Resource Usage Analysis for the π-Calculus

Naoki Kobayashi[1], Kohei Suenaga[2], and Lucian Wischik[3]

[1] Tohoku University, koba@ecei.tohoku.ac.jp
[2] University of Tokyo, kohei@yl.is.s.u-tokyo.ac.jp
[3] Microsoft Corporation, lwischik@microsoft.com

Abstract. We propose a type-based resource usage analysis for the π-calculus extended with resource creation/access primitives. The goal of the resource usage analysis is to statically check that a program accesses resources such as files and memory in a valid manner. Our type system is an extension of previous behavioral type systems for the pi-calculus, and can guarantee the safety property that no invalid access is performed, as well as the property that necessary accesses (such as the close operation for a file) are eventually performed unless the program diverges. A sound type inference algorithm for the type system is also developed to free the programmer from the burden of writing complex type annotations. Based on the algorithm, we have implemented a prototype resource usage analyzer for the π-calculus. To the authors' knowledge, ours is the first type-based resource usage analysis that deals with an expressive concurrent language like the π-calculus.

1 Introduction

Computer programs access many external resources, such as files, library functions, device drivers, etc. Such resources are often associated with certain access protocols; for example, an opened file should be eventually closed and after the file has been closed, no read/write access is allowed. The aim of resource usage analysis [9] is to statically check that programs conform to such access protocols. Although a number of approaches, including type systems and model checking, have been proposed so far for the resource usage analysis or similar analyses [1, 5–7, 9], most of them focused on analysis of sequential programs, and did not treat concurrent programs, especially those involving dynamic creation/passing of channels and resources.

In the present paper, we propose a type-based method of resource usage analysis for *concurrent languages*. Dealing with concurrency is especially important because concurrent programs are hard to debug, and also because actual programs accessing resources are often concurrent. We use the π-calculus (extended with resource primitives) as a target language so that our analysis can be applied to a wide range of concurrency primitives (including those for dynamically creating and passing channels) in a uniform manner.

For the purpose of analyzing resource usage, we extend previous behavioral type systems for the π-calculus [3, 8]. The idea of the behavioral types [3, 8] is to use CCS-like processes as types. The types express abstract behavior of processes, so that certain properties of processes can be verified by verifying the corresponding properties of their types, using, for example, model checking techniques. The latter properties (of CCS-like types) are more amenable to automatic verification techniques like model checking than the former ones, because the types do not have channel mobility and also because the types typically represent only the behavior of a part of the entire process.

Following the previous behavioral types, we use CCS-like types to express resource-wise access behaviors of a process and construct a type system which guarantees that any well-typed process uses resources in a valid manner. The main contributions of the present paper are:

- Adaption of behavioral types (for pure π-calculus) [3, 8] to the π-calculus extended with resource access primitives.
- Realization of fully automatic verification (while making the analysis more precise than [8]). Igarashi and Kobayashi [8] gave only an abstract type system, without giving a concrete type inference algorithm. Chaki et al. [3] requires type annotations. The full automation was enabled by a combination of a number of small ideas, like inclusion of hiding and renaming as type constructors (Igarashi and Kobayashi [8] used a fragment without hiding and renaming, and Chaki et al. [3] used a fragment without renaming), approximation of a CCS-like type by a Petri net (to reduce the problem of checking conformance of inferred types to resource usage specification).
- Verification of not only the usual safety property that an invalid resource access does not occur, but also an extended safety (which we call *partial liveness*) that necessary resource accesses (e.g. closing of a file) are eventually performed unless the whole process diverges. The partial liveness is not guaranteed by Chaki et al.'s type system [3]. A noteworthy point about our type system for guaranteeing the partial liveness is that it is parameterized by a mechanism that guarantees deadlock-freedom (in the sense of Kobayashi's definition [13]). So, our type system can be combined with *any* mechanism (model checking, abstract interpretation, another type system, or whatever) to verify deadlock-freedom.
- Implementation of a prototype resource usage analyzer based on the proposed method. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

The rest of this paper is structured as follows. Section 2 introduces an extension of the π-calculus with primitives for creating and accessing resources. Section 3 introduces a type system for resource usage analysis. Section 4 gives a type inference algorithm for the type system. Section 5 presents our prototypical implementation. Section 6 discusses related work. Section 7 concludes. For lack of space, proofs and some explanations have been omitted. They are found in the full version of this paper [15].

2 Language

Let x, y, z range over a countably infinite set **Var** of variables, let values v range over variables and also the two constant values **true** and **false**, let tags t range over $\{\emptyset, \mathbf{c}\}$, let ξ range over a set of *access labels*, and let Φ (called a *trace set*) denote a set of sequences of access labels, possibly ending with a special label $\downarrow$, that is closed under the prefix operation. We write $\widetilde{x}$ for a sequence $x_1, \ldots, x_n$ of variables, and similarly $\widetilde{v}$, and define $\Phi^{-\xi} = \{s \mid \xi s \in \Phi\}$. Let L range over *reduction labels* $\{x^\xi \mid x \in \mathbf{Var}\} \cup \{\tau\}$.

$$P ::= \mathbf{0} \mid (P \,|\, Q) \mid \textbf{if } v \textbf{ then } P \textbf{ else } Q \ \mid (\nu x)\, P \mid *P$$
$$\mid \overline{x}_t\langle\widetilde{v}\rangle.\, P \mid x_t(\widetilde{y}).\, P \mid (\mathfrak{N}^\Phi x)P \mid \mathbf{acc}_\xi(x).P$$

Structural preorder $\preceq$ is as follows. $P \equiv Q$ stands for $(P \preceq Q) \wedge (Q \preceq P)$.

$$P \,|\, \mathbf{0} \equiv P \qquad P \,|\, Q \equiv Q \,|\, P \qquad P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R \qquad *P \preceq *P \,|\, P$$

$$(\nu x)\, P \,|\, Q \preceq (\nu x)\,(P \,|\, Q) \text{ and } (\mathfrak{N}^\Phi x)P \,|\, Q \preceq (\mathfrak{N}^\Phi x)(P \,|\, Q) \textit{if } x \textit{ not free in } Q$$

$$\frac{P \preceq P' \quad Q \preceq Q'}{P \,|\, Q \preceq P' \,|\, Q'} \qquad \frac{P \preceq Q}{(\nu x)\, P \preceq (\nu x)\, Q} \qquad \frac{P \preceq Q}{(\mathfrak{N}^\Phi x)P \preceq (\mathfrak{N}^\Phi x)Q}$$

Labeled relation $\xrightarrow{L}$ is as follows. Write $P \longrightarrow Q$ when $P \xrightarrow{L} Q$ for some L, and $\longrightarrow^*$ for reflexive and transitive closure of $\longrightarrow$. Define $target(x^\xi) = \{x\}$ and $target(\tau) = \emptyset$.

$$\overline{x}_t\langle\widetilde{z}\rangle.\, P \,|\, x_{t'}(\widetilde{y}).\, Q \xrightarrow{\tau} P \,|\, [\widetilde{z}/\widetilde{y}]Q \qquad \mathbf{acc}_\xi(x).P \xrightarrow{x^\xi} P \qquad \begin{array}{l} \textbf{if true then } P \textbf{ else } Q \xrightarrow{\tau} P \\ \textbf{if false then } P \textbf{ else } Q \xrightarrow{\tau} Q \end{array}$$

$$\frac{P \xrightarrow{L} Q}{P \,|\, R \xrightarrow{L} Q \,|\, R} \qquad \frac{P \xrightarrow{L} Q \qquad x \notin target(L)}{(\nu x)\, P \xrightarrow{L} (\nu x)\, Q} \qquad \frac{P \xrightarrow{x^\xi} Q}{(\mathfrak{N}^\Phi x)P \xrightarrow{\tau} (\mathfrak{N}^{\Phi^{-\xi}} x)Q}$$

$$\frac{P \xrightarrow{L} Q \qquad x \notin target(L)}{(\mathfrak{N}^\Phi x)P \xrightarrow{L} (\mathfrak{N}^\Phi x)Q} \qquad \frac{P \preceq P' \qquad P' \xrightarrow{L} Q' \qquad Q' \preceq Q}{P \xrightarrow{L} Q}$$

Fig. 1. Process language

The process language P is in Figure 1. The first line is standard π-calculus – $(\nu x)\, P$ declares a new channel with bound name x, $*P$ is replication, and parallel composition $\mid$ binds less tightly than the prefixes. We often omit trailing $\mathbf{0}$. Bound and free variables are defined as normal. We identify processes up to α-conversion, and so assume that bound variables are always different from each other and from free variables.

The input command $x_t(\widetilde{y}).\, P$ waits for input on channel x with bound formal parameters $\widetilde{y}$ and then behaves as P. The output command $\overline{x}_t\langle\widetilde{v}\rangle.\, P$ sends $\widetilde{v}$ along x and then behaves as P. The attribute t is either $\mathbf{c}$ (indicating that if the input is executed then it will succeed unless the whole process diverges) or $\emptyset$ (which does not give the guarantee). We often omit $\emptyset$. Note that the attributes do not affect the operational semantics of processes. Typically the attributes have been inferred by some deadlock analysis tool such as `TyPiCal` [10–12, 14]. For this paper, we assume that the correctness of the attributes are ensured by whichever deadlock-analysis tool used to make the annotations.

For the final line in the definition of processes, $(\mathfrak{N}^{\Phi}x)P$ declares a resource with bound name x which is to be accessed according to specification Φ, and $\mathbf{acc}_{\xi}(x).P$ performs access ξ on resource x and then behaves like P. Resources here are an abstraction of real-world resources such as files or objects. In this paper we consider accesses such as I for initialize, R for read, W for write and C for close. For example, $(\mathfrak{N}^{(I(R+W)^{*}C\downarrow)^{\#}}x)P$ creates a resource that should be first initialized, read or written an arbitrary number of times, and then closed. The symbol $\downarrow$ at the end indicates that the final close is required eventually to occur. Here, $(S)^{\#}$ is the prefix closure of S, i.e., $\{s \mid ss' \in S\}$. We write ϵ for the empty access sequence. We write $\mathbf{init}(x).P$ for $\mathbf{acc}_{I}(x).P$, and similarly $\mathbf{read}(x)$, $\mathbf{write}(x)$, $\mathbf{close}(x)$. We do not fix the syntax of Φ. Our type system is independent of the choice of the language for describing the specificaiton Φ (except for the sub-algorithm for type-checking discussed in Section 4.1, where we assume that Φ is a regular language).

We treat resources as primitives in this paper, and give operational semantics where $\mathbf{acc}_{\xi}(x).P$ is non-blocking. This is for simplicity. It would also be possible to treat a resource with (say) three access labels as a tuple of three channels. This would allow previous work [3, 8] to infer some of the properties of this paper, albeit with less precision and more complexity. Also in this paper we have specifications Φ apply only to a single resource. To model a program with two co-declared resources as in [8] with intertwined specifications, we would instead merge them into a single resource with a single specification.

The operational semantics of the language are given in Figure 1, through a *structural preorder* $\preceq$ and a labeled reduction relation $\xrightarrow{L}$. Notice that invalid resource access sets $\Phi = \emptyset$, valid access removes a prefix from Φ, and complete access results in $\Phi = \{\epsilon, \downarrow\}$).

$$(\mathfrak{N}^{(IC\downarrow)^{\#}}x)\mathbf{read}(x).\mathbf{0} \to (\mathfrak{N}^{\emptyset}x)\mathbf{0} \qquad \text{(invalid access)}$$
$$(\mathfrak{N}^{(IC\downarrow)^{\#}}x)\mathbf{init}(x).\mathbf{0} \to (\mathfrak{N}^{(C\downarrow)^{\#}}x)\mathbf{0} \qquad \text{(valid access)}$$
$$(\mathfrak{N}^{(IC\downarrow)^{\#}}x)\mathbf{init}(x).\mathbf{close}(x).\mathbf{0} \to (\mathfrak{N}^{\{\epsilon,\downarrow\}}x)\mathbf{0} \qquad \text{(complete access)}$$

We are concerned with the following properties.

Definition 1 1. A process P is *safe* if it does not contain a sub-expression of the form $(\mathfrak{N}^{\emptyset}x)Q$.
2. A process P is *partially live* if $\downarrow \in \Phi$ whenever $P \longrightarrow^{*}\preceq (\widetilde{\nu\mathfrak{N}})(\mathfrak{N}^{\Phi}x)Q \not\longrightarrow$.

The first property means that the process has not performed any invalid access. The second property means that necessary accesses are eventually performed before the whole process converges. In the next section, we shall develop a type system that guarantees the safety and partial liveness.

Example 1. The following example process is safe and partially live. It uses internal synchronization to ensure that the resource x is accessed in a valid order.

$$
\begin{array}{ll}
(\mathfrak{N}^{(IR^*C)^\#}x)(\nu y)\,(\nu z)\,(& \\
\quad \mathbf{init}(x).(\overline{y}\langle\,\rangle \,|\, \overline{y}\langle\,\rangle) & \textit{/* initialize x, and send signals */} \\
\quad |\, y_{\mathbf{c}}(\,).\,\mathbf{read}(x).\overline{z}\langle\,\rangle & \textit{/* wait on y, then read x, and signal on z */} \\
\quad |\, y_{\mathbf{c}}(\,).\,\mathbf{read}(x).\overline{z}\langle\,\rangle & \textit{/* wait on y, then read x, and signal on z */} \\
\quad |\, z_{\mathbf{c}}(\,).\,z_{\mathbf{c}}(\,).\,\mathbf{close}(x)\,) & \textit{/* wait on z, then close x */}
\end{array}
$$

3 Type System

3.1 Types

We first introduce the syntax of types. We use two categories of types: value types and behavioral types. The latter describes how a process accesses resources and communicates through channels. As mentioned in Section 1, we use CCS processes for behavioral types.

Definition 2 (types) The sets of *value types* σ and *behavioral types* A are defined by:

$$
\begin{array}{l}
\sigma ::= \mathbf{bool} \mid \mathbf{res} \mid \mathbf{chan}\langle(x_1:\sigma_1,\ldots,x_n:\sigma_n)A\rangle \\
A ::= \mathbf{0} \mid \alpha \mid a_t.A \mid x^{\xi}.A \mid \tau_t.A \mid (A_1 \mid A_2) \mid A_1 \oplus A_2 \mid *A \\
\qquad \mid \langle y_1/x_1,\ldots,y_n/x_n\rangle A \mid (\nu x)\,A \mid \mu\alpha.A \mid A{\uparrow_S} \mid A{\downarrow_S} \\
a \text{ (communication labels) } ::= x \mid \overline{x}
\end{array}
$$

A behavioral type A, which is a CCS process, describes what kind of communication and resource access a process may perform. $\mathbf{0}$ describes a process that performs no communication or resource access. The types $x_t.\,A$, $\overline{x}_t.\,A$, $x^{\xi}.A$ and $\tau_t.A$ describes process that first perform an action and then behave according to A; the actions are, respectively, an input on x, an output on x, an access operation ξ on x, and the invisible action. Attributes t denote whether an action is guaranteed to succeed. $A_1 \,|\, A_2$ describes a process that performs communications and resource access according to A_1 and A_2, possibly in parallel. $A_1 \oplus A_2$ describes a process that behaves according to either A_1 or A_2. $*A$ describes a process that behaves like A an arbitrary number of times, possibly in parallel. $\langle y_1/x_1,\ldots,y_n/x_n\rangle A$, abbreviated to $\langle\widetilde{y}/\widetilde{x}\rangle A$, denotes simultaneous renaming of $\widetilde{x}$ with $\widetilde{y}$ in A. $(\nu x)\,A$ describes a process that behaves like A for some hidden channel x. For example, $(\nu x)\,(x.\,\overline{y} \,|\, \overline{x})$ describes a process that performs an output on y after the invisible action on x. The type $\mu\alpha.A$ describes a process that behaves like a recursive process defined by $\alpha \overset{\triangle}{=} A$. The type $A{\uparrow_S}$ describes a process that behaves like A, except that actions whose targets are in S are replaced by the invisible action τ, while $A{\downarrow_S}$ describes a process that behaves like

A, except that actions whose targets are not in S are replaced by τ. The formal semantics of behavioral types is defined later using labeled transition semantics.

As for value types, **bool** is the type of booleans. **res** is the type of resources. The type $\mathbf{chan}\langle(x_1 : \sigma_1, \ldots, x_n : \sigma_n)A\rangle$, abbreviated to $\mathbf{chan}\langle(\widetilde{x} : \widetilde{\sigma})A\rangle$, describes channels carrying tuples consisting of values of types $\sigma_1, \ldots, \sigma_n$. Here the type A approximates how a receiver on the channel may use the elements $x_1, \ldots, x_n$ of each tuple for communications and resource access. For example, $\mathbf{chan}\langle(x : \mathbf{res}, y : \mathbf{res})x^R.y^C\rangle$ describes channels carrying a pair of resources, where a party who receives the actual pair (x', y') will first read x' and then close y'. We sometimes omit $\widetilde{\sigma}$ and write $\mathbf{chan}\langle(\widetilde{x})A\rangle$ for $\mathbf{chan}\langle(\widetilde{x} : \widetilde{\sigma})A\rangle$. When $\widetilde{x}$ is empty, we also write $\mathbf{chan}\langle\rangle$.

Note that $\langle\widetilde{y}/\widetilde{x}\rangle$ is treated as a constructor rather than an operator for performing the actual substitution. We write $[\widetilde{y}/\widetilde{x}]$ for the latter throughout this paper. $\langle\widetilde{y}/\widetilde{x}\rangle A$ is slightly different from the *relabeling* of the standard CCS [17]: $\langle y/x\rangle(x \,|\, \overline{y})$ allows the communication on y, but the relabeling of CCS does not. This difference calls for the introduction of a special transition label $\{x, \overline{y}\}$ in Section 3.2.

$(\nu x)\, A$, $\langle\widetilde{y}/\widetilde{x}\rangle A$, and $A{\uparrow}_S$ bind x, $\widetilde{x}$, and the variables in S respectively. We write $\mathbf{FV}(A)$ for the set of free variables in A. We identify behavioral types up to renaming of bound variables. In the rest of this paper, we require that every channel type $\mathbf{chan}\langle(x_1 : \sigma_1, \ldots, x_n : \sigma_n)A\rangle$ must satisfy $\mathbf{FV}(A) \subseteq \{x_1, \ldots, x_n\}$. For example, $\mathbf{chan}\langle(x{:}\mathbf{res})x^R\rangle$ is a valid type but $\mathbf{chan}\langle(x{:}\mathbf{res})y^R\rangle$ is not. By abuse of notation, we write $\langle v_1/x_1, \ldots, v_n/x_n\rangle A$ for $\langle v_{i_1}/x_{i_1}, \ldots, v_{i_k}/x_{i_k}\rangle A$ where $\{v_{i_1}, \ldots, v_{i_k}\} = \{v_1, \ldots, v_n\}\backslash\{\mathbf{true}, \mathbf{false}\}$. For example, $\langle\mathbf{true}/x, y/z\rangle A$ stands for $\langle y/z\rangle A$.

3.2 Semantics of behavioral types

We give a labeled transition relation $\xrightarrow{l}$ for behavioral types. The transition labels l are

$$l \;::=\; x \mid \overline{x} \mid x^{\xi} \mid \tau \mid \{x, \overline{y}\}$$

The label $\{x, \overline{y}\}$ indicates the potential to react in the presence of a substitution that identifies x and y. We also extend *target* to the function on transition labels by:

$$\mathit{target}(x) = \mathit{target}(\overline{x}) = \{x\} \qquad \mathit{target}(\{x, \overline{y}\}) = \{x, y\}$$

Figure 2 shows a part of the definition of the transition relation $\xrightarrow{l}$ on behavioral types. For the complete definition, see the full paper [15]. We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$. We also write $\overset{l}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{l}\Longrightarrow$.

Remark 1. $(\nu x)\, A$ should not be confused with $A{\uparrow}_{\{x\}}$. $(\nu x)\, A$ is the hiding operator of CCS, while $A{\uparrow}_{\{x\}}$ just replaces any actions on x with τ [8]. For example, $(\nu x)\,(x.\, y^{\xi})$ cannot make any transition, but $(x.\, y^{\xi}){\uparrow}_{\{x\}} \xrightarrow{\tau}\xrightarrow{y^{\xi}} \mathbf{0}{\uparrow}_{\{x\}}$.

$$a_t.A \xrightarrow{a} A \qquad x^{\xi}.A \xrightarrow{x^{\xi}} A \qquad \tau_t.A \xrightarrow{\tau} A$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A{\uparrow_S} \xrightarrow{\tau} A'{\uparrow_S}} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A{\uparrow_S} \xrightarrow{l} A'{\uparrow_S}}$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A{\downarrow_S} \xrightarrow{l} A'{\downarrow_S}} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A{\downarrow_S} \xrightarrow{\tau} A'{\downarrow_S}}$$

Fig. 2. A Part of Definition of Transition semantics of behavioral types

We next define a predicate *disabled*(A, S) inductively as follows.

$$\begin{array}{rl}
\mathit{disabled}(\mathbf{0}, S) & \\
\mathit{disabled}(x^{\xi}.A, S) & \text{if } \mathit{disabled}(A, S) \text{ and } x \notin S \\
\mathit{disabled}(a_{\mathbf{c}}.A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}(a_{\emptyset}.A, S) & \\
\mathit{disabled}(\tau_{\mathbf{c}}.A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}(\tau_{\emptyset}.A, S) & \\
\mathit{disabled}(A_1 \mid A_2, S) & \text{if } \mathit{disabled}(A_1, S) \text{ and } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(A_1 \oplus A_2, S) & \text{if } \mathit{disabled}(A_1, S) \text{ or } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(*A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}((\nu x)\, A, S) & \text{if } \mathit{disabled}(A, S \backslash \{x\}) \\
\mathit{disabled}(A{\uparrow_{S'}}, S) & \text{if } \mathit{disabled}(A, S \backslash S') \\
\mathit{disabled}(A{\downarrow_{S'}}, S) & \text{if } \mathit{disabled}(A, S \cap S') \\
\mathit{disabled}(\langle \widetilde{y}/\widetilde{x} \rangle A, S) & \text{if } \mathit{disabled}(A, \{z \mid [\widetilde{y}/\widetilde{x}]z \in S\}) \\
\mathit{disabled}(\mu\alpha.A, S) & \text{if } \mathit{disabled}([\mu\alpha.A/\alpha]A, S)
\end{array}$$

Intuitively, *disabled*(A, S) means that A describes a process that may get blocked without accessing any resources in S.

The set $\mathbf{etraces}_x(A)$ defined below is the set of possible access sequences on x described by A.

Definition 3 (extended traces) The set $\mathbf{etraces}_x(A)$ of extended traces is:

$$\begin{array}{l}
\{\xi_1 \cdots \xi_n \downarrow \mid \exists B. A{\downarrow_{\{x\}}} \xRightarrow{x^{\xi_1}} \cdots \xRightarrow{x^{\xi_n}} B \wedge \mathit{disabled}(B, \{x\})\} \\
\cup \{\xi_1 \cdots \xi_n \mid \exists B. A{\downarrow_{\{x\}}} \xRightarrow{x^{\xi_1}} \cdots \xRightarrow{x^{\xi_n}} B\}
\end{array}$$

We define the subtyping relation $A_1 \leq A_2$ below. Intuitively, $A_1 \leq A_2$ means that a process behaving according to A_1 can also be viewed as a process behaving according to A_2. To put in another way, $A_1 \leq A_2$ means that A_2

simulates A_1.[4] We define $\leq$ for only *closed* types, i.e., those not containing free type variables.

Definition 4 (subtyping) The subtyping relation $\leq$ on closed behavioral types is the largest relation that satisfies the following properties:

- $A_1 \leq A_2$ and $A_1 \xrightarrow{l} A_1'$ implies $A_2 \xRightarrow{l} A_2'$ and $A_1' \leq A_2'$ for some A_2'.
- $\mathit{disabled}(A_1, S)$ implies $\mathit{disabled}(A_2, S)$ for any set S of variables.

We often write $A_1 \geq A_2$ for $A_2 \leq A_1$, and write $A_1 \approx A_2$ for $A_1 \leq A_2 \wedge A_2 \leq A_1$.

3.3 Typing

We consider two kinds of judgments, $\Gamma \triangleright v : \sigma$ for values, and $\Gamma \triangleright P : A$ for processes. Γ is a mapping from a finite set of variables to value types. In $\Gamma \triangleright P : A$, the type environment Γ describes the types of the variables, and A describes the possible behaviors of P. For example, $x : \mathbf{chan}\langle(b : \mathbf{bool})\mathbf{0}\rangle \triangleright P : \overline{x} \,|\, \overline{x}$ implies that P may send booleans along the channel x twice. The judgment $y : \mathbf{chan}\langle(x : \mathbf{chan}\langle(b : \mathbf{bool})\mathbf{0}\rangle)\overline{x}\rangle \triangleright Q : y$ means that Q may perform an input on y once, and then it may send a boolean on the received value. Note that in the judgment $\Gamma \triangleright P : A$, the type A is an approximation of the behavior of P on free channels. P may do less than what is specified by A, but must not do more; for example, $x : \mathbf{chan}\langle(\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle : \overline{x} \,|\, \overline{x}$ holds but $x : \mathbf{chan}\langle(\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle.\, \overline{x}\langle\,\rangle : \overline{x}$ does not. Because of this invariant, if A does not perform any invalid access, neither does P.

We write $\mathit{dom}(\Gamma)$ for the domain of Γ. We write $\emptyset$ for the empty type environment, and write $x_1 : \tau_1, \ldots, x_n : \tau_n$ (where $x_1, \ldots, x_n$ are distinct from each other) for the type environment Γ such that $\mathit{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Δ such that $\mathit{dom}(\Delta) = \mathit{dom}(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, and $\Delta(y) = \Gamma(y)$ for $y \in \mathit{dom}(\Gamma)$. We define the *value judgment* relation $\Gamma \triangleright v{:}\sigma$ to be the least relation closed under

$$\Gamma,\, x{:}\sigma \triangleright x{:}\sigma \qquad \Gamma \triangleright \mathbf{true}{:}\mathbf{bool} \qquad \Gamma \triangleright \mathbf{false}{:}\mathbf{bool}.$$

We write $\Gamma \triangleright \widetilde{v}{:}\widetilde{\sigma}$ as an abbreviation for $(\Gamma \triangleright v_1{:}\sigma_1) \wedge \cdots \wedge (\Gamma \triangleright v_n{:}\sigma_n)$.

Figure 3 gives the rules for the relation $\Gamma \triangleright P : A$. We explain key rules below. In rule (T-Out), the first premise $\Gamma \triangleright P : A_2$ implies that the continuation of the output process behaves like A_2, and the second premise $\Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle$ implies that the tuple of values $\widetilde{v}$ being sent may be used by an input process according to $\langle\widetilde{v}/\widetilde{y}\rangle A_1$. Therefore, the whole behavior of the output process is described by $\overline{x}.\,(\langle\widetilde{v}/\widetilde{y}\rangle A_1 \,|\, A_2)$. Note that, as in previous behavioral type systems [3, 8], the resource access and communications made on $\widetilde{v}$ by the receiver of $\widetilde{v}$ are counted as the behavior of the output process. In rule (T-In), the first

[4] Note that the subtyping relation defined here is the converse of the one used in Igarashi and Kobayashi's generic type system [8].

$$\frac{\Gamma \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad \Gamma \triangleright \widetilde{v} : \widetilde{\sigma}}{\Gamma \triangleright \overline{x}_t\langle\widetilde{v}\rangle.\, P : \overline{x}_t.\, (\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)} \quad \text{(T-Out)}$$

$$\frac{\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1}{\Gamma \triangleright x_t(\widetilde{y}).\, P : x_t.\, (A_2{\uparrow_{\{\widetilde{y}\}}})} \quad \text{(T-In)}$$

$$\frac{\Gamma \triangleright P_1 : A_1 \qquad \Gamma \triangleright P_2 : A_2}{\Gamma \triangleright P_1 \mid P_2 : A_1 \mid A_2} \quad \text{(T-Par)} \qquad \Gamma \triangleright \mathbf{0} : \mathbf{0} \quad \text{(T-Zero)}$$

$$\frac{\Gamma \triangleright P : A}{\Gamma \triangleright *P : *A} \quad \text{(T-Rep)} \qquad \frac{\Gamma \triangleright P : A \qquad \Gamma \triangleright x : \mathbf{res}}{\Gamma \triangleright \mathbf{acc}_\xi(x).P : x^\xi.A} \quad \text{(T-Acc)}$$

$$\frac{\Gamma \triangleright v : \mathbf{bool} \qquad \Gamma \triangleright P : A \qquad \Gamma \triangleright Q : A}{\Gamma \triangleright \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : A} \quad \text{(T-If)} \qquad \frac{\Gamma, x : \mathbf{res} \triangleright P : A \qquad \mathbf{etraces}_x(A) \subseteq \Phi}{\Gamma \triangleright (\mathfrak{N}^\Phi x)P : A{\uparrow_{\{x\}}}} \quad \text{(T-NewR)}$$

$$\frac{\Gamma, x : \mathbf{chan}\langle(\widetilde{y} : \widetilde{\sigma})A_1\rangle \triangleright P : A_2}{\Gamma \triangleright (\nu x)\, P : (\nu x)\, A_2} \quad \text{(T-New)} \qquad \frac{\Gamma \triangleright P : A' \qquad A' \leq A}{\Gamma \triangleright P : A} \quad \text{(T-Sub)}$$

Fig. 3. Typing Rules

premise implies that the continuation of the input process behaves like A_2. Following previous behavioral type systems [3, 8], we split A_2 into two parts: $A_2{\downarrow_{\{\widetilde{y}\}}}$ and $A_2{\uparrow_{\{\widetilde{y}\}}}$. The first part describes the behavior on the received values $\widetilde{y}$ and is taken into account in the channel type. The second part describes the resource access and communications performed on other values, and is taken into account in the behavioral type of the input process. The condition $A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1$ requires that the access and communication behavior on $\widetilde{y}$ conforms to A_1, the channel arguments' behavior. In (T-New), the premise implies that P behaves like A, so that $(\nu x)\, P$ behaves like $(\nu x)\, A$. Here, we only require that x is a channel, unlike in the previous behavioral type systems for the π-calculus [8, 10]. That is because we are only interested in the resource access behavior; the communication behavior is used only for accurately inferring the resource access behavior. In (T-NewR), we check that the process's behavior A conforms to the resource usage specification Φ. Rule (T-Sub) allows the type A' of a process to be replaced by its approximation A.

Example 2. Consider the process $P = (\nu s)\, (*s(n, x, r).\, P_1 \mid (\mathfrak{N}^\Phi x)P_2)$, where:

$$P_1 {=} \mathbf{if}\ n = 0\ \mathbf{then}\ \overline{r}\langle\rangle\ \mathbf{else}\ (\nu r')\, (\overline{s}\langle n-1, x, r'\rangle \mid r'_{\mathbf{c}}().\, \mathbf{read}(x).\overline{r}\langle\rangle)$$
$$P_2 {=} (\nu r)\, (\mathbf{init}(x).\overline{s}\langle 100, x, r\rangle \mid r_{\mathbf{c}}().\, \mathbf{close}(x)) \quad \Phi = (IR^*C\downarrow)^\#$$

Let $A_1 = \mu\alpha.(\overline{r} \oplus (\nu r')\, (\langle r'/r\rangle\alpha | r'_{\mathbf{c}}.\, x^R.\overline{r})$ and
let $\Gamma = s{:}\mathbf{chan}\langle(n{:}\mathbf{int},\ x{:}\mathbf{res},\ r{:}\mathbf{chan}\langle\rangle)\, A_1\rangle$. Then

$$\Gamma, n{:}\mathbf{int}, x{:}\mathbf{res}, r{:}\mathbf{chan}\langle\rangle \triangleright P_1 : A_1 \quad \Gamma \triangleright *s(n, x, r).\, P_1 : *s.\, (A_1{\uparrow_{\{n,x,r\}}}) \approx *s$$
$$\Gamma \triangleright P_2 : (\nu r)\, (x^I.A_1 | r_{\mathbf{c}}.\, x^C)$$

So long as $\mathbf{etraces}_x((\nu r)\,(x^I.A_1|r.\,x^C)) \subseteq \Phi$, we obtain $\emptyset \triangleright P : \mathbf{0}$. See Section 4.1 for the algorithm that establishes $\mathbf{etraces}_x(\cdot) \subseteq \Phi$. □

Remark 2. The type A_1 in the example above demonstrates how recursion, hiding, and renaming are used together. In general, in order to type a recursive process of the form $*s(x).\,(\nu y)\,(\cdots \overline{s}\langle y\rangle \cdots)$, we need to find a type that satisfies $(\nu y)\,(\cdots \langle y/x\rangle A \cdots) \;\leq\; A$. Moreover, for the type inference (in Section 4), we must find the *least* such A. Thanks to the type constructors for recursion, hiding, and renaming, we can always do that: A can be expressed by $\mu\alpha.(\nu y)\,(\cdots \langle y/x\rangle \alpha \cdots)$.

The following theorem states that no well-typed process performs an invalid access to a resource.

Theorem 1 (type soundness (safety)). *Suppose that P is safe. If $\Gamma \triangleright P : A$ and $P \longrightarrow^* Q$, then Q is safe.*

Theorem 2 below states that well-typed programs eventually perform all the necessary resource accesses (unless the whole process diverges).

Definition 5 (well-annotatedness) P is *active* if $P \preceq (\widetilde{\nu \mathfrak{N}})(\overline{x}_{\mathbf{c}}\langle\widetilde{v}\rangle.\,Q \mid R)$ or $P \preceq (\widetilde{\nu \mathfrak{N}})(x_{\mathbf{c}}(\widetilde{y}).\,Q \mid R)$. P is *well-annotated* if for any P' such that $P \longrightarrow^* P'$ and $active(P')$, there exists P'' such that $P' \longrightarrow P''$.

Theorem 2. *If $well\_annotated(P)$ and $\emptyset \triangleright P : A$, then P is partially live.*

4 Type Inference Algorithm

This section discusses an algorithm which takes a closed process P as an input and checks whether $\emptyset \triangleright P : \mathbf{0}$ holds. The algorithm consists of the following steps.

1. Extract constraints on type variables based on the (syntax-directed version of) typing rules.
2. Reduce constraints to trace inclusion constraints of the form $\{\mathbf{etraces}_{x_1}(A_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_n}(A_n) \subseteq \Phi_n\}$
3. Decide whether the trace inclusion constraints are satisfied.

The algorithm for Step 3 is sound but not complete.

The first two steps are fairly standard [9, 10]. Based on the typing rules, we can transform $\emptyset \triangleright P : \mathbf{0}$ to equivalent constraints of the form:

$$\{\alpha_1 \;\geq\; A_1, \ldots, \alpha_n \;\geq\; A_n, \mathbf{etraces}_{x_1}(B_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_m}(B_m) \subseteq \Phi_m\}$$

where $\alpha_1, \ldots, \alpha_n$ are different from each other. Each subtype constraint $\alpha \;\geq\; A$ can be replaced by $\alpha \;\geq\; \mu\alpha.A$. Therefore, the above constraints can be further reduced to:

$$\{\mathbf{etraces}_{x_1}([\widetilde{A'}/\widetilde{\alpha}]B_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_m}([\widetilde{A'}/\widetilde{\alpha}]B_m) \subseteq \Phi_m\}$$

Here, $A'_1, \ldots, A'_n$ are the least solutions for the subtype constraints. Thus, we have reduced type checking to the validity of trace inclusion constraints of the form $\mathbf{etraces}_x(A) \subseteq \Phi$.

Example 3. Recall Example 2. We obtain the constraint $\mathbf{etraces}_x(A_1) \subseteq (IR^*C)^\#$ where

$$A_1 = (\nu r)\,(x^I.\overline{s}.\,A_2 \,|\, r.\,x^C) \qquad A_3 \;=\; \mu\alpha_2.\alpha_2$$
$$A_2 = \mu\alpha_1.\overline{r}.\,A_3 \oplus (\nu r')\,(\overline{s}.\,\langle r'/r\rangle\alpha_1 \,|\, r'.\,x^R.\overline{r}.\,A_3){\downarrow_{\{n,x,r\}}}.$$

4.1 Step 3: Constraint Solving

We present an approximate algorithm for checking how to check a trace inclusion constraint $\mathbf{etraces}_x(A) \subseteq \Phi$ when the trace set Φ is a regular language. (Actually, we can extend the algorithm to deal with the case where Φ is a deterministic Petri net language: see the full version [15].)

The algorithm consists of the following three steps.

- Approximate the behavior of $A{\downarrow_{\{x\}}}$ by a (labeled) Petri net $N_{A_1,x}$.
- Construct a Petri net $N_{A_1',x} \,\|\, M_\Phi$ that simultaneously simulates $N_{A_1,x}$ and a minimized deterministic automaton M_Φ that accepts Φ.
- Check that $N_{A_1',x} \,\|\, M_\Phi$ does not reach any invalid state. Here, the set of invalid states consists of (1) states where $N_{A_1,x}$ can make a ξ-transition while M_Φ cannot, and (2) states where $N_{A_1,x}$ is disabled (in other words, can make a $\downarrow$-transition) while M_Φ cannot make a $\downarrow$-transition.

The last part amounts to solving a reachability problem of Petri nets. In the implementation, we further approximate the Petri net by a finite state machine.

We sketch the first step of the algorithm with an example below. Attributes are omitted below for simplicity. Please consult the full version [15] for more details and the other two steps. In Example 3 above, we have reduced the typability of the process to the equivalent constraint $\mathbf{etraces}_x(A_1) \subseteq \Phi$ where $\Phi = (IR^*C\downarrow)^\#$ and

$$A_1{\downarrow_{\{x\}}} \approx (\nu r)\,(x^I.A_2'' \mid r.\,x^C) \qquad A_2'' = \overline{r} \oplus (\nu r')\,(\langle r'/r\rangle A_2'' \mid r'.\,x^R.\overline{r})$$

Here, we have omitted $A_3 = \mu\alpha.\alpha$ since it is insignificant.

Approximate the behavior of $A_1{\downarrow_{\{x\}}}$ by a Petri net [19] $N_{A_1,x}$. This part is similar to the translation of usage expressions into Petri nets in Kobayashi's previous work [10, 11, 14]. Since the behavioral types are more expressive (having recursion, hiding, and renaming), however, we need to approximate the behavior of a behavioral type unlike in the previous work. In this case $A_1{\downarrow_{\{x\}}}$ is infinite. To make it tractable we make a sound approximation A_1' by pushing (ν) to top level, and we eliminate $\langle r'/r\rangle$:

$$A_1' = (\nu r, r')\,(x^I.A_2' \mid r.\,x^C) \quad A_2' = \overline{r} \oplus (A_3' \mid r'.\,x^R.\overline{r}) \quad A_3' = \overline{r'} \oplus (A_3' \mid r'.\,x^R.\overline{r'})$$

Then $N_{A_1',x}$ is as pictured in Figure 4. (Here we treat $A_1 \oplus A_2$ as $\tau.A_1 \oplus \tau.A_2$ for clarity. We also use a version of Petri nets with labeled transitions.) The rectangles are the places of the net, and the dots labeled by τ, x^R, etc. are the transitions of the net. Write i_x for the number of tokens at node B_x. The behavior A_1' corresponds to the initial marking $\{i_1{=}1,\ i_{10}{=}1\}$. We say that the nodes $\widetilde{B}$

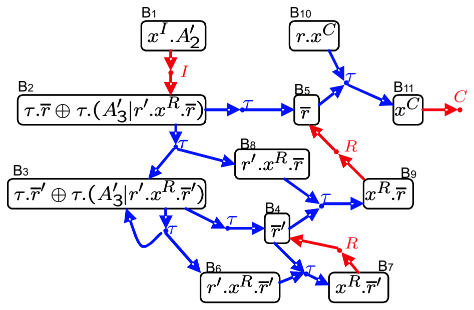

Fig. 4. $N_{A'_1,x}$

together with the restricted names (r, r') constitute a *basis* for A'_1. Note here that $\mathbf{etraces}_x(A_1) \subseteq \mathbf{etraces}_x(A'_1) = \mathbf{ptraces}(N_{A'_1,x})$ where $\mathbf{ptraces}(N_{A'_1,x})$ is the set of traces of the Petri net. Thus, $\mathbf{ptraces}(N_{A'_1,x}) \subseteq \Phi$ is a sufficient condition for $\mathbf{etraces}_x(A_1) \subseteq \Phi$. The key point here is that A'_1 still has infinite states, but all its reachable states can be expressed in the form $(\nu r, r')\,(i_1 B_1 \mid \cdots \mid i_{11} B_{11})$ (where $i_k B_k$ is the parallel composition of i_k copies of B_k), a linear combination of finitely many processes $\widetilde{B}$. That is why we could express A'_1 by the Petri net as above.

5 Implementation

We have implemented a prototype resource usage analyzer based on the type system proposed in this paper. We have tested all the examples given in the present paper. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

The analyzer takes a pi-calculus program as an input, and uses `TyPiCal`[11] to annotate each input or output action with an attribute on whether the action is guaranteed to succeed automatically. The annotated program is then analyzed based on the algorithm described in Section 4.

The followings are some design decisions we made in the current implementation. We restrict the resource usage specification (Φ) to the regular languages although in future we may extend it to deterministic Petri net languages. In the algorithm for checking $\mathbf{etraces}_x(A) \subseteq \Phi$, we blindly approximate A by pushing all of its ν-prefixes to the top-level. In future we might utilize an existing model checker to handle the case where A is already finite. To solve the reachability problems of Petri nets, we approximate the number of tokens in each place by an element of the finite set $\{0, 1, 2, \text{“3 or more”}\}$. That approximation reduces Petri nets to finite state machines, so we can use BDD to compute an approximation of the reachable states.

Figure 5 shows a part of a successful run of the analyzer. The first process (on the second line) of the input program runs a server, which returns a new, initialized resource. We write `!` and `?` for output and input actions. The resource access

Input:

```
new create,s in
  *(create?(r).newR {init(read|write)*close }, x in acc(x,init).r!(x))
| *(new r in create!(r) | r?(y).new c in s!(false,y,c)
          | s!(false,y,c) | c?().c?().acc(y,close))
| *(s?(b,x,r).if b then r!() else acc(x,read).s!(b,x,r))
```

Output:

```
(*** The result of lock-freedom analysis ***)
new create, s in
  *create??(r). newR {init(read|write)*close}, x in acc(x, init). r!!(x)
| *(new r in create!!(r) | r??(y).new c in s!!(false,y,c)
          | s!!(false,y,c) | c??().c??().acc(y,close))
...
No error found
```

Fig. 5. A Sample Run of the Analyzer.

specification is here expressed by a regular expression `init(read|write)*close`. The second process runs infinitely many client processes, each of which sends a request for a new resource, and after receiving it, reads and closes it. The third process (on the 6th line) is a tail-recursive version of the replicated service in Example 2. Here, a boolean is passed as the first argument of s instead of an integer, as the current system is not adapted to handle integers; it does not affect the analysis, since the system ignores the value and simply inspects both branches of the conditional. Note that the program creates infinitely many resources and has infinitely many states. The first output is the annotated version of the input program produced by `TyPiCal`, where `!!` and `??` are an output and an input with the attribute **c**.

6 Related Work

Resource usage analysis and similar analyses have recently been studied extensively, and a variety of methods from type systems to model checking have been proposed [1, 5–7, 9, 16, 20]. However, only a few of them deal with concurrent languages. To our knowledge, none of them deal with the partial liveness property. Nguyen and Rathke [18] propose an effect-type system for a kind of resource usage analysis for functional languages extended with threads and monitors. In their language, neither resources nor monitors can be created dynamically. On the other hand, our target language is π-calculus, so that our type system can be applied to programs that may create infinitely many resources (due to the existence of primitives for dynamic creation of resources: recall the example in Figure 5), and also to programs that use a wide range of communication and synchronization primitives.

Model checking technologies [2, 4, 21, 22] can of course be applicable to concurrent languages, but naive applications of model checking technologies would suffer from the state explosion problem, especially for expressive concurrent languages like π-calculus, where resources and communication channels can be dynamically created and passed around. Actually, our type-based analysis can be considered as a kind of abstract model checking. The behavioral types extracted by (the first two steps of) the type inference algorithm are abstract concurrent programs, each of which captures the access behavior on each resource. Then, conformance of the abstract program with respect to the resource usage specification is checked as a model checking problem. From that perspective, a nice point about our approach is that our type, which describes a resource-wise behavior, has much smaller state space than the whole program. In particular, if infinitely many resources are dynamically created, the whole program has infinite states, but it is often the case that our behavioral types are still finite (indeed so for the example in Figure 5).

Technically, closest to our type system are that of Igarashi and Kobayashi [8] and that of Chaki, Rajamani, and Rehof [3]. Those type systems are developed for checking the communication behavior of a process, but by viewing a set of channels as a resource, it is possible to use those type systems directly for the resource usage analysis. As mentioned in Section 1, the main contributions of the present work with respect to those type systems are realization of automatic verification while keeping enough precision, and verification of the partial liveness. The parameterization of the type system with an arbitrary mechanism to guarantee deadlock-freedom opens a new possibility of integrating type-based techniques with other verification techniques (the current implementation uses another type-based analyzer to infer deadlock-freedom, but one can replace that part with a model checker or an abstract interpreter).

7 Conclusion

We have presented a type-based technique for verifying resource usage of concurrent programs. Future work includes more serious assessment of the effectiveness of our analysis and extensions of the type system to deal with other typical synchronization primitives like join-patterns and external choice.

Acknowledgments

We would like to thank Andrew Gordon, Jakob Rehof, and Eijiro Sumii for useful discussions and comments. We would also like to thank anonymous referees for useful comments and suggestions.

References

1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods 2004*, volume 2999 of *LNCS*, pages 1–20. Springer-Verlag, 2004.

2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of POPL*, pages 1–3, 2002.
3. S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. of POPL*, pages 45–57, 2002.
4. M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.
5. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of PLDI*, pages 59–69, 2001.
6. R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proc. of PLDI*, 2002.
7. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. of PLDI*, pages 1–12, 2002.
8. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
9. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Prog. Lang. Syst.*, 27(2):264–313, 2005. Preliminary summary appeared in Proceedings of POPL 2002.
10. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica.* to appear.
11. N. Kobayashi. TyPiCal: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.
12. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Prog. Lang. Syst.*, 20(2):436–482, 1998.
13. N. Kobayashi. A type system for lock-free processes. *Info. Comput.*, 177:122–159, 2002.
14. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR2000*, volume 1877 of *LNCS*, pages 489–503. Springer-Verlag, August 2000.
15. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the pi-calculus. Full version, 2005. `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/usage-pi.pdf`.
16. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, volume 2895 of *LNCS*, pages 212–229, 2003.
17. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
18. N. Nguyen and J. Rathke. Typed static analysis for concurrent, policy-based, resource access control. draft.
19. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.
20. C. Skalka and S. Smith. History effects and verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 107–128, 2004.
21. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *CAV'94: Computer Aided Verification*, volume 818 of *LNCS*, pages 428–440. Springer-Verlag, 1994.
22. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *Proceedings of VMCAI 2003*, volume 2575 of *LNCS*, pages 116–131. Springer-Verlag, 2003.