VNU Journal of Science, Natural Sciences and Technology 24 (2008) 82-91

Finding upper bounds of component instances with deallocation beyond local scope

Hoang A. Truong*

College of Technology, VNU, 144 Xuan Thuy Road, Cau Giay District, Hanoi, Vietnam

Received 31October 2007

Abstract. We develop an abstract component language and a static type system that can tells us the maximum resources a program may use. We prove that the upper resource bound is sharp and we point out a polynomial algorithm that can infer the sharp bound. Knowing the maximal resources a program may request allows us to adjust resource usage of the program and to prevent it from raising exceptions or behaving unexpectedly on systems that do not have enough resources. This work extends our previous works in one crucial point: the deallocation primitive can free an instance beyond its local scope. This semantics makes the language much closer to practical ones.

1. Introduction

Any software program needs resources to run. These resources can be physical components such as memory or communication ports, or they can be virtual components of the operating system or the underlying runtime machine such as file handles or TCP/IP sockets. As most of these resources are limited, any computer program should be prepared for the out-of-resource situation at runtime.

There are several solutions to the problem, ranging from dynamic checking, testing to static analysis. Runtime checking for failure every time the program requests for a resource is costly. These dynamic checks increase the program size and reduce its performance. On embedded and handheld devices only a small overhead is significant. Even when the dynamic checks are inserted, the program still stops working when the system does not have enough resource. Testing is always necessary, but it does not cover all possibilities. Furthermore, testing may not be applicable for modern applications which are extensible, composable from modules of thirdparties and these modules can be updated automatically. The last method is the best if possible. It allows us to detect potential problems at compile time, before the program is deployed.

Component software is built from various components, possibly developed by third-parties [1,2]. These components may in turn use other components and so on. Upon execution, instances of these components and their sub-compnents are created and discarded. Since each instance uses some resources, some components are required to have only a certain number of simultaneously active instances.

In this paper we explore the possibility of a type system [3-5], a branch of static analysis,

* E-mail: hoangta@vnu.edu.vn

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 does not directly control actual resources. Instead we will abstract them by the number of instances. Using types and effects systems [6,7], we can infer every specific resource by adding annotations to components using the resource.

This work extended our previous work [8] by allowing the deallocation operate beyond local store. The simple change in the operational semantics requires additional information in type expressions and some typing rules also need changes substantially. As it is unusual to allow deallocation go beyond a thread, we leave out the parallel composition for simplicity. The type system can be extended with the similar rule in [8] if we add the parallel composition to the language.

The paper is organized as follows. Section 2 introduces the component language and a small-step operational semantics. Section 3 defines types and the typing relation. Section 4 shows several important properties of the system, among them are type soundness and sharpness of resource bounds. Last, Section 5 concludes.

2. A component language

2.1. Syntax

Component programs, declarations and expressions are defined in Table 2.1. We use extended Backus-Naur Form with infix | for choice and overlining for Kleene closure (zero or more iterations).

Table 1. Syntax

$Prog$	::=	$Decls : E$	Program
$Decls$	::=	$x \prec E$	Declarations
E	::=		Expression
		ϵ	Empty
	|	`new`x	Instantiation
	|	`del`x	Deallocation
	|	$(E + E)$	Choice
	|	$\{E\}$	Scope
	|	$E\ E$	Sequencing

Component names, ranged over by x, y, z, are collected in a set C. Component expressions, ranged over by A, .., E, can be empty expressions – used for startup, or they can be formed by two primitives new and `del` for creating and deleting an instance of a component, respectively, or they can be assembled by three composition operators: choice, denoted by +, scope, denoted by {}, and sequencing denoted by juxtaposition.

A new component x can be introduced from a component expression E by a declaration of the form $x \prec E$, which states that component x deploys the component expression E. We also call E the *body* of x. For startup, we can declare a so-called *primitive component* by giving its

body an empty expression $x \prec \varepsilon$. A primitive component is the one that does not depend on any other components, so it can be used to represent some specific resource such as a serial communication port.

A component program is defined by a list of component declarations followed by a main expression, which will be the startup expression when the program is executed.

2.2. Operational semantics

Table 2.2 defines formally the operational semantics by a transition system between configurations. A configuration is a stack of pairs of a multiset and an expression. A configuration is *terminal* if it has the form (M, ε). We denote a stack ST of n element by $(M_1, E_1) \circ ... \circ (M_n, E_n)$ where (M_1, E_1) is the bottom, (M_n, E_n) is the top of the stack, and '$\circ$' is the stack separator.

Table 2. Transition rules

(osNew) $x \prec A \in Decls$
$\mathbb{T} \circ (M, \mathtt{new}\, x E) \rightsquigarrow \mathbb{T} \circ (M + x, AE)$

(osDel) $x \in M_k$ and $x \notin M_{k+1}, ..., M_n$
$\mathbb{T} \circ (M_k, E_k) \circ .. \circ (M_n, \mathtt{del}\, x E)$
$\rightsquigarrow \mathbb{T} \circ (M_k - x, E_k) \circ .. \circ (M_n, E)$

(osChoice) $i \in \{1, 2\}$
$\mathbb{T} \circ (M, (A_1 + A_2) E) \rightsquigarrow \mathbb{T} \circ (M, A_i E)$

(osPush)
$\mathbb{T} \circ (M, \{A\} E) \rightsquigarrow \mathbb{T} \circ (M, E) \circ ([], A)$

(osPop)
$\mathbb{T} \circ (M, E) \circ (M', \epsilon) \rightsquigarrow \mathbb{T} \circ (M, E)$

By the rules osNew, osDel, and osChoice we only rewrite the pair at the top of the 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 first store from the top of the stack, if there exists one. If there exists no instance of x in the whole stack, 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 and rules osPush and osPop are for the scope operator.

Up until now we have fully described the component language. Now we take a look at how specific resources are measured to answers the usual questions like how much memory or how many serial communication ports a program uses?

Given a program, a natural way to infer the maximum amount of a resource that the program needs is to annotate the usage of that resource that each component directly uses. That is, we have a function for each resource that maps every component name to the amount of the resource that the component directly uses. Then we can run the program and calculate the total resource consumption of each execution state by taking the sum of resources occupied by all existing instances. For example, if a program has four components *a, b, c, d* and components *a* and *c* each uses 1KB of memory, components *b* and *d* each uses 2KB. Then at the state ([*b, e, d, d*], *E*), the program occupies 7KB of memory.

In the above method, we need to examine all possible states of the program to know the maximum resources that the program needs. In general, these methods are not applicable to detect these maxima since testing all possible runs is usually impossible due to a possible exponential number of such runs or circular dependencies of components. The type system in the next section can tell us the maximum resource consumption for a class of programs and it inspires a polynomial algorithm to find such an upper bound.

3. A type system

The main purpose of our type system is to find out the maximum number of coexistent instances that a program can create during the running of the program. We will need the maximum number for each component, so that mean we need to find a set of pair *x, n* for each component *x*. This is exactly the notion of multiset, which is a set with multiple occurrences of elements. Therefore multiset is the right data structure for storing these maxima in a type expression.

Another important aspect of most type systems is the property so-called compositionality. That is, type of an expression can be computed from types of its subexpressions. In our language, the choice composition is not rather straightforward since the maximum if $A+B$ is maximum of two maxima of A and B, while the sequential composition is the much more sophisticated.

When composing AB we need to know the maximum number of instances of A. During the running of B, we need to know the maximum number of instances that A left after its execution. So we need another multiset. But B can be a deallocation such as *delx*, this multiset should also has negative elements. So it needs to be a *signed multiset*. A signed multiset is a multiset but with negative occurrences of elements.

Another point we need the type system to be able to detect is the safety of deallocation. When composing AB and B may have some deallocations, then we need to make sure that A has at least enough instances created so that deallocation in B can be executed safely. Therefore, we need another multiset for storing the minimum number of instances that B needs and this multiset will allow B to be composed safely with any A that can create such minima.

Last, when an expression A is enclosed in a scope, $\{A\}$ will not increase the number of instances of after the execution of $\{A\}$, but it still can delete instances in the environment, as we can see the rule osDel. The maximum deallocation is exactly the safety multiset mentioned in previous paragraph. The minimum, however is the minimum number of instances that barely guarantees the safety of deallocation in A, in the run that has the least

number of deallocations. Therefore, we need two safety stores for typing scope expressions.

Types are tuples of three multisets and two signed multisets. We let $X,..,Z$ range over types.

Definition 3.1. (Types). *Types of component expressions are tuples*

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

where X^s, X^r, X^i *are multisets and* X^o, X^l *are signed multisets.*

Multisets are denoted by [...], where sets are denoted, as usual, by {...}. $M(x)$ 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 $M - [x]$. Domain of M, also called *support set*, notation dom(M), is the set of elements that occur in M: $\text{dom}(M) = \{x \mid M(x) \neq 0\}$.

Similarly, a signed multiset M, also denoted by [...], over a set S is a map from S to $\mathbb{Z}$, the set of integers. For example, [x, -y, -y] is a signed multiset where the multiplicity of x is 1 and the multiplicity of y is -2. Signed multisets are also called hybrid set [9]. The analogous operations of multisets are defined 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 have additive union: $(M + N)(x) = M(x) + N(x)$; subtraction: $(M - N)(x) = M(x) - N(x)$; 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 $x \in M$; domain or support set $\text{dom}(M) = \{x \mid M(x) \neq 0\}$. Last, we define $[M]^-$ be the multiset received from M by removing all elements with positive occurrences:

$$[M]^-(x) = \begin{cases} M(x), & \text{if } M(x) < 0 \\ 0, & \text{if } M(x) \geq 0 \end{cases}$$

Having the meanings of each part of a type, Table 3 describes all typing rules. Before looking at that table, we need to clarify some terminologies. A basis or *typing environment* is a list of declarations: $x_1 \prec E_1,...,x_n \prec E_n$. An empty basis is denoted by $\varnothing$. Let Γ, Δ range over bases. The *domain of a basis* $\Gamma = x_1 \prec E_1,...,x_n \prec E_n$, notation dom($\Gamma$), is the set $\{x_1,...,x_n\}$ .A *typing judgement* (or *just judgement*) is a tuple of the form:

$$\Gamma \vdash E : X$$

and it asserts that expression E has type X in the environment Γ. A typing judgement can be regarded as *valid* or *invalid*. Valid ones are identified by the following definitions.

Definition 3.2. (Valid typing judgements). *Valid typing judgements* $\Gamma \vdash A : X$ *are derived by applying the typing rules in Table 3 in the usual inductive way.*

By the term usual inductive way we mean a valid judgement is one that can be obtained as the root of a tree of judgements, where each judgement is obtained from the ones immediately above it by some typing rule in Table 3. Such a tree of judgements is called a *typing derivation*.

Table 3. Typing rules

(Axiom)

$$\varnothing \vdash \epsilon : ([],[],[],[],[])$$

(Weaken)

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

(New)

$$\frac{\Gamma \vdash A : X \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x \prec A \vdash \mathtt{new}\, x : (X^s, X^r, X^r + x, X^v + x, X^l - x)}$$

(Del)

$$\frac{\Gamma \vdash A : X \quad x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathtt{del}\, x : ([x],[x],[],[-x],[-x])}$$

(Seq)

$$\frac{\begin{array}{c}\Gamma \vdash A : X \quad \Gamma \vdash B : Y \quad A, B \neq \epsilon \\ Z^s = X^s \cup (Y^s - X^l) \quad Z^r = X^r \cup (Y^r - X^v) \\ Z^r = X^r \cup (X^v + Y^r) \quad Z^v = X^v + Y^v \quad Z^l = X^l + Y^l\end{array}}{\Gamma \vdash AB : Z}$$

(Choice)

$$\frac{\begin{array}{c}\Gamma \vdash A : X \quad \Gamma \vdash B : Y \\ Z = X^s \cup Y^s, X^r \cap Y^r, X^r \cup Y^r, X^v \cup Y^v, X^l \cap Y^l\end{array}}{\Gamma \vdash (A + B) : Z}$$

(Scope)

$$\frac{\Gamma \vdash A : X}{\Gamma \vdash \{A\} : (X^s, X^r, X^r, -X^r, -X^s)}$$

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. The first multiset of the type of an expression is for the safety of deallocations in the expression. First, we 4 still need X^s for the safety of deallocations in A. Second, since there are at least X^l instances after the execution of A, we need at least $\left(Y^s - X^l\right)$ for the safety of B. Therefore, we need $X^s \cup \left(Y^s - X^l\right)$ instances for the safety of deallocations in AB. The second multiset is analogous, but for the

minimal safety of deallocations. The third multiset is the maximum instances that AB can reach. It can be the maximum of A or the maximal outcome of A together with the maximum of B. The remaining two signed multisets, $X^0 + Y^0$ and $X^1 + Y^1$, are easy referring to the semantics of them.

Other typing rules are straightforward. The rule Axiom is used for startup. The rule WeakenB allows us to extend a basis so that the rules Seq, Choice may be applied. The rule New accumulates a new instance in type expression while the rule Del reduces by one instance. In the rule Del, the first two multisets are for the safety of the deallocation. The third multiset in the type of `del` x is empty since it has no effect to the maximum in composition, but the last two multisets are both $[-x]$ since del x removes one x from the environment. The judgement $\Gamma \vdash A : X$ in the premise of this rule only guarantees that the basis Γ is legal.

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 a list of the program declarations. As mentioned in the Introduction Section 1, we have an polynomial algorithm (cf. [9]) which can automatically decide whether a program is well-typed or not.

Definition 3.3. (Well-typed programs). *Program Prog = Decls; E is well-typed if there exists a reordering Γ of declarations in Decls such that $\Gamma \vdash E : X$.*

4. Soundness and sharpness

We state several important properties of the type system and left out some supporting properties that are similar as in [8].

4.1. Soundness properties

One of the most important properties of static type systems is the soundness. 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 does not exists 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 maximum stated in its type, and the maximum is sharp.

Our proof of the type soundness is based on the approach of Wright and Felleisen [10]. 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 nonterminal state, from which it cannot move to another state. First we need to define what a well-typed configuration means.

Definition 4.1. (Well-typed configuration). *Configuration $\mathbb{T} = (M_1, E_1) \circ \ldots \circ (M_n, E_n)$ is well-typed with respect to a basis Γ, notation $\Gamma \vDash \mathbb{T}$, if for $h = 1..n$ there exists Z_h such that*

$$\Gamma \vdash E_h : Z_h \text{ and } [\mathbb{T}|_h] + \text{rets}_{\mathbb{T}}(h+1) - Z_h^+ \geq 0$$

where

$$\text{rets}_{\mathbb{T}}(h) = [M_h + \text{rets}_{\mathbb{T}}(h+1) - Z_h^+]^-$$

Note that we have simplified the definition of rets for trivial cases that $\text{rets}_{\mathbb{T}}(k) = 0$ for $k > n$.

The two standard lemmas for soundness property are stated as follows.

Lemma 4.2. (Preservation). *If* $\Gamma \vDash \mathbb{T}$ *and* $\mathbb{T} \rightsquigarrow \mathbb{T}'$, *then* $\Gamma \vDash \mathbb{T}'$.

Lemma 4.3. (Progress). *If* $\Gamma \vDash \mathbb{T}$, *then either* $\mathbb{T}$ *is terminated or there exists a configuration* $\mathbb{T}'$ *such that* $\mathbb{T} \rightsquigarrow \mathbb{T}'$.

Next, we show an invariant which allows us to infer the resource bounds of well-typed programs. The invariant is about the monotonicity of the maximum number of instances that a well-typed configuration $\mathbb{T} = \{M_1, E_1\}$ o...o (M_n, E_n) can reach. We calculate the maximum number as follows.

$$\mathsf{maxins}(\mathbb{T}) = \bigcup_{h=1}^{n} \mathsf{maxins}(\mathbb{T}, h)$$

Where

$$\mathsf{maxins}(\mathbb{T}, h) = [\mathbb{T}|_h] + X'_h + \mathsf{reto}_{\mathbb{T}}(h+1)$$

$$\mathsf{reto}_{\mathbb{T}}(h) = [M_h + \mathsf{reto}_{\mathbb{T}}(h+1) + [X'_h]^-]^-$$

Where X_h is the type of E_h. During transition, this maximum number of instances that the configuration can generate does not increase. Furthermore, when the maximum is not reach for some component, there exists a next configuration such that the maximum is the same. This allows us to prove the sharpness of the type system.

Lemma 4.4. (Invariant of maxins). *If* $\Gamma \vDash \mathbb{T}$ *and* $\mathbb{T} \rightsquigarrow \mathbb{T}'$, *then*

- $\mathsf{maxins}(\mathbb{T}) \supseteq \mathsf{maxins}(\mathbb{T}')$
- *If* $[\mathbb{T}](z) < \mathsf{maxins}(\mathbb{T})(z)$ *for some z, then there exists* $\mathbb{T}''$ *such that* $\mathbb{T} \rightsquigarrow \mathbb{T}''$ *and* $\mathsf{maxins}(\mathbb{T}) = \mathsf{maxins}(\mathbb{T}'')$

Now we can state the type soundness together with the upper bound of instances that a welltyped program always respects.

Theorem 4.5. (Soundness). *Let Prog = Decls; E be well-typed, that is,* $\Gamma \vdash E : X$ *for some reordering* Γ *of Decls and some type X. Then for any* $\mathbb{T}$ *such that* $([], E) \rightsquigarrow^* \mathbb{T}$ *we have that T is not stuck and* $[\mathbb{T}] \subseteq X'$.

4.2. Termination and sharpness

Before presenting the sharpness property, we need to show that any welltyped program terminates in a finite number of steps. The common tool for proving the termination of programs (cf. [11, 5]) is to find a *termination function* which maps program states to a *well-founded set*. A well-founded set is a set S with an ordering > on elements of S such that there can be no infinite descending sequences of elements. We choose the set of natural numbers N and the usual ordering > to be a well-founded set (N,>). The termination function mts is defined for expressions and for configurations as follows.

$$\mathsf{mts}(E) = \begin{cases} 0, \text{ if } E = \epsilon \\ 1, \text{ if } E = \texttt{del}\,x \\ 1 + \mathsf{mts}(A), \text{ if } E = \texttt{new}\,x, x \prec A \in Decls \\ \mathsf{mts}(A) + \mathsf{mts}(B), \text{ if } E = AB \\ 2 + \mathsf{mts}(A), \text{ if } E = \{A\} \\ 1 + \max(\mathsf{mts}(A), \mathsf{mts}(B)), \text{ if } E = (A + B) \end{cases}$$

The integers 0, 1 and 2 in the definition are the corresponding steps of the operational semantics. The function is defined for a stack $\mathbb{T} = (M_1, E_1) \circ \ldots \circ (M_n, E_n)$ as follows:

$$\mathrm{mts}(\mathbb{T}) = \sum_{i=1}^{n} \mathrm{mts}(E_i) + n - 1$$

Here n - 1 is the number of osPop steps.

Note that, if E is the main expression of a well-typed program, then mts(E) is the maximum transition steps that the program takes to terminate in any run, not all possible runs of the program because there may be an exponential number of such runs. The following theorem guarantees the termination of any well-typed program.

Theoram 4.6. (Termination)

1. If $\Gamma \vDash \mathbb{T}$ *and* $\mathbb{T} \rightsquigarrow \mathbb{T}'$, *then* mts($T$) > mts ($\mathbb{T}'$)

2. A well-typed program always terminates in a finite number of steps.

Last, the sharpness of the type system shows that there is a run of any welltyped program such that the maximum number of instances reaches the bound expressed in program's type.

Theorem 4.7 (Sharpness). *Let Prog = Decls; E be welltyped, that is,* $\Gamma \vdash E : X$ *for some reordering* Γ *of Decls and some type X. Then for any* $z \in X^i$, *there exists a sequence of configurations* $([],E) = \mathbb{T}_0 \rightsquigarrow \mathbb{T}_1 \rightsquigarrow \ldots \rightsquigarrow \mathbb{T}_n$ *such that* $[\mathbb{T}](z) = Xi(z)$.

4.3. Type inference

Type inference is similar to those in our previous works [8, 9]. We have a polynomial type inference algorithm that can infer the type of a program if there is one, and it reports failure otherwise.

5. Related works and conclusion

There are several other works on static and analysis of memory use. In [12,13] Chin et. Al. presented a type system that can capture memory bounds of object-oriented programs. He provided a framework in [13] for inferring abstract size of programs as exact as possible (since they used Pres-burger formulae for size information). Our language has an explicit deallocation primitive and our computation of resource bounds is exact. Crary and Weirich [14] presented decidable type systems for low level languages which are capable of specifying and certifying that their programs will terminate within a given amount of time, but the type system does not infer any bounds given by programmers. In contrast, out type systems focus on high level languages and they can infer the sharp upper bounds of resources. Hofmann [15] showed that linear type systems can ensure that programs do not increase the size of their input so that exponential growth of immediate results can be avoided, even with the presence of iterated recursion. His languages are functional while ours are imperative.

We have presented an abstract component language that focuses on two primitives for manipulating resources (allocation and deallocation) and three composition operators: sequencing, choice, and scope. These operators are of particular relevant to the dynamic semantics of the two primitives for allocating and freeing resources. Then we have developed a static type system that can find the sharp resource bounds of a component program. The type inference algorithm is polynomial as

shown in our previous works. Due to space limitations, proofs are not included here. We plan to provide a technical report that contains all proofs.

We have left out some features such as loops, function calls, and recursions to simplify the system. Adding finite loops and function calls would not be difficult and would not cause substantial changes to the type systems. We plan to consider them in the future.

This work was partly supported by the research project No. 204006 entitled "Modern Methods for Building Intelligent Systems" granted by the National IT Fundamental Research Program of Vietnam.

References

[1] C. Szyperski, "Component Software—Beyond Object- Oriented Programming", *Addison-Wesley / ACM Press*, 2nd edition, 2002.

[2] T.L. Thai, H.Q. Lam, "NET Framework Essentials", *A Nutshell Handbook. O'Reilly & Associates, Inc.*, 3rd edition, Aug. 2003.

[3] H.P. Barendregt, "Lambda Calculi with Types", *Oxford Univertity Press* Vol. 2 (1992) 117.

[4] L. Cardelli, "Type systems", *ACM Comput. Surv.*, 28(1) (1996) 263.

[5] B.C. Pierce, editor, "Types and Programming Languages", *MIT Press*, 2002

[6] F. Nielson, H. R. Nielson, "Type and effect systems", In Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel), Springer–Verlag., London, UK, (1999) 114.

[7] F. Nielson., "Annotated type and effect systems", *ACM Comput. Surv*, 28(2) (1996) 344.

[8] H. Truong, M. Bezem, "Finding resource bounds in the presence of explicit deallocation", *In D. V. Hung and M. Wirsing, editors, ICTAC*, Lecture Notes in Computer Science, Springer, Vol. 3722 (2005) 227.

[9] M. Bezem, H. Truong, "A type system for the safe instantiation of components", *Electronic Notes in Theoretical Computer Science*, 97 (2004) 197.

[10] A.K. Wright, M. Felleisen, "A syntactic approach to type soundness", *Information and Computation*, 115(1) (1994) 38.

[11] N. Dershowitz, Z. Manna, "Proving termination with multiset orderings", *Communications of the ACM*, 22(8) (1979) 465.

[12] S. Q. Wei-Ngan Chin, Huu Hai Nguyen, M. Rinard, "Memory usage verification for OO programs", *In C. Hankin and I. Siveroni, editors, The 12th International Static Analysis Symposium (SAS'05)*, London, UK, Sept. 2005.

[13] W.N. Chin, S.C. Khoo, "Calculating sized types", *Higher-Order and Symbolic Computation*, 14(2-3) (2001) 261.

[14] K. Crazy, D. Walker, G. Morrisett, "Typed memory management in a calculus of capabilities", In *POPL'99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM Press (1999) 262.

[15] M. Hofmann, "Linear types and non size-increasing poly-nomial time computation", In *LICS'99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, Washington. DC, USA, IEEE Computer Science, (1999) 464.

[16] A. Syropoulos, "Mathematics of multisets", In *WMP '00: Proceedings of the Workshop on Multiset Processing*, London, UK, SpringerVerlag (2001) 347.