Java Jr.: Fully Abstract Trace Semantics for a Core Java Language

Alan Jeffrey[1,2,⋆] and Julian Rathke[3]

[1] Bell Labs, Lucent Technologies, Chicago, IL, USA
[2] DePaul University, Chicago, IL, USA
[3] University of Sussex, Brighton, UK

Abstract. We introduce an expressive yet semantically clean core Java-like language, Java Jr., and provide it with a formal operational semantics based on traces of observable actions which represent interaction across package boundaries. A detailed example based on the Observer Pattern is used to demonstrate the intuitive character of the semantic model. We also show that our semantic trace equivalence is fully-abstract with respect to a natural notion of testing equivalence for object systems. This is the first such result for a full class-based OO-language with inheritance.

1 Introduction

Operational semantics as a modelling tool for program behaviour originated in the early 1960s in early work of McCarthy [19] and found some popularity in modelling programming languages such as ALGOL and LISP [20, 30, 13] and the lambda-calculus, [18]. Later, this approach to modelling was championed by Plotkin [25, 26] and has since been applied extensively and successfully for providing semantic descriptions of simple programming languages and computational models [31, 21, 22, 15, 29, 1, 11, 8]. As these modelling techniques began to be applied to to larger scale languages, their semantic descriptions became more complex [27, 28, 4, 9, 6, 3].

There has been a considerable research effort towards formalising operational behaviour of Java and Java-like languages, for example [4, 11, 16, 6, 9, 24, 3]. Indeed [3] is a special volume journal dedicated to semantic techniques for the Java language which collects together much of the interesting work on this topic to date. None of these, however, address the issue of program equivalence and extensional descriptions of object behaviour. The papers cited above tend to analyse subsets of the Java language for issues related to type safety rather than equivalence. Banerjee and Naumann [5] provide a denotational semantics for a subset of Java, but do not prove a correspondence with an operational model. With this in mind we propose an experimental class-based Java-like language, designed to have a straightforward semantic description of the interactive behaviour of its programs. We call this language Java Jr.

⋆ This material is based upon work supported by the National Science Foundation under Grant No. 0430175.

M. Sagiv (Ed.): ESOP 2005, LNCS 3444, pp. 423–438, 2005.

To address the issue of program equivalence in Java Jr. we make use of Morris' theory of testing [23], later refined by Hennessy [15]. It is a robust theory based on observability of basic events during computation in context and has been applied to many languages, including models of functional programming such as the λ-calculus [2] and PCF [25], concurrent languages such as the π-calculus [10], and object-oriented languages such as the σ-calculus [17].

The definition of testing equivalence involves a universal quantification over all possible test harnesses for programs, which often makes establishing equivalences difficult. For this reason it is commonplace to investigate alternative characterisations of semantic equivalence which offer simpler proof techniques. We provide an alternative characterisation of testing equivalence in Java Jr. by describing sequences of interactions which programs may engage in with arbitrary test harnesses. These are defined as traces derived from a labelled transition system [7]. The key result we prove is that programs which exhibit the same set of traces are exactly those which are testing equivalent. This property of our trace model is known as full abstraction.

For the remainder of the paper, we will present an overview of the Java Jr. language followed by its formal syntax and operational model. We will then discuss issues of typeability and how well-typed object components can grouped together to form larger, well-typed systems. In Section 4, we define our notion of testing equivalence for Java Jr. and in the following section introduce the trace model. The full abstraction result is outlined in Section 6 and we then close with remarks about future work.

2 The Java Jr. Language

Java Jr. is a small, single threaded, subset of the Java language which allows for the declaration of classes and interfaces in packages. It includes two extensions of Java: it allows for packages to contain object declarations (rather than requiring them to be static fields inside classes), and it allows for explicit specification of the signature of a package. We shall discuss these in more detail below.

An example Java Jr. program is given in Figure 1: it provides a simple implementation of the *Observer* pattern from [12]. Observer objects can register themselves with a Subject (in this case, we just provide a singleton Subject), and any calls to `notify` on the Subject result in `update` calls on all the registered Observers.

We will use the following terminology throughout this paper: a *package* consists of a sequence of *declarations* and a *component* consists of a sequence of packages. We use the metavariables C, P and D to represent components, packages and declarations respectively. We will also use the overbar notation to denote sequences, for example $\bar{P}$ refers to a sequence of packages. The metavariable v is used throughout the paper to refer to a fully specified object reference including the package name and object identity, using the usual Java $p.o$ syntax. We also use the metavariable t to refer to types of the language, that is, fully specified class or interface names. For example, in Figure 1, `observer.singleton` is a fully specified object reference, and `observer.Subject` is a fully specified interface name.

The notion of packages are central to Java Jr. They delimit our semantic descriptions by identifying the boundaries of observable interactions. Statically, only interfaces and

```
{ package observer;
  interface Subject extends ε {
    System.void addObserver (observer.Observer o);
    System.void notify ();
  }
  interface Observer extends ε {
    System.void update ();
  }
  class SubjectImpl extends Object implements Subject {
    observer.List contents;
    SubjectImpl (observer.List contents) { super (); this.contents = contents; }
    public System.void addObserver (observer.Observer o) {
      return (this.contents = new observer.Cons (o, this.contents), System.unit);
    }
    public System.void notify () { return this.contents.updateAll (); }
  }
  class List extends Object implements ε {
    observer.List () { super (); }
    public System.void updateAll () { return System.unit; }
  }
  class Cons extends List {
    observer.Observer hd; observer.List tl;
    Cons (observer.Observer hd, observer.List tl) { super (); this.hd = hd; this.tl = tl; }
    public System.void updateAll () { return (this.hd.update(), this.tl.updateAll ()); }
  }
  object observer.SubjectImpl singleton implements observer.Subject {
    contents = observer.list_nil;
  }
  object observer.List list_nil implements ε { }
}
```

Fig. 1. Definition of the `observer` package in Java Jr.

public objects (and not classes or private objects) are visible across package boundaries; dynamically, only publicly visible method calls (and not fields, constructors, or private methods) are visible across package boundaries. In particular, code placed in a package p, cannot create instances of objects using classes in a different package q. Nor can this code access fields of objects created in q directly. In line with software engineering good practice, each of these operations must be provided by factory, accessor and mutator methods. Moreover, all packages in Java Jr. are *sealed*, that is new classes, objects and interfaces may not be added to existing packages.

Where Java Jr. differs significantly from Java is in the provision for statically available methods and members. Rather than modelling the intricacies of Java's `static` modifier, we allow packages to contain explicit object declarations of the form:

$$\mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\{f_1 = v_1; , \ldots , f_n = v_n;\}$$

Such a declaration indicates that an object with identity o is an instance of class t with initial field assignments $f_i = v_i$; which may change during program execution. Object declarations also contain a list of interface types $\bar{t}$ which the object is said to *implement*. These are the externally visible types for the object, as opposed to the class name t, which is only internally visible within the package. If the list of interface types is empty, then the object is considered private to the package. For example, in Figure 1 we have:

```
{ package observer;
  interface Subject extends ε {
    System.void addObserver (observer.Observer o);
    System.void notify ();
  }
  interface Observer {
    System.void update ();
  }
  extern observer.Subject singleton;
}
```

Fig. 2. External view of of the `observer` package

- Object `singleton` is declared as having class `SubjectImpl`, and implementing `Subject`, so within the `observer` package we have `singleton:SubjectImpl` but externally we only have `singleton:Subject`.
- Object `list_nil` implements no interfaces, so within the `observer` package we have `list_nil:List` but externally it is inaccessible.

In Java, all packages are *export* packages, that is they contain both the signature of the package and its implementation: in contrast, languages like C allow for importation of externally defined entities, and for the importer to give the signature of the imported entity. In defining a notion of equivalence for Java programs, we found it necessary to be formal about the notion of package interface, since the external behaviour of a package crucially depends on the types of external entities.

For this reason, our other extension of Java is to allow for *import* packages, which do not contain class or object declarations, and instead only contain interface declarations and *extern* declarations, of the form:

$$\mathsf{extern}\ \bar{t}\ o;$$

Such an declaration within an import package p declares that any export package which implements p must provide an object named o with public types $\bar{t}$. For example, in Figure 2 we give the *external view* of the `observer` package.

2.1 Formal Syntax and Semantics of Java Jr.

We present a formal grammar for the Java Jr. language in Figure 3. For the most part this syntax is imported directly from Java.

The only novel Java Jr. expression is of the form $E\ \mathsf{in}\ p$ which has no effect upon runtime behaviour but is used simply as an annotation to assist typechecking. This operator is effectively a type coercion of the following form:

> If the expression E is well-typed to run in package p with return type t, then the expression $E\ \mathsf{in}\ p$ is well-typed to run in any package q with return type t, as long as t is a visible type in q.

In order to present the dynamic and static semantics of our language we found it useful to make recourse to a number of auxiliary, syntactically defined functions. The definitions of these are largely obvious and are too numerous to list here. One of the most important

Components:	$C ::= \bar{P}$
Packages:	$P ::= \{\mathsf{package}\ p; \bar{D}\}$
Declarations:	$D ::= \mathsf{class}\ c\ \mathsf{extends}\ t\ \mathsf{implements}\ \bar{t}\,\{K\,\bar{G}\,\bar{M}\}$
	$\mid\ \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}$
	$\mid\ \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}$
	$\mid\ \mathsf{extern}\ \bar{t}\ o; \quad (\bar{t} \neq \varepsilon)$
Constructors:	$K ::= c\,(\bar{t}\ \bar{f}, \bar{u}\ \bar{g})\,\{\mathsf{super}\,(\bar{f}); \mathsf{this}.\bar{g} = \bar{g};\}$
Fields:	$F ::= f = v;$
Field types:	$G ::= t\ f;$
Methods:	$M ::= \mathsf{public}\ t\ m\,(\bar{t}\ \bar{x})\,\{\mathsf{return}\ E;\}$
Method types:	$N ::= t\ m\,(\bar{t}\ \bar{x});$
Expressions:	$E ::= v \mid x \mid E.m(\bar{E}) \mid E.f \mid E.f = E$
	$\mid\ \mathsf{new}\ t\,(\bar{E}) \mid (E == E\,?\,E : E) \mid E, E \mid E\ \mathsf{in}\ p$
Compound names:	$p, \ldots, w ::= \bar{a}$

Simple names range over Object, and $a, \ldots, o$ and Variables range over this and $x, \ldots, z$. We also assume that sequences of field identifiers and variables, $\bar{f}$ and $\bar{x}$, and names in $\bar{P}, \bar{D}, \bar{F}, \bar{G}, \bar{M}, \bar{N}$ are always pairwise distinct.

Fig. 3. Syntax of the Java Jr. language

of these is the updating function $C + C'$ which is an asymmetric operator in which each declaration $\{\mathsf{package}\ p; D\}$ within C' overrides any declaration with the same full name present in C, is included in package p of C if C contains this package, and is simply appended to C otherwise. We write $C.p.n$ for the declaration $\{\mathsf{package}\ p; D\}$ where package p in C declares D with name n. Another crucial definition is

- $C.p$ is an *export package* if there is a n such that $C.p.n = \{\mathsf{package}\ p; D\}$ where D is either a class or an object declaration.
- $C.p$ is an *import package* if it is not an export package.

2.2 Dynamic Semantics

A Java Jr. component C, will exhibit no dynamic behaviour until a thread of execution is provided. As Java Jr. is a single threaded language we need not concern ourselves with thread identities and synchronisation and we may model the single active thread simply by a Java Jr. expression E. Given this, it is not difficult to define a relation $\rightarrow$ of the form

$$(C \vdash E) \rightarrow (C' \vdash E')$$

to model the evaluation of the thread E with respect to the component C. In order to define the reduction relation it is useful to identify what is typically referred to as *evaluation contexts* [32]. The grammar of all possible evaluation contexts of the language is given by

$$\begin{aligned}\mathcal{E} ::= {} & \cdot \mid \mathcal{E}.m(\bar{E}) \mid v.m(\bar{v}, \mathcal{E}, \bar{E}) \mid \mathcal{E}.f \mid \mathcal{E}.f = E \mid v.f = \mathcal{E} \mid \mathsf{new}\ t\,(\bar{v}, \mathcal{E}, \bar{E}) \\ & \mid (\mathcal{E} == E\,?\,E_T : E_F) \mid (v == \mathcal{E}\,?\,E_T : E_F) \mid \mathcal{E}, E \mid \mathcal{E}\ \mathsf{in}\ p\end{aligned}$$

We also list, in Figure 4, the proof rules which define the reduction relation itself. For the most part, these rules are reasonably straightforward. Two points of interest are:

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad \mathsf{public}\ u\ m\,(\bar{u}\ \bar{x})\,\{\mathsf{return}\ E;\} \in C.t.\mathsf{methods}}{(C \vdash \mathcal{E}[v.m\,(\bar{v})]) \to (C \vdash \mathcal{E}[E[v/\mathsf{this}, \bar{v}/\bar{x}]\ \mathsf{in}\ p])}$$

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad f = w; \in \bar{F}}{(C \vdash \mathcal{E}[v.f]) \to (C \vdash \mathcal{E}[w])}$$

$$\frac{C.v = \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}\}\} \quad (f = u;) \in \bar{F} \quad C' = C + \{\mathsf{package}\ p; \mathsf{object}\ t\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}'\}\} \quad \bar{F}' = \bar{F} + (f = w;)}{(C \vdash \mathcal{E}[v.f = w]) \to (C' \vdash \mathcal{E}[w])}$$

$$\frac{C.p.c.\mathsf{fields} = \bar{t}\ \bar{f}; \quad p.o \notin \mathsf{dom}\,(C) \quad C' = C + \{\mathsf{package}\ p; \mathsf{object}\ p.c\ o\ \mathsf{implements}\ \varepsilon\,\{\bar{f} = \bar{v};\}\}}{(C \vdash \mathcal{E}[\mathsf{new}\ p.c\,(\bar{v})]) \to (C' \vdash \mathcal{E}[p.o])} \qquad \frac{}{(C \vdash \mathcal{E}[v\ \mathsf{in}\ p]) \to (C \vdash \mathcal{E}[v])}$$

$$\frac{}{(C \vdash \mathcal{E}[(v == v\,?\,E : E')]) \to (C \vdash \mathcal{E}[E])} \qquad \frac{v \neq w}{(C \vdash \mathcal{E}[(v == w\,?\,E : E')]) \to (C \vdash \mathcal{E}[E'])}$$

Fig. 4. Rules for reductions $(C \vdash E) \to (C' \vdash E')$

- In the rule for generating new objects, the new object is always stored within the same package as the class it is instantiating.
- The result of a method call is to inline the method body E, say, within the current evaluation context. Note that before doing this E is wrapped with the coercion $E\ \mathsf{in}\ p$ where p is the package of the receiver. This facilitates type-safe embedding of external code within a package at runtime.

Note that the statically defined component C is modified during reduction as it also models the runtime heap as well as the program class table.

2.3 Static Semantics

As with Java itself, Java Jr. is a statically typed class-based language. It uses the package mechanism to enforce *visibility*: in Java Jr., classes are always package protected, and interfaces are always public, conforming to the common discipline of *programming to an interface*. In order to check that a Java Jr. program respects package visibility, the type system tracks the current package of each class, method and expression, for example the type judgement for an expression is:

$$C \vdash E : t\ \mathsf{in}\ p$$

This indicates that the expression E could potentially access all protected fields and methods in p but cannot access anything outside of p except public methods declared in interfaces.

We close this section by confirming that Java Jr. satisfies Subject Reduction for the runtime type system.

Proposition 1 (Subject Reduction). *For any well-typed component* $\vdash C : \mathsf{component}$ *such that* $C \vdash E : t$ in p *and* $(C \vdash E) \rightarrow^* (C' \vdash E')$ *we have that* $\vdash C' : \mathsf{component}$ *and* $C' \vdash E' : t$ in p.

3 Linking and Compatibility

A fundamental property of components ought to be that they should be compositional: it should be possible to replace a subcomponent with an an equivalent subcomponent without affecting the whole system. In Section 4 we will discuss the dynamic properties of equivalence, and in this section we will discuss the static properties. Our goal is to provide a characterisation for when we can replace a subcomponent of a well-typed system and ensure that the new system is still well-typed.

When first need to discuss what *linking* means in the context of Java Jr. Consider two components C_1, which contains an import package p, and C_2, which contains an export package p. As long as C_1 and C_2 are *linkable*, we should be able to find a component $C_1 \between C_2$ where C_1's import of p is satisfied by C_2's export.

We can now define when it is possible to link two declarations. Declarations D_1 and D_2 are *linkable* if one of the following cases holds:

- D_1 is object $t\ o$ implements $\bar{t}\{\bar{F}\}$ and D_2 is extern $\bar{t}\ o$;
- D_2 is object $t\ o$ implements $\bar{t}\{\bar{F}\}$ and D_1 is extern $\bar{t}\ o$;
- $D_1 = D_2$ and are interface or extern declarations.

We define when it is possible to link two packages of the same name. Given packages P_1 and P_2 we say that these are *linkable* if one of the following cases holds:

- P_1 is an export package and P_2 is an import package, and for each v such that $P_2.v = \{\mathsf{package}\ p; D_2\}$ we have that $P_1.v = \{\mathsf{package}\ p; D_1\}$ where D_1 and D_2 are linkable.
- Symmetrically, when P_1 is an import package and P_2 is an export package.
- P_1 and P_2 are both import packages, and for each v such that $P_1.v = \{\mathsf{package}\ p; D_1\}$ and $P_2.v = \{\mathsf{package}\ p; D_2\}$ we have that $D_1 = D_2$.

We define when it is possible to link two components: C_1 and C_2 are *linkable* if

- for any $P_1 \in C_1$ and $P_2 \in C_2$ with $\mathsf{name}(P_1) = \mathsf{name}(P_2)$ we have that P_1 and P_2 are linkable.

The above definitions outline the formal requirements for two components C_1 and C_2 to be linked to form the larger component $C_1 \between C_2$ given by:

$$C_1 \between C_2 = (C_1.\mathsf{imports} + C_2.\mathsf{imports}) + (C_1.\mathsf{exports} + C_2.\mathsf{exports})$$

where $C.\mathsf{exports}$ is the component containing all of the export packages of C, and similarly for $C.\mathsf{imports}$.

Proposition 2. *If* $\vdash C_1 : \mathsf{component}$ *and* $\vdash C_2 : \mathsf{component}$ *and* C_1 *and* C_2 *are linkable then* $\vdash C_1 \between C_2 : \mathsf{component}$.

We can now address our goal of providing a characterisation for when we can replace a subcomponent C_1 of a well-typed system $C_1 \between C$ by a replacement component C_2 and be sure that $C_2 \between C$ is still well-typed. We shall call such components C_1 and C_2 *compatible*, defined as:

$$\begin{array}{c}\text{for all } C,\ C \text{ and } C_1 \text{ are linkable}\\ \text{if and only if } C \text{ and } C_2 \text{ are linkable}\end{array}$$

This definition, although appealing for its intuitive character, may be a little intractable due to the use of the quantification over all components C. For this reason we seek to provide a direct syntactic characterisation of compatibility. Two components C_1 and C_2 are *interface compatible* when:

$$\begin{array}{c}\text{for all } t,\ C_1.t = \{\mathsf{package}\ p; \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}\}\\ \text{if and only if } C_2.t = \{\mathsf{package}\ p; \mathsf{interface}\ i\ \mathsf{extends}\ \bar{t}\,\{\bar{N}\}\}\end{array}$$

Two components C_1 and C_2 are *extern compatible* when:

$$\begin{array}{c}\text{for all } v,\ C_1.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\}\\ \text{if and only if } C_2.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\}\end{array}$$

Two components C_1 and C_2 are *object compatible* when:

$$\begin{array}{c}\text{for all } v \text{ and } \bar{t} \neq \varepsilon,\ C_1.v = \{\mathsf{package}\ p; \mathsf{object}\ t_1\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}_1\}\}\\ \text{if and only if } C_2.v = \{\mathsf{package}\ p; \mathsf{object}\ t_2\ o\ \mathsf{implements}\ \bar{t}\,\{\bar{F}_2\}\}\end{array}$$

Two components C_1 and C_2 are *package compatible* when:

$$\begin{array}{c}\text{for all } p,\ p \in \mathsf{dom}\,(C_1) \text{ and } C_1.p \text{ is an export package}\\ \text{if and only if } p \in \mathsf{dom}\,(C_2) \text{ and } C_2.p \text{ is an export package}\end{array}$$

For readers familiar with Java's notion of *binary compatibility* [14–Chapter 13], these are stronger requirements, justified by the following result.

Proposition 3. *Components* $\vdash C_1 : \mathsf{component}$ *and* $\vdash C_2 : \mathsf{component}$ *are compatible if and only if they are interface, extern, object and package compatible.*

4 Contextual Equivalence

The question of whether two programs are equal lies at the heart of semantics. An initial requirement for equivalence clearly should be that the programs, or components, are at least compatible. Further to this, we adopt an established means of defining equivalence by making use of contextual testing [15, 23]; programs are considered equal when they pass exactly the same tests.

In the case of Java Jr., a test is any component which can be linked against the component being tested, and the resulting system passes a test by printing an appropriate message using a chosen method `System.out.print(Object)`. The remainder of this section will now formalise this notion of testing.

Define a special component System as:

```
{ package System;
  interface Output { Object print(Object msg); }
  extern System.Output out;
}
```

We say that a component C *accepts* System if C and System are linkable and C.`System` is not an export package. Note that if $\texttt{System} \notin \mathsf{dom}(C)$ then C trivially accepts System. For compatible components $\vdash C_1 : \mathsf{component}$ and $\vdash C_2 : \mathsf{component}$ which accept System, define $C_1 \lesssim C_2$ as:

for all $\vdash C : \mathsf{component}$ linkable with C_1 and $C \between C_1 \vdash E : \mathsf{Object\ in} * $ and for all $C \vdash v : \mathsf{Object\ in} *$ we have

$$(C \between C_1 \vdash E) \rightarrow^* (C_1' \vdash \mathcal{E}_1[\texttt{System.out.print}(v)]) \quad \text{implies}$$
$$(C \between C_2 \vdash E) \rightarrow^* (C_2' \vdash \mathcal{E}_2[\texttt{System.out.print}(v)])$$

We say that well-typed C_1 and C_2 are contextually equivalent, $C_1 \simeq C_2$ whenever both

$$C_1 \lesssim C_2 \qquad \text{and} \qquad C_2 \lesssim C_1 .$$

Although this definition is appealing in the sense of being extensional and robust, it is rather intractable as a means of identifying equivalent programs, due to the quantification over all well-typed components C. We will now establish a simpler trace-based method for establishing contextual equivalence for Java Jr.

5 Trace Semantics

We will now discuss the trace semantics of Java Jr., which provides a description of the external behaviour of a component as a series of method calls and returns. The semantics of a component describes all possible interactions it could engage in with some unknown testing component. Each interaction takes the form of a sequence of basic actions α given by:

$$\gamma ::= v.m(\bar{v}) \mid \mathsf{return}\ v \mid \mathsf{new}(v) . \gamma$$
$$a ::= \gamma? \mid \gamma! \qquad \alpha ::= a \mid \tau$$

Each visible action is either a method call $v.m(\bar{v})$ or method return v. They are decorated $\gamma?$ if the message goes from the environment to the process, or $\gamma!$ if the message comes from the process to the environment. Moreover, actions may mention new objects which have not previously been seen: these are indicated by $\mathsf{new}(v) . \gamma$. The final action τ is used to represent interaction internal to the component under test.

We define *traces* as sequences $\bar{a}$ of visible actions, considered up to *alpha equivalence*, viewing $\mathsf{new}(v) . \bar{a}$ as a binder of v in $\bar{a}$:

$$\bar{a} . \mathsf{new}(\bar{v}) . \bar{b} \equiv \bar{a} . \mathsf{new}(\bar{w}) . \bar{b}[\bar{w}/\bar{v}] \quad \text{when } \bar{w} \notin \bar{b}$$

We will now describe the rules which generate traces from the component syntax. Before we can do this though it is useful to present an auxiliary notion.

The downcasting of imported names $C + \mathsf{extern}\ t\ v;$ is given by:

$$\begin{aligned} C + \mathsf{extern}\ t\ v; &= C \quad (\text{when } C \vdash v : t \text{ in } *) \\ C + \mathsf{extern}\ t\ v; &= C + \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}, t\ o;\} \\ &\qquad (\text{otherwise, where } C.v = \{\mathsf{package}\ p; \mathsf{extern}\ \bar{t}\ o;\} \\ &\qquad \text{and } C.\bar{t}.\mathsf{headers} \cup C.t.\mathsf{headers} \text{ are compatible}) \end{aligned}$$

Similarly, the downcasting of exported names $C + \mathsf{object}\ t\ v;$ is given by:

$$\begin{aligned} C + \mathsf{object}\ t\ v; &= C \quad (\text{when } C \vdash v : t \text{ in } *) \\ C + \mathsf{object}\ t\ v; &= C + \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \bar{t}, t\ \{\bar{F}\}\} \\ &\qquad (\text{otherwise, when } C.v = \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \bar{t}\ \{\bar{F}\}\} \\ &\qquad \text{and } C \vdash u <: t \text{ in } p \text{ and } \bar{t} \neq \varepsilon) \end{aligned}$$

Notice that downcasting with $t\ v$ has no effect in case the object reference v is already known to the component at the (public) type t. Otherwise, the appropriate import or export declaration is updated.

In order to generate traces we need to describe all possible interactions of components with an unknown testing component and unknown thread. We build these interactions up from sequences of single basic actions which the component can engage in. There are essentially two modes of interaction we need to consider here. One is the situation in which the unknown testing component and thread is executing in its code and may call in to a method of the component under test. The other is the situation in which the component under test has been called and is executing some of its known code. We represent these two scenarios using the following *states*:

$$\Sigma ::= (C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \mid (C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})$$

where block represents unknown code being executed by the testing environment and $\bar{\mathcal{E}}$ represents the component C's view of the evaluation stack. In fact this stack is formed from a *sequence* of evaluation contexts as the view of the full evaluation stack is only partial. The types of these evaluation contexts is also recorded and uses the notation $\mathcal{E} : t \to u$ to indicate that the hole in $\mathcal{E}$ is to be filled with an expression of type t, and doing so will yield an expression of type u.

We now define a relation $\Sigma \stackrel{\bar{b}}{\Longrightarrow} \Sigma'$ between (well-typed) states which describes the sequences of actions a component can engage in. The defining rules for this relation are presented in Figure 5. From here we are now in a position to define the semantics of a component as

$$\mathsf{Traces}(C) = \{\bar{a} \mid (C \vdash \mathsf{block} \rhd \varepsilon : \varepsilon) \stackrel{\bar{b}}{\Longrightarrow} \Sigma \text{ and } \bar{a} \equiv \bar{b}\}$$

In Figure 6, we show an example of our Observer example above. We define a component Test which contains (an external declaration of) an object which will be registered with the observer service:

```
{ package observer.test;
  interface Test { void run (); }
  extern observer.test.Test test;
}
```

$$\frac{(C \vdash E) \to (C' \vdash E')}{(C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\tau} (C' \vdash E' : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

Silent transitions

$$\frac{C.v \text{ is an export} \quad C \vdash v : u \text{ in } * \quad s\, m\,(\bar{s}\,\bar{x}); \in C.u.\mathsf{headers} \quad C' = C + \mathsf{extern}\ \bar{s}\ \bar{v};}{(C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{v.m\,(\bar{v})?} (C' \vdash v.m\,(\bar{v}) : s \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

$$\frac{C' = C + \mathsf{extern}\ t\ v;}{(C \vdash \mathsf{block} \rhd \mathcal{E}, \bar{\mathcal{E}} : t \to u, \bar{t} \to \bar{u}) \xrightarrow{\mathsf{return}\ v?} (C' \vdash \mathcal{E}[v] : u \rhd \bar{\mathcal{E}} : \bar{t})}$$

Input transitions

$$\frac{C.v \text{ is an import} \quad C \vdash v : u \text{ in } * \quad s\, m\,(\bar{s}\,\bar{x}); \in C.u.\mathsf{headers} \quad C' = C + \mathsf{object}\ \bar{s}\ \bar{v};}{(C \vdash \mathcal{E}[v.m\,(\bar{v})] : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{v.m\,(\bar{v})!} (C' \vdash \mathsf{block} \rhd \mathcal{E}, \bar{\mathcal{E}} : s \to t, \bar{t} \to \bar{u})}$$

$$\frac{C' = C + \mathsf{object}\ t\ v;}{(C \vdash v : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{return}\ v!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u})}$$

Output transitions

$$\frac{\begin{array}{c} C.p \text{ is an import package} \quad p.o \notin \mathsf{dom}\,(C) \quad p.o \in \mathsf{fn}\,(\gamma?) \\ C'' = C + \{\mathsf{package}\ p; \mathsf{extern}\ \mathsf{Object}\ o;\} \\ (C'' \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\gamma?} (C' \vdash E' : t' \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}') \end{array}}{(C \vdash \mathsf{block} \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{new}(p.o).\gamma?} (C' \vdash E' : t' \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}')}$$

$$\frac{\begin{array}{c} C.p.o = \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \varepsilon\,\{\bar{F}\}\} \quad p.o \in \mathsf{fn}\,(\gamma!) \\ C'' = C + \{\mathsf{package}\ p; \mathsf{object}\ u\ o\ \mathsf{implements}\ \mathsf{Object}\,\{\bar{F}\}\} \\ (C'' \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\gamma!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}') \end{array}}{(C \vdash E : t \rhd \bar{\mathcal{E}} : \bar{t} \to \bar{u}) \xrightarrow{\mathsf{new}(p.o).\gamma!} (C' \vdash \mathsf{block} \rhd \bar{\mathcal{E}}' : \bar{t}' \to \bar{u}')}$$

Fresh name transitions

$$\frac{}{\Sigma \xRightarrow{\varepsilon} \Sigma} \qquad \frac{\Sigma \xRightarrow{\bar{a}} \Sigma' \xRightarrow{\bar{a}'} \Sigma''}{\Sigma \xRightarrow{\bar{a}\,\bar{a}'} \Sigma''} \qquad \frac{\Sigma \xrightarrow{\tau} \Sigma'}{\Sigma \xRightarrow{\varepsilon} \Sigma'} \qquad \frac{\Sigma \xrightarrow{a} \Sigma'}{\Sigma \xRightarrow{a} \Sigma'}$$

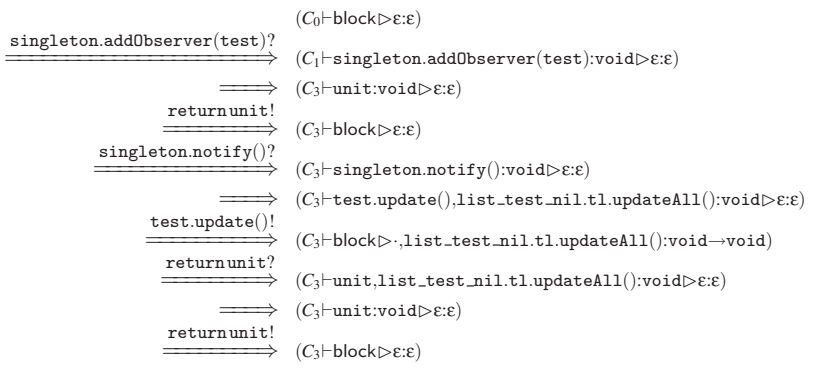
Concatenating actions

Fig. 5. Generating rules for labelled transitions

Note that during this example the type of the `test` object changes: initially it is just `observer.test.Test`, but after the first action it also has type `observer.Observer`.

6 Full Abstraction

Having built our trace model of components we now need to verify that the notion of equivalence induced by the model (equality on trace sets), does actually coincide with the intuitive notion of testing equivalence $\simeq$ defined earlier. This is the content of the Full Abstraction Theorem.



C_0 = Components System, Test and that defined in Figure 1.
$C_1 = C_0 +$ {package observer.test; extern Test, Observer test; }
$C_2 = C_1 +$ {package observer; object Cons list_test_nil { hd=test; tl=list_nil; }}
$C_3 = C_2 +$ {package observer; object SubjectImpl singleton ... { contents = list_test_nil; }}

Fig. 6. Example trace of `observer`

Theorem 1 (Full Abstraction). *For all well-typed components C_1 and C_2, we have*

$$\mathsf{Traces}(C_1) = \mathsf{Traces}(C_2) \quad \textit{if and only if} \quad C_1 \simeq C_2$$

The proof of this theorem is non-trivial and unfortunately too long to present in full here. It breaks in to two parts, soundness and completeness, which together are sufficient to show full abstraction. For the remainder of this section we will outline the proof technique.

6.1 Soundness of Trace Inclusion for Testing

We show the soundness of our model with respect to testing equivalence, that is,

$$\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2) \quad \text{implies} \quad C_1 \lesssim C_2.$$

To demonstrate this we observe that the traces are defined in such a way as to guarantee that every (internal) interaction between C_i and a test component can be decomposed in to complementary traces (traces which are identical except for the reversal of the ! and ? annotations) and conversely that pairs of complementary traces can also be (re)composed to obtain an internal reduction. This is a not a straightforward property to express as the result of an active thread in C_1 and C is typically an intertwining of code from each component. To describe the possible states of the system we overload the notation $\between$ to define how to *merge* complementary states of the labelled transition system.

Define $\bar{t}_1 \to \bar{u}_1$ and $\bar{t}_2 \to \bar{u}_2$ to be *mergeable* for t as:

- ε and ε are mergeable for Object.
- If $\bar{t}_2 \to \bar{u}_2$ and $\bar{t}_1 \to \bar{u}_1$ are mergeable for u
  then $\bar{t}_1 \to \bar{u}_1$ and $t \to u, \bar{t}_2 \to \bar{u}_2$ are mergeable for t.

Define $(C_1 \vdash E : t \rhd \bar{\mathcal{E}}_1 : \bar{t}_1 \to \bar{u}_1)$ and $(C_2 \vdash \mathsf{block} \rhd \bar{\mathcal{E}}_2 : \bar{t}_2 \to \bar{u}_2)$ are *mergeable* when

- C_1 and C_2 are linkable and $\bar{t}_1 \to \bar{u}_1$ and $\bar{t}_2 \to \bar{u}_2$ are mergeable for t

We define the partial merge $\bar{\mathcal{E}}_1 \between \bar{\mathcal{E}}_2$ of context stacks as

$$\varepsilon \between \varepsilon = \cdot$$
$$\bar{\mathcal{E}}_1 \between (\mathcal{E}_2, \bar{\mathcal{E}}_2) = (\bar{\mathcal{E}}_2 \between \bar{\mathcal{E}}_1)[\mathcal{E}_2]$$

When Σ_1 and Σ_2 are mergeable we define $\Sigma_1 \between \Sigma_2$ as the state given by:

$$(C_1 \vdash E_1 : t_1 \rhd \bar{\mathcal{E}}_1 : \bar{t}_1 \to \bar{u}_1) \between (C_2 \vdash \mathsf{block} \rhd \bar{\mathcal{E}}_2 : \bar{t}_2 \to \bar{u}_2) = (C_1 \between C_2 \vdash (\bar{\mathcal{E}}_1 \between \bar{\mathcal{E}}_2)[E_1])$$

Proposition 4. *If* $\vdash \Sigma_1 : \mathsf{state}$ *and* $\vdash \Sigma_2 : \mathsf{state}$ *are mergeable and* $\Sigma_1 \between \Sigma_2 = (C \vdash E)$, *then* $\vdash C : \mathsf{component}$ *and* $C \vdash E : \mathsf{Object\ in} * .$

We write $\bar{a}^{-1}$ for the trace $\bar{a}$ with the input and output annotations reversed. We also define the *external ordering*, $C \sqsubseteq^{ext} C'$ as the preorder on components generated by $C \sqsubseteq^{ext} C + \mathsf{object}\ t\ v;$. Note that whenever $C \sqsubseteq^{ext} C'$ then C' differs only in that it may contain more external interface types for object definitions.

Proposition 5 (Trace Composition/Decomposition). *If* Σ_1 *and* Σ_2 *are mergeable such that* $\Sigma_1 \between \Sigma_2 = (C \vdash E)$ *then*

1. *If* $\Sigma_1 \xrightarrow{\bar{a}} \Sigma_1'$ *and* $\Sigma_2 \xrightarrow{\bar{a}^{-1}} \Sigma_2'$ *then* $(C \vdash E) \to^* (C' \vdash E')$ *where either*
   - Σ_1' *and* Σ_2' *are mergeable such that* $\Sigma_1' \between \Sigma_2' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$, *or*
   - Σ_2' *and* Σ_1' *are mergeable such that* $\Sigma_2' \between \Sigma_1' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$.
2. *If* $(C \vdash E) \to^* (C' \vdash E')$ *then there exists* $\bar{a}$ *such that* $\Sigma_1 \xrightarrow{\bar{a}} \Sigma_1'$ *and* $\Sigma_2 \xrightarrow{\bar{a}^{-1}} \Sigma_2'$ *where either*
   - Σ_1' *and* Σ_2' *are mergeable such that* $\Sigma_1' \between \Sigma_2' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$, *or*
   - Σ_2' *and* Σ_1' *are mergeable such that* $\Sigma_2' \between \Sigma_1' = (C'' \vdash E')$ *with* $C' \sqsubseteq^{ext} C''$.

Theorem 2 (Soundness of traces for may testing). *For compatible* C_1 *and* C_2 *which accept* `System`, *if* $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ *then* $C_1 \lesssim C_2$.

Proof: (Sketch) Suppose that $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ and also suppose that C is a testing component such that $C \between C_1$ prints the message "Hello" during evaluation. We can use Trace Decomposition on the interaction between C and C_1 which caused this string to be produced. This gives a pair of complementary traces. Now, because $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$ we also know that C_2 must perform the same traces as C_1. Therefore, when we link C with C_2, because these components can respectively perform the given pair of complementary traces, these may be re-composed using Trace Composition to give an internal evaluation of $C \between C_2$ which will also print the message "Hello". This can be done for any testing component and any message. So this serves to demonstrate that $C_1 \lesssim C_2$. □

6.2 Completeness

The converse property, completeness,

$$C_1 \lesssim C_2 \quad \text{if and only if} \quad \mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$$

also relies on the Trace Composition/Decomposition property (Proposition 5). In addition to this though we also need to show a definability result which states that for every (odd length) trace from a well-typed component we can find a component and expression which will exhibit this trace, and only this trace (up to renaming of fresh names).

Proposition 6 (Definability). *If we have* $\vdash C : \mathsf{component}$ *and (for* $\bar{a}$ *of odd length)* $(C \vdash \mathsf{block} \rhd \varepsilon : \varepsilon) \stackrel{\bar{a}}{\Longrightarrow} \cdots$ *then we can find* C'*,* E *such that* $\vdash C' : \mathsf{component}$*,* C *and* C' *are linkable,* $C' \vdash E : \mathsf{Object}\ \mathsf{in} * $ *and* $(C' \vdash E : \mathsf{Object} \rhd \varepsilon : \varepsilon) \stackrel{\bar{b}}{\Longrightarrow} \cdots$ *(for* $\bar{b}$ *of odd length) if and only if* $\bar{b} \equiv \leq \bar{a}^{-1}$.

Theorem 3 (Completeness of traces for may testing). *For compatible* C_1 *and* C_2*, if* $C_1 \lesssim C_2$ *then* $\mathsf{Traces}(C_1) \subseteq \mathsf{Traces}(C_2)$.

Proof: (Sketch) Suppose that $C_1 \lesssim C_2$ and also suppose that C_1 has a trace $\bar{a}$. We can suppose (wlog) that $\bar{a}$ is even length. We must show that C_2 also has this trace. We do this by first applying our definability result to the complement of $\bar{a}$ extended with a visible action of an outgoing call to `System.print` with message, "Hello", say. This yields a component `C_def`. Given this, we use Trace Composition with $\bar{a}$ and its complement by linking `C_def` and C_1 to yield an internal evaluation which will ultimately print "Hello". Because, $C_1 \lesssim C_2$ and because `C_def` acts as a test, we know that `C_def` $\between C_2$ must also evaluate and eventually print the message "Hello". We use Trace Decomposition to split this internal evaluation into separate traces to see that C_2 must perform *some* trace complementary to that of `C_def`. But, given that `C_def` performs the unique (up to $\equiv$) trace $\bar{a}^{-1}$, we must have that C_2 has the trace $\bar{a}$ also. □

7 Conclusions and Further Work

We have described a novel core Java-like language, Java Jr., which allows package and class-based definitions of object systems. The language is specifically designed to be semantically clean by using the packaging system to enforce all cross-package interaction to be limited to method invocation and return. We provided Java Jr. with a static type system, whose types act as interfaces to components for building large systems and we presented a simple notion of linking for plugging well-typed components together. This notion of linking was also used to give an extensional definition of a component's suitability for substitution with other components from a structural, or static point of view. We proceeded to develop an extensional definition of a component's suitability for substitution from a behavioural, or dynamic point of view, drawing on a body of work in the literature on process testing [15]. Importantly, our trace semantics for Java Jr. components turn out to capture the notion of behavioural testing precisely.

This Full Abstraction property is a major result of this work and is the first such result for a class-based object language with packages and subtyping.

Although the core language only supports a limited number of Java features, it already has enough power to encode some of the more sophisticated control-flow operations and, to some extent, is robust enough to allow the addition of extra features without disrupting the higher-level semantics, as long as these new features do not create new cross-package interaction. For instance, it is straightforward, but slightly cumbersome to include primitive types and constants in Java Jr. These would not seriously affect the validity of the full abstraction theorem. We also anticipate that Java exceptions could readily be included within our framework, although this would require exception interfaces in addition to Java's exception classes, if we want Java Jr. exceptions to be thrown and caught across package boundaries.

Unfortunately, some of Java's features have significant impact on our model: in particular, explicit downwards typecasts and concurrency. Downwards casts affect our trace semantics in a fairly significant way. At present we maintain strict public interfaces to classes and only release objects at given interfaces. This type security is maintained in Java Jr. as there is no possibility for code to use a received object at any lower type. This is reflected in the trace semantics by recording lists of interface types at which component and environment object names have been leaked. Allowing downwards casts would enable code in a given package to receive objects declared in a different package and discover their private types. This breaks the *programming to an interface* discipline of Java Jr.

Similarly, the trace model is built around the notion of a single thread of control. The straightforward alternation of control between component and environment in the trace semantics is a direct consequence of this. Fortunately, introducing named threads and providing an interleaved trace model is achievable. Earlier work of ours [17] provides such a model for a small concurrent object-based calculus. The extra complications of classes and subtyping present are unlikely to affect the integration of the concurrent thread model within Java Jr.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.
3. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1999.
4. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Proc. Formal Underpinnings of Java*, 1998.
5. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Functional Programming*, 2005. To appear.
6. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
7. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
8. L. Cardelli and A. Gordon. Mobile ambients. In *Proc. Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

9. S. Drossopoulou and S. Eisenbach. Towards an operational semantics and proof of type soundness for Java. In *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer-Verlag, 1999.
10. M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the pi-calculus. In *Proc. IEEE Logic in Computer Science*, page 43. IEEE Computer Society, 1996.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. ACM Principles of Programming Languages*, pages 171–183, 1998.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
13. M. J. C. Gordon. *Experimental Programming Reports*. PhD thesis, School of AI, University of Edinburgh, 1973.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley, 2000.
15. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
16. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
17. A.S.A Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proc. IEEE Logic in Computer Science*, pages 101–112. IEEE Computer Society Press, 2002.
18. P.J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation: Part I. *Communications of the ACM*, 8(2):89–101, 1965.
19. J. McCarthey. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing 1962*, pages 21–28, 1963.
20. J. McCarthy. A formal description of a subset of algol. In *Proc. IFIP WG Formal Language Description Languages for Computer Programming*, pages 1–12. North Holland, 1966.
21. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
22. R. Milner. *Communication and mobile systems: the π-calculus*. Cambridge University Press, 1999.
23. J. H. Morris. *Lambda-calculus models of programming languages*. PhD thesis, MIT, 1968.
24. T. Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In *Proc. Marktobderdorf Summer School*. IOS Press, 2003. To appear.
25. G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
26. G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
27. M. Tofte R. Milner and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
28. J. Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, 1992. Technical Report TR 92-1285.
29. D. Sangiorgi and D. Walker. *The pi-calculus: A Theory of mobile processes*. Cambridge University Press, 2001.
30. T. B. Steel. A formalization of semantics for programming language description. In *Proc. IFIP WG Formal Language Description Languages for Computer Programming*, pages 25–36, 1969.
31. G.J. Sussman and Jr G.L. Steele. Scheme:an interpreter for extended lambda-calculus. Technical Report Memo 349, MIT AI Lab, 1975.
32. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.