Finding Resource Bounds in the Presence of Explicit Deallocation ⋆

Hoang Truong and Marc Bezem

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

Abstract. A software program requesting a resource that is not available usually raises an out-of-resource exception. Component software is software that has been assembled from standardized, reusable components which, in turn, may also composed from other components. Due to the independent development and reuse of components, component software has a high risk of causing out-of-resource exceptions. We present a small component language and develop a type system which can statically prevent this type of errors .
This work continues our previous works [3, 18] by including explicit deallocation. We prove that the type system is sound with respect to safe deallocation and that sharp resource bounds can be computed statically.

1 Introduction

Component software is built from various components, possibly developed by third-parties [15, 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

⋆ This research was supported by the Research Council of Norway.

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. [1, 4, 11]. 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, 12]. In component software, typing has been studied in relation to integrating components such as type-safe composition [21] 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, deallocation and composition. In the previous work [3, 18], the main features are instantiation and reuse, sequential composition, choice, parallel composition and scope and the deallocation of instances is controlled by scope mechanism. In this work, we consider sequencing and parallel composition, choice and scope, add an explicit deallocation primitive, which allows us to imperatively remove an instance in the same scope. For the sake of simplicity, we do not consider the reuse primitive. However, we believe that the combination of all the features is feasible.

Though abstract, the strength of the primitives for composition is considerable. Choice allows us to model both conditionals and non-determinism. 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. Parallel composition allows several threads of execution. Sequential composition is associative.

We use a small-step operational semantics and as a result, we can prove the soundness of our type system using the standard technique of Wright and Felleisen [20].

The type inference algorithm for this system is almost the same as in [3]. We still have a polynomial time type inference algorithm. Polynomial type inference is of crucial importance since examining all possible executions of the operational semantics is (at least) exponential.

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. The soundness and several other properties of the system are presented in Section 4. Finally, we outline some future directions.

2 A Component Language

2.1 Syntax

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).

Table 1. Syntax

Prog	$::= \overline{\mathit{Decls}}; E$	Program
Decls	$::= \overline{x \prec E}$	Declarations
E	$::=$	Expression
	$\mid \ \epsilon$	Empty
	$\mid \ \mathtt{new}\,x$	Instantiation
	$\mid \ \mathtt{del}\,x$	Deallocation
	$\mid \ (E + E)$	Choice
	$\mid \ (E \parallel E)$	Parallel
	$\mid \ \{E\}$	Scope
	$\mid \ E\ E$	Sequencing

Let $a, \ldots, z$ range over component names and $A, \ldots, E$ range over expressions. We collect all component names in a set $\mathbb{C}$.

The main ingredients in the component language are component declaration and expression. We have two primitives ($\mathtt{new}$ and $\mathtt{del}$) for creating and deleting 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 uses no subcomponents then E is ϵ and we call x a *primitive component*. A *component program* consists 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. In this example, d and e are primitive components. Component a is the parallel composition of $\{\mathtt{new}\,d\}\,\mathtt{new}\,e$ and $\mathtt{new}\,d$ followed by a deallocation of d. Component b has a choice expression before deleting an instance of e.

$$
\begin{array}{l}
d \prec \epsilon \qquad e \prec \epsilon \\
a \prec (\{\mathtt{new}\,d\}\,\mathtt{new}\,e \parallel \mathtt{new}\,d)\,\mathtt{del}\,d \\
b \prec (\mathtt{new}\,a + \mathtt{new}\,e\,\mathtt{new}\,d)\,\mathtt{del}\,e; \\
\mathtt{new}\,b
\end{array}
$$

2.2 Operational Semantics

Informally, expression E can be viewed as a sequence of commands of the form $\mathtt{new}\,x$, $\mathtt{del}\,x, (A+B), (A \parallel B), \{A\}$ in imperative programming languages and the execution is sequential from left to right. In the operational semantics E is paired with a local store, modelled by a multiset. The first three commands act locally. When executing a command of the form $\mathtt{new}\,x$, a new instance of x is created in the local store and the execution continues with the 'body' A, if the declaration of x is $x \prec A$ and $A \neq \epsilon$. If $A = \epsilon$ the execution proceeds to the next command after $\mathtt{new}\,x$. Executing $\mathtt{del}\,x$ simply removes a x in the local store then continues with the next command. Executing $(A+B)$ means to choose A or B to execute with the same store.

When the current command is of the form $\{E\}$ the execution of the commands after $\{E\}$, say A, is suspended, and the execution is transferred to E with a new empty local store. When the execution of the new pair $([], E)$ terminates in pair (M, ϵ), the instances in M are *discarded* and the execution resumes to the expression A and its local store. We will use stacks for this scope mechanism.

Executing $(E_1 \parallel E_2)$ suspends the execution of the commands after it and creates two new empty stores for each E_1 and E_2 and these two new pairs $([], E_1)$ and $([], E_2)$, called child threads, are executed concurrently. When a thread terminates in the pair (M, ϵ) the instances in M are *returned* to the store at the top of its parent thread. When all the child threads terminated, the execution resumes to the parent thread. The formal model is detailed as follows.

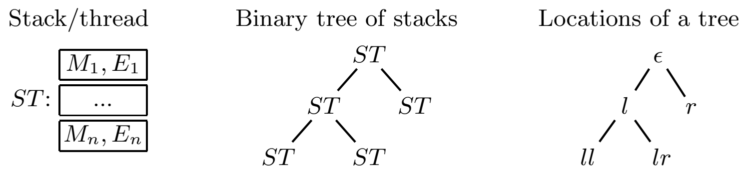

Fig. 1. Illustration of a tree of stacks

The operational semantics is defined by a rewriting system [16] of *configurations*. A configuration is a binary tree $\mathbb{T}$ of threads. A thread is a stack ST of pairs of a local store and an 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. A configuration is *terminal* if it has only one (root) thread of the form (M, ϵ). Figure 1 illustrates stacks and configurations. The syntax of stacks and configurations is as follows.

$$
\begin{array}{lll}
ST \ ::= (M_1, E_1) \circ \ldots \circ (M_n, E_n) & & \text{Stack} \\
\mathbb{T}, \mathbb{S} ::= & & \text{Configurations} \\
\qquad \mathsf{Lf}(ST) & & \quad \text{Leaf} \\
\quad \mid \ \mathsf{Nd}(ST, \mathbb{T}) & & \quad \text{Node with one branch} \\
\quad \mid \ \mathsf{Nd}(ST, \mathbb{T}, \mathbb{T}) & & \quad \text{Node with two branches}
\end{array}
$$

The above stack ST has n elements where (M_1, E_1) is the bottom, (M_n, E_n) is the top of the stack, and '$\circ$' is the stack separator. A node in our binary trees may have no child nodes $\mathsf{Lf}(ST)$, or one branch $\mathsf{Nd}(ST, \mathbb{T})$, or two branches $\mathsf{Nd}(ST, \mathbb{T}, \mathbb{T})$.

We assign to each node in our tree a *location*, illustrated in Figure 1. 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 $\mathbb{T}$.

By $\mathbb{T}[\,]_\alpha$ we denote a tree with a hole at the leaf location α. Filling this hole with another tree $\mathbb{S}$ is denoted by $\mathbb{T}[\![\mathbb{S}]\!]_\alpha$. One step reduction is defined first by choosing an arbitrary active thread. Then depending on the pattern of the chosen thread and the state of the configuration, the appropriate rewrite rule can be applied. The rewriting rules for these patterns or subconfigurations, notation $\mathbb{S} \rightsquigarrow \mathbb{S}'$, are called the basic reduction relation. The configuration $\mathbb{T}[\![\mathbb{S}]\!]_\alpha$ can take a step to $\mathbb{T}[\![\mathbb{S}']\!]_\alpha$, notation $\mathbb{T}[\![\mathbb{S}]\!]_\alpha \longrightarrow \mathbb{T}[\![\mathbb{S}']\!]_\alpha$, if $\mathbb{S} \rightsquigarrow \mathbb{S}'$. As usual, $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

Table 2. Basic reduction rules

(osNew) $\quad x \prec A \in \mathit{Decls}$
$\mathsf{Lf}(ST \circ (M, \mathtt{new}\, xE)) \rightsquigarrow \mathsf{Lf}(ST \circ (M + x, AE))$

(osDel) $\quad x \in M$
$\mathsf{Lf}(ST \circ (M, \mathtt{del}\, xE)) \rightsquigarrow \mathsf{Lf}(ST \circ (M - x, E))$

(osChoice) $\quad i \in \{1, 2\}$
$\mathsf{Lf}(ST \circ (M, (A_1 + A_2)E)) \rightsquigarrow \mathsf{Lf}(ST \circ (M, A_i E))$

(osPush)
$\mathsf{Lf}(ST \circ (M, \{A\}E)) \rightsquigarrow \mathsf{Lf}(ST \circ (M, E) \circ ([\,], A))$

(osPop)
$\mathsf{Lf}(ST \circ (M, E) \circ (M', \epsilon)) \rightsquigarrow \mathsf{Lf}(ST \circ (M, E))$

(osParIntr)
$\mathsf{Lf}(ST \circ (M, (A \parallel B)E)) \rightsquigarrow \mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}(([\,], A)), \mathsf{Lf}(([\,], B)))$

(osParElimL)
$\mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}((M', \epsilon)), \mathbb{S}) \rightsquigarrow \mathsf{Nd}(ST \circ (M + M', E), \mathbb{S})$

(osParElimR)
$\mathsf{Nd}(ST \circ (M, E), \mathbb{S}, \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Nd}(ST \circ (M + M', E), \mathbb{S})$

(osParElim)
$\mathsf{Nd}(ST \circ (M, E), \mathsf{Lf}((M', \epsilon))) \rightsquigarrow \mathsf{Lf}(ST \circ (M + M', E))$

The basic reduction relation is described in Table 2. Each basic reduction rule has two lines. The first line contains a rule name followed by a list of conditions. The second line has the form $\mathbb{S} \rightsquigarrow \mathbb{S}'$, which states that if a configuration $\mathbb{T}$ has a subconfiguration of the form $\mathbb{S}$ and all the conditions in the first line hold, then we can replace the subconfiguration $\mathbb{S}$ of $\mathbb{T}$ by subconfiguration $\mathbb{S}'$ and get the new state $\mathbb{T}[\![\mathbb{S}']\!]$.

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

By the rules osNew, osDel, and osChoice we only rewrite the pair at the top of a leaf stack. The rule osNew first creates a new instance of component x in the local store. Then if x is a primitive component it continues to execute the remaining expression E; otherwise, it continues to execute A before executing the remaining expression E. The rule osDel deallocates an instance of x in the local store if there exists one. If there exists no instance of x in the local store, the execution is stuck. Note that here we have abstracted away the specific instance that will be deleted. The rule osChoice selects a branch to execute.

The next two rules change the shape of a leaf stack. Rule osPush pushes an element on the top of the leaf stack. The rule osPop pops the stack when the stack has at least two elements. That means no stack in any configuration is empty. The last four rules change the tree structure of the configuration. By the rule osParIntr, a leaf is replaced by a branch of a node and two leaves. In contrast, by the rules osParElimR, osParElimL, osParElim, a leaf is removed from the tree and the instances left at the leaf are returned to the store at the top of the parent thread. When appropriate, the parent node may be promoted to be an active thread (osParElim).

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 $\mathsf{Lf}([\,], \mathtt{new}\, b)$. After the first step, there are two possibilities by the rule osChoice.

$$
\begin{array}{rll}
(\mathsf{Start}) & ([\,], \mathtt{new}\, b) & \\
(\mathsf{osNew}) \longrightarrow & ([b], (\,\mathtt{new}\, a + \mathtt{new}\, e\, \mathtt{new}\, d)\, \mathtt{del}\, e) & \\
(\mathsf{osChoice}) \longrightarrow & ([b], \mathtt{new}\, a\, \mathtt{del}\, e) & (\text{or } ([b], \mathtt{new}\, e\, \mathtt{new}\, d\, \mathtt{del}\, e))
\end{array}
$$

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

$$
\begin{array}{rl}
 & ([b],\, \mathtt{new}\, a\, \mathtt{del}\, e) \\
(\mathsf{osNew}) \longrightarrow & ([b,a], (\{\, \mathtt{new}\, d\}\, \mathtt{new}\, e \parallel \mathtt{new}\, d)\, \mathtt{del}\, d\, \mathtt{del}\, e) \\
(\mathsf{osParIntr}) \longrightarrow & ([b,a],\, \mathtt{del}\, d\, \mathtt{del}\, e)\ \langle \begin{array}{l} ([\,], \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e) \\ \boxed{([\,],\, \mathtt{new}\, d)} \end{array} \\
(\mathsf{osNew}) \longrightarrow & ([b,a],\, \mathtt{del}\, d\, \mathtt{del}\, e)\ \langle \begin{array}{l} \boxed{([\,], \{\, \mathtt{new}\, d\}\, \mathtt{new}\, e)} \\ ([d], \epsilon) \end{array} \\
(\mathsf{osPush}) \longrightarrow & ([b,a],\, \mathtt{del}\, d\, \mathtt{del}\, e)\ \langle \begin{array}{l} \boxed{([\,],\, \mathtt{new}\, e) \circ ([\,],\, \mathtt{new}\, d)} \\ ([d], \epsilon) \end{array} \\
(\mathsf{osNew}) \longrightarrow & ([b,a],\, \mathtt{del}\, d\, \mathtt{del}\, e)\ \langle \begin{array}{l} ([\,],\, \mathtt{new}\, e) \circ ([d], \epsilon) \\ \boxed{([d], \epsilon)} \end{array} \\
(\mathsf{osParElimL}) \longrightarrow & ([b,a,d],\, \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoonup ([\,],\, \mathtt{new}\, e) \circ ([d], \epsilon) \\
(\mathsf{osPop}) \longrightarrow & ([b,a,d],\, \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoonup ([\,],\, \mathtt{new}\, e) \\
(\mathsf{osNew}) \longrightarrow & ([b,a,d],\, \mathtt{del}\, d\, \mathtt{del}\, e) \leftharpoonup ([e], \epsilon) \\
(\mathsf{osParElim}) \longrightarrow & ([b,a,d,e],\, \mathtt{del}\, d\, \mathtt{del}\, e) \\
(\mathsf{osDel}) \longrightarrow & ([b,a,e],\, \mathtt{del}\, e) \\
(\mathsf{osDel}) \longrightarrow & ([b,a], \epsilon) \quad \text{(terminal)}
\end{array}
$$

As mentioned in Section 1, here we have abstracted resources by the number of instances. When we want to account for specific resources, we can annotate the source program with the resource consumption of relevant component. Then the maximum resources the component program will use can be computed from our inferred types and the annotation. Another way to find how much resources a component program will probably use is declaring the specific resources as primitive components. Other components will then instantiate these resources in their declarations if they use the resources. Then our type system in the next section can tell us the maximum resources the program needs.

3 Type System

We have two main goals in designing the type system. The first one comes from the rule osDel of the dynamic semantics, where the program is stuck if the next operation is a deallocation of a component and there exists no instance of that component in the local store. In other words, the type system must guarantee the safety of the deallocation operation. We solve the problem by keeping a store in the typing environment, a technique inspired by linear type systems [19, 12]. For the second goal, we want to find the upper bounds of resources that a program may request. Since we have abstracted the specific resources in the instances, the upper bounds become the maximum numbers of simultaneously active instances. In the rest of this section, we first define types and explain them informally; Then then we present the formal typing rules and some typing examples.

Before defining types, we extend the notion of multiset in Section 2 to the notion of *signed multiset*. Recall, a multiset over a set of elements S can be viewed as a map from S to the set of natural numbers $\mathbb{N}$. Similarly, a signed multiset M, also denoted by [...], over a set S is a map from S to the set of integers $\mathbb{Z}$. The analogous operations of multisets are overloaded for signed multisets. $M(x)$ is the 'multiplicity' of x (can be negative); $M(x) = 0$ when x is not an element of M, notation $x \notin M$. Let M, N be signed multisets, then we define additive union: $(M + N)(x) = M(x) + N(x)$; substraction: $(M - N)(x) = M(x) - N(x)$; union: $(M \cup N)(x) = max(M(x), N(x))$; intersection: $(M \cap N)(x) = min(M(x), N(x))$; inclusion: $M \subseteq N$ if $M(x) \leq N(x)$ for all $x \in M$. For example, $[x, -y, -y]$ is a signed multiset where the multiplicity of x is 1 and the multiplicity of y is -2.

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

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

where X^i is a multiset and X^o, X^l are signed multisets.

Intuitively, the meaning of each part of a type triple is as follows. Suppose X is the type of an expression E. Then X^i is the upper bound of the number of simultaneously active instances for all components during the execution of E. Multisets are the right data structure to store this information. Next, X^o is the maximum number of instances that 'survive' at the end of the execution when executing E alone, as in [3, 18]. In this paper, we have the deallocation primitive and its behaviour is opposite to instantiation so we use signed multisets. Moreover we want compositionality of typing, so in composition X^o is the maximal net effect (with respect to the change in the number of instances) to the runtime environment *before* and *after* the execution of E. Similarly, X^i in composition is the effect on the maximum during the execution. The pair $\langle X^i, X^o \rangle$ is enough to calculate the upper bound.

Besides, we want the safety of the deallocation primitives in composition. When sequencing E and $\mathtt{del}\, x$ the safety of $\mathtt{del}\, x$ depends on the minimum outcome of E. Therefore we need X^l, which is the minimum number of surviving instances after the execution of E. Like X^o, in composition, X^l is the minimal net effect to the runtime environment before and after the execution of E. The discrepancy between X^o and X^l is caused by choice composition $+$. More explanation is given shortly in the exposition of typing rules below.

A *basis* is a list of declarations: $x_1 \prec E_1, \ldots, x_n \prec E_n$. Empty basis is denoted by $\varnothing$. Let Γ, Δ range over bases. The domain of basis $\Gamma = x_1 \prec E_1, \ldots, x_n \prec E_n$, notation $\mathsf{dom}(\Gamma)$, is the set $\{x_1, \ldots, x_n\}$. A *store* is a multiset (no negative multiplicities) of component names. Let σ range over stores. An *environment* is a pair of a store and a basis. A typing judgment is a tuple of the form

$$\sigma, \Gamma \vdash E : X$$

and it asserts that expression E has type X in the environment σ, Γ.

Definition 2 (Valid typing judgments). *Valid typing judgments* $\sigma, \Gamma \vdash A\!:\!X$ *are derived by applying the typing rules in Table 3 in the usual inductive way.*

Table 3. Typing rules

(Axiom)
$$\frac{}{[\,], \varnothing \vdash \epsilon\!:\!\langle [\,], [\,], [\,] \rangle}$$

(WeakenB)
$$\frac{\sigma_1, \Gamma \vdash A\!:\!X \quad \sigma_2, \Gamma \vdash B\!:\!Y \quad x \notin \mathsf{dom}(\Gamma)}{\sigma_1, \Gamma, x \prec B \vdash A\!:\!X}$$

(WeakenS)
$$\frac{\sigma, \Gamma \vdash A\!:\!X \quad \sigma \subseteq \sigma_1}{\sigma_1, \Gamma \vdash A\!:\!X}$$

(New)
$$\frac{\sigma, \Gamma \vdash A\!:\!X \quad x \notin \mathsf{dom}(\Gamma)}{\sigma, \Gamma, x \prec A \vdash \mathtt{new}\, x\!:\!\langle X^i + x, X^o + x, X^l + x \rangle}$$

(Del)
$$\frac{\sigma, \Gamma \vdash A\!:\!X \quad x \in \mathsf{dom}(\Gamma)}{[x], \Gamma \vdash \mathtt{del}\, x\!:\!\langle [\,], [-x], [-x] \rangle}$$

(Seq)
$$\frac{\sigma_1, \Gamma \vdash A\!:\!X \quad \sigma_2, \Gamma \vdash B\!:\!Y \quad A, B \neq \epsilon}{\sigma_1 \cup (\sigma_2 - X^l), \Gamma \vdash AB\!:\!\langle X^i \cup (X^o + Y^i), X^o + Y^o, X^l + Y^l \rangle}$$

(Choice)
$$\frac{\sigma_1, \Gamma \vdash A\!:\!X \quad \sigma_2, \Gamma \vdash B\!:\!Y}{\sigma_1 \cup \sigma_2, \Gamma \vdash (A + B)\!:\!\langle X^i \cup Y^i, X^o \cup Y^o, X^l \cap Y^l \rangle}$$

(Parallel)
$$\frac{[\,], \Gamma \vdash A\!:\!X \quad [\,], \Gamma \vdash B\!:\!Y}{[\,], \Gamma \vdash (A \parallel B)\!:\!\langle X^i + Y^i, X^o + Y^o, X^l + Y^l \rangle}$$

(Scope)
$$\frac{[\,], \Gamma \vdash A\!:\!X}{[\,], \Gamma \vdash \{A\}\!:\!\langle X^i, [\,], [\,] \rangle}$$

These typing rules deserve some further explanation. The most critical rule is Seq because sequencing two expressions can lead to increase in instances of the composed expression. First, the semantics of the store in the typing judgment requires that the store always has enough elements for deallocation commands in the expression. So we need to increase the store when the minimum outcome of A and its store, $X^l + \sigma_1$, is not enough for σ_2. Consider a component x. The premise of the rule Seq tells us that we need a store σ_1 for executing A. Thereafter, we have at least $X^l(x)$ instances of x, where $X^l(x) \in \mathbb{Z}$. Again by the premise of the rule Seq we need $\sigma_2(x)$ instances for safely executing B. Therefore we must start the execution of AB with at least $(\sigma_2 - X^l)(x)$ in the store (more than $\sigma_2(x)$ if $X^l(x) < 0$). Second, in the type expression of AB, the maximum is the maximum of A or of the outcome of A together with the maximum of B. So the first part of the type of AB is $X^i \cup (X^o + Y^i)$. The remaining parts, $X^o + Y^o$ and $X^l + Y^l$, are easy referring to the semantics of these parts of the types.

Other typing rules are straightforward. The rule Axiom is used for startup. The rules WeakenB allows us to extend the type environments so that the rules Seq, Choice, Parallel may be applied. The rule WeakenS plays a technical role in some proofs and is a natural rule anyway: enlarging the store should preserve typing. The rule New accumulates a new instance in type expression while the rule Del reduces by one instance. The first signed multiset in the type of $\mathtt{del}\, x$ is empty since it has no effect to the maximum in composition, but the last two multisets are both $[-x]$ since $\mathtt{del}\,x$ reduces the local stores by one x. The judgment $\sigma, \Gamma \vdash A\!:\!X$ in the premise of this rule only guarantees that the basis Γ is legal. The rules Parallel and Scope require an empty store in the environment because the semantics of deallocation applies to local store only.

Now we can define the notion of *well-typed program* with respect to our type system. Basically, a program is well-typed if we can derive a type for the main expression of the program from an empty store and a list of the program declarations. As mentioned in the Introduction Section 1, we have an polynomial algorithm (cf. [3]) which can automatically decide whether a program is well-typed or not, and if so, infer a type.

Definition 3 (Well-typed programs). *Program Prog* $=$ *Decls*$;E$ *is well-typed if there exists a reordering* Γ *of declarations in Decls such that* $[\,], \Gamma \vdash E\!:\!X$.

Using the example in Section 2.1 we derive type for $\mathtt{new}\,b$. Note that we omitted some side conditions as they can be checked easily and we shortened the rule names to the first two characters (we do not use the rule WeakenS so WeakenB is abbreviated to We). The signed multisets are simplified as well. The elements of a signed multiset are listed in a string with the multiplicities as superscripts, multiplicity 1 is not shown as supperscript and elements with multiplicity 0 are not shown. The rule Axiom is also simplified.

$$\mathsf{We}\,\dfrac{\mathsf{Sc}\,\dfrac{\mathsf{Ne}\,\dfrac{[\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}{[\,],d\mathrel{-\!\!\prec}\epsilon \vdash \mathtt{new}\,d\!:\!\langle d,d,d\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon \vdash \{\mathtt{new}\,d\}\!:\!\langle d,[\,],[\,]\rangle} \quad \mathsf{We}\,\dfrac{[\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle \quad [\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}{[\,],d\mathrel{-\!\!\prec}\epsilon \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash \{\mathtt{new}\,d\}\!:\!\langle d,[\,],[\,]\rangle} \tag{1}$$

$$\mathsf{Se}\,\dfrac{(1) \quad \mathsf{Ne}\,\dfrac{\mathsf{We}\,\dfrac{[\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle \quad [\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}{[\,],d\mathrel{-\!\!\prec}\epsilon \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash \mathtt{new}\,e\!:\!\langle e,e,e\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash \{\mathtt{new}\,d\}\,\mathtt{new}\,e\!:\!\langle de,e,e\rangle} \tag{2}$$

$$\mathsf{Pa}\,\dfrac{(2) \quad \mathsf{We}\,\dfrac{\mathsf{Ne}\,\dfrac{[\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}{[\,],d\mathrel{-\!\!\prec}\epsilon \vdash \mathtt{new}\,d\!:\!\langle d,d,d\rangle} \quad [\,],\varnothing \vdash \epsilon\!:\!\langle[\,],[\,],[\,]\rangle}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash \mathtt{new}\,d\!:\!\langle d,d,d\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash (\{\mathtt{new}\,d\}\,\mathtt{new}\,e \parallel \mathtt{new}\,d)\!:\!\langle d^2e,de,de\rangle} \tag{3}$$

$$\mathsf{Ne}\,\dfrac{\mathsf{Se}\,\dfrac{(3) \quad \mathsf{De}\,\dfrac{(3) \quad d \in \mathsf{dom}(d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon)}{[d],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash \mathtt{del}\,d\!:\!\langle[\,],d^{-1},d^{-1}\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon \vdash (\{\mathtt{new}\,d\}\,\mathtt{new}\,e \parallel \mathtt{new}\,d)\,\mathtt{del}\,d\!:\!\langle d^2e,e,e\rangle}}{[\,],d\mathrel{-\!\!\prec}\epsilon,e\mathrel{-\!\!\prec}\epsilon,a\mathrel{-\!\!\prec}(\{\mathtt{new}\,d\}\,\mathtt{new}\,e \parallel \mathtt{new}\,d)\,\mathtt{del}\,d \vdash \mathtt{new}\,a\!:\!\langle ad^2e,ae,ae\rangle} \tag{4}$$

Similarly, we can derive $\Gamma \vdash \mathtt{new}\,b\!:\!\langle abd^2e, abd, b\rangle$ where $\Gamma = d\mathrel{-\!\!\prec}\epsilon, e\mathrel{-\!\!\prec}\epsilon, a\mathrel{-\!\!\prec}(\{\mathtt{new}\,d\}\,\mathtt{new}\,e \parallel \mathtt{new}\,d)\,\mathtt{del}\,d, b\mathrel{-\!\!\prec}(\mathtt{new}\,a + \mathtt{new}\,e\,\mathtt{new}\,d)\,\mathtt{del}\,e$.

By the example we illustrate how we can infer the specific resources. If component a and d each creates a database connection, then from the type of $\mathtt{new}\,b$, we know that the program, in particular the main expression $\mathtt{new}\,b$, may need three database connections (one by a and two by d). From another point of view, we regard d as a database connection component, then we know that the program needs maximum two database connections.

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 model, type errors occur when the program tries to delete an instance which is not in the local store or when the program tries to instantiate a component x but there is no declaration of x. We will prove that these two situations will not happen. Besides, we will prove an additional important property which guarantees that a well-typed program will not create more instances than a certain maximum stated in its type.

Our proof of the type soundness is based on the approach of Wright and Felleisen [20]. 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*. We start with some auxiliary definitions.

First, 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 $\mathbb{T}$ and $\mathbb{T}(\alpha)$ the stack at location α in $\mathbb{T}$.

Second, 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 $\alpha_1 < \alpha_2$.

Next, we formalize the notion of *subtree*. Given a tree $\mathbb{T}$. The set of positions $\mathcal{L} = \{\alpha_i.k_i \in \mathbb{T} \mid 1 \leq i \leq m\}$ is *valid* if $\alpha_i.k_i \not\leq \alpha_j.k_j$ for all $i \neq j$. The tree $\mathbb{T}'$ obtained from $\mathbb{T}$ by removing all elements at positions $\alpha.k \geq \alpha_i.k_i$ for all $1 \leq i \leq m$ is a subtree of $\mathbb{T}$, notation $\mathbb{T}' \sqsubseteq_{\mathcal{L}} \mathbb{T}$ or $\mathbb{T}' = \mathbb{T}|_{\mathcal{L}}$. Consequently, $\mathbb{T}'$ has the same root as $\mathbb{T}$. When $\mathcal{L}$ is empty, we get $\mathbb{T}' = \mathbb{T}$.

We denote by $\mathsf{hi}(ST)$ the height of the stack ST. By $\mathbb{T}(\alpha.k) = (M, E)$ we denote that the element at position $\alpha.k$ is the pair (M, E). We denote by $[\mathbb{T}(\alpha.k)]$ the store M at position $\alpha.k$, by $[\mathbb{T}(\alpha)]$ the additive union of all stores in the stack at location α, and by $[\mathbb{T}]$ the multiset of all active instances in the tree $\mathbb{T}$, i.e. $[\mathbb{T}] = \biguplus_{\alpha \in \mathbb{T}} [\mathbb{T}(\alpha)]$.

Now we calculate the multiset of instances that will be returned to a position $\alpha.k$. Due to the non-determinism of $\mathsf{osChoice}$, we can only calculate the upper bound and the lower bound of the collection. The minimal number of instances returned to a position $\alpha.k$, denoted by function $\mathsf{retl}_{\mathbb{T}}(\alpha.k)$, is zero if k is not the top of the stack at location α, or α is a leaf. Otherwise, it contains those in the multisets at the bottom of its child nodes and the minimal number of instances which survive the expressions there. Since the bottom of a child node of $\alpha.k$ may receive instances from its child nodes ($\mathsf{osParElimL}, \mathsf{osParElimR}, \mathsf{osParElim}$) and so on, we need to call the function recursively.

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

where $\mathbb{T}(\beta.1) = (M, E)$ and $M + \mathsf{retl}_{\mathbb{T}}(\beta.1), \Gamma \vdash E : X$. Note that this recursive definition is well-defined since first it is well-defined for all the positions at all leaves. Then it is well-defined for the top position of the parents of all leaves. And so on until the root.

The maximal number of instances that will be returned to a position $\alpha.k$, denoted by function $\mathsf{reto}_{\mathbb{T}}(\alpha.k)$, is calculated analogously.

$$\mathsf{reto}_{\mathbb{T}}(\alpha.k) = \begin{cases} [\,], & \text{if } k < \mathsf{hi}(\mathbb{T}(\alpha)) \text{ or } \alpha \in \mathsf{leaves}(\mathbb{T}) \\ \biguplus_{\beta \in \{\alpha l, \alpha r\}} (M + X^o + \mathsf{reto}_{\mathbb{T}}(\beta.1)), & \text{otherwise} \end{cases}$$

where $\mathbb{T}(\beta.1) = (M, E)$ and $M + \mathsf{retl}_{\mathbb{T}}(\beta.1), \Gamma \vdash E : X$.

By Lemma 5 below, these two functions always return multisets even though signed multisets X^l, X^o appear in their definitions.

Now we can define the notion of well-typed configuration. It guarantees that the local store always has enough elements for typing its executing expression. Hence deallocation operations are always safe to execute.

Definition 4 (Well-typed configuration). *Configuration $\mathbb{T}$ is well-typed with respect to a basis Γ, notation $\Gamma \models \mathbb{T}$, if for all pair (M, E) at position $\alpha.k \in \mathbb{T}$ there exists X such that*

$$M + \mathsf{retl}_{\mathbb{T}}(\alpha.k), \Gamma \vdash E : X$$

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 $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then $\mathbb{T}'$ is well-typed.*

Lemma 2 (Progress). *If $\Gamma \models \mathbb{T}$, then either $\mathbb{T}$ is terminal or there exists a configuration $\mathbb{T}'$ such that $\mathbb{T} \longrightarrow \mathbb{T}'$.*

Next, we show some additional invariants which allow us to infer the resource bounds of a well-typed program. Then we state the soundness theorem which contains both goals mentioned at the beginning of the section.

Consider the pair (M, E) at position $\alpha.k$ in a well-typed configuration $\mathbb{T}$. By Definition 4 we have $M + \mathsf{retl}_{\mathbb{T}}(\alpha.k), \Gamma \vdash E : X$ for some X. The maximum number of instances involved in the execution of the pair (M, E) is computed by:

$$\mathsf{io}_{\mathbb{T}}(\alpha.k) = M + \mathsf{reto}_{\mathbb{T}}(\alpha.k) + X^i$$

Lemma 3 (Invariants of retl, reto, and io). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all positions $\alpha.k$ in both configurations $\mathbb{T}$ and $\mathbb{T}'$ we have:*

1. $\mathsf{retl}_{\mathbb{T}}(\alpha.k) \subseteq \mathsf{retl}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
2. $\mathsf{reto}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{reto}_{\mathbb{T}'}(\alpha.k)$ *if* $\mathbb{T}(\alpha.k) = \mathbb{T}'(\alpha.k)$,
3. $\mathsf{io}_{\mathbb{T}}(\alpha.k) \supseteq \mathsf{io}_{\mathbb{T}'}(\alpha.k)$.

Note that the inclusions are related to choice: less options means smaller maxima and larger minima.

The maximum number of instances of a subtree $\mathbb{T}|_{\mathcal{L}}$ includes the maximum of its leaves and all the active instances in all the stores inside the subtree.

$$\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) = \biguplus_{\alpha.k<\mathcal{L}'} [\mathbb{T}(\alpha.k)] + \biguplus_{\alpha.k\in\mathcal{L}'} \mathsf{io}_{\mathbb{T}}(\alpha.k)$$

where $\mathcal{L}'$ is the set of all positions at the top of leaves of subtree $\mathbb{T}|_{\mathcal{L}}$, i.e. $\mathcal{L}' = \{\alpha.\mathsf{hi}(\mathbb{T}|_{\mathcal{L}}(\alpha)) \mid \alpha \in \mathsf{leaves}(\mathbb{T}|_{\mathcal{L}})\}$.

By the monotonicity of the function io, the function *maxins* also has this property.

Lemma 4 (Invariant of maxins). *If $\Gamma \models \mathbb{T}$ and $\mathbb{T} \longrightarrow \mathbb{T}'$, then for all valid set of positions $\mathcal{L}'$ of $\mathbb{T}'$ there exists a valid set of positions $\mathcal{L}$ of $\mathbb{T}$ such that*

$$\mathsf{maxins}(\mathbb{T}|_{\mathcal{L}}) \supseteq \mathsf{maxins}(\mathbb{T}'|_{\mathcal{L}'})$$

Now we can state the soundness property together with the upper bounds of instances that a well-typed program always respects.

Theorem 1 (Soundness). *If program $Prog = Decls; E$ is well-typed, then there exists a multiset M such that for every sequence of reductions $\mathsf{Lf}([\,], E) \longrightarrow^* \mathbb{T}$ we have $\mathbb{T}$ is not stuck and $[\mathbb{T}] \subseteq M$.*

4.2 Typing Properties

This section lists some properties of the type system. They are needed to prove the lemmas and theorem in the previous section. We start with some definitions.

Let $\Gamma = x_1 \prec A_1, \ldots, x_n \prec A_n$ be a basis. Γ is called *legal* if $\sigma, \Gamma \vdash A : X$ for some store σ, 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 an *initial segment* of Γ, if $\Delta = x_1 \prec A_1, \ldots, x_j \prec A_j$ for some $1 \leq j \leq n$.

We use X^* for any of X^i, X^o and X^l. Recall X^* are maps, we denote by $\mathsf{dom}(X^*) = \{x \mid X^*(x) \neq 0\}$ the domain of X^*. For multiset M we denote $\mathsf{dom}(M) = \{x \mid M(x) \neq 0\}$. Let $\mathsf{var}(E)$ denote the set of variables occurring in an expression:

$$\begin{aligned} &\mathsf{var}(\,\mathtt{new}\,x) = \mathsf{var}(\,\mathtt{del}\,x) = \{x\}, \quad \mathsf{var}(\{A\}) = \mathsf{var}(A), \\ &\mathsf{var}(AB) = \mathsf{var}((A+B)) = \mathsf{var}((A \parallel B)) = \mathsf{var}(A) \cup \mathsf{var}(B) \end{aligned}$$

The following lemma collects a number of simple properties of a valid typing judgment.

Lemma 5 (Legal typing). *If $\sigma, \Gamma \vdash A : X$, then*

1. $\mathsf{var}(A) \subseteq \mathsf{dom}(\Gamma)$, $\mathsf{dom}(X^*) \subseteq \mathsf{dom}(\Gamma)$,
2. *every variable in* $\mathsf{dom}(\Gamma)$ *is declared only once in* Γ,
3. $X^i \supseteq X^o \supseteq X^l$, $X^i \supseteq []$,
4. $\sigma + X^* \supseteq [\,]$.

The following lemmas show the associativity of the sequential composition and the significance of the order of declarations in a legal basis.

Lemma 6 (Associativity). *If $\sigma_i, \Gamma \vdash A_i : X_i$, for $i \in \{1, 2, 3\}$, then the typing judgments for $(A_1 A_2) A_3$ and $A_1(A_2 A_3)$ are the same.*

The following lemma is important in that it allows us to find a syntax-directed derivation of the type of an expression. This lemma is sometimes called the *inversion lemma of the typing relation* [11].

Lemma 7 (Generation).

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

5 Conclusions and Research Directions

This work follows a more liberal approach compared to our previous works [3, 18] where the resource bounds, i.e. the maximum number of instances for each component, are known in advance and the type system checks these bounds in typing rules. The dynamic semantics of the deallocation primitive here applies to local stores only. Even though this style is rather common in practice, we plan to extend the semantics of deallocation so that it can operate beyond scopes and even threads. 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, communication among threads and location of resources.

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, 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. B. Pierce. *Types and Programming Languages*. MIT Press, ISBN 0-262-16209-1, February 2002.
12. B. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, ISBN 0-262-16228-8, January 2005.
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. Thai, H. Lam. *.NET Framework Essentials*. 3nd Edition, ISBN 0-596-00302-1, August 2003.
18. H. Truong. Guaranteeing Resource Bounds for Component Software. Martin Steffen, Gianluigi Zavattaro, editors. In *Proceedings of FMOODS'05*, Athens, Greece, June 2005. *LNCS 3535*, Springer, ISBN: 3-540-26181-8. pp. 179-194, May 2005.
19. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.
20. A. K. Wright and M. Felleisen, A Syntactic Approach to Type Soundness. In *Information and Computation*, Vol. 115, No. 1, pp. 38–94, 1994.
21. M. Zenger, Type-Safe Prototype-Based Component Evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.