Useless-Code Elimination and Program Slicing for the Pi-Calculus

Naoki Kobayashi

Department of Computer Science
Tokyo Institute of Technology
kobayasi@cs.titech.ac.jp

Abstract. In this paper, we study program transformations called *useless-code elimination* and *program slicing* in the context of the π-calculus. The aim of useless-code elimination and program slicing is to simplify a program by eliminating useless or unimportant part of the program. We define formal correctness criteria for those transformations, present a type-based transformation method, and prove its correctness.

1 Introduction

In this paper, we study useless-code elimination and program slicing[1] for the π-calculus [15, 21]. The purpose of *useless-code elimination* [3, 6, 11, 23] (UCE, in short) is to optimize a program by eliminating part of code that does not contribute to the observable program behavior. (It is more aggressive optimization than deadcode elimination in the sense that UCE may eliminate part of code that is executed in the original program, as long as that part does not affect the observable behavior.) *Program slicing* [19, 22] can be considered a generalization of UCE: It aims to eliminate part of code that does not contribute to a certain interesting part of the program behavior. Program slicing is useful for program optimization, debugging, verification, etc. For example, suppose that a concurrent program suffers from a race condition. To find the cause of the race, one may perform program slicing to obtain a simplified program that exhibits the same race condition, so that one can inspect the simplified program instead of the original program. As another example, suppose that one wants to verify that a concurrent program satisfies a certain property. Then, one may perform program slicing to extract a part of the code that is relevant to the property to be verified, and then perform verification.

Both UCE [3, 6, 11, 23] and program slicing [8, 19, 22] have been extensively studied for sequential programs, but they have been relatively unexplored for concurrent programs. There are some pieces of work on program slicing for concurrent programs [9, 14, 24], but the slicing algorithms are too specific to a particular language or a thread model, so that it is not clear whether they can be applied to other concurrent languages. Moreover, only a few of them [9] give

[1] There are several kinds of program slicing [8, 22]: Throughout this paper, we use the term program slicing to refer to the static, backward slicing.

a formal correctness criterion. The aim of this paper is, therefore, to establish formal, general foundations for UCE and program slicing by using the π-calculus as a target of the study. By using the π-calculus, (i) we can reuse standard definitions of process equivalence and type theories for defining correctness criteria of UCE and program slicing and developing transformation algorithms, (ii) we hope that the techniques developed here can be used in a wide range of concurrent programming languages (since the π-calculus is general enough to model various concurrency primitives), and (iii) we hope that we can clarify the relationship between program slicing and information flow analysis for concurrent languages [10, 18], which are both instances of dependency analyses [2] but have been studied independently in the context of concurrent languages.

The main technical contributions of this paper are summarized as follows:

- We give a general, formal definition of the problems of UCE and program slicing for the π-calculus by using the idea of (weak) barbed congruence [21]. The idea of barbed congruence is to compare two processes by putting them into the same context and check whether they can simulate each other. Since correctness criteria of program slicing may vary depending on its purpose, we parameterize the barbed congruence with the set of contexts that processes can be put into, so that various criteria can be obtained by changing the set of contexts.
- We present a *type-based* method for UCE and program slicing, and prove its correctness. Thanks to the use of the type-based technique, we can separate the issue of *what is* a valid transformation and the issue of *how to obtain* a valid transformation: The former can be specified in a declarative manner by using rules similar to typing rules, and the latter can be reduced to the problem of type reconstruction. We can also use a standard proof technique for showing soundness of the transformation. A distinguishing feature of our *type-based* transformation is that it works for *untyped* programs. We enable an application of a type-based transformation to untyped programs by introducing a special type describing untyped values. We think that dealing with untyped programs is important for programs that run in open environments, where we cannot assume that an environment (in particular, a malicious one) respects types. Note that a transformation that is correct in the typed world may be incorrect in the untyped world: For example, although $x \sim y$ (which should be read "x and y are equivalent") holds under the assumption that both x and y have the same singleton type (hence having the same value), $x \sim y$ does not hold in the untyped world. For another example, $\mathbf{if}\ b\ \mathbf{then}\ e\ \mathbf{else}\ e\ \sim e$ in the typed world, but $\mathbf{if}\ b\ \mathbf{then}\ e\ \mathbf{else}\ e\ \not\sim e$ in the untyped world, since b in the lefthand process may not be a boolean, in which case a run-time error is reported. We expect that the idea of our type-based transformation for untyped programs can be used for applying other type-based program analysis and transformation techniques to untyped programs.

The rest of this paper is structured as follows. Section 2 introduces the syntax and operational semantics of the language. Section 3 gives a formal definition of the problems of UCE and program slicing. Section 4 presents a type-based transformation method for UCE and program slicing. Section 5 discusses extensions of our method. Section 6 discusses related work and Section 7 concludes. Proofs and more examples are given in the full version of this paper, available from `http://www.kb.cs.titech.ac.jp/~kobayasi/publications.html`.

2 Language

2.1 Syntax

Definition 1. *The sets of* processes *and* values *are defined by:*

$$\begin{array}{lcl} P \ \textit{(processes)} & ::= & \mathbf{0} \mid v![v_1,\ldots,v_n].\,P \mid v?[y_1,\ldots,y_n].\,P \\ & & \mid (P \mid Q) \mid (\nu x)\,P \mid *P \mid \textbf{if } v \textbf{ then } P \textbf{ else } Q \\ v \ \textit{(values)} & ::= & x \mid \star \mid \textbf{true} \mid \textbf{false} \end{array}$$

Process $\mathbf{0}$ does nothing. Process $v![v_1,\ldots,v_n].\,P$ sends a tuple $[v_1,\ldots,v_n]$ of values on channel v, and behaves like P after the tuple is received. Process $v?[y_1,\ldots,y_n].\,P$ waits to receive a tuple $[v_1,\ldots,v_n]$ of values, binds $y_1,\ldots,y_n$ to $v_1,\ldots,v_n$, and behaves like P. $P \mid Q$ runs P and Q in parallel. Process $(\nu x)\,P$ creates a fresh communication channel, binds x to it, and behaves like P. Process $*P$ runs infinitely many copies of P in parallel. Process **if** v **then** P **else** Q behaves like P if v is **true** and behaves like Q if v is **false**. A communication channel is represented by a variable. The value $\star$ denotes an empty value, corresponding to the null tuple in functional language ML [16] (the element of a singleton type **unit**). In examples, we will use other values such as strings and integers.

We assume that the input prefix $v?[y_1,\ldots,y_n]$ binds variables $y_1,\ldots,y_n$, and the ν-prefix (νx) binds x. The set of free variables (i.e., those that are not bound by any prefix) in P is denoted by $\mathbf{FV}(P)$. We assume that bound variables are implicitly renamed as necessary to ensure that bound variables are different from each other and free variables.

We give a higher precedence to prefixes and conditionals (**if**) than to the parallel composition, so that $(\nu y)\,x?[y].\,P \mid Q$ means $((\nu y)\,(x?[y].\,P)) \mid Q$. A sequence $v_1,\ldots,v_n$ is abbreviated to $\tilde{v}$. $(\nu\tilde{u})\,P$ is an abbreviated form of $(\nu u_1)\cdots(\nu u_n)\,P$. We often abbreviate $x![\tilde{v}].\,\mathbf{0}$ to $x![\tilde{v}]$.

2.2 Operational Semantics

Following the standard reduction semantics [21], we define the operational semantics by using a structural relation and a reduction relation. Unlike in the standard semantics, however, we do not require the structural relation to be a congruence relation, for the sake of technical convenience (see Remark 1 below).

Definition 2. *A* structural preorder $\preceq$ *is the least reflexive and transitive relation relation on processes closed under the rules in Figure 1. A* reduction relation $\longrightarrow$ *is the least relation closed under the rules in Figure 1.*

$$P \mid Q \preceq Q \mid P \quad \text{(SP-Commut)}$$

$$*P \preceq *P \mid P \quad \text{(SP-Rep)}$$

$$P \mid (Q \mid R) \preceq (P \mid Q) \mid R \quad \text{(SP-Assoc)}$$

$$\frac{P \preceq P' \qquad Q \preceq Q'}{P \mid Q \preceq P' \mid Q'} \quad \text{(SP-Par)}$$

$$\frac{x \text{ is not free in } Q}{(\nu x)\, P \mid Q \preceq (\nu x)\,(P \mid Q)} \quad \text{(SP-New)}$$

$$\frac{P \preceq Q}{(\nu x)\, P \preceq (\nu x)\, Q} \quad \text{(SP-CNew)}$$

$$x![v_1, \ldots, v_n].\, P \mid x?[z_1, \ldots, z_n].\, Q \longrightarrow P \mid [v_1/z_1, \ldots, v_n/z_n]Q \quad \text{(R-Com)}$$

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \quad \text{(R-Par)}$$

$$\textbf{if true then } P \textbf{ else } Q \longrightarrow P \quad \text{(R-IfT)}$$

$$\frac{P \longrightarrow Q}{(\nu x)\, P \longrightarrow (\nu x)\, Q} \quad \text{(R-New)}$$

$$\textbf{if false then } P \textbf{ else } Q \longrightarrow Q \quad \text{(R-IfF)}$$

$$\frac{P \preceq P' \qquad P' \longrightarrow Q' \qquad Q' \preceq Q}{P \longrightarrow Q} \quad \text{(R-SP)}$$

Fig. 1. Structural Preorder and Reduction Relation

Intuitively, $P \longrightarrow Q$ means that P is reduced to Q in one step (by a communication or reduction of a conditional expression).

Remark 1. If the relation $\preceq$ were symmetric, allowing $*P \mid P \preceq *P$, then Theorem 2, one of the main theorems about the correctness of the transformation given later, would not hold.

Example 1. Let *CPoint* be the following process ("_" denotes an unused bound variable):

$$\begin{aligned}&(\nu\mathit{state})\,(\nu\mathit{cstate})\,(\mathit{state}![1,2] \mid \mathit{cstate}![\text{“}\mathit{black}\text{”}]\\ &\quad \mid *\mathit{set}?[x,y,c,r].\,\mathit{state}?[\_,\_].\,\mathit{cstate}?[\_].\,(\mathit{state}![x,y] \mid \mathit{cstate}![c] \mid r![\,])\\ &\quad \mid *\mathit{getxy}?[r].\,(\mathit{state}?[x,y].\,(\mathit{state}![x,y] \mid r![x,y]))\\ &\quad \mid *\mathit{getc}?[r].\,(\mathit{cstate}?[c].\,(\mathit{cstate}![c] \mid r![c])))\end{aligned}$$

CPoint works as a colored point object that provides three kinds of services (called *methods* below) along channels *set*, *getxy*, and *getc*. The channel *state* is used to store the current coordinate, and the channel *cstate* is used to store the current color. The processes $*\mathit{set}?[x,y,c].\,\cdots$, $*\mathit{getxy}?[r].\,\cdots$, and $*\mathit{getc}?[r].\,\cdots$ implement methods *set* (for updating the current state), *getxy* (for reading the current coordinate), and *getc* (for reading the current color) respectively. If *CPoint* receives a tuple of values $[x,y,c,r]$ along *set*, it invalidates the current coordinate and color (by $\mathit{state}?[\_,\_].\,\mathit{cstate}?[\_].$), sets its current coordinate and color to $[x,y]$ and c (by $\mathit{state}![x,y] \mid \mathit{cstate}![c]$), and sends a message to r to confirm that the update has completed. If *CPoint* receives a channel r along *getxy*, then it reads the current coordinate, and sends it to r.

3 Useless-Code Elimination and Program Slicing

In this section, we define the problems of UCE and program slicing.

A *context* is a term obtained by replacing a sub-process of a process with $\langle\,\rangle$. We write $\mathcal{C}$ for the set of all contexts. We write $C\langle P\rangle$ for the process obtained by replacing $\langle\,\rangle$ in C with P.

We say that a process Q is a *slice* of P, written $P \gg Q$, if Q is a process obtained by removing or replacing some sub-processes of P with $\mathbf{0}$, and replacing some values with $\star$. The formal definition is given below.

Definition 3 (slice). *A binary relation $P \gg Q$ on processes and a binary relation $v \gg_V v'$ on values are the least reflexive and transitive relations closed under the following rules:*

$$v \gg_V \star \qquad\qquad P \gg \mathbf{0} \qquad\qquad C\langle P\rangle \gg P$$

$$\frac{v \gg_V v' \qquad \forall i \in \{1,\dots,n\}.v_i \gg_V v_i'}{v![v_1,\dots,v_n].\,P \gg v'![v_1',\dots,v_n'].\,P} \qquad \frac{v \gg_V v'}{v?[y_1,\dots,y_n].\,P \gg v'?[y_1,\dots,y_n].\,P}$$

$$\frac{v \gg_V v'}{\mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ \gg \mathbf{if}\ v'\ \mathbf{then}\ P\ \mathbf{else}\ Q}$$

$$\frac{P \gg Q}{C\langle P\rangle \gg C\langle Q\rangle}$$

The rule $P \gg \mathbf{0}$ allows elimination a sub-process, while $C\langle P\rangle \gg P$ allows elimination of a context. For example, the latter allows $x?[\,].\,y![\,]$ to be replaced by $y![\,]$.

The purpose of UCE and program slicing is to obtain slices that preserve a certain behavior of the original process. To define what behavior should be preserved, we adapt the barbed congruence [21]. The idea of the barbed congruence is to compare two processes by putting them into various contexts, and check whether they exhibit the same observational behavior. The contextual definition of the barbed congruence allows us to express slicing criteria in terms of the set of contexts in which processes are compared. We could instead use testing equivalence [17], another contextual definition of process equivalence. We have chosen the barbed congruence since it is stronger than the testing equivalence and also since the co-inductive definition of the equivalence is convenient for proving the correctness of the type-based transformation given in Section 4.

The usual definition of barbed congruence uses as observables (which are called *barbs* in the terminology of barbed congruence) information about on which channel a process may perform input or output. Since we are working on an untyped calculus, we also add a type error as an observable. We first define a predicate $P \Uparrow$, which means that a type error has occurred in P, and then define the set of barbs (observables) $Barbs(P)$.

Definition 4. *We write $P \Uparrow$ if one of the following conditions holds:*

– $P \preceq (\nu\tilde{u})\,(\mathbf{if}\ v\ \mathbf{then}\ Q_1\ \mathbf{else}\ Q_2\ \mid R)$ *with* $v \notin \{\mathbf{true}, \mathbf{false}\}$
– $P \preceq (\nu\tilde{u})\,(v![\tilde{w}].\,Q \mid R)$ *with* $v \in \{\mathbf{true}, \mathbf{false}, \star\}$
– $P \preceq (\nu\tilde{u})\,(v?[\tilde{y}].\,Q \mid R)$ *with* $v \in \{\mathbf{true}, \mathbf{false}, \star\}$.
– $P \preceq (\nu\tilde{u})\,(x![v_1, \ldots, v_m].\,P \mid x?[y_1, \ldots, y_n].\,Q \mid R)$ *with* $m \neq n$.

Definition 5. *The (weak) barbs of P, written $Barbs(P)$, is defined by:*

$$\begin{aligned} Barbs(P) = &\{x^! \mid P \longrightarrow^* \preceq (\nu\tilde{u})\,(x![v_1, \ldots, v_n].\,Q \mid R) \wedge x \notin \{\tilde{u}\}\} \\ &\cup \{x^? \mid P \longrightarrow^* \preceq (\nu\tilde{u})\,(x?[y_1, \ldots, y_n].\,Q \mid R) \wedge x \notin \{\tilde{u}\}\} \\ &\cup \{\mathbf{error} \mid P \longrightarrow^* Q \wedge Q \Uparrow\} \end{aligned}$$

Now we define process equivalence relations. Intuitively, processes P and Q are weakly barbed bisimilar if they can simulate each other. P and Q are weakly barbed congruent if they are bisimilar in any context. In the case of program slicing, we may not be interested in running processes P and Q in arbitrary contexts. We therefore introduce the notion of barbed S-congruence, which uses only contexts in S to compare two processes.

Definition 6 (barbed bisimulation, congruence). *A binary relation $\mathcal{R}$ on processes is a weak barbed bisimulation if the following conditions hold for any $(P, Q) \in \mathcal{R}$.*

1. *If $P \longrightarrow P'$, then there exists Q' such that $Q \longrightarrow^* Q'$ and $(P', Q') \in \mathcal{R}$.*
2. *If $Q \longrightarrow Q'$, then there exists P' such that $P \longrightarrow^* P'$ and $(P', Q') \in \mathcal{R}$.*
3. *$Barbs(P) = Barbs(Q)$.*

P and Q are weakly barbed bisimilar, written $P \overset{\bullet}{\approx} Q$, if $(P, Q) \in \mathcal{R}$ holds for some weak barbed bisimulation $\mathcal{R}$. Let S be a set of contexts. P and Q are weakly barbed S-congruent, written $P \approx_S Q$, if for any context $C \in S$, $C\langle P\rangle \overset{\bullet}{\approx} C\langle Q\rangle$ holds.

We can now define the criteria of UCE and program slicing.

Definition 7 (proper slice, program slicing, UCE). *Let S be a set of contexts. Q is a* proper S-slice *of P if $P \gg Q$ and $P \approx_S Q$. The problem of* program slicing *is, given a process P and a set S of contexts, to find a proper S-slice of P. The problem of* useless-code elimination *is, given a process P, to find a proper $\mathcal{C}$-slice of P.*

As defined above, the problem of UCE can be considered an instance of the program slicing problem. Conversely, a certain class of program slicing problems can be reduced to the problem of UCE.

– If one is interested in only the communication behavior of P on a certain set $\{\tilde{x}\}$ of channels, then one can perform program slicing by choosing the set of contexts: $\{C\langle(\nu\tilde{u})\,\langle\,\rangle\rangle \mid C \in \mathcal{C}\}$, where $\{\tilde{u}\} = \mathbf{FV}(P)\backslash\{\tilde{x}\}$. Alternatively, one can first modify a given process P into $(\nu\tilde{u})\,P$, perform UCE, and then remove the ν-prefixes $(\nu\tilde{u})$.

– If one is interested in the first argument of messages sent on x by a given process but not in the second argument, one can achieve program slicing by choosing the set of contexts: $\{C\langle(\nu x)\,(\langle\,\rangle \,|\, x?[y,z].\,w![y])\rangle \mid C \in \mathcal{C}\}$. Alternatively, one can first modify a given process P into $(\nu x)\,(P \,|\, x?[y,z].\,w![y])$, perform UCE, and then remove the context $(\nu x)\,(\langle\,\rangle \,|\, x?[y,z].\,w![y])$.

Some of the standard program slicing problems do not directly match the criterion given in Definition 7, but we can pre-process an input program so that those problems match our slicing criterion.

– Program slicing is often specified by a set of interesting program points, so that the behavior in those program points is preserved. Let us assume that a label is attached to each interesting program point, like $x?^{l}[y].\,P$. Then, before applying program slicing, we can transform $x?^{l}[y].\,P$ into $x?[y].\,(l![\,] \,|\, P)$. In this way, we can ensure that interesting program points are executed in the original program if and only if they are executed in a sliced program.
– If we want to ensure also that states are preserved at a set of interesting program points, we can pre-process a given program to replace a process of the form $x?^{l}[y].\,P$ with $x?[y].\,(l![\tilde{u}] \,|\, P)$, before performing program slicing. Here, $\{\tilde{u}\}$ is the set of live variables at l. This criterion roughly corresponds to the correctness criterion that Hatcliff et al. [9] used for slicing multi-threaded Java programs.

Example 2. Recall the process *CPoint* in Example 1. Suppose that every client is interested only in services *set* and *getxy* (in other words, no client accesses channel *getc*). Then, we let a set S of contexts be $\{C\langle(\nu getc)\,\langle\,\rangle\rangle \mid C \in \mathcal{C}\}$. The following process is a proper S-slice of *CPoint*:

$$\begin{array}{l}(\nu state)\,(state![1,2]\\ \qquad |\, *set?[x,y,\_,r].\,state?[\_,\_].\,(state![x,y] \,|\, r![\,])\\ \qquad |\, *getxy?[r].\,(state?[x,y].\,(state![x,y] \,|\, r![x,y])))\end{array}$$

In addition to the process $*getc?[r].\,\cdots$, communications on channel *cstate* has been removed above. How to perform this kind of optimization is discussed in the rest of this paper.

4 Type-Based Useless-Code Elimination

In this section, we present a type-based method for UCE. As discussed in Section 3, a certain class of program slicing problems can be turned into the problem of UCE, so that the method presented here can also be used for program slicing.

4.1 Basic Ideas

An idea of our type-based UCE comes from our earlier work on type-based UCE for a typed λ-calculus [11]. The aim of our UCE for the λ-calculus [11] was to replace sub-terms with the empty value $\star$, so that the value of the whole expression does not change. This can be reduced to the problem of replacing as many sub-terms as possible with $\star$ so that the resulting term has the same type as the original term. This is intuitively understood as follows: The only possibility that the evaluation result is changed by the replacement is that the replaced sub-terms are actually used during evaluation, so that the replacement of them with $\star$ introduces a run-time error. If the resulting term is well-typed, however, such a run-time error cannot happen.

The idea described above can be used in UCE for the π-calculus for replacing value expressions with $\star$. In addition, we need to address the following issues to deal with the (untyped) π-calculus:

- What is a sufficient criterion for replacing sub-processes with $\mathbf{0}$? Unlike in the case of the λ-calculus, preservation of the well-typedness is not a sufficient condition. For example, if x is a channel that can be accessed by a context, replacing $x![\,]$ with $\mathbf{0}$ obviously changes the observable behavior.
- How to deal with untyped processes: Our previous type-based UCE for the λ-calculus [11] was for typed programs and it was only proved that the effect of the transformation cannot be observed by any *well-typed* context. As argued in Section 1, to optimize programs that are executed in open environments, one cannot assume that the environment respects typing.

For the first issue above, We distinguish between the type of processes that contain interesting communications and the type of processes that do not, and we enforce by transformation rules that if P contains interesting communications, input or output operations that guard execution of P are also considered interesting communications. For the second issue, following type systems for reasoning about security properties [1, 4, 7], we introduce a type **Un** describing untyped values, so that any process can be typed.

4.2 Types

Definition 8 (types). *The sets of* types *and* process types *are defined by:*

$$\begin{aligned} &\tau \ \textit{(types)} ::= \mathbf{Un} \mid \mathbf{bool} \mid [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \mid \mathbf{unit} \\ &I \ \textit{(process types)} ::= \mathbf{N} \mid \mathbf{I} \end{aligned}$$

Type **Un** describes untyped (or untrusted) values. For example, values provided by an unknown environment should be given this type. Type **bool** is the type of booleans, and $[\tau_1, \ldots, \tau_n]$ **chan** is the type of channels that are used for transmitting a tuple of values of types $\tau_1, \ldots, \tau_n$.

The process type **I** describes a process that is informative (may contain important communications) and **N** describes a process that is *non-informative* (contains no important communications).

Definition 9 (subtyping relation). *The subtyping relation $\tau_1 \leq \tau_2$ is the least reflexive relation satisfying the following laws:*

$$\mathbf{unit} \leq \mathbf{Un} \qquad \mathbf{bool} \leq \mathbf{Un} \qquad [\mathbf{Un}, \ldots, \mathbf{Un}]\ \mathbf{chan} \leq \mathbf{Un}$$

The binary relation $I_1 \leq I_2$ on process types is the least reflexive relation satisfying the law: $\mathbf{N} \leq \mathbf{I}$.

A *type environment* is a mapping form a finite set of variables to types. We use meta-variables Γ and Δ to denote type environments. We write $\emptyset$ for the typing environment whose domain is empty, and write $dom(\Gamma)$ for the domain of Γ. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment obtained by extending the type environment Γ with the binding of x to τ. We abbreviate $(\cdots((\emptyset, x_1 : \tau_1), x_2 : \tau_2), \cdots), x_n : \tau_n$ to $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$.

4.3 Transformation Rules

We introduce a relation $\Gamma \vdash P : I \Longrightarrow P' \dashv \Gamma'$, which means that process P that is well-typed under Γ is transformed into process P that is well-typed under Γ'. We call Γ an *input type environment*, and Γ' an *output type environment*. For example, using the input type environment $x : [\mathbf{bool}]\ \mathbf{chan}$, we can transform the process $x![\mathbf{true}]$ in one of the following ways.

$$\begin{array}{l} x : [\mathbf{bool}]\ \mathbf{chan} \vdash x![\mathbf{true}] : \mathbf{I} \Longrightarrow x![\mathbf{true}] \dashv x : [\mathbf{bool}]\ \mathbf{chan} \\ x : [\mathbf{bool}]\ \mathbf{chan} \vdash x![\mathbf{true}] : \mathbf{I} \Longrightarrow x![\star] \dashv x : [\mathbf{unit}]\ \mathbf{chan} \\ x : [\mathbf{bool}]\ \mathbf{chan} \vdash x![\mathbf{true}] : \mathbf{N} \Longrightarrow \mathbf{0} \dashv x : \mathbf{unit} \end{array}$$

In the first transformation, x remains to be a channel for sending booleans, so that the process remains unchanged. In the second transformation, x is changed into a channel for sending an empty value, and the value **true** is replaced by $\star$. In the third transformation, the output on x is judged to be unimportant, so that the type of x is replaced by **unit** and the whole process is replaced by $\mathbf{0}$. It depends on the output type environment which of the above transformations should be chosen.

The relation $\Gamma \vdash P : I \Longrightarrow Q \dashv \Delta$ is defined as the least relation closed under the transformation rules given in Figure 3. In the rules, we use an auxiliary relation $\Gamma \vdash v : \tau \Longrightarrow v' : \tau' \dashv \Gamma'$ given in Figure 2. It means that the value v that has type τ under Γ is transformed into the value v' that has type τ' under Γ'. When performing UCE, we are particularly interested in transformations of the form:

$$x_1 : \mathbf{Un}, \cdots, x_n : \mathbf{Un} \vdash P : I \Longrightarrow Q \dashv x_1 : \mathbf{Un}, \cdots, x_n : \mathbf{Un}.$$

Variables $x_1, \ldots, x_n$ are used as an interface with the environment. Since we cannot generally assume that the environment behaves in a well-typed manner, we transform a given process on the assumption that those variables have type **Un**. As is clear from the transformation rules, however, type environments other than $x_1 : \mathbf{Un}, \cdots, x_n : \mathbf{Un}$ may appear in intermediate steps of derivation.

We now explain transformation rules. The rules in Figure 2 are used for transformation of values. There are two kinds of rules: those whose name end with ELIM and the other rules. The former rules are used for replacing values with $\star$. The type of the resulting term is given type **unit**. The latter rules are used for preserving the value. The condition $\tau \neq \mathbf{unit}$ in TRV-VAR ensures that a variable whose type is changed into **unit** does not appear in the result of transformation.

The rules in Figure 3 are used for transformation of processes. Again, there are two kinds of rules: those whose names end with ELIM and the other rules. The former class of rules are used for replacing the process with $\mathbf{0}$, while the latter class of rules are used for preserving the outermost structure of the process. In the former class of rules (TR-OUTELIM, TR-INELIM, and TR-IFELIM), it is ensured that all the sub-processes have type $\mathbf{N}$ (so that no important process is eliminated) and that a value being accessed has an appropriate type (so that no process that may cause a run-time error is eliminated). The second condition is, for example, ensured by the condition $\Gamma(x) = [\tau_1, \ldots, \tau_n]\ \mathbf{chan}$ in TR-OUTELIM. It ensures that x must be a communication channel.

In the latter class of rules, sub-processes are transformed using the same input and output type environments: For example, in TR-PAR, sub-processes P and Q are transformed using the same input type environment Γ and the same output type environment Γ'. This ensures that a process is transformed in a consistent manner: For example, there is no such case that x is judged to be important in one sub-process while it is judged to be unimportant and removed in another sub-process.

Note that the condition $[\tau_1, \ldots, \tau_n]\ \mathbf{chan} \leq \tau$ in TR-OUT allows τ to be type $\mathbf{Un}$ (and in this case, $\tau_1, \ldots, \tau_n$ must be $\mathbf{Un}$). Thus, TR-OUT can be used to transform an ill-typed process like $\mathbf{true}![\mathbf{false}]$. In fact, we can derive $\emptyset \vdash \mathbf{true}![\mathbf{false}] : \mathbf{I} \Longrightarrow \mathbf{true}![\mathbf{false}] \dashv \emptyset$ from $\emptyset \vdash \mathbf{true} : \mathbf{Un} \Longrightarrow \mathbf{true} : \mathbf{Un} \dashv \emptyset$ and $\emptyset \vdash \mathbf{false} : \mathbf{Un} \Longrightarrow \mathbf{false} : \mathbf{Un} \dashv \emptyset$.

Remark 2. We did not include rules for eliminating (νx), $*$, and parallel composition, so that the output of transformation may contain redundant occurrences of those constructors. We leave elimination of such redundancy to a post-processing phase. For example, $(\nu x)\,(x![\,]\,|\,x?[\,].\,\mathbf{0})$ is transformed into $(\nu x)\,(\mathbf{0}\,|\,\mathbf{0})$. We can then clean up this code to obtain $\mathbf{0}$ in a post-processing phase.

Example 3. The process $(\nu x)\,(x![\mathbf{true}, \mathbf{false}]\,|\,x?[z, w].\,y![z])$ can be transformed as follows.

$$\begin{array}{l} y : \mathbf{Un} \vdash (\nu x)\,(x![\mathbf{true}, \mathbf{false}]\,|\,x?[z, w].\,y![z]) : \mathbf{I} \Longrightarrow \\ \qquad\qquad (\nu x)\,(x![\mathbf{true}, \star]\,|\,x?[z, w].\,y![z]) \dashv y : \mathbf{Un} \end{array}$$

Example 4. Let us consider the following process:

$$y?[b, r].\,(\nu x)\,(x?[\,].\,\mathbf{0}\,|\,r![\,]\,|\,\mathbf{if}\ b\ \mathbf{then}\ x![\,]\ \mathbf{else}\ \mathbf{0}\ )$$

Let $\Gamma = y : \mathbf{Un}, b : \mathbf{Un}, r : \mathbf{Un}, x : [\,]\ \mathbf{chan}$ and $\Delta = y : \mathbf{Un}, b : \mathbf{Un}, r : \mathbf{Un}, x : \mathbf{unit}$. Then, sub-processes are transformed as follows.

$$\begin{array}{l} \Gamma \vdash x?[\,].\,\mathbf{0} : \mathbf{N} \Longrightarrow \mathbf{0} \dashv \Delta \\ \Gamma \vdash r![\,] : \mathbf{I} \Longrightarrow r![\,] \dashv \Delta \\ \Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ x![\,]\ \mathbf{else}\ \mathbf{0}\ : \mathbf{I} \Longrightarrow \mathbf{if}\ b\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ \mathbf{0}\ \dashv \Delta \end{array}$$

$$\frac{\mathbf{bool} \leq \tau \qquad b \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash b : \tau \Longrightarrow b : \tau \dashv \Gamma'} \quad (\text{TrV-Bool})$$

$$\frac{\mathbf{bool} \leq \tau \qquad b \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash b : \tau \Longrightarrow \star : \mathbf{unit} \dashv \Gamma'} \quad (\text{TrV-BoolElim})$$

$$\frac{\mathbf{unit} \leq \tau' \leq \tau}{\Gamma \vdash \star : \tau \Longrightarrow \star : \tau' \dashv \Gamma'} \quad (\text{TrV-Unit})$$

$$\frac{\tau \neq \mathbf{unit}}{\Gamma, x : \tau \vdash x : \tau \Longrightarrow x : \tau' \dashv \Gamma', x : \tau'} \quad (\text{TrV-Var})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \Longrightarrow \star : \mathbf{unit} \dashv \Gamma'} \quad (\text{TrV-VarElim})$$

Fig. 2. Transformation Rules for Values

Using Tr-Par, Tr-New, and Tr-In, we obtain:

$$\begin{aligned} y : \mathbf{Un} \vdash y?[b, r].\,(\nu x)\,(x?[\,].\,\mathbf{0} \mid r![\,] \mid \mathbf{if}\ b\ \mathbf{then}\ x![\,]\ \mathbf{else}\ \mathbf{0}\ ) : \mathbf{I} \Longrightarrow \\ y?[b, r].\,(\nu x)\,(\mathbf{0} \mid r![\,] \mid \mathbf{if}\ b\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ \mathbf{0}\ ) \dashv y : \mathbf{Un} \end{aligned}$$

Note that **if** b **then** $x![\,]$ **else 0** cannot be replaced by **0**: Since y has type **Un**, a non-boolean value may be assigned to b, so that replacing **if** b **then** $x![\,]$ **else 0** with **0** changes the observable behavior. If it is guaranteed that y is used according to type $[\mathbf{bool}, [\,]\ \mathbf{chan}]\ \mathbf{chan}$ by the environment, then the if-expression can be removed as follows:

$$\begin{aligned} y : [\mathbf{bool}, [\,]\ \mathbf{chan}]\ \mathbf{chan} \vdash y?[b, r].\,(\nu x)\,(x?[\,].\,\mathbf{0} \mid r![\,] \mid \mathbf{if}\ b\ \mathbf{then}\ x![\,]\ \mathbf{else}\ \mathbf{0}\ ) : \mathbf{I} \\ \Longrightarrow y?[b, r].\,(\nu x)\,(\mathbf{0} \mid r![\,] \mid \mathbf{0}) \dashv y : [\mathbf{bool}, [\,]\ \mathbf{chan}]\ \mathbf{chan} \end{aligned}$$

Example 5. Let us consider the following, simplified version of *CPoint* in Example 1.

$$\begin{aligned} &(\nu \mathit{state})\,(\nu \mathit{cstate})\,(\mathit{state}![1, 2] \mid \mathit{cstate}![\text{“}\mathit{black}\text{”}] \\ &\quad \mid *\mathit{set}?[x, y, c].\,(\mathit{state}?[\_, \_].\,\mathit{state}![x, y] \mid \mathit{cstate}?[c'].\,\mathit{cstate}![c]) \\ &\quad \mid *\mathit{getxy}?[r].\,(\mathit{state}?[x, y].\,(\mathit{state}![x, y] \mid r![x, y])) \\ &\quad \mid *\mathit{getc}?[r].\,(\mathit{cstate}?[c].\,(\mathit{cstate}![c] \mid r![c]))) \end{aligned}$$

Here, we have omitted a reply of the method *set*: How to deal with the original version of *CPoint* is discussed in Section 5.

The above process is transformed into:

$$\begin{aligned} &(\nu \mathit{state})\,(\nu \mathit{cstate})\,(\mathit{state}![1, 2] \mid \mathbf{0} \\ &\quad \mid *\mathit{set}?[x, y, \_].\,(\mathit{state}?[x', y'].\,\mathit{state}![x, y] \mid \mathbf{0}) \\ &\quad \mid *\mathit{getxy}?[r].\,(\mathit{state}?[x, y].\,(\mathit{state}![x, y] \mid r![x, y])) \mid \mathbf{0}) \end{aligned}$$

under the input type environment $\mathit{set} : \mathbf{Un}, \mathit{getxy} : \mathbf{Un}, \mathit{getc} : [[\,]\ \mathbf{chan}]\ \mathbf{chan}$ and the output type environment $\mathit{set} : \mathbf{Un}, \mathit{getxy} : \mathbf{Un}, \mathit{getc} : \mathbf{unit}$. If every client is

$$\Gamma \vdash \mathbf{0} : I \Longrightarrow \mathbf{0} \dashv \Gamma' \qquad \text{(Tr-Zero)}$$

$$\frac{\Gamma \vdash P : I_1 \Longrightarrow P' \dashv \Gamma' \qquad \Gamma \vdash Q : I_2 \Longrightarrow Q' \dashv \Gamma' \qquad I_1, I_2 \leq I}{\Gamma \vdash P \,|\, Q : I \Longrightarrow P' \,|\, Q' \dashv \Gamma'} \qquad \text{(Tr-Par)}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : I \Longrightarrow P' \dashv \Gamma' \qquad \Gamma \vdash v : \tau \Longrightarrow v' : \tau' \dashv \Gamma' \\ [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \leq \tau \qquad [\tau'_1, \ldots, \tau'_n]\ \mathbf{chan} \leq \tau' \\ \Gamma \vdash v_i : \tau_i \Longrightarrow v'_i : \tau'_i \dashv \Gamma' \ \text{(for each } i \in \{1, \ldots, n\})\end{array}}{\Gamma \vdash v![v_1, \ldots, v_n].\, P : \mathbf{I} \Longrightarrow v'![v'_1, \ldots, v'_n].\, P' \dashv \Gamma'} \qquad \text{(Tr-Out)}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathbf{N} \Longrightarrow P' \dashv \Gamma' \\ \Gamma(x) = [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \qquad \Gamma'(x) = \mathbf{unit} \\ \Gamma \vdash v_i : \tau_i \Longrightarrow \star : \mathbf{unit} \dashv \Gamma' \ \text{(for each } i \in \{1, \ldots, n\})\end{array}}{\Gamma \vdash x![v_1, \ldots, v_n].\, P : \mathbf{N} \Longrightarrow \mathbf{0} \dashv \Gamma'} \qquad \text{(Tr-OutElim)}$$

$$\frac{\begin{array}{c}\Gamma, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash P : I \Longrightarrow P' \dashv \Gamma', y_1 : \tau'_1, \ldots, y_n : \tau'_n \\ \Gamma \vdash v : \tau \Longrightarrow v' : \tau' \dashv \Gamma' \\ [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \leq \tau \qquad [\tau'_1, \ldots, \tau'_n]\ \mathbf{chan} \leq \tau'\end{array}}{\Gamma \vdash v?[y_1, \ldots, y_n].\, P : \mathbf{I} \Longrightarrow v'?[y_1, \ldots, y_n].\, P' \dashv \Gamma'} \qquad \text{(Tr-In)}$$

$$\frac{\begin{array}{c}\Gamma, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash P : \mathbf{N} \Longrightarrow P' \dashv \Gamma', y_1 : \mathbf{unit}, \ldots, y_n : \mathbf{unit} \\ \Gamma(x) = [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \qquad \Gamma'(x) = \mathbf{unit}\end{array}}{\Gamma \vdash x?[y_1, \ldots, y_n].\, P : \mathbf{N} \Longrightarrow \mathbf{0} \dashv \Gamma'} \qquad \text{(Tr-InElim)}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau \vdash P : I \Longrightarrow P' \dashv \Gamma', x : \tau' \\ [\tau_1, \ldots, \tau_n]\ \mathbf{chan} \leq \tau\end{array}}{\Gamma \vdash (\nu x)\, P : I \Longrightarrow (\nu x)\, P' \dashv \Gamma'} \qquad \text{(Tr-New)}$$

$$\frac{\Gamma \vdash P : I \Longrightarrow P' \dashv \Gamma'}{\Gamma \vdash *P : I \Longrightarrow *P' \dashv \Gamma'} \qquad \text{(Tr-Rep)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \tau \Longrightarrow v' : \tau' \dashv \Gamma' \qquad \mathbf{bool} \leq \tau, \tau' \\ \Gamma \vdash P : I_1 \Longrightarrow P' \dashv \Gamma' \qquad \Gamma \vdash Q : I_2 \Longrightarrow Q' \dashv \Gamma'\end{array}}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : \mathbf{I} \Longrightarrow \mathbf{if}\ v'\ \mathbf{then}\ P'\ \mathbf{else}\ Q'\ \dashv \Gamma'} \qquad \text{(Tr-If)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathbf{bool} \Longrightarrow v' : \tau' \dashv \Gamma' \\ \Gamma \vdash P : \mathbf{N} \Longrightarrow P' \dashv \Gamma' \qquad \Gamma \vdash Q : \mathbf{N} \Longrightarrow Q' \dashv \Gamma'\end{array}}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : \mathbf{N} \Longrightarrow \mathbf{0} \dashv \Gamma'} \qquad \text{(Tr-IfElim)}$$

Fig. 3. Transformation Rules for Processes

interested only in the first component of the return value of the *getxy* method, the above process can be further transformed into

$$
\begin{aligned}
&(\nu state)\,(\nu cstate)\,(state![1,\star]\,|\,\mathbf{0}\\
&\quad |\,{*set?}[x,\_,\_].\,(state?[x',\_].\,state![x,\star]\,|\,\mathbf{0})\\
&\quad |\,{*getxy?}[r].\,(state?[x,\_].\,(state![x,\star]\,|\,r![x,\star]))\\
&\quad |\,\mathbf{0})
\end{aligned}
$$

by using the input type environment $set\!:\!\mathbf{Un}, getxy\!:\![[\mathbf{Un},\mathbf{int}]\ \mathbf{chan}]\ \mathbf{chan}, getc\!:\![[\,]\ \mathbf{chan}]\ \mathbf{chan}$ and the output type environment:
$set : \mathbf{Un}, getxy : [[\mathbf{Un},\mathbf{unit}]\ \mathbf{chan}]\ \mathbf{chan}, getc : [[\,]\ \mathbf{chan}]\ \mathbf{chan}$.

4.4 Soundness of Transformation

The soundness of our type-based transformation is stated as follows.

Theorem 1. *Suppose that $\Gamma(x) = \mathbf{Un}$ for every $x \in dom(\Gamma)$. If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Gamma$, then Q is a proper $\mathcal{C}$-slice of P.*

We give a proof sketch below. A full proof is given in the full version of this paper. We first introduce a relation $P \equiv_0 Q$, which means intuitively that P and Q are equal except for garbage processes $\mathbf{0}$.

Definition 10. *A binary relation $\equiv_0$ on processes is the least congruence relation that satisfies the following rules:*

$$
P\,|\,\mathbf{0} \equiv_0 P \qquad\qquad *\mathbf{0} \equiv_0 \mathbf{0} \qquad\qquad (\nu x)\,\mathbf{0} \equiv_0 \mathbf{0} \qquad\qquad \frac{P \preceq Q}{P \equiv_0 Q}
$$

The following theorem, which corresponds to the subject reduction property of the usual type system, states that the transformation relation is preserved by reduction.

Theorem 2.

1. *If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Delta$ and $P \longrightarrow P'$, then there exists Q' such that (i) $Q \equiv_0 Q'$ or $Q \longrightarrow Q'$, and (ii) $\Gamma \vdash P' : I' \Longrightarrow Q' \dashv \Delta$ with $I' \leq I$.*
2. *If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Delta$ and $Q \longrightarrow Q'$, then there exists P' such that $P \longrightarrow P'$ and $\Gamma \vdash P' : I' \Longrightarrow Q' \dashv \Delta$ with $I' \leq I$.*

Definition 11. *The strong barbs of P, written $SBarbs(P)$, is defined by:*

$$
\begin{aligned}
SBarbs(P) = &\{x^! \mid P \equiv (\nu\tilde{u})\,(x![v_1,\ldots,v_n].\,Q\,|\,R) \wedge x \notin \{\tilde{u}\}\}\\
&\cup\{x^? \mid P \equiv (\nu\tilde{u})\,(x?[y_1,\ldots,y_n].\,Q\,|\,R) \wedge x \notin \{\tilde{u}\}\}\\
&\cup\{\mathbf{error} \mid P \Uparrow\}
\end{aligned}
$$

Theorem 3 (transformation preserves barbs). *If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Delta$ and $\Gamma(x) = \mathbf{Un}$ for every $x \in dom(\Gamma)$, then $SBarbs(P) = SBarbs(Q)$.*

Theorems 2 and 3 imply that our transformation preserves the weak barbed bisimilarity.

Theorem 4. *Suppose that $\Gamma(x) = \mathbf{Un}$ for every $x \in dom(\Gamma)$. If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Gamma$, then $P \overset{\bullet}{\approx} Q$.*

Lemma 1. *Suppose that $\Gamma(x) = \mathbf{Un}$ for every $x \in dom(\Gamma)$. If $\Gamma \vdash P : I \Longrightarrow Q \dashv \Gamma$ and C is a context, then $\Delta \vdash C\langle P\rangle : \mathbf{I} \Longrightarrow C\langle Q\rangle \dashv \Delta$ holds for Δ such that $\Delta(x) = \mathbf{Un}$ for every $x \in dom(\Delta)$.*

Theorem 4 and Lemma 1 imply the soundness of the transformation rules.
Proof of Theorem 1. Suppose that $\Gamma \vdash P$ and $\Gamma \vdash P : I \Longrightarrow Q \dashv \Gamma$ hold. $P \gg Q$ is trivial by the transformation rules. To show $P \approx_{\mathcal{C}} Q$, it is sufficient to show that $C\langle P\rangle \overset{\bullet}{\approx} C\langle Q\rangle$ holds for any context C. By Lemma 1, $\Delta \vdash C\langle P\rangle : \mathbf{I} \Longrightarrow C\langle Q\rangle \dashv \Delta$ holds for Δ such that $\Delta(x) = \mathbf{Un}$ for every $x \in dom(\Delta)$. So, $C\langle P\rangle \overset{\bullet}{\approx} C\langle Q\rangle$ follows from Theorem 4. □

4.5 How to Find Valid Transformations

Theorem 1 implies that we can perform UCE for a process$P$ by finding Q such that $x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un} \vdash P : I \Longrightarrow Q \dashv x_1 : \mathbf{Un}, \ldots, x_n : \mathbf{Un}$ holds. This can be carried out by performing type reconstruction in a demand-driven manner, in the same way as the type-based UCE for the λ-calculus [11]. For example, since Tr-OutElim is preferred to Tr-Out, the type constraints required by Tr-Out are added only when it turns out to be impossible to apply Tr-OutElim. We conjecture that a valid transformation can be obtained in time linear in the size of a given process as is the case for the type-based UCE for the λ-calculus [11].

5 Discussion

5.1 Extension of the Transformation Method

Using the type-based transformation described in Section 4 alone is actually insufficient in many cases, so that we need to combine it with other transformations. For example, consider the process $(\nu x)\,(x![\,]\,|\,x?[\,].\,y![\,])$. Although $y![\,]$ is a proper $\mathcal{C}$-slice, $y : \mathbf{Un} \vdash (\nu x)\,(x![\,]\,|\,x?[\,].\,y![\,]) : I \Longrightarrow y![\,] \dashv y : \mathbf{Un}$ does not hold: Rule Tr-InElim allows elimination of the input on x in $x?[\tilde{y}].\,P$ only when its continuation part P has type $\mathbf{N}$, so that we cannot replace $x?[\,].\,y![\,]$ with $y![\,]$. For the same reason, communications on channel *cstate* in Example 1 cannot be eliminated by our type-based transformation, since in the method *set*, the continuation of the input on *cstate* contains an important communication $r![\,]$. (In Example 5, we have omitted the communication on r, so that communications on *cstate* could be eliminated.)

The above problem can be solved by using type systems that can guarantee lock-freedom [12, 20]. For example, our type system for lock-freedom [12] can guarantee that for certain input or output processes, there are always corresponding output or input processes. If $x?[\,].\,(P\,|\,Q)$ is such a process (more precisely, if the input is annotated with the time limit 0 [12]) and if all the outputs on x are asynchronous (in the sense that the continuation part is $\mathbf{0}$), then $x?[\,].\,(P\,|\,Q)$ is equivalent to $(x?[\,].\,P)\,|\,Q$. Using this fact, the above process can be first replaced with $(\nu x)\,(x![\,]\,|\,x?[\,].\,\mathbf{0}\,|\,y![\,])$, and then it is replaced with $(\nu x)\,(\mathbf{0}\,|\,\mathbf{0}\,|\,y![\,])$ by the transformation in Section 4. In a similar manner, we can eliminate communications on *cstate* from *CPoint*: See Appendix A.

5.2 Limitation of the Slicing Criteria

We think that our slicing criteria given in Section 3 (combined with some preprocessing) cover a wide range of program slicing problems, but they are by no means universal. For example, they are not sufficient for the purpose of finding the cause of a deadlock. Consider the deadlocked process: $(\nu x)\,(\nu y)\,(x?[\,].\,(P\,|\,y![\,])\,|\,y?[\,].\,(Q\,|\,x![\,]))$. If one wants to find the cause of the deadlock by looking at a slice of the process, $(\nu x)\,(\nu y)\,(x?[\,].\,y![\,]\,|\,y?[\,].\,x![\,])$ would be an ideal slice. According to the slicing criterion in Section 3, however, $\mathbf{0}$, which is not useful for finding the cause of the deadlock, is also a proper slice. It is left for future work to find an appropriate criterion for this kind of program slicing. (Fortunately, our type-based transformation given in Section 3 does NOT transform $(\nu x)\,(\nu y)\,(x?[\,].\,(P\,|\,y![\,])\,|\,y?[\,].\,(Q\,|\,x![\,]))$ into $\mathbf{0}$, but it is a coincidence, coming from the limitation of our type-based transformation.)

6 Related Work

There are recently several pierces of work on program slicing for concurrent programs [5, 9, 14, 24]. They use either a specific programming language (Java) or more restrictive thread models than the π-calculus. They describe slicing algorithms in a procedural manner, as opposed to our declarative specification using type-based transformation rules. As for the quality of sliced programs, our type-based method can eliminate less code than those based on flow graphs [5, 9], since our method does not take into account the direction of data flow. We can overcome the problem by using subtyping as in type-based UCE for functional languages [3, 6]. Hatcliff et al. [9] gave a formal correctness criterion of program slicing for multi-threaded Java programs, by using a variant of bisimulation. As mentioned in Section 3, we think that this criterion can be considered an instance of our general slicing criterion (modulo preprocessing).

Our type-based method for UCE is related with three lines of research. First, the idea of using types for eliminating useless expressions has its origin in type-based UCE for functional programs [3, 6, 11]. The main differences are that in the present work, we deal with concurrency primitives and untyped programs. Secondly, the idea of introducing a special type **Un** to deal with untrusted values has been used in type systems for reasoning about security properties [1, 4, 7]. Thirdly, our method is also related with type-based information flow analysis for the π-calculus [10, 18], since UCE, program slicing, and information flow analysis can all be considered instances of the dependency analysis [2]. In fact, UCE can be used for information flow analysis and vice versa: We can verify that information about some secret data is not leaked by a process P, by performing UCE on P and checking that the secret data are removed by UCE. Conversely, we can check that a certain term in a process P is unnecessary, by marking it as secret data and performing information flow analysis to check that information about the marked data is not leaked by P. An important difference between the present work and the type-based information flow analyses for the π-calculus is that the previous work on type-based information flow analyses restrict the environment of a process to well-typed environments, while we do not impose such restriction. Our type-based UCE described in Section 4 is not powerful enough to perform as accurate information flow analysis as Honda and Yoshida's type system [10], but with the extension sketched in Section 5, the accuracy of our analysis seems to become comparable to that of Honda and Yoshida's type system. Unlike Honda and Yoshida's type system, the extended analysis can be performed automatically without programmers' annotation [13].

7 Conclusion

We gave a formal definition of the problems of useless-code elimination and program slicing for the π-calculus and presented a type-based method that is applicable to untyped programs. We hope that this work serves as a basis for further formal studies of UCE and program slicing for concurrent programs.

Acknowledgment We thank Atsushi Shiro for discussions on the subject, and Reynald Affeldt, Haruo Hosoya, and Atsushi Igarashi for comments on a draft of this paper.

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the Association for Computing Machinery (JACM)*, 46(5):749–786, 1999.
2. M. Abadi, A. Banerjee, N. Heintze, and J. G. Rieck. A core calculus of dependency. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 147–169, 1999.
3. S. Berardi, M. Coppo, F. Damiani, and P. Giannini. Type-based useless-code elimination for functional programs. In *Proceedings of SAIG 2000*, volume 1924 of *Lecture Notes in Computer Science*, pages 172–189. Springer-Verlag, 2000.
4. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 365–379. Springer-Verlag, 2000.
5. J. Cheng. Slicing concurrent programs - a graph-theoretical approach. In *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 1993.

6. F. Damiani. Useless-code detection and elimination for pcf with algebraic data types. In *Proceedings of TLCA'99*, volume 1581 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, 1999.
7. A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 145–159. IEEE Computer Society Press, 2001.
8. M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
9. J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal stufy of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings of SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1999.
10. K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2002.
11. N. Kobayashi. Type-based useless variable elimination. *Higher-Order and Symbolic Computation*, 14(2-3):221–260, 2001.
12. N. Kobayashi. A type system for lock-free processes. *Information and Computation*, 177:122–159, 2002.
13. N. Kobayashi. Type-based information flow analysis for the pi-calculus. submitted, 2003.
14. J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).
15. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
16. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
17. R. D. Nicola and M. C. B. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
18. F. Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
19. T. Reps and T. Turnidge. Program specialization via program slicing. In *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 409–429, 1996.
20. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1-2):457–493, 1999.
21. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
22. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–181, 1995.
23. M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 291–302, 1999.
24. J. Zhao. Slicing concurrent Java programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages 126–133, May 1999.

Appendix

A Program slicing for *CPoint* in Example 1

In this appendix, we show how to achieve program slicing for *CPoint* in Example 1, by using the transformation in Section 4 and the extension sketched in Section 5.

Suppose that every client is interested only in services *set* and *getxy*. Then, program slicing is achieved by performing UCE for $(\nu\, getc)\, CPoint$. By using the transformation in Section 4, $(\nu\, getc)\, CPoint$ is transformed into the following process (let $set : \mathbf{Un}, getxy : \mathbf{Un}$ be the input and output type environment):

$$\begin{array}{l}(\nu\, getc)\,(\nu\, state)\,(\nu\, cstate)\,(state![1,2] \mid cstate![\star] \\ \qquad \mid *set?[x,y,c,r].\, state?[\_,\_].\, cstate?[\_].\,(state![x,y] \mid cstate![\star] \mid r![\,]) \\ \qquad \mid *getxy?[r].\,(state?[x,y].\,(state![x,y] \mid r![x,y])) \\ \qquad \mid *getc?[\_].\,(cstate?[\_].\,(cstate![\star] \mid \mathbf{0})))\end{array}$$

The argument of channel *cstate* has been removed (replaced by $\star$) and a reply on channel *getc* has been removed, but communications on *cstate* have not been removed yet.

Now, as mentioned in Section 5, using the type system for lock-freedom [12], we can replace the above process with the following process:

$$\begin{array}{l}(\nu\, getc)\,(\nu\, state)\,(\nu\, cstate)\,(state![1,2] \mid cstate![\star] \\ \qquad \mid *set?[x,y,\_,r].\, state?[\_,\_].\,(cstate?[\_].\, cstate![\star] \mid state![x,y] \mid r![\,]) \\ \qquad \mid *getxy?[r].\,(state?[x,y].\,(state![x,y] \mid r![x,y])) \\ \qquad \mid *getc?[\_].\,(cstate?[\_].\,(state![\star] \mid \mathbf{0})))\end{array}$$

Here, the sub-process $cstate?[\_].\,(state![x,y] \mid cstate![\star] \mid r![\,])$ has been replaced with $(cstate?[\_].\, cstate![\star]) \mid state![x,y] \mid r![\,]$, so that the dependency between communications on *cstate* and r is removed.

Now, by applying the transformation in Section 4 again, we obtain the following optimized program:

$$\begin{array}{l}(\nu\, getc)\,(\nu\, state)\,(\nu\, cstate)\,(state![1,2] \mid \mathbf{0} \\ \qquad \mid *set?[x,y,\_,r].\, state?[\_,\_].\,(\mathbf{0} \mid state![x,y] \mid r![\,]) \\ \qquad \mid *getxy?[r].\,(state?[x,y].\,(state![x,y] \mid r![x,y])) \\ \qquad \mid \mathbf{0})\end{array}$$

This is identical to the sliced program given in Example 2 modulo garbage processes $\mathbf{0}$ and ν-prefixes.