $\mathtt{reu}\,d : \langle [d], [d], [], [], [d] \rangle$, $\mathtt{reu}\,d\{\,\mathtt{new}\,d\} : \langle [d, d], [d], [d], [], [d] \rangle$, $\mathtt{reu}\,d\{\,\mathtt{new}\,a\} : \langle [a, d, d], [d], [a, d], [], [d] \rangle$, $(\,\mathtt{reu}\,d \parallel \mathtt{new}\,e) : \langle [d, e], [d, e], [e], [e], [d, e] \rangle$, $(\,\mathtt{reu}\,d + \mathtt{new}\,e) : \langle [d, e], [d, e], [e], [e], [] \rangle$, where $d \prec \epsilon$ and $a \prec \mathtt{new}\,d$ like in the example program in Section 2.1.

The intuitions from the above paragraphs will be indispensable for understanding the typing rules later in this section, in particular the sequencing rule. We will explain more when describing each typing rule, but before that we have to prepare with some preliminary definitions.

Let $\mathcal{R}$ be the requirement that some components in $\mathbb{C}$ can have at most a certain number of simultaneous instances. $\mathcal{R}$ can be viewed as a total function from $\mathbb{C}$ to $\mathbb{N} \cup \{\infty\}$. Then $\mathcal{R}(x) \in \mathbb{N}$ is the maximum allowed number of simultaneously active instances of x; $\mathcal{R}(x) = \infty$ expresses that x can have any number of instances. By convention $n < \infty$ for all $n \in \mathbb{N}$. For any multiset M, we denote $M \subseteq \mathcal{R}$ when $M(x) \leq \mathcal{R}(x)$ for all $x \in M$.

A *basis* or an *environment* is an list of declarations: $x_1 \prec A_1, \ldots, x_n \prec A_n$ with distinct variables $x_i \neq x_j$ for all $i \neq j$, as in [1]. Let Γ, Δ range over bases. The domain of basis $\Gamma = x_1 \prec A_1, \ldots, x_n \prec A_n$, denoted by $Dom(\Gamma)$, is the set $\{x_1, \ldots, x_n\}$. A typing judgment is a tuple of the form

$$\Gamma \vdash_{\mathcal{R}} A : X$$

and it asserts that expression A has type X in the environment Γ, with respect to requirement $\mathcal{R}$. We leave out subscript $\mathcal{R}$ when $\mathcal{R}$ is clear from context.

Definition 2 (Typing rules). *Type judgments $\Gamma \vdash A : X$ are derived by applying the typing rules in Table 4 in the usual inductive way.*

In rule Seq in Table 4, expression $M!_N$, where M, N are multisets, is defined as follows:

$$(M!_N)(x) = \begin{cases} 0, & \text{if } x \in N \\ M(x), & \text{otherwise} \end{cases}$$

Besides the intuition given in the beginning of this section, some further explanation of these typing rules is in order. The rule Axiom requires no premise and is used to take-off. The rules New and Reu allow us to type expressions `new` x and `reu` x, respectively. The rule Weaken is used to expand bases so that we can combine typings in the other rules. The side condition $x \notin Dom(\Gamma)$ in the rules Weaken, New and Reu keeps the expanded basis well-formed. The rules Choice and Scope are easy to understand recalling the semantics of the corresponding reduction rules osChoice, osPush and osPop. In the rule Parallel, since we have no specific schedule for two parallel threads, both can generate their maximum numbers of instances for any component. To be on the safe side, we have to prepare for the worst case and therefore the type of two parallel expressions is additive union of their types. The side condition follows naturally.

The most critical rule is Seq because sequencing two expressions can lead to increase in instances of the composed expression. Let us start with the first component of type expression for AB. After expression A is executed, there are at most $X^o(x)$ instances of component x. If x is not in the system state after the execution of A, then at most $Y^i(x)$ instances of x can be created when executing B. Otherwise, at most $Y^j(x)$ additional instances of x can be created. If we take the maximum of $(X^o \uplus Y^j)(x)$ and $Y^i(x)$ to be the maximum number

(Axiom)

$$\frac{}{\vdash \epsilon : \langle [], [], [], [], [] \rangle}$$

(Weaken)

$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad x \notin Dom(\Gamma)}{\Gamma, x \prec B \vdash A : X}$$

(New)

$$\frac{\Gamma \vdash A : X \quad x \notin Dom(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : \langle X^i + x, X^o + x, X^j + x, X^p + x, X^l + x \rangle}$$

(Reu)

$$\frac{\Gamma \vdash A : X \quad x \notin Dom(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{reu}\, x : \langle X^i + x, X^o + x, X^j, X^p, X^l + x \rangle}$$

(Seq)

$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad X^o \uplus Y^j \subseteq \mathcal{R} \quad A, B \neq \epsilon}{\Gamma \vdash AB : \langle X^i \cup (X^o \uplus Y^j) \cup Y^i!_{X^l}, (X^o \uplus Y^p) \cup Y^o!_{X^l}, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p, X^l \cup Y^l \rangle}$$

(Choice)

$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y}{\Gamma \vdash (A + B) : \langle X^i \cup Y^i, X^o \cup Y^o, X^j \cup Y^j, X^p \cup Y^p, X^l \cap Y^l \rangle}$$

(Parallel)

$$\frac{\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad X^i \uplus Y^i \subseteq \mathcal{R}}{\Gamma \vdash (A \parallel B) : \langle X^i \uplus Y^i, X^o \uplus Y^o, X^j \uplus Y^j, X^p \uplus Y^p, X^l \cup Y^l \rangle}$$

(Scope)

$$\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : \langle X^i, [], X^j, [], [] \rangle}$$

Table 4. Typing rules

of x which can be created after the execution of A and during the execution of B, then we do not obtain the *sharp* upper bound. For example, let $A = \mathtt{reu}\, x$ and $B = (\mathtt{reu}\, x \parallel \mathtt{reu}\, x)$. Executing B alone can create two instances of x. However, executing AB creates only one instance of x.

To remedy the situation we need to know whether an instance of x is always in the system state after the execution of A or not. If it is, then we know that at most $Y^j(x)$ additional instances can be created; otherwise, $Y^i(x)$ additional instances can be created when executing B. Therefore the maximum number of x after execution of A and during execution of B are $(X^o \uplus Y^j)(x)$, or $(X^l \uplus Y^j)(x)$ if $X^l(x) \geq 1$, or $Y^i(x)$ if $X^l(x) = 0$. Since $X^o \supseteq X^l$, the number becomes $((X^o \uplus Y^j) \cup Y^i!_{X^l})(x)$.

In addition, because executing A can create at most $X^i(x)$ instances, the first component of type of AB is the maximum of $X^i(x)$ and $((X^o \uplus Y^j) \cup Y^i!_{X^l})(x)$. Last, since X^i and Y^i satisfy the requirement $\mathcal{R}$, we only require an additional side condition $X^o \uplus Y^j \subseteq \mathcal{R}$ which means $X^o(x) + Y^j(x) \leq \mathcal{R}(x)$ for each $x \in \mathbb{C}$.

Analogously, after executing AB, the maximal number of surviving instances of x are $X^o(x) + Y^p(x)$, or $Y^o(x)$ if there is a run of A which ends with no surviving instance of x. Hence the surviving instances of AB are $(X^o \uplus Y^p) \cup Y^o!_{X^l}$.

By a similar reasoning, when we start with a stack containing at least one instance of every component, we can calculate the second and the last components in the type expression for AB and the whole type expression of AB is $\langle X^i \cup (X^o \uplus Y^j) \cup Y^i!_{X^l}, (X^o \uplus Y^p) \cup Y^o!_{X^l}, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p, X^l \cup Y^l \rangle$.

Using the example in Section 2.1 with assumption that $\mathcal{R} = \{b \mapsto 1, e \mapsto 2, a, d \mapsto 4\}$, we derive type for $\texttt{reu}\,b$. Note that we omitted some side conditions as they can be checked easily and we shortened the rule names to the first two characters. The rule $\textsf{Axiom}$ is also simplified.

$$\textsf{We}\,\dfrac{\textsf{Sc}\,\dfrac{\textsf{Re}\,\dfrac{\vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \prec \epsilon \vdash \texttt{reu}\,d : \langle [d], [d], [], [], [d] \rangle}}{d \prec \epsilon \vdash \{\,\texttt{reu}\,d\} : \langle [d], [], [], [], [] \rangle} \quad \textsf{We}\,\dfrac{\vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \prec \epsilon \vdash \epsilon : \langle [], [], [], [], [] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \{\,\texttt{reu}\,d\} : \langle [d], [], [], [], [] \rangle} \tag{1}$$

$$\textsf{Se}\,\dfrac{(1) \quad \textsf{Re}\,\dfrac{\textsf{We}\,\dfrac{\vdash \epsilon : \langle [], [], [], [], [] \rangle \quad \vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \prec \epsilon \vdash \epsilon : \langle [], [], [], [], [] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \texttt{reu}\,e : \langle [e], [e], [], [], [e] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \{\,\texttt{reu}\,d\}\,\texttt{reu}\,e : \langle [d, e], [e], [], [], [e] \rangle} \tag{2}$$

$$\textsf{Ne}\,\dfrac{\textsf{Pa}\,\dfrac{\textsf{We}\,\dfrac{\textsf{Ne}\,\dfrac{\vdash \epsilon : \langle [], [], [], [], [] \rangle}{d \prec \epsilon \vdash \texttt{new}\,d : \langle [d], [d], [d, [d], [d] \rangle}}{d \prec \epsilon, e \prec \epsilon \vdash \texttt{new}\,d : \langle [d], [d], [d, [d], [d] \rangle} \quad (2)}{d \prec \epsilon, e \prec \epsilon \vdash (\,\texttt{new}\,d \parallel \{\,\texttt{reu}\,d\}\,\texttt{reu}\,e) : \langle [d, d, e], [d, e], [d], [d], [d, e] \rangle}}{d \prec \epsilon, e \prec \epsilon, a \prec (\,\texttt{new}\,d \parallel \{\,\texttt{reu}\,d\}\,\texttt{reu}\,e) \vdash \texttt{new}\,a : \langle [a, d, d, e], [a, d, e], [a, d], [a, d], [a, d, e] \rangle}$$

Similarly, we can derive $\Gamma \vdash \texttt{reu}\,b : \langle [b, a, d, d, e], [b, a, d, e], [a, d, e], [a, d, e], [a, b, d, e] \rangle$ where $\Gamma = d \prec \epsilon, e \prec \epsilon, a \prec (\,\texttt{new}\,d \parallel \{\,\texttt{reu}\,d\}\,\texttt{reu}\,d), b \prec (\,\texttt{reu}\,d\{\,\texttt{new}\,a\} + \texttt{new}\,e\,\texttt{new}\,a)\,\texttt{reu}\,d$.

In this example expression $\texttt{reu}\,b$ is typable. If $\mathcal{R}(d) = 1$, the expression would not be typable as the side condition when paralleling $\texttt{new}\,d$ and $\{\,\texttt{reu}\,d\}\,\texttt{reu}\,e$ would not be satisfied. Also, note that the above type derivation is not the only one but, as we will see later, the type for any expression is unique.

As mentioned at the Section 1, we can infer specific resource consumption from our types by adding annotations to the source programs. For example, if component a and d each create a database connection, then from the type of b, we know that the program, in particular the main expression $\texttt{reu}\,b$, may need three database connections (since the first component in the type of b has one a and two d's). From another point of view, we view d as a database connection component, then we know that the program needs two database connections.

We end this section with the definition of *well-typed program*.

Definition 3 (Well-typed programs). *Let $\mathcal{R}$ be a requirement. Program $Prog = Decls; E$ is well-typed w.r.t. $\mathcal{R}$ if there exists a reordering Γ of declarations in Decls such that $\Gamma \vdash_R E : X$.*

4 Formal Properties

4.1 Type Soundness

A fundamental property of static type systems is *type soundness* or *safety* [4]. It states that well-typed programs cannot cause type errors. In our case, type errors occur when a configuration violates requirement $\mathcal{R}$, that is, there exists a component x whose the number of its active instances is greater than the allowed number, $\mathcal{R}(x)$.

Our proof of the type soundness is based on the approach of Wright and Felleisen [18]. We will prove two main lemmas: Preservation and Progress. The first lemma states that well-typedness is preserved under reduction. The latter guarantees that well-typed programs cannot get stuck, that is, move to a non-terminal state, from which it cannot move to another state. In order to use this technique, we need to define the notion of *well-typed configuration*. Before giving the formal definition of well-typed configuration we need some auxiliary definitions.

The first notion is *subtree*. Given a tree $\mathbb{T}$ and a set of positions $\mathcal{L} = \{\alpha_i.k_i \in \mathbb{T} \mid 1 \leq i \leq m\}$ such that $\alpha_i.k_i \not\leq \alpha_j.k_j$ for all $i \neq j$ and for all leaf $\alpha \in \mathsf{leaves}(\mathbb{T})$ there exists a h such that $\alpha \geq \alpha_h \in \mathcal{L}$. That means for every path from the root of $\mathbb{T}$ to one of its leaves we select one and only one position for set $\mathcal{L}$. In the sequel, we assume that $\mathcal{L}$ always satisfies these conditions. The tree $\mathbb{S}$ obtained from $\mathbb{T}$ by keeping only elements at positions $\alpha.k \leq \alpha_i.k_i$ for $1 \leq i \leq m$ is a subtree of $\mathbb{T}$, notation $\mathbb{S} \sqsubseteq_{\mathcal{L}} \mathbb{T}$. Consequently, $\mathsf{leaves}(\mathbb{S}) = \{\alpha_1, .., \alpha_m\}$ and $\mathsf{hi}(\mathbb{S}(\alpha_i)) = k_i$ for all $1 \leq i \leq m$.

The next one is the notion of the reusable instances for the expression E at an arbitrary position $\alpha.k$. Recall that we have defined the reusable instances for an expression in a leaf node in Section 2.2. Now we extend this notion for an arbitrary position $\alpha.k$. Due to the nondeterminism of our operational semantics, the collection of reusable instances for an expression in a non-leaf node is also non-deterministic, but we can calculate its sharp upper bound and a lower bound. Note that due to the semantics of `reu`, it is enough for the latter being a collection of instances which E can always reuse; it needs not to be a sharp lower bound. We define the latter first and denote this collection by $\mathsf{reul}_{\mathbb{T}}(\alpha.k)$.

The element of $\mathsf{reul}_{\mathbb{T}}(\alpha.k)$ is not only those in $\mathsf{reuLf}_{\mathbb{T}}(\alpha.k)$ but also ones returned from its child nodes, $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$ (see the rules $\mathsf{osParElim1}, \mathsf{osParElim2}$ in Table 2.)

$$\mathsf{reul}_{\mathbb{T}}(\alpha.k) = \mathsf{reuLf}_{\mathbb{T}}(\alpha.k) \cup \mathsf{retl}_{\mathbb{T}}(\alpha.k)$$

The set of instances returned to $\alpha.k$ is empty if $\alpha.k$ is not at the top of α. Otherwise, it contains instances which will be generated at the bottom of its child nodes. Since the child nodes may have more children, we need to make recursive calls to them.

$$\mathsf{retl}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha.k \notin \mathbb{T} \\ \bigcup_{\beta \in \{\alpha l, \alpha r\}} ([\mathbb{T}(\beta.1)] \cup X^l \cup \mathsf{retl}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where X is the type of the expression at position $\beta.1$ and $\mathbb{T}(\beta.1)$ is the multiset at position $\beta.1$.

Analogously, for the sharp upper bound, the maximal number of instances returned to a position $\alpha.k$ ($\mathsf{retop}_{\mathbb{T}}(\alpha.k)$) is zero if k is not at the top of the stack at α. Otherwise, it contains the one in the multisets at the bottom of its child nodes and the maximal number of instances which survive the expressions here. This number, $\mathsf{op}_{\mathbb{T}}(\alpha.k)$, is calculated as in the sequencing typing rule Seq. Last, the child nodes of $\alpha.k$ may received instances from its child nodes and so on, so we need to call the function recursively. To simplify the definition of the function retl and retop with recursion, we let the function return an empty multiset for invalid positions $\alpha.k \notin \mathbb{T}$.

$$\mathsf{retop}_{\mathbb{T}}(\alpha.k) = \begin{cases} [], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha.k \notin \mathbb{T} \\ \biguplus_{\beta \in \{\alpha l, \alpha r\}}([\mathbb{T}(\beta.1)] \uplus \mathsf{op}_{\mathbb{T}}(\beta.1) \uplus \mathsf{retop}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where

$$\mathsf{op}_{\mathbb{T}}(\alpha.k) = X^p \cup X^o!_{\mathsf{reul}_{\mathbb{T}}(\alpha.k)}$$

Here X is the type of the expression at position $\alpha.k$.

We are going to define the central notion of well-typed configuration. Its main statement is that the total number of active instances in the configuration respects the requirement $\mathcal{R}$. Since the leaves of the configuration tree may generate more instances, we need to include these instances to the above total number. Furthermore, because the tree can shrink and when it shrinks, some nodes eventually become leaves we need to prove for these future states also. The function $\mathsf{ij}_{\mathbb{T}}(\alpha.k)$ below returns a multiset which is the maximal number of instances which can be generated by the expression at the position $\alpha.k$. As in the sequencing typing rule Seq, this number is bounded by the maximal number returned from its child nodes ($\mathsf{retop}_{\mathbb{T}}(\alpha.k)$) and the additional instances (X^j) for components that indeed are reused, where X is the type of the expression at position $\alpha.k$. For runs after which x may not be in the set of reusable instances, an additional bound $X^i(x)$ should be taken into account. This explains the definition of the function ij.

$$\mathsf{ij}_{\mathbb{T}}(\alpha.k) = (\mathsf{retop}_{\mathbb{T}}(\alpha.k) \uplus X^j) \cup X^i!_{\mathsf{reul}_{\mathbb{T}}(\alpha.k)}$$

Now we are ready to define the notion of a *well-typed configuration*. The first clause requires that all expressions in the configuration are well-typed. The second one contains the safety behaviour of the configuration. It requires that the total number of existing instances in the configuration and the ones which may be generated by expressions in the future still respect the requirement $\mathcal{R}$.

Definition 4 (Well-typed configuration). *Let Γ be a legal basis. Configuration $\mathbb{T}$ is well-typed with respect to requirement $\mathcal{R}$ if*

1. *for every E occurring in $\mathbb{T}$ there exists X such that $\Gamma \vdash E : X$, and*
2. *for all $\mathbb{S} \sqsubseteq_{\mathcal{L}} \mathbb{T}$:*
$$[\mathbb{S}] \uplus \biguplus_{\alpha.k \in \mathcal{L}} \mathsf{ij}_{\mathbb{T}}(\alpha.k) \subseteq \mathcal{R}$$

Having the definition of well-typed configuration, the two main lemmas mentioned at the beginning of the section are stated as follows.

Lemma 1 (Preservation). *If $\mathbb{T}$ is a well-typed configuration and $\mathbb{T} \longrightarrow \mathbb{T}'$, then $\mathbb{T}'$ is well-typed.*

Lemma 2 (Progress). *If $\mathbb{T}$ is a well-typed configuration, then either*

1. *there exists configuration $\mathbb{T}'$ such that $\mathbb{T} \longrightarrow \mathbb{T}'$ or*
2. *$\mathbb{T}$ is terminal.*

Finally, the type soundness property allows us to safely execute well-typed component programs. That is, during the execution of the programs the number of active instances of any component never exceeds the allowed number.

Theorem 1 (Soundness). *Let $\mathcal{R}$ be a requirement, Γ be a basis, E be an expression and suppose $\Gamma \vdash_{\mathcal{R}} E : X$ for some X. Let $\mathbb{T} = \mathsf{Lf}([], E)$. Then for every sequence of reductions $\mathbb{T} \longrightarrow^* \mathbb{T}'$ we have $[\mathbb{T}'] \subseteq \mathcal{R}$.*

4.2 Other Properties

The section lists some fundamental properties of our type system. These properties are needed to prove the lemmas and theorem in the previous section. Most of these properties are analogous to those in [3]. We start by giving some definitions. In the sequel we use X^* for any of X^i, X^o, X^j, X^p and X^l.

Following [1] we fix some terminology on bases or environments.

Definition 5 (Bases). *Let $\Gamma = x_1 \prec A_1, \ldots, x_n \prec A_n$ be a basis.*

- *Γ is called* legal *if $\Gamma \vdash A : X$ for some expression A and type X.*
- *A declaration $x \prec A$* is in *Γ, notation $x \prec A \in \Gamma$, if $x \equiv x_i$ and $A \equiv A_i$ for some i.*
- *Δ is* part *of Γ, notation $\Delta \subseteq \Gamma$, if $\Delta = x_{i_1} \prec A_{i_1}, \ldots, x_{i_k} \prec A_{i_k}$ with $1 \leq i_1 < \ldots < i_k \leq n$. Note that the order is preserved.*
- *Δ is an* initial segment *of Γ, if $\Delta = x_1 \prec A_1, \ldots, x_j \prec A_j$ for some $1 \leq j \leq n$.*

For any expression E, let $\mathsf{var}(E)$ denote the set of variables occurring in E:

$$\mathsf{var}(\mathtt{new}\, x) = \mathsf{var}(\mathtt{reu}\, x) = \{x\}, \quad \mathsf{var}(\{A\}) = \mathsf{var}(A),$$
$$\mathsf{var}(AB) = \mathsf{var}((A+B)) = \mathsf{var}((A \parallel B)) = \mathsf{var}(A) \cup \mathsf{var}(B)$$

The following lemma collects a number of simple properties of a typing judgment. It also shows some relations among multisets of A and any legal basis always has distinct declarations.

Lemma 3 (Legal typing). *If $\Gamma \vdash A : X$, then*

1. *elements of $\mathsf{var}(A)$, X^* are in $Dom(\Gamma)$,*
2. *$\Gamma \vdash \epsilon : \langle [], [], [], [], [] \rangle$,*
3. *every variable in $Dom(\Gamma)$ is declared only once in Γ,*
4. *$X^o \subseteq X^i \subseteq \mathcal{R}$ and $X^p \subseteq X^j \subseteq \mathcal{R}$,*
5. *$X^j \subseteq X^i$, $X^p \subseteq X^o$, and $X^l \subseteq X^o$.*

The following lemma is important in that it allows us to find a syntax-directed derivation of the type of an expression and hence it allows us to calculate the types of sub-expressions and is used in type inference. This lemma is sometimes called the *inversion lemma of the typing relation* [12].

Lemma 4 (Generation).

1. *If* $\Gamma \vdash \mathtt{new}\, x : X$, *then* $x \in X^p$ *and there exists bases* Δ, Δ' *and expression* A *such that* $\Gamma = \Delta, x \prec A, \Delta'$, *and* $\Delta \vdash A : \langle X^i - x, X^o - x, X^j - x, X^p - x, X^l - x \rangle$.
2. *If* $\Gamma \vdash \mathtt{reu}\, x : X$, *then* $x \in X^o$ *and there exists bases* Δ, Δ' *and expression* A *such that* $\Gamma = \Delta, x \prec A, \Delta'$, *and* $\Delta \vdash A : \langle X^i - x, X^o - x, X^j, X^p, X^l - x \rangle$.
3. *If* $\Gamma \vdash AB : Z$ *with* $A, B \neq \epsilon$, *then there exists* X, Y *such that* $\Gamma \vdash A : X$, $\Gamma \vdash B : Y$ *and* $Z = \langle X^i \cup (X^o \uplus Y^j) \cup Y^i!_{X^l}, (X^o \uplus Y^p) \cup Y^o!_{X^l}, X^j \cup (X^p \uplus Y^j), X^p \uplus Y^p, X^l \cup Y^l \rangle$.
4. *If* $\Gamma \vdash (A + B) : Z$, *then there exists* X, Y *such that* $\Gamma \vdash A : X$, $\Gamma \vdash B : Y$ *and* $Z = \langle X^i \cup Y^i, X^o \cup Y^o, X^j \cup Y^j, X^p \cup Y^p, X^l \cap Y^l \rangle$.
5. *If* $\Gamma \vdash (A \parallel B) : Z$, *then there exists* X, Y *such that* $\Gamma \vdash A : X$, $\Gamma \vdash B : Y$, *and* $Z = \langle X^i \uplus Y^i, X^o \uplus Y^o, X^j \uplus Y^j, X^p \uplus Y^p, X^l \cup Y^l \rangle$.
6. *If* $\Gamma \vdash \{A\} : \langle X^i, [], X^j, [], [] \rangle$, *then there exists multisets* X^o, X^p, *and* X^l *such that* $\Gamma \vdash A : X$.

The next lemma stresses the significance of the order of declarations in a legal basis in our type system. Besides, because of the weakening rule, there can be many legal bases under which a well-typed expression can be derived. Thus, its 'inversion' is stated in the lemma following.

Lemma 5 (Legal monotonicity).

1. *If* $\Gamma = \Delta, x \prec E, \Delta'$ *is legal, then* $\Delta \vdash E : X$ *for some* X.
2. *If* $\Gamma \vdash E : X$, $\Gamma \subseteq \Gamma'$ *and* Γ' *is legal, then* $\Gamma' \vdash E : X$.

Lemma 6 (Strengthening). *If* $\Gamma, x \prec A \vdash B : Y$ *and* $x \notin \mathsf{var}(B)$, *then* $\Gamma \vdash B : Y$ *and* $x \notin Y^i$.

Last, in our type system, when an expression has a type this type is unique. This property is stated in the following proposition.

Proposition 1 (Uniqueness of types). *If* $\Gamma \vdash A : X$ *and* $\Gamma \vdash A : Y$, *then* $X^i = Y^i$, $X^o = Y^o$, $X^j = Y^j$, $X^p = Y^p$, *and* $X^l = Y^l$.

5 Research Directions

In a slightly more liberal approach one leaves out the side condition from the typing rule Seq and takes the types as counting the maximum number of simultaneously active instances of each component. These maxima can then be compared to the available resources.

We are well aware of the level of abstraction of the component language and plan to incorporate more language features. These include recursion in component declarations, explicit deallocation primitive, and communication among threads. For example, suppose d, e are primitive components, then $a \prec (\{\mathtt{new}\, d\}\, \mathtt{reu}\, a + \mathtt{new}\, e)$ is bounded by $\{a, e, d\}$, despite that it has one infinite execution trace.

References

1. H. Barendregt. Lambda Calculi with Types. In: Abramsky, Gabbay, Maibaum (Eds.), *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press. 1992.
2. M. Bezem and H. Truong. A Type System for the Safe Instantiation of Components. In *Electronic Notes in Theoretical Computer Science* Vol. 97, July 2004.
3. M. Bezem and H. Truong. Counting Instances of Software Components, In *Proceedings of LRPP'04*, July 2004.
4. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208-2236. CRC Press, 1997.
5. K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262-275, San Antonio, TX, USA, January 1999.
6. K. Crary and S. Weirich. Resource Bound Certification. In *the Twenty-Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184-198, Boston, MA, USA, January 2000.
7. B. Dushnik and E. W. Miller. *Partially Ordered Sets*, American Journal of Mathematics, Vol. 63, 1941.
8. R. Englander. *Developing Java Beans.* 1st Edition, ISBN 1-56592-289-1, June 1997.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., ISBN 0201633612, 1994.
10. E. Meijer and C. Szyperski. Overcoming Independent Extensibility Challenges, *Communications of the ACM*, Vol. 45, No. 10, pp. 41–44, October 2002.
11. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
12. B. Pierce. *Types and Programming Languages.* MIT Press, ISBN 0262162091, February 2002.
13. J. C. Seco. Adding Type Safety to Component Programming. In *Proc. of The PhD Student's Workshop* in FMOODS'02, University of Twente, the Netherlands, March 2002.
14. F. Smith, D. Walker and G. Morrisett. Alias Types. In *European Symposium on Programming*, Berlin, Germany, March 2000.
15. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*, 2nd edition, Addison-Wesley, ISBN 0201745720, 2002.
16. Terese. *Term Rewriting Systems*, Cambridge Tracts in Theoretical Computer Science, Vol. 55, Cambridge University Press, 2003
17. T. L. Thai, Hoang Lam. *.NET Framework Essentials.* 3nd Edition, ISBN 0-596-00302-1, August 2003.
18. A. K. Wright and M. Felleisen, A Syntactic Approach to Type Soundness. In *Information and Computation*, Vol. 115, No. 1, pp. 38–94, 1994.
19. M. Zenger, Type-Safe Prototype-Based Component Evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.
20. M. Zenger, Programming Language Abstractions for Extensible Software Components, PhD Thesis, No. 2930, EPFL, Switzerland, March 2004.