A π-Calculus Semantics of Java: The Full Definition

Bart Jacobs and Frank Piessens
{bart.jacobs,frank.piessens}@cs.kuleuven.ac.be

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
Fax +3216327996

Abstract. We present a formal semantics of the concurrent object-oriented programming language Java, as a mapping of Java programs to π-calculus processes. Our semantics shows how Java features such as polymorphism, typecasts, exceptions, per-thread memory caching, and native method invocations can together be modelled in the π-calculus.
Keywords. Semantics, object-oriented programming language, concurrency, π-calculus.

1 Introduction

Soon after the introduction of the π-calculus, Walker and others [10, 7] noticed that this calculus was very suitable for specifying the semantics of concurrent object-oriented languages. Walker illustrated this by showing how a toy OO language could be translated. The calculus has been used by different authors to give semantics to academic concurrent object-oriented languages (or specific communication constructs in these languages) (a representative example is [6]), and languages have been built as syntactic sugar over the calculus [5]. But to our best knowledge, nobody has yet taken the effort to apply these ideas to a real-life language such as Java or C♯. The only reasonably complete semantics of, for instance, the Java language is based on Abstract State Machines [8]. Other approaches to giving a formal semantics to Java either limit themselves to a small subset of the language, or to sequential execution [2, 9, 4].

While working on a project about verification of Java programs, we needed a formal semantics of a very large subset of Java, and we decided to base this semantics on the π-calculus, extending Walker's ideas. Our semantics covers almost the full Java Language Specification [3]. Section 9 gives details of the differences between our specification of Java and the official JLS.

In this paper, we report on how this π-calculus based semantics is structured, and we discuss both the positive and negative aspects of the semantics. In order to deal with all the complexities of the Java language, we had to extend the original approach of Walker in many ways. For instance, to deal with concurrency

in Java, a formalization of the Java memory model had to be developed, and an explicit representation of thread-local storage is present in our semantics.

We assume that the reader of this paper is familiar with both the π-calculus, including the syntactic sugar and the typing introduced in the book by Walker and Sangiorgi [7], and with the Java Language Specification. The rest of this paper is structured as follows. In Section 2 we describe the general philosophy underlying the mapping of Java programs to processes. In Section 3 we show the definition of the mapping. In Sections 4 and 5, we analogously describe the memory model. In Sections 6, 7, and 8, we analogously describe the mapping of Java statements and expressions to processes. In Sections 9 and 10, we discuss our semantics and related work. The notational conventions used in this text are described in the Appendix.

2 Programs — Semantic Domain

Our notion of program does not commit to a particular way to use a program. A program is simply a stand-alone set of classes and interfaces. There is no notion of a main method, and there is also no notion of execution of a program. Rather, there is the notion of the use of, or interaction with the program at run-time. We refer to the entity that uses the program as the *client*.

Let p be a Java program. Our semantics maps this program onto a π-calculus process $P = [\![p]\!]$ whose free names correspond to the ways in which the program p can be used. The free names of P are of the type *ProgramChannel*, defined as follows:

$$
\begin{aligned}
\mathsf{data}\ \mathit{ProgramChannel} \;=\; & \mathit{MethodChannel}(\mathit{TypeName}, \mathit{Signature}) \\
\mid\; & \mathit{NewObjectChannel}(\mathit{TypeName}) \\
\mid\; & \mathit{NewPublicSuperChannel}(\mathit{TypeName}) \\
\mid\; & \mathit{NewInternalSuperChannel}(\mathit{TypeName}) \\
\mid\; & \mathit{UpcastChannel}(\mathit{TypeName}, \mathit{TypeName}) \\
\mid\; & \mathit{IdentityServerChannel}
\end{aligned}
$$

2.1 Identities

All program channels are functional, i.e. stateless, except for *IdentityServerChannel*. This channel is of type $\mathsf{o}(\mathit{NaturalNumber})$. Each time an input action is performed on this channel, a new natural number is retrieved. As we will see further, these are used as identities of objects, threads, and field master variables.

2.2 Statically-Bound Method Invocations

The primary way to use a program is by invoking one of the program's methods. Our semantics specifies the observable behavior of a program when one of its

methods is invoked. One or more of the following things can happen when a method is invoked:

- The invocation completes normally
- The invocation completes abruptly because of an exception
- A method not implemented inside the program is invoked by the program

The first two things can occur only once for each invocation. The third, however, can occur more than once, and, if new threads are started inside the program, some of these program-to-client invocations can even occur after the client-to-program invocation completes. We believe that our treatment of this kind of invocations, which are known as callbacks, is a particular strength of our semantics.

There are two kinds of invocations of methods not implemented inside a program: native method invocations and dynamically-bound invocations on objects passed (directly or indirectly) as arguments of an invocation to the program, when the objects are instances of classes declared outside of the program, i.e. classes declared by the client. (See Section 2.5.)

It may seem wrong to treat native method invocations as observable events. Indeed, when looking from the point of view of the developer of an application, the observable events are e.g. the appearances of windows and buttons on the screen, rather than the underlying native method invocations, which are not even well-documented. But, when defining a semantics of the Java language (as opposed to the Java Platform, which includes the class libraries and the native libraries), we consider every interaction between the program and its environment (i.e. the client) to be observable.

The behavior of a method invocation depends on which thread performs the invocation, in two ways: firstly, some thread T is allowed to enter a synchronized statement owned by some thread T' only if $T = T'$; secondly, the behavior of a method which accesses fields is more deterministic if previous accesses occurred in the same thread, because of Java's memory model, which intends to permit efficient implementations on common hardware platforms. The memory model allows an optimizing compiler to cache variables in CPU registers, and since each thread conceptually has its own set of registers, these caches act like per-thread "working copies" of the variables.

This dependency of the behavior of a method invocation on which thread performs the invocation is reflected in our semantics by the fact that a so-called context channel must be passed as an argument of each output action that corresponds with a method invocation.

Our semantics defines a process $\mathit{NewContext}(x)$, parameterized on a channel name x. This process encapsulates the semantics of creating a new thread. $\mathit{NewContext}(x)$ retrieves a new number from the identity server and creates a new context on channel x, which can then be used for output actions corresponding to method invocations.

Suppose the program p declares a class C which declares a static void method m with no arguments. Then, the type of the channel $\mathit{MethodChannel}(C, m())$ is $\sharp(\mathit{Context}, \mathsf{o}(), \mathsf{o}(\mathit{Exception}))$. The first argument is the context corresponding to

the thread that invokes the method. The second argument is the channel that is signaled upon normal completion of the invocation. The third argument is the channel that is signaled upon abrupt completion because of an exception. The exception object is sent as an argument.

The process $P = [\![p]\!]$ corresponding to a Java program p is output-only on all method channels that correspond to native methods. When the program p invokes a native method, this corresponds to the process P performing an output action on the corresponding method channel. The client of the program might wish to complete the invocation normally; this corresponds to an output action by the client on the second argument of the action corresponding to the invocation.

2.3 Creating Objects

Another common way to use a program is by creating an instance of one of the program's classes. Suppose the program p declares a class C. Then the type of the channel $\mathit{NewObjectChannel}(C)$ is $\sharp(\mathsf{o}(\mathit{Object}_C))$. The client of program p creating an instance of class C corresponds to performing an output action on $\mathit{NewObjectChannel}(C)$, passing as an argument a channel on which the program sends the newly created object. Note that we do not consider constructor invocation to be part of object creation; we do not distinguish constructor invocation from method invocation. (The Java Virtual Machine follows the same approach.)

2.4 Using Objects

An object can be used in the following ways:

- invocation of an instance method
- use or assignment of an instance field
- identity comparison with another object
- typecast
- synchronization on the object

The channel type Object_C for a given class C with superclass S is defined as shown in Figure 1. The following auxiliary types are used:

$$\mathit{NaturalNumber} = \mathsf{o}(\mathsf{i}(), \mathsf{i}(\mathit{NaturalNumber}))$$
$$\mathit{Lock} = \sharp()$$

Typecasts Performing an upcast of an object of type C to a supertype T corresponds to performing an output action on the $\mathit{UpcastChannel}(C, T)$ channel, passing as arguments the object and a channel on which the program will send a view of the object of type T.

Performing a dynamic cast of an object to a type T corresponds to performing an output action on the object's dynamic cast channel, passing as arguments an

$$
\begin{array}{lll}
\mathit{Object}_C & = & \sharp(\mathit{NaturalNumber}, & \text{identity} \\
& & \mathit{Lock}, & \text{lock} \\
& & \mathit{Variable}(\mathit{Option}(\mathit{NaturalNumber})), & \text{lock owner} \\
& & \mathit{Fields}_{\mathrm{private},C}, & \text{private fields} \\
& & \mathit{Fields}_{\mathrm{internal},C}, & \text{internal fields} \\
& & \mathit{Fields}_{\mathrm{public},C}, & \text{public fields} \\
& & \mathit{Methods}_{\mathrm{internal},C}, & \text{internal methods} \\
& & \mathit{Methods}_{\mathrm{public},C}, & \text{public methods} \\
& & \mathit{Object}_S, & \text{super-object} \\
& & \mathit{DynamicCast}) & \text{dynamic cast}
\end{array}
$$

Fig. 1. Definition of Object_C

encoding $\mathit{Enc}(T)$ of the name of T as a natural number and a channel on which the object will send a view of itself of type T. A dynamic cast channel is of type *DynamicCast*, which is defined as follows:

$$\mathit{DynamicCast} = \mathsf{o}(\mathit{Enc}(T) : \mathit{NaturalNumber}, \mathsf{i}(\mathit{Option}(\mathit{Object}_T)))$$

(The above type definition is informal; it reflects that the type of the second argument of an output action on the dynamic cast channel depends on the value of the first argument.)

Fields $\mathit{Fields}_{a,C} = \mathsf{i}(\ldots, \mathit{Field}(T_i), \ldots)$ where $\ldots, f_i : T_i, \ldots$ are the fields of accessibility a declared or inherited by class C.

$\mathit{Field}(T) = \mathsf{i}(\mathit{Variable}(T), \mathit{NaturalNumber})$

Through the channel corresponding with some field, retrieved from a particular object, the client can receive the variable containing the field's value, as well as a unique natural number distinguishing this variable from all other such variables in the same object and in other objects. Whenever an object of some class C is created, such a variable is created for each field declared in C and for each field declared in a superclass of C.

Dynamically-Bound Method Invocations $\mathit{Methods}_{a,C} = \mathsf{i}(\ldots, \mathit{Method}(\boldsymbol{p_i}, r_i), \ldots)$ where $\ldots, m_i(\boldsymbol{p_i}) : r_i, \ldots$ are the methods of accessibility a declared or inherited by class C.

$\mathit{Method}(\boldsymbol{p}, r) = \mathsf{o}(\mathit{Context}, \mathsf{i}([\![r]\!]), \mathsf{i}(\mathit{Exception}), [\![p_1]\!], \ldots, [\![p_n]\!])$

Note that the target object need not be passed as an argument of the output action corresponding with a dynamically-bound method invocation.

2.5 Declaring Subclasses

Yet another way to use a program is by declaring a class D that extends a class C declared in the program. In itself, this does not cause any run-time behavior. It is when an instance of the client-declared class D is created, that the client needs to request a new super-object of class C from the program. As part of the request, the client includes pointers to the methods declared or inherited by D (with the `this` parameter already bound) which override methods declared or inherited by C. Depending on whether C and D are in the same package, respectively in different packages, the *NewInternalSuperChannel*(C) channel or the *NewPublicSuperChannel*(C) channel are used for this. This reflects the fact that internal methods in C can only be overridden if D is in the same package.

The *NewInternalSuperChannel*(C) channel is of the following type:

$$\sharp(\mathit{Methods}_{\mathrm{public},C}, \mathit{Methods}_{\mathrm{package},C}, \sharp(\mathit{Object}_C))$$

The *NewPublicSuperChannel*(C) channel is of the following type:

$$\sharp(\mathit{Methods}_{\mathrm{public},C}, \sharp(\mathit{Object}_C))$$

(The $\mathit{Methods}_{a,C}$ types are as defined in the previous subsection.)

3 Programs — The Mapping

A Java program is a finite set of class and interface declarations:

$$p = \{C_1, \ldots, C_n, I_1, \ldots, I_m\}$$

The program p is mapped onto a process as follows:

$$[\![p]\!] = \mathit{IdServer} \mid [\![C_1]\!] \mid \cdots \mid [\![C_n]\!] \mid [\![I_1]\!] \mid \cdots \mid [\![I_m]\!]$$

IdServer, the identity server, is defined as follows:

$$\mathit{IdServer} = \mathsf{new}\, l\, (!l(n).\overline{\mathit{identity\text{-}server}}\langle n\rangle.\bar{l}\langle \mathsf{Succ}\ n\rangle \mid \bar{l}\langle \mathsf{Zero}\rangle)$$

3.1 Interface Declarations

Let T be the name of interface I; let $I_1, \ldots, I_k$ be the superinterfaces of I, including I itself. Let T_i be the name of interface I_i.

$$[\![I]\!] = U_1 \mid \cdots \mid U_k$$

Let $s_1, \ldots, s_q$ be the signatures of the methods declared or inherited by I_i. The upcast server U_i is defined as follows:

$$U_i = !T{::}\uparrow T_i(o\, r).\bar{r}\langle\{o, \mathsf{public\text{-}methods} \leftarrow \{s_1 \leftarrow o.\mathsf{public\text{-}methods}.s_1, \ldots, s_q \leftarrow o.\mathsf{public\text{-}methods}.s_q\}\}\rangle$$

where $T{::}\uparrow T_i$ is a shorthand for *UpcastChannel*(T, T_i). The process U_i accepts an object o of type Object_T and returns a copy of o in which the public-methods item has been updated to leave out the methods of interface T that are not methods of interface T_i.

3.2 Class Declarations

Let $s_1, \ldots, s_q$ be the signatures of the methods declared or inherited by class C. Let T be the name of C. Let $T_1, \ldots, T_k$ be the names of the superclasses and superinterfaces of C, including C itself.

$$[\![C]\!] = N \mid N_{SP} \mid N_{SI} \mid M_1 \mid \cdots \mid M_q \mid U_1 \mid \cdots \mid U_k$$

Upcast Servers

- If T_i is C:
$$U_i = !T :: \uparrow T_i(o\, r).\overline{r}\langle o \rangle$$
- If T_i is a superinterface of C:
Let $s_1^I, \ldots, s_q^I$ be the signatures of the methods declared or inherited by T_i.
$$U_i = !T :: \uparrow T_i(o\, r).\overline{r}\langle \{o, \mathsf{public\text{-}methods} \leftarrow a\} \rangle$$
where
$$a = \{s_1^I \leftarrow o.\mathsf{public\text{-}methods}.s_1^I, \ldots, s_q^I \leftarrow o.\mathsf{public\text{-}methods}.s_q^I\}$$
- If T_i is a superclass of C:
Let S be the immediate superclass of C.
$$U_i = !T :: \uparrow T_i(o\, r).\overline{S :: \uparrow T_i}\langle o.\mathsf{super}\, r \rangle$$

Method Servers If the method declared or inherited by C with signature s_i is static, then let

$$\boldsymbol{a} = \mathit{context}, \mathit{return}, \mathit{throw}, a_1, \ldots, a_n$$

Otherwise:

$$\boldsymbol{a} = \mathit{context}, \mathit{return}, \mathit{throw}, \mathit{this}, a_1, \ldots, a_n$$

Declared Methods Suppose C declares the method.

If this method is declared abstract or native, then $M_i = 0$. Otherwise, let $p_1, \ldots, p_n$ be the parameter names and let S be the statement that constitutes the method body.

$$M_i = !T :: s_i(\boldsymbol{a}).\mathsf{new}\, p_1 \cdots p_n\, (\overline{p_1}\langle a_1 \rangle \mid \cdots \mid \overline{p_n}\langle a_n \rangle \mid \mathsf{new}\ \mathit{break}\ ([\![S]\!];\ \overline{\mathit{return}}))$$

Inherited Methods Suppose C inherits the method from its immediate superclass S.

Let $\boldsymbol{b}$ be the result of substitution of $\mathit{this}.\mathsf{super}$ for this in $\boldsymbol{a}$.

$$M_i = !T :: s_i(\boldsymbol{a}).\overline{S :: s_i}\langle \boldsymbol{b} \rangle$$

New Internal Super Server

Root Class If C is the root class of the program (i.e. the one that does not extend a superclass):

Let $\ldots, f_i^s, \ldots$ be the names of the private instance fields declared in C. Let $\ldots, f_i^i, \ldots$ be the names of the internal instance fields declared in C. Let $\ldots, f_i^p, \ldots$ be the names of the public instance fields declared in C.

These names are used as labels and as π-calculus names, depending on the context.

Let $\boldsymbol{f} = f_1, \ldots, f_r$ be the names of all fields declared in C.

$$N_{SI} = !T{::}\textit{new-internal-super}(p\,i\,r).\mathsf{new}\,o\,l\,v\,s\,d\,\boldsymbol{f}\;\textit{identity-server}(n).(O \mid L \mid F)$$

$$\begin{aligned} O = \overline{r}\langle\{&\mathsf{identity} \leftarrow n,\\ &\mathsf{lock} \leftarrow l,\\ &\mathsf{lock\text{-}owner} \leftarrow v,\\ &\mathsf{private\text{-}fields} \leftarrow \{\ldots, f_i^s \leftarrow f_i^s, \ldots\},\\ &\mathsf{internal\text{-}fields} \leftarrow \{\ldots, f_i^i \leftarrow f_i^i, \ldots\},\\ &\mathsf{public\text{-}fields} \leftarrow \{\ldots, f_i^p \leftarrow f_i^p, \ldots\},\\ &\mathsf{internal\text{-}methods} \leftarrow i,\\ &\mathsf{public\text{-}methods} \leftarrow p,\\ &\mathsf{super} \leftarrow s,\\ &\mathsf{dynamic\text{-}cast} \leftarrow d\}\rangle \end{aligned}$$

$$L = ([v] \leftarrow \mathsf{Nothing};\, \mathsf{new}\, w\, (!w.\bar{l}.l.\overline{w} \mid \overline{w}))$$

$$F = F_1 \mid \cdots \mid F_r$$

Let t_i be the type of field f_i. Let $(t_i)_0$ be the default value of type t_i.

$$F_i = \textit{identity-server}(n).\mathsf{new}\, m\, (\overline{m}\langle(t_i)_0\rangle \mid !\overline{f_i}\langle n\, m\rangle)$$

Non-Root Class, Superclass in Same Package Let S be the name of the immediate superclass.

Let $\ldots, s_i^{s,p}, \ldots$ be the signatures of the public instance methods declared or inherited by S. Let $\ldots, s_i^{s,i,i}, \ldots$ be the signatures of the internal instance methods declared or inherited by S which are also signatures of internal instance methods declared or inherited by C. Let $\ldots, s_i^{s,i,p}, \ldots$ be the signatures of the internal instance methods declared or inherited by S which are overridden by public methods declared in C.

$$N_{SI} = !C{::}\mathit{new\text{-}internal\text{-}super}(p\, i\, r).(s \Leftarrow \overline{S{::}\mathit{new\text{-}internal\text{-}super}}\langle a_1\, a_2\rangle;\ P)$$

where

$$a_1 = \{\ldots, s_i^{s,p} \leftarrow p.s_i^{s,p}, \ldots\}$$

and

$$a_2 = \{\ldots, s_i^{s,i,i} \leftarrow i.s_i^{s,i,i}, \ldots, \ldots, s_i^{s,i,p} \leftarrow p.s_i^{s,i,p}, \ldots\}$$

Let $\boldsymbol{f} = f_1, \ldots, f_r$ be the instance fields declared by C.

$$P = \mathsf{new}\ \boldsymbol{f}\ (O \mid F)$$

Let $\ldots, f_i^s, \ldots$ be the private fields declared by C. Let $\ldots, f_i^{i,d}, \ldots$ be the internal fields declared by C. Let $\ldots, f_i^{i,i}, \ldots$ be the internal fields inherited by C. Let $\ldots, f_i^{p,d}, \ldots$ be the public fields declared by C. Let $\ldots, f_i^{p,i}, \ldots$ be the public fields inherited by C.

$$\begin{aligned}
O = \overline{r}\langle\{&\mathsf{identity} \leftarrow s.\mathsf{identity},\\
&\mathsf{lock} \leftarrow s.\mathsf{lock},\\
&\mathsf{lock\text{-}owner} \leftarrow s.\mathsf{lock\text{-}owner},\\
&\mathsf{private\text{-}fields} \leftarrow \{\ldots, f_i^s \leftarrow f_i^s, \ldots\},\\
&\mathsf{internal\text{-}fields} \leftarrow \{\ldots, f_i^{i,d} \leftarrow f_i^{i,d}, \ldots, \ldots, f_i^{i,i} \leftarrow s.\mathsf{internal\text{-}fields}.f_i^{i,i}, \ldots\},\\
&\mathsf{public\text{-}fields} \leftarrow \{\ldots, f_i^{p,d} \leftarrow f_i^{p,d}, \ldots, \ldots, f_i^{p,i} \leftarrow s.\mathsf{public\text{-}fields}.f_i^{p,i}, \ldots\},\\
&\mathsf{internal\text{-}methods} \leftarrow i,\\
&\mathsf{public\text{-}methods} \leftarrow p,\\
&\mathsf{super} \leftarrow s,\\
&\mathsf{dynamic\text{-}cast} \leftarrow s.\mathsf{dynamic\text{-}cast}\}\rangle
\end{aligned}$$

$$F = F_1 \mid \cdots \mid F_r$$

Let t_i be the type of field f_i. Let $(t_i)_0$ be the default value of type t_i.

$$F_i = \mathit{identity\text{-}server}(n).\mathsf{new}\, m\, (\overline{m}\langle(t_i)_0\rangle \mid !\overline{f_i}\langle n\, m\rangle)$$

Non-Root Class, Superclass in Different Package Let S be the name of the immediate superclass.

Let $\ldots, s_i^{s,p}, \ldots$ be the signatures of the public instance methods declared or inherited by S.

$$\begin{aligned}
N_{SI} = &!C{::}\mathit{new\text{-}internal\text{-}super}(p\, i\, r).\\
&(s \Leftarrow \overline{S{::}\mathit{new\text{-}public\text{-}super}}\langle\{\ldots, s_i^{s,p} \leftarrow p.s_i^{s,p}, \ldots\}\rangle;\ P)
\end{aligned}$$

Let $\boldsymbol{f} = f_1, \ldots, f_r$ be the fields declared by C.

$$P = \mathsf{new}\ \boldsymbol{f}\ (O \mid F)$$

Let $\ldots, f_i^s, \ldots$ be the private fields declared by C. Let $\ldots, f_i^i, \ldots$ be the internal fields declared by C. Let $\ldots, f_i^{p,d}, \ldots$ be the public fields declared by C. Let $\ldots, f_i^{p,i}, \ldots$ be the public fields inherited by C.

$$\begin{aligned}
O = \overline{r}\langle\{&\mathsf{identity} \leftarrow s.\mathsf{identity},\\
&\mathsf{lock} \leftarrow s.\mathsf{lock},\\
&\mathsf{lock\text{-}owner} \leftarrow s.\mathsf{lock\text{-}owner},\\
&\mathsf{private\text{-}fields} \leftarrow \{\ldots, f_i^s \leftarrow f_i^s, \ldots\},\\
&\mathsf{internal\text{-}fields} \leftarrow \{\ldots, f_i^i \leftarrow f_i^i, \ldots\},\\
&\mathsf{public\text{-}fields} \leftarrow \{\ldots, f_i^{p,d} \leftarrow f_i^{p,d}, \ldots, \ldots, f_i^{p,i} \leftarrow s.\mathsf{public\text{-}fields}.f_i^{p,i}, \ldots\},\\
&\mathsf{internal\text{-}methods} \leftarrow i,\\
&\mathsf{public\text{-}methods} \leftarrow p,\\
&\mathsf{super} \leftarrow s,\\
&\mathsf{dynamic\text{-}cast} \leftarrow s.\mathsf{dynamic\text{-}cast}\}\rangle
\end{aligned}$$

$$F = F_1 \mid \cdots \mid F_r$$

Let t_i be the type of field f_i. Let $(t_i)_0$ be the default value of type t_i.

$$F_i = \mathit{identity\text{-}server}(n).\mathsf{new}\, m\, (\overline{m}\langle(t_i)_0\rangle \mid !\overline{f_i}\langle n\, m\rangle)$$

New Public Super Server Let $\ldots, s_i^i, \cdots = s_1^i, \ldots, s_r^i$ be the signatures of the internal instance methods declared or inherited by C.

$$\begin{aligned}
N_{SP} = &!T{::}\mathit{new\text{-}public\text{-}super}(p\, r).\\
&\mathsf{new}\ \cdots\ s_i^i\ \cdots\ (o \Leftarrow \overline{C{::}\mathit{new\text{-}internal\text{-}super}}\langle p\,\{\ldots, s_i^i \leftarrow s_i^i, \ldots\}\rangle;\ P)
\end{aligned}$$

$$P = \overline{r}\langle o\rangle.(P_1 \mid \cdots \mid P_r)$$

Let $s_i^i = m(t_1, \ldots, t_n)$.

$$P_i = !s_i^i(c\, r\, t\, a_1\ \cdots\ a_n).\overline{C{::}s_i^i}\langle c\, r\, t\, o\, a_1\ \cdots\ a_n\rangle$$

New Object Server Let $\ldots, s_i^p, \cdots = s_1^p, \ldots, s_r^p$ be the signatures of the public instance methods declared or inherited by C.

$$N = !T{::}new(r).\mathsf{new} \cdots s_i^p \cdots (o \Leftarrow \overline{C{::}new\text{-}public\text{-}super}\langle\{\ldots, s_i^p \leftarrow s_i^p, \ldots\}\rangle; P)$$

$$P = \overline{r}\langle o\rangle.(D \mid P_1 \mid \cdots \mid P_r)$$

$$D = !o.\mathsf{dynamic\text{-}cast}(n\, r).D_0$$

Let m be a natural number that is greater than all encodings of names of superclasses or superinterfaces of C.

$$D_m = \overline{r}\langle\mathsf{Null}\rangle$$

For each $i < m$:

$$D_i = \mathsf{case}\ n\ \mathsf{of}\ \begin{cases}\mathsf{Zero} & \Rightarrow D_i' \\ \mathsf{Succ}\ n & \Rightarrow D_{i+1}\end{cases}$$

If i is the encoding of a superclass or superinterface type name T':

$$D_i' = (o \Leftarrow \overline{T{::} \uparrow T'}\langle o\rangle; \overline{r}\langle\mathsf{Object}\ o\rangle)$$

Otherwise:

$$D_i' = \overline{r}\langle\mathsf{Null}\rangle$$

Let $s_i^p = m(t_1, \ldots, t_n)$.

$$P_i = !s_i^p(c\, r\, t\, a_1 \cdots a_n).\overline{T{::}s_i^p}\langle c\, r\, t\, o\, a_1 \cdots a_n\rangle$$

4 Memory Model — Semantic Domain

As will be discussed below, a statement or expression accessing a field retrieves a working copy for the field and then accesses the field via this working copy. In our semantics, the definition of the behavior of working copies is implied in the definition of the *NewContext* process mentioned above.

We see a working copy holding values of type T as a state machine with a set of states given by the constructors *Empty*, *Clean*(T), and *Dirty*(T), and a set of actions given by the constructors *Use*(T), *Assign*(T), *Lock*, *Unlock*, *Load*(T), and *Store*(T).

When a working copy is created, it is in the *Empty* state. The *Use* and *Assign* actions correspond to input and output actions on the working copy's channel, which occur as part of the evaluation of field use and assign expressions in the program. The *Lock* and *Unlock* actions occur when the "lock working copies", respectively the "unlock working copies" channels of the context are signalled

as part of the execution of synchronized statements. The *Load* and *Store* actions are always enabled; they can occur at any time. This reflects the unspecified nature of the optimizations performed by just-in-time bytecode compilers and CPUs with respect to memory access.

The transition table is as follows:

	Empty	*Clean(w)*	*Dirty(w)*
Use(w)	—	*Clean(w)*	*Dirty(w)*
Assign(w′)	*Dirty(w′)*	*Dirty(w′)*	*Dirty(w′)*
Lock	*Empty*	*Empty*	*Dirty(w)*
Unlock	*Empty*	*Clean(w)*	—
Store(w)	—	—	*Clean(w)*
Load(w′)	*Clean(w′)*	*Clean(w′)*	—

A dash — means that that action is not allowed in that state.

The entry in the transition table for the *Lock* action in the *Dirty(w)* state is somewhat problematic; this is discussed in Section 9.

5 Memory Model—The Mapping

In this section, we give a textual description of the mapping of the memory model. The actual π-calculus process expressions can be found on the website [1]; they have been omitted here because they mostly concern straightforward but tedious manipulation of data structures.

The channel x in the process *NewContext*(x) is of type *Context*, which is defined as follows:

$$
\begin{aligned}
\mathit{Context} = \mathsf{o}(&\mathit{NaturalNumber}, && \text{context identity}\\
&\mathsf{i}(\forall T.(\mathit{Field}(T), \mathsf{o}(\sharp([\![T]\!])))), && \text{get working copy}\\
&\mathsf{i}(\mathsf{o}()), && \text{lock working copies}\\
&\mathsf{i}(\mathsf{o}()), && \text{unlock working copies}
\end{aligned}
$$

The channel x is called the context channel.

In Section 6 we explain how the execution of statements and the evaluation of expressions interacts with this process. Here, we describe the process's behavior.

The first element of the tuple which can be received from the context channel is the context identity. The *NewContext*(x) process receives a new identity from the identity server before it starts serving the tuple on the context channel.

The second element is a channel through which the process receives requests for working copies of fields. A request for a working copy is a tuple containing a field and a channel on which to return the working copy. A field is a tuple containing a variable (i.e. the master variable for the field) and a field identity. The process keeps a table with associations of field identities with working copies. The table is initially empty. When there is no entry in the table for the field identity of the field passed in the request, a new working copy is created and a

new entry is added to the table, linking the field identity in the request to the new working copy. Creation of a new working copy comes down to the creation of a new state machine which implements the transition table displayed in the previous section.

The third element in the tuple which can be received from the context channel is a channel through which a request can be sent to perform the *Lock* action on all working copies in the context's table. The process signals the channel provided in the request after all *Lock* actions have been performed.

The fourth and last element in the tuple is a channel through which a request can be sent to perform the *Unlock* action on all working copies in the context's table. Again, the process signals the channel provided in the request after all *Unlock* actions have been performed.

6 Statements and Expressions — Semantic Domain

Somewhat independent of the above (from the client's point of view), we also define a "behavior space" for statements and expressions. A *phrase* is a statement or an expression. Our semantics maps each phrase ϕ to a π-calculus process $P = [\![\phi]\!]$. The free names of P are of the following type:

$$
\begin{aligned}
\mathsf{data}\ \mathit{PhraseChannel} \ = \ & \mathit{DoneChannel} \\
& \mid \mathit{ThrowChannel} \\
& \mid \mathit{ReturnChannel} \\
& \mid \mathit{BreakChannel}(\mathit{Label}) \\
& \mid \mathit{ContextChannel} \\
& \mid \mathit{LocalVariableChannel}(\mathit{LocalVariableName}) \\
& \mid \mathit{ThisChannel} \\
& \mid \mathit{ProgramChannel}(\mathit{ProgramChannel})
\end{aligned}
$$

In a way, each phrase in a program has the same point of view as the client of the program. Phrases invoke methods and instantiate classes rather than declaring them. P is output-only on $\mathit{ProgramChannel}(c)$, for each program channel c.

If ϕ is a statement or an invocation of a void method, *DoneChannel* is of type $\mathsf{o}()$. Otherwise, *DoneChannel* is of type $\mathsf{o}([\![T]\!])$, with T the type of ϕ. $[\![T]\!]$ refers to the mapping of Java types to π-calculus types which is implied by our semantics. Normal completion of ϕ corresponds to an output action on *DoneChannel*, with the result of evaluation, if any, as an argument.

ThrowChannel : $\mathsf{o}(\mathit{Exception})$ Abrupt completion of ϕ because of an exception corresponds to an output action on this channel.

If the method containing ϕ is declared void, *ReturnChannel* : $\mathsf{o}()$; otherwise, *ReturnChannel* : $\mathsf{o}([\![T]\!])$, with T the return type of the method containing ϕ.

Abrupt completion because of a return statement corresponds to an output action on this channel.

For any label l, $\mathit{BreakChannel}(l) : \mathsf{o}()$. Abrupt completion because of a break statement corresponds to an output action on this channel.

$\mathit{ContextChannel} : \mathit{Context}$ Like invocation of a method, the behavior of execution of a phrase depends on the thread that performs the execution. The *ContextChannel* reflects this fact in our semantics of statements and expressions. The context channel corresponds to the thread in which the phrase is executed.

$$
\begin{array}{lr}
\mathit{Context} = \mathsf{i}(\mathit{NaturalNumber}, & \text{context identity} \\
\quad \mathsf{o}(\forall T.(\mathit{Field}(T), \mathsf{i}(\sharp([\![T]\!])))), & \text{get working copy} \\
\quad \mathsf{o}(\mathsf{i}()), & \text{lock working copies} \\
\quad \mathsf{o}(\mathsf{i}()), & \text{unlock working copies}
\end{array}
$$

A phrase uses the context identity for comparison with the lock owner of an object, and to register the current thread as the lock owner of an object.

A phrase uses the "get working copy" channel to retrieve a per-thread working copy for a given field. A phrase never directly reads or writes a field's master variable; it interacts with the working copy only. A use of a field corresponds with an input action on the working copy's channel; an assign corresponds with an output action on the same channel.

A phrase performs an output action on the "lock working copies" channel and then performs an input action on the output action's argument immediately after it acquires the lock on an object. It performs the same protocol on the "unlock working copies" channel immediately before it releases the lock on an object.

If ϕ is in the scope of a local variable $n : \mathit{LocalVariableName}$ of type T, then $\mathit{LocalVariableChannel}(n) : \sharp([\![T]\!])$. A use or assign of a local variable corresponds to an input action on the corresponding local variable channel, followed by an output action on the same channel. The argument of the input action is the variable's current value. This value is immediately passed through as the argument of the output action in case of a use; in case of an assign, the argument of the output action is the newly assigned value.

Let C be the class containing ϕ. $\mathit{ThisChannel} : \mathit{Object}_C$.

7 Expressions — The Mapping

In this section we define a translation that associates a Java expression E appearing in a Java program P with a π-calculus process $[\![E]\!]_P$. We will usually omit the subscript.

Below, we use the names *done*, *return*, etc. as shorthands for *DoneChannel*, *ReturnChannel*, etc. Also, we use n as a shorthand for $\mathit{LocalVariableChannel}(n)$ and c as a shorthand for $\mathit{ProgramChannel}(c)$.

7.1 Some Sugar

The use of the term *local* in the following definition refers to thread-local working copies of fields, rather than to block-local variables.

$$\mathsf{struct}\ \mathrm{context} = \{\mathsf{context\text{-}identity}, \mathsf{get\text{-}local}, \mathsf{lock\text{-}locals}, \mathsf{unlock\text{-}locals}\}$$

$$\begin{aligned}\mathsf{struct}\ \mathrm{object} = \{&\mathsf{identity} : \mathrm{nat}, \mathsf{lock}, \mathsf{lock\text{-}owner},\\ &\mathsf{private\text{-}fields}, \mathsf{internal\text{-}fields}, \mathsf{internal\text{-}methods}, \mathsf{public\text{-}fields}, \mathsf{public\text{-}methods},\\ &\mathsf{super}, \mathsf{dynamic\text{-}cast}\}\end{aligned}$$

$$\mathsf{data}\ \mathrm{ref} = \mathsf{Null} \mid \mathsf{Object}\ \mathrm{object}$$

$$(v \leftarrow P;\, Q) = \mathsf{new}\ \mathit{start}\,(\mathsf{new}\ \mathit{done}\,(P \mid \mathit{done}(w).\overline{\mathit{start}}\langle w\rangle) \mid \mathit{start}(v).Q)$$

$$\mathit{NotNull}(P) = \mathsf{case}\ r\ \mathsf{of}\ \begin{cases}\mathsf{Null} & \Rightarrow \mathit{ThrowNPE}\\ \mathsf{Object}\ o & \Rightarrow P\end{cases}$$

with

$$\mathit{ThrowNPE} = (r \leftarrow [\![\texttt{new NullPointerException()}]\!];\, \overline{\mathit{throw}}\langle r\rangle)$$

7.2 Literal Expressions

Let i be a literal of type `int`.

$$[\![i]\!] = \overline{\mathit{done}}\langle i\rangle$$

7.3 Operators on Primitive Values

Let E_1 and E_2 be expressions of type `int`.

$$[\![E_1 + E_2]\!] = (i_1 \leftarrow [\![E_1]\!];\, i_2 \leftarrow [\![E_2]\!];\, \overline{\mathit{done}}\langle i_1 + i_2\rangle)$$

7.4 Reference Comparison

$$[\![E_1\ \texttt{==}\ E_2]\!] = (r_1 \leftarrow [\![E_1]\!];\, r_2 \leftarrow [\![E_2]\!];\, \mathsf{case}\ r_1\ \mathsf{of}\ \begin{cases}\mathsf{Null} & \Rightarrow P_1\\ \mathsf{Object}\ o_1 & \Rightarrow P_2\end{cases})$$

with

$$P_1 = \mathsf{case}\ r_2\ \mathsf{of}\ \begin{cases}\mathsf{Null} & \Rightarrow \overline{\mathit{done}}\langle\texttt{true}\rangle\\ \mathsf{Object}\ o_2 & \Rightarrow \overline{\mathit{done}}\langle\texttt{false}\rangle\end{cases}$$

$$P_2 = \mathsf{case}\ r_2\ \mathsf{of}\ \begin{cases}\mathsf{Null} & \Rightarrow \overline{\mathit{done}}\langle\texttt{false}\rangle\\ \mathsf{Object}\ o_2 & \Rightarrow \overline{\mathit{done}}\langle o_1.\mathsf{identity} = o_2.\mathsf{identity}\rangle\end{cases}$$

7.5 The Conditional Expression

Let E_C be an expression of type `boolean` and let E_T and E_F be expressions of some type T.

$$[\![E_C \ ?\ E_T : E_F]\!] = (b \leftarrow [\![E_C]\!];\ (b\ ?\ [\![E_T]\!] : [\![E_F]\!]))$$

7.6 This

$$[\![\texttt{this}]\!] = \overline{done}\langle \mathsf{Object}\ this\rangle$$

7.7 Local Variables

$$[\![n]\!] = n(v).(\overline{n}\langle v\rangle \mid \overline{done}\langle v\rangle)$$

$$[\![n\ \texttt{=}\ E]\!] = (v \leftarrow [\![E]\!];\ n(w).(\overline{n}\langle v\rangle \mid \overline{done}\langle v\rangle))$$

7.8 Fields

Let the static type of expression E be C, where C is a class declared in program P.

Let a be private, internal, or public, according to the accessibility of field f in class C.

$$\mathit{GetField}(p, C, f, P) = \mathit{NotNull}(l \Leftarrow \overline{context.\mathsf{get\text{-}local}}\langle o.a\text{-}\mathsf{fields}.f\rangle;\ P)$$

$$[\![E\,.\,f]\!]_p = (r \leftarrow [\![E]\!];\ \mathit{GetField}(p, C, f, (l(v).\overline{done}\langle v\rangle)))$$

Suppose the following expression appears in a class D that extends a class C:

$$[\![\texttt{super}\,.\,f]\!] = [\![\texttt{((}C\texttt{) this)}\,.\,f]\!]$$

$$[\![E_1\,.\,f\ \texttt{=}\ E_2]\!] = (r \leftarrow [\![E_1]\!];\ v \leftarrow [\![E_2]\!];\ \mathit{GetField}(p, C, f, (\bar{l}\langle v\rangle.\overline{done}\langle v\rangle)))$$

7.9 Static Method Invocation

Suppose in the following expression overload resolution selects the method with signature s.

$$[\![C\,.\,m\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!] = (v_1 \leftarrow [\![E_1]\!];\ \cdots\ ;\ v_n \leftarrow [\![E_n]\!];\ \overline{C{::}s}\langle context\ done\ throw\ v_1\ \cdots\ v_n\rangle)$$

7.10 Instance Method Invocation

Suppose in the following expression the type of E is T and overload resolution selects the method with signature s.

If the method is private:

$$[\![E\texttt{.}m\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!] =$$
$$(r \leftarrow [\![E]\!]; v_1 \leftarrow [\![E_1]\!]; \cdots ; v_n \leftarrow [\![E_n]\!]; \mathit{NotNull}(\overline{T\texttt{::}s}\langle \mathit{context}\ \mathit{done}\ \mathit{throw}\ o\ v_1\ \cdots\ v_n\rangle))$$

Otherwise, let a be internal or public, as appropriate.

$$[\![E\texttt{.}m\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!] =$$
$$(r \leftarrow [\![E]\!]; v_1 \leftarrow [\![E_1]\!]; \cdots ; v_n \leftarrow [\![E_n]\!]; \mathit{NotNull}(\overline{o.a\text{-}\mathsf{methods}.s}\langle \mathit{context}\ \mathit{done}\ \mathit{throw}\ v_1\ \cdots\ v_n\rangle))$$

7.11 Super Invocation

Let S be the immediate superclass of the class in which the expression appears. Suppose in the following expression overload resolution selects the method with signature s.

$$[\![\texttt{super.}m\texttt{(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)}]\!] =$$
$$(v_1 \leftarrow [\![E_1]\!]; \cdots ; v_n \leftarrow [\![E_n]\!]; \overline{S\texttt{::}s}\langle \mathit{context}\ \mathit{done}\ \mathit{throw}\ (\mathit{this}.\mathsf{super})\ v_1\ \cdots\ v_n\rangle)$$

7.12 Typecast Expressions

Let expression E be of type T_1.

If T_2 is a superclass or a superinterface of T_1 (that is, if the typecast is a widening conversion, i.e. an upcast):

$$[\![(T_2)\ \ E]\!] = (r \leftarrow [\![E]\!]; \mathsf{case}\ r\ \mathsf{of}\ \begin{cases} \mathsf{Null} & \Rightarrow \overline{\mathit{done}}\langle\mathsf{Null}\rangle \\ \mathsf{Object}\ o_1 & \Rightarrow (o_2 \Leftarrow \overline{T_1\texttt{::}\uparrow T_2}\langle o_1\rangle;\ \overline{\mathit{done}}\langle\mathsf{Object}\ o_2\rangle) \end{cases})$$

Otherwise:

$$[\![E\ \ \mathsf{as}\ \ T_2]\!] = (r \leftarrow [\![E]\!]; \mathsf{case}\ r\ \mathsf{of}\ \begin{cases} \mathsf{Null} & \Rightarrow \overline{\mathit{done}}\langle\mathsf{Null}\rangle \\ \mathsf{Object}\ o & \Rightarrow \overline{o.\mathsf{dynamic\text{-}cast}}\langle[\![T_2]\!]\ \mathit{done}\rangle \end{cases})$$

Here, $[\![T]\!]$ is some encoding of the type name T in the datatype nat. For example, one could take the UTF-8 encoding of the type name and interpret it as a natural number.

Downcast and instanceof expressions can be translated using as expressions.

7.13 Class Instance Creation

$$[\![\texttt{new}\ \ C]\!] = (o \Leftarrow \overline{C{::}new}\langle\rangle;\ \overline{done}\langle \mathsf{Object}\ o\rangle)$$

8 Statements — The Mapping

$$\mathsf{data}\ \mathrm{completion} = \mathsf{Done}^0 \mid \mathsf{Return}^1 \mid \mathsf{Throw}^1 \mid \mathsf{Break}^1$$

$$\mathit{TryFinally}(B, G) = \mathsf{new}\ \mathit{start}\ (\mathsf{new}\ \mathit{done return throw break}\ P_1 \mid P_2)$$

with

$$P_1 = B \mid \mathit{done}.\overline{\mathit{start}}\langle\mathsf{Done}\rangle \mid \mathit{return}(v).\overline{\mathit{start}}\langle\mathsf{Return}\ v\rangle \mid \mathit{throw}(v).\overline{\mathit{start}}\langle\mathsf{Throw}\ v\rangle \mid \mathit{break}(v).\overline{\mathit{start}}\langle\mathsf{Break}\ v\rangle$$

$$P_2 = \mathit{start}(c).(G;\ \mathsf{case}\ c\ \mathsf{of}\ \begin{cases} \mathsf{Done} & \Rightarrow \overline{\mathit{done}} \\ \mathsf{Return}\ v & \Rightarrow \overline{\mathit{return}}\langle v\rangle \\ \mathsf{Throw}\ v & \Rightarrow \overline{\mathit{throw}}\langle v\rangle \\ \mathsf{Break}\ v & \Rightarrow \overline{\mathit{break}}\langle v\rangle \end{cases})$$

8.1 Expression Statement

$$[\![E\,;]\!] = (v \leftarrow [\![E]\!];\ \overline{\mathit{done}})$$

8.2 Return Statement

$$[\![\texttt{return}\ \ E\,;]\!] = (v \leftarrow [\![E]\!];\ \overline{\mathit{return}}\langle v\rangle)$$

8.3 Sequential Composition Statement

$$[\![S_1\,;\ \ S_2\,;]\!] = ([\![S_1]\!];\ [\![S_2]\!])$$

8.4 Local Variable Declaration

$$[\![\{T\ \ n\ \texttt{=}\ E;\ \ S\}]\!] = (v \leftarrow [\![E]\!];\ \mathsf{new}\ n\ (\overline{n}\langle v\rangle \mid [\![S]\!]))$$

8.5 Empty Statement

$$[\![;]\!] = \overline{\mathit{done}}$$

8.6 Labelled Statement

$$[\![l\texttt{:}\ \ S]\!] = \mathsf{new}\ \mathit{start}\ (\mathsf{new}\ \mathit{break}\ ([\![S]\!] \mid \mathit{break}(v).\overline{\mathit{start}}\langle v\rangle) \mid P)$$

with

$$P = \mathit{start}(v).\mathsf{case}\ v\ \mathsf{of}\ \begin{cases} \mathsf{Zero} & \Rightarrow \overline{\mathit{done}} \\ \mathsf{Succ}\ w & \Rightarrow \overline{\mathit{break}}\langle w\rangle \end{cases}$$

8.7 If Statement

$$[\![\texttt{if (}E\texttt{)}\ \ S_1\ \ \texttt{else}\ \ S_2]\!] = (b \leftarrow [\![E]\!];\ (b?[\![S_1]\!] : [\![S_2]\!]))$$

8.8 While Statement

$$[\![\texttt{while (}E\texttt{)}\ \ S]\!] = \mathsf{new}\ l\ (!l.(b \leftarrow [\![E]\!];\ (b?([\![S]\!];\ \bar{l}) : \overline{\mathit{done}})) \mid \bar{l})$$

8.9 Break Statement

$$[\![\texttt{break}\ \ l]\!] = \overline{\mathit{break}}\langle [\![l]\!]\rangle$$

where $[\![l]\!]$ is a natural number, encoded in datatype nat, equal to the number of enclosing labelled statements to be skipped. That is, the labelled statement with label l is the $n+1$-th most enclosing labelled statement of this break statement.

A Java program can be rewritten, preserving semantics, into one without continue statements or unlabelled break statements.

8.10 Throw Statement

$$[\![\texttt{throw}\ \ E]\!] = (r \leftarrow [\![E]\!];\ \mathit{NotNull}(\overline{\mathit{throw}}\langle o\rangle))$$

8.11 Try-Catch Statement

$$[\![\texttt{try}\ \ S_1\ \ \texttt{catch (}T\ \ n\texttt{)}\ \ S_2]\!] = \mathsf{new}\ \mathit{start}\ (\mathsf{new}\ \mathit{throw}\ ([\![S_1]\!]\ |\mathit{throw}(o).\overline{\mathit{start}}\langle o\rangle)\ |P)$$

where

$$P = \mathit{start}(o).(r \Leftarrow \overline{o.\mathsf{dynamic\text{-}cast}}\langle [\![T]\!]\rangle;\ \mathsf{case}\ r\ \mathsf{of}\ \begin{cases} \mathsf{Null} & \Rightarrow \overline{\mathit{throw}}\langle o\rangle \\ \mathsf{Object}\ o & \Rightarrow \mathsf{new}\ n\ (\overline{n}\langle r\rangle \mid [\![S_2]\!]) \end{cases})$$

8.12 Try-Finally Statement

$$[\![\texttt{try}\ \ S_1\ \ \texttt{finally}\ \ S_2]\!] = \mathit{TryFinally}([\![S_1]\!], [\![S_2]\!])$$

8.13 Synchronized Statement

$$[\![\texttt{synchronized (}E\texttt{)}\ \ S]\!] = (r \leftarrow [\![E]\!];\ \mathit{NotNull}(P_1))$$

where

$$P_1 = (x \leftarrow [o.\mathsf{lock\text{-}owner}];\ \mathsf{case}\ x\ \mathsf{of}\ \begin{cases} \mathsf{Nothing} \Rightarrow P_2 \\ \mathsf{Just}\ n \ \ \Rightarrow (n = \mathit{context}.\mathsf{context\text{-}identity}\ ?\ [\![S]\!]\ :\ P_2) \end{cases})$$

$$P_2 = o.\mathsf{lock}.([o.\mathsf{lock\text{-}owner}] \leftarrow \mathsf{Just}\ \mathit{context}.\mathsf{context\text{-}identity};\ \Leftarrow \overline{\mathit{context}.\mathsf{lock\text{-}locals}}\langle\rangle;\ \mathit{TryFinally}([\![S]\!], G))$$

$$G = (\Leftarrow \overline{\mathit{context}.\mathsf{unlock\text{-}locals}}\langle\rangle;\ [o.\mathsf{lock\text{-}owner}] \leftarrow \mathsf{Nothing};\ \overline{o.\mathsf{lock}}.\overline{\mathit{done}})$$

8.14 Asynchronous Statement

This statement, which does not exist in Java, can be used to implement `Thread.start()`. In Java, `Thread.start()` is a native method implemented in a C library. By defining this statement, we can write multithreaded programs without using native methods or C libraries.

The statement starts a new thread to execute its body and completes normally without waiting for the new thread to finish. No jump labels are in scope in S. Local variables are in scope only if they are final. `this` is in scope. It is a compile-time error for a return statement to appear in S.

$$[\![\texttt{async}\ \ S]\!] = \mathit{NewContext}(\mathsf{new}\ \mathit{done}\ \mathit{throw}\ [\![S]\!]) \mid \overline{\mathit{done}}$$

9 Discussion

There are a number of features of Java which are not covered in our current definition of the semantics. Some of these features, such as operations on primitive values and `for` loops, are entirely analogous to features we did cover. Others, specifically volatile fields, arrays, static fields, static initializers, the `wait`, `notify`, and `notifyAll` methods of class `Object`, and string literals, can probably be added without much difficulty. There are two features, dynamic class loading and reflection, which would be more difficult to add.

Our semantics does not capture accessibility issues. For example, we do not restrict channels which correspond to private methods. We do, however, correctly model the impact of accessibility modifiers on the behavior of programs. For example, if class D extends class C and C and D are in different packages, then D does not inherit the internal members of C. Also, a method declared in D never overrides an internal method declared in C.

9.1 Memory Model

Our memory model differs from the memory model specified in the JLS on two points. Firstly, in our memory model, variables of type long and double are atomic; in the JLS, they are not. Secondly, we allow the following sequence of actions by a thread on some variable: *Assign-Lock-Use*, whereas the JLS does not. Our semantics is strictly weaker than the JLS, since all programs which are correct with respect to our semantics are correct with respect to the JLS, but the converse is not true. Note that if all assignments to shared variables in a program are inside synchronized blocks, then for this program our memory model is equivalent to the JLS memory model.

After an assign, a working copy is in the *Dirty* state. With respect to *Lock* actions in the *Dirty* state, there are two possible design decisions: either we allow them, or we do not, in which case a *Store* action must occur before the *Lock* action. Neither choice is entirely JLS-compliant. In the latter case, the sequence *Assign-Lock-Assign* is disallowed, but this sequence is allowed by the JLS, so then our semantics would be stronger than the JLS.

When using our semantics for verification of programs, it must not be stronger than the JLS. When using it for verification of implementations of the Java language itself, it must not be weaker than the JLS. We chose the former alternative, because our goal is program verification.

To be compliant with the JLS, we would need to look forward in time to see if the next action after the *Lock* would be an *Assign* or some other action. In our semantics, we cannot look forward, since it is an operational semantics. This is a disadvantage of the operational semantics approach over the declarative semantics approach.

10 Related Work

As mentioned in the introduction, some work has already been done in the area of using the π-calculus to define the semantics of object-oriented programming languages (e.g. [10, 7]). However, to our knowledge, this is the first time this has been applied to a widely-used language, combining inheritance, overriding (distinguishing public and internal methods), a semantics for unsynchronized access to shared variables, and many other features.

Pict [5] is a programming language based on the π-calculus, and, like in this article, the semantics is given by translation to the π-calculus. There is no explicit notion of inheritance or overriding in Pict, although this functionality can be implemented on top of Pict easily.

A number of people have defined a semantics of Java (or some subset of it) in formalisms other than the π-calculus. Two examples are the LOOP project [2], which uses a weakest precondition-based semantics, and project Bali [9, 4]. Neither project supports concurrency. Another example is [8], which gives a rather complete semantics of Java using abstract state machines. A notable difference with our semantics is that these abstract state machines are modularised

along the structure of the Java language itself (imperative features, module features, object-oriented features, etc.), whereas in our semantics the process for a program is modularized along the structure of the program. This might make reasoning about program modules easier.

References

1. B. Jacobs and F. Piessens. A π-calculus semantics of Java—the definition. At `http://www.cs.kuleuven.ac.be/~bartj/javapi`.
2. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in LNCS, pages 284–299, 2001.
3. B. Joy, G. Steele, J. Gosling, and G. Bracha. *Java Language Specification (*2^{nd} *Edition)*. Addison-Wesley, 2000.
4. T. Nipkow et al. Project Bali. At `http://isabelle.in.tum.de/Bali`.
5. B. B. Pierce and D. N. Turner. A programming language based on the π-calculus. In G. Plotkin et al., editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
6. B. Robben, F. Piessens, and W. Joosen. Formalizing Correlate—from practice to π. In D. Duke and A. Evans, editors, *Proceedings of the Second International BCS-FACS Northern Formal Methods Workshop (NFMW'97)*, Electronic Workshops in Computing, pages 1–16, 1997.
7. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
8. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.
9. D. von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. `http://isabelle.in.tum.de/Bali/papers/CPE01.html`.
10. D. Walker. Objects in the π-calculus. *Information and Computation*, 115:253–271, 1995.

Appendix Notational Conventions

Our semantics uses the pure untyped monadic π-calculus. Type judgments are informal. In this appendix, we introduce some syntactic sugar.

Unless stated otherwise, all names introduced in the right-hand side of a definition are assumed not to appear free in the left-hand side.

10.1 Blocking Calls

Let $r \notin \mathrm{fn}(w_1 \cdots w_n P)$.

$$(v \Leftarrow \overline{x}\langle w_1 \cdots w_n \rangle;\ P) = \mathsf{new}\, r\, (\overline{x}\langle w_1 \cdots w_n r \rangle.r(v).P)$$

10.2 Values

Let I be some finite set. Let $\boldsymbol{w}$ be a list of names that contains one name w_j for each j in I. Let one such $\boldsymbol{w}$ be chosen for each finite set I. Let the names in $\boldsymbol{w}$ not be used anywhere else.

Let $i \in I$.

$$\langle i \rangle = \mathsf{new}\, v\, \langle v \rangle.(!v(\boldsymbol{w}).\overline{w_i})$$

$$(i : I)P_i = (v)(\mathsf{new}\, \boldsymbol{w}\, (\overline{v}\langle \boldsymbol{w} \rangle \mid \sum_{i \in I} w_i.P_i))$$

The syntax $(i)P_i$ will be used if no confusion is possible.

10.3 Datatypes

Let $\mathsf{data}\ T = C_1^{m_1} \mid \cdots \mid C_n^{m_n}$.

$$\langle C_i v_1 \cdots v_{m_i} \rangle = \mathsf{new}\, v\, \langle v \rangle.(!v(w_1 \cdots w_n).\overline{w_i}\langle v_1 \cdots v_{m_i} \rangle)$$

Let $w_1, \ldots, w_n$ not be used in $P_1, \ldots, P_n$.

$$\left(\mathsf{case}\ x\ \mathsf{of}\ \begin{cases} C_1 v_1^1 \cdots v_{m_1}^1 & \Rightarrow P_1 \\ \vdots & \\ C_n v_1^n \cdots v_{m_n}^n & \Rightarrow P_n \end{cases} \right) = \mathsf{new}\, w_1 \cdots w_n\, \overline{x}\langle w_1 \cdots w_n \rangle. \sum_i w_i(v_1^i \cdots v_{m_i}^i).P_i$$

Some Datatypes

$$\mathsf{data}\ \mathrm{nat} = \mathsf{Zero} \mid \mathsf{Succ}\ \mathrm{nat}$$

Equality on Pure Datatypes A datatype T_i is pure if there is a set $S = \{T_1, \ldots, T_n\}$ of datatypes such that the types of all parameters of all constructors of all datatypes in S are in S.

$$P[v_1 = v_2] = \mathsf{new}\, eq_1 \cdots eq_n\, (Q_1 \mid \cdots \mid Q_n \mid (b \Leftarrow \overline{eq_i}\langle v_1 v_2 \rangle;\, P[b]))$$

with

$$Q_i = !eq_i(v_1 v_2 r).R_i$$

We shall define R_i by example. Suppose $T_i = \mathrm{nat}$.

$$R_i = \mathsf{case}\ v_1\ \mathsf{of}\ \begin{cases} \mathsf{Zero} & \Rightarrow \mathsf{case}\ v_2\ \mathsf{of}\ \begin{cases} \mathsf{Zero} & \Rightarrow \overline{r}\langle \mathtt{true} \rangle \\ \mathsf{Succ}\ w_2 & \Rightarrow \overline{r}\langle \mathtt{false} \rangle \end{cases} \\ \mathsf{Succ}\ w_1 & \Rightarrow \mathsf{case}\ v_2\ \mathsf{of}\ \begin{cases} \mathsf{Zero} & \Rightarrow \overline{r}\langle \mathtt{false} \rangle \\ \mathsf{Succ}\ w_2 & \Rightarrow (b \Leftarrow eq_i\langle w_1 w_2 \rangle;\, \overline{r}\langle b \rangle) \end{cases} \end{cases}$$

10.4 Structures

Let $\mathsf{struct}\ T = \{l_1 \cdots l_n\}$.

$$\langle\{l_1 \leftarrow v_1, \ldots, l_n \leftarrow v_n\}\rangle = \mathsf{new}\, v\, \langle v\rangle.(!\overline{v}\langle v_1 \cdots v_n\rangle)$$

$$P[v.l_i] = v(v_1 \cdots v_n).P[v_i]$$

10.5 Variables

A variable x with initial value v is introduced as follows: $\mathsf{new}\, x\, (\overline{x}\langle v\rangle.P)$.

$$(v \leftarrow [x];\, P) = x(v).(\overline{x}\langle v\rangle \mid P)$$

Let $w \notin \mathrm{fn}(P, v, x)$.

$$([x] \leftarrow v;\, P) = x(w).(\overline{x}\langle v\rangle \mid P)$$