Coinductive big-step operational semantics for type soundness of Java-like languages [*]

Davide Ancona
DISI, University of Genova, Italy
davide@disi.unige.it

ABSTRACT

We define a coinductive semantics for a simple Java-like language by simply interpreting coinductively the rules of a standard big-step operational semantics.

We prove that such a semantics is sound w.r.t. the usual small-step operational semantics, and then prove soundness of a conventional nominal type system w.r.t. the coinductive semantics. From these two results, soundness of the type system w.r.t. the small-step semantics can be easily deduced.

This new proposed approach not only opens up new possibilities for proving type soundness, but also provides useful insights on the connection between coinductive big-step operational semantics and type systems.

Categories and Subject Descriptors

D.3.1 [**Programming languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics and meanings of programs**]: Semantics of Programming Languages—*Operational semantics*; F.3.3 [**Logics and meanings of programs**]: Studies of Program Constructs —*Type structure*

General Terms

Languages, Theory

Keywords

coinduction, operational semantics, type soundness, Java

1. INTRODUCTION

It is well known that standard inductive big-step operational semantics are less amenable to prove soundness of type systems than small-step semantics; several important motivations for this statement can be found in the related literature [10, 11]. In this paper we show that this drawback can be overcome when coinductive big-step semantics are considered.

This work is mainly inspired on previous work by Ancona et al. on abstract compilation for object-oriented languages [7, 3, 6, 5, 4], a novel approach which aims at reconciling type analysis and symbolic execution, where programs are compiled into a constraint logic program, and type analysis corresponds to solving a certain goal w.r.t. the coinductive semantics of CLP. In particular, the contribution of this paper w.r.t. abstract compilation is investigating new proof techniques to obtain simpler proofs of soundness for abstract compilation schemes. Another important source of inspiration is the work by Leroy and Grall, whose main purpose is seeking new techniques to be easily automatized for proving the correctness of compilation of high-level functional languages down to lower-level languages.

After defining the standard small-step semantics of our reference language (Section 2), in Section 3 we define a coinductive semantics by simply interpreting coinductively the rules of a standard inductive big-step semantics for the reference language. We prove that such a semantics is sound w.r.t. the usual small-step semantics; this is an interesting result since it allows one to prove soundness of a type system in terms of the coinductive big-step operational semantics, and then directly deduce soundness w.r.t. the small-step semantics. In Section 4 we show this possibility by proving soundness of a conventional nominal type system w.r.t. the coinductive big-step operational semantics. Finally, in Section 5 we outline conclusions and related work.

[*]This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

2. DEFINITION OF THE LANGUAGE

In this section we present our simple Java-like language, which will be used as reference language throughout the paper, together with its standard call-by-value small-step operational semantics (abbreviated with ISS).

The syntax of the language is defined in Figure 1.

The language is a modest variation of Featherweight Java (FJ) [9], where the main differences concern the introduction of conditional expressions and boolean values, and the omission of type casts.

Standard syntactic restrictions are implicitly imposed in the figure. Bars denote sequences of n items, where n is the superscript of the bar and the first index is 1. Sometimes this notation is abused, as in $\overline{f}^h = \overline{e'}^h$; which is a shorthand for $f_1 = e'_1; \ldots f_h = e'_h;$.

A program consists of a sequence of class declarations and a main expression. Types can only be class names and the primitive type **bool**; we assume that the language supports

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{cd}}^n\ e \\
\mathit{cd} & ::= & \textbf{class}\ c_1\ \textbf{extends}\ c_2\ \{\ \overline{\mathit{fd}}^n\ \overline{\mathit{md}}^k\ \}\quad (c_1 \neq \texttt{Object}) \\
\mathit{fd} & ::= & \tau\ f; \\
\mathit{md} & ::= & \tau_0\ m(\overline{\tau\ x}^n)\ \{e\}\quad x_i \neq \texttt{this}\ \forall i = 1..n \\
\tau & ::= & c \mid \textbf{bool} \\
e & ::= & \textbf{new}\ c(\overline{e}^n) \mid x \mid e.f \mid e_0.m(\overline{e}^n) \mid \textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2 \\
 & & \mid \textbf{false} \mid \textbf{true}
\end{array}
$$

Assumptions: $n, k \geq 0$, inheritance is acyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Figure 1: Syntax of the language

boxing conversions, hence *bool* is a subtype[1] of the predefined class `Object`, which is the top type.

A class declaration contains field and method declarations; in contrast with FJ, constructors are not declared, but every class is equipped with an implicit constructor with parameters corresponding to all fields, in the same order as they are inherited and declared. For instance, the classes defined below

```
class P extends Object {bool b; P p;}
class C extends P {C c;}
```

have the following implicit constructors:

```
P(bool b,P p) {super(); this.b=b; this.p=p;}
C(bool b,P p,C c) {super(b,p); this.c=c;}
```

Method declarations are standard; in the body, the target object can be accessed via the implicit parameter `this`, therefore all explicitly declared formal parameters must be different from `this`. Expressions include instance creation, variables, field selection, method invocation, conditional expressions, and boolean literals.

The definition of the conventional small-step operational semantics of the language can be found in Figure 2. We follow the approach of FJ, even though for simplicity we have preferred to restrict the semantics to the deterministic call-by-value evaluation strategy.

Values are either the literals **false** or **true**, or object expressions in normal form having shape **new** $c(\overline{v}^n)$. As happens for FJ, the semantics of object creation is more liberal than the expected one; indeed, **new** $c(\overline{v}^n)$ is always a correct expression which reduces to itself in zero steps, even when class c is not declared, or the number of arguments does not match the number of fields of c. As we will see, the big-step semantics follows a less liberal semantics, more in accordance with the standard semantics of mainstream object-oriented languages.

As usual, the reduction relation $\rightarrow$ should be indexed over the collection of all class declarations contained in the program (called class table), however for brevity we leave implicit such an index in all judgments defined in the paper. The reflexive and transitive closure of $\rightarrow$ is denoted by $\xrightarrow{*}$.

The definition of the standard auxiliary functions *fields* and *meth* is standard [2]. For compactness, such functions provide semantic and type information at once, since they are instrumental for the definition of both the semantics and the type system of the language. Function *fields* returns the list of all fields which are either inherited or declared in the class, in the standard order and with the corresponding declared types. In the case of the predefined class `Object` the returned list is empty (ϵ); field hiding is not supported, hence *fields* is not defined if a class declares a field with the same name of an inherited one. Function *meth* performs standard method look-up: if $\mathit{meth}(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau$, then look-up of method m starting from class c returns the corresponding declaration where $\overline{\tau}^n\ \overline{x}^n$ are the formal parameters with their declared types, and e and τ are the body and the declared returned type, respectively. If $\mathit{meth}(c, m)$ is undefined, then it means that look-up of m from c fails.

In rule (fld), if f_i is a field of the class, then the expression reduces to the corresponding value passed to the implicit constructor. If the selected field is not in such a list, then the evaluation of the expression gets stuck.

In rule (inv), if method look-up succeeds starting from the class of the target object, then the corresponding body is executed, where the implicit parameter `this` and the formal parameters are substituted with the target object and the argument values, respectively. The notation $e_i[\overline{x}^n \mapsto \overline{v}^n]$ denotes parallel substitution of the distinct variables $\overline{x}^n$ with values $\overline{v}^n$ in the expression e.

Rules for conditional expressions (ift) and (iff), and for context closure (ctx) are straightforward. Contexts are the standard ones corresponding to left-to-right, call-by-value strategy.

3. A COINDUCTIVE SEMANTICS

In this section we define a call-by-value coinductive big-step operational semantics (abbreviated with CBS) for our language.

Such a semantics is obtained by simply interpreting coinductively the definition of values and the rules of a pretty standard inductive big-step operational semantics (with no rules for error handling). In Section 4 we prove soundness of the standard nominal type system of the language w.r.t. CBS. The claim of soundness we prove is quite simple: if an expression is well-typed, then it evaluates to a value.

We recall that interpreting recursive definitions or rules coinductively is equivalent to considering also infinite terms and proof trees [7, 4, 11, 15]. The CBS judgment uses value environments (see below), just for uniformity with the type judgment defined in Section 4. Value environments are not strictly necessary, since the rule for method invocation can be equivalently defined with parallel substitution as in ISS; however, it is not difficult to prove that the two semantics are equivalent. Values are separated from expressions, since expressions are always finite terms, whereas values can be also infinite. Such a separation is further stressed by the fact that values belong to a different syntactic category, that is, even finite values are different from expressions.

$$
\mathbb{v}, \mathbb{u} \quad ::= \quad \mathit{obj}(c, [\overline{f}^n \mapsto \overline{\mathbb{v}}^n]) \mid \mathit{false} \mid \mathit{true} \quad (\mathit{coind.\ def.})
$$

If the recursive definition above were interpreted inductively, we would obtain a standard definition of values for our big-step semantics. It is important to recall that **false** and **true** are expressions of our language, and values in ISS (denoted by the meta-variable v), whereas *false* and *true* are not expressions, but just the corresponding values in CBS (denoted by the meta-variable $\mathbb{v}$). Similarly **new** $c(\textbf{true})$ is both an expression and a value in ISS, whereas $\mathit{obj}(c, [f \mapsto \mathit{true}])$ is

[1]Besides mimicking what happens in Java, this assumption ensures the existence of the join operator (least upper bound) between types, without introducing union types. This allows a simpler typing rule for conditional expressions in the type system defined in Section 4.

$$\begin{array}{rcl} v & ::= & \textbf{new } c(\overline{v}^n) \mid \textbf{false} \mid \textbf{true} \\ \mathcal{C}[\,] & ::= & \square \mid \textbf{new } c(\overline{v}^n, \square, \overline{e}^k) \mid \square.f \mid \square.m(\overline{e}^n) \mid v.m(\overline{v}^n, \square, \overline{e}^k) \mid \textbf{if } (\square)\ e_1 \textbf{ else } e_2 \end{array}$$

$$(\text{fld})\ \frac{\mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n, \quad 1 \le i \le n}{\textbf{new } c(\overline{v}^n).f_i \rightarrow v_i}$$

$$(\text{inv})\ \frac{\mathit{meth}(c, m) = \overline{\tau}^n\ \overline{x}^n.e{:}\tau}{\textbf{new } c(\overline{v}^k).m(\overline{v'}^n) \rightarrow e[\texttt{this} \mapsto \textbf{new } c(\overline{v}^k), \overline{x}^n \mapsto \overline{v'}^n]}$$

$$(\text{ift})\ \frac{}{\textbf{if } (\textbf{true})\ e_1 \textbf{ else } e_2 \rightarrow e_1} \qquad (\text{iff})\ \frac{}{\textbf{if } (\textbf{false})\ e_1 \textbf{ else } e_2 \rightarrow e_2} \qquad (\text{ctx})\ \frac{e \rightarrow e'}{\mathcal{C}[e] \rightarrow \mathcal{C}[e']}$$

Figure 2: Call-by-value inductive small-step operational semantics

the corresponding value in CBS (assuming that the only field of c is f), and is not an expression.

Since the definition above is coinductive, object values may be also infinite in CBS. For instance, the value $\mathbb{v}$ defined by the equation

$$\mathbb{v} = \mathit{obj}(\mathit{List}, [\mathit{hd} \mapsto \mathit{obj}(\mathit{Elem}, [\,]), \mathit{tl} \mapsto \mathbb{v}])$$

represents an infinite list; in our language, such a value can only be returned by an infinite computation. Of course in a lazy language this value could be returned also by a terminating expression, or in an imperative language a terminating expression could evaluate into a circular list; however, the important point here is that type correct expressions which do not terminate always return a value in CBS: as explained in case 2 in the second part of this section, without infinite values the claim of soundness proved in Section 4 would not hold.

CBS is defined in Figure 3. Thicker lines manifest that rules are interpreted coinductively. An environment Π is a finite sequence $\overline{x_i}^n \mapsto \overline{\mathbb{v}}^n$, where all variables $\overline{x_i}^n$ are distinct, denoting a finite function mapping variables to values ($\emptyset$ denotes the empty environment, $\mathit{dom}(\Pi)$ the domain of Π). Environments model stack frames of method invocations.

Rules (VAR), (FAL), and (TRU) are straightforward. Evaluation of instance creation (NEW) succeeds only if $\mathit{fields}(c)$ is defined (that is, if c and its ancestors are declared in the program and no field is hidden), and returns a list of fields whose length must coincide with the number of arguments; then all arguments are evaluated and the obtained values are associated with the corresponding fields in the object value. For field selection (FLD) the target expression is evaluated; then evaluation succeeds only if an object value is returned, and the selected field is present in the object value; in this case the corresponding associated value is returned. For method invocation (INV) all expressions denoting the target object and the arguments are evaluated. If the value corresponding to the target is an object of class c, method look-up starting from c succeeds and returns a method declaration with a number of formal parameters coinciding with the number of passed arguments, then the method body is evaluated in the environment where `this` and the formal parameters are associated with their corresponding values. If such an evaluation succeeds, then the returned value is the value of the method invocation. Finally, rules (IFT) and (IFF) are straightforward.

Note that CBS of object creation is less liberal than ISS: as an example, **new** c`()` is a value in ISS, whereas the same expression may not evaluate to a value in CBS; this happens if either c is not declared in the program, or if c contains at least one field.

We have already observed that if the definition of values and the evaluation rules are interpreted inductively, then we obtain a standard inductive big-step operational semantics. Obviously, if an expression evaluates to a value in the inductive semantics, then the same value is obtained in the coinductive one; however, this case concerns terminating expressions, whereas what we do really care about here is the behavior of CBS for non terminating expressions. We show that three different cases may occur. We leave to the reader the easy check that all expressions e considered in the examples below do not terminate w.r.t. ISS, that is, there exists no normal form e' s.t. $e \xrightarrow{*} e'$.

Case 1: There exist many values $\mathbb{v}$ s.t. $\emptyset \vdash e \Rightarrow \mathbb{v}$

Let us consider the expression $e =$ **new** `C().m()`, where `C` is declared as follows:

```
class C extends Object {bool m() {this.m()}}
```

Then $\emptyset \vdash e \Rightarrow \mathbb{v}$ for all values $\mathbb{v}$, as shown in the proof tree of Figure 4. Ellipsis means that such a tree is infinite (hence, it cannot be a valid proof for an inductive system), although regular, that is, it can be folded into a finite graph, because of the repeated finite pattern originated from the judgment $\Pi \vdash$ `this.m()` $\Rightarrow \mathbb{v}$.

$$\dfrac{\dfrac{}{\emptyset \vdash \textbf{new}\ \texttt{C()} \Rightarrow \mathbb{u}} \qquad \dfrac{\dfrac{}{\Pi \vdash \texttt{this} \Rightarrow \mathbb{u}} \quad \dfrac{\vdots}{\Pi \vdash \texttt{this.m()} \Rightarrow \mathbb{v}}}{\Pi \vdash \texttt{this.m()} \Rightarrow \mathbb{v}}}{\emptyset \vdash \textbf{new}\ \texttt{C().m()} \Rightarrow \mathbb{v}}$$

Figure 4: Proof tree for $\emptyset \vdash e \Rightarrow \mathbb{v}$, where $\mathbb{u} = \mathit{obj}(\texttt{C}, [\,])$, $\Pi = \texttt{this} \mapsto \mathbb{u}$

There are also cases where finitely many values are returned. For instance,

$$\emptyset \vdash \textbf{if}\texttt{(}\textbf{new}\ \texttt{C().m())}\ \textbf{true}\ \textbf{else}\ \textbf{false} \Rightarrow \mathit{true}$$
$$\emptyset \vdash \textbf{if}\texttt{(}\textbf{new}\ \texttt{C().m())}\ \textbf{true}\ \textbf{else}\ \textbf{false} \Rightarrow \mathit{false}$$

and no other values can be returned.

Case 2: There exists a unique value $\mathbb{v}$ s.t. $\emptyset \vdash e \Rightarrow \mathbb{v}$

We consider two possible cases, depending on the fact that the returned value is finite or infinite. For both cases we assume that `C` is declared as in case 1. For the former case, the expression **if**`(`**new** `C().m())` **true else true** trivially evaluates to the unique value *true* (although with two different proof trees). For the latter, let us add the following declarations:

```
class P extends Object {H m(){new H(this.m())}}
class H extends P {H h;}
```

$$(\textsc{var})\ \frac{\Pi(x)=\mathbb{v}}{\Pi\vdash x\Rightarrow\mathbb{v}}\qquad(\textsc{fal})\ \frac{}{\Pi\vdash\textbf{false}\Rightarrow\mathit{false}}\qquad(\textsc{tru})\ \frac{}{\Pi\vdash\textbf{true}\Rightarrow\mathit{true}}$$

$$(\textsc{new})\ \frac{\forall\, i=1..n\ \Pi\vdash e_i\Rightarrow\mathbb{v}_i\quad \mathit{fields}(c)=\overline{\tau}^n\ \overline{f}^n}{\Pi\vdash\textbf{new}\ c(\overline{e}^n)\Rightarrow \mathit{obj}(c,[\overline{f}^n\mapsto\overline{\mathbb{v}}^n])}\qquad(\textsc{ift})\ \frac{\Pi\vdash e\Rightarrow\mathit{true}\qquad \Pi\vdash e_1\Rightarrow\mathbb{v}}{\Pi\vdash\textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2\Rightarrow\mathbb{v}}\qquad(\textsc{iff})\ \frac{\Pi\vdash e\Rightarrow\mathit{false}\qquad \Pi\vdash e_2\Rightarrow\mathbb{v}}{\Pi\vdash\textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2\Rightarrow\mathbb{v}}$$

$$(\textsc{fld})\ \frac{\Pi\vdash e\Rightarrow\mathit{obj}(c,[\overline{f}^n\mapsto\overline{\mathbb{v}}^n])\quad 1\le i\le n}{\Pi\vdash e.f_i\Rightarrow\mathbb{v}_i}\qquad(\textsc{inv})\ \frac{\begin{array}{l}\forall\, i=0..n\ \Pi\vdash e_i\Rightarrow\mathbb{v}_i\quad \mathtt{this}\mapsto\mathbb{v}_0,\overline{x}^n\mapsto\overline{\mathbb{v}}^n\vdash e\Rightarrow\mathbb{v}\\ \mathbb{v}_0=\mathit{obj}(c,[\ldots])\quad \mathit{meth}(c,m)=\overline{\tau}^n\ \overline{x}^n.e{:}\tau\end{array}}{\Pi\vdash e_0.m(\overline{e}^n)\Rightarrow\mathbb{v}}$$

Figure 3: Call-by-value coinductive big-step operational semantics

Figure 5 shows the unique proof tree for $\emptyset\vdash\textbf{new}\ \mathtt{P().m()}\Rightarrow \mathit{obj}(\mathtt{H},[\mathtt{h}\mapsto\mathbb{v}])$; ellipsis means that such a tree is infinite, but, again, it can be folded into a finite graph. Indeed, the proof tree must verify the following invariant:

$$\begin{array}{l}\Pi\vdash\textbf{new}\ \mathtt{H(this.m())}\Rightarrow\mathbb{v}\ \text{iff}\\ \Pi\vdash\textbf{new}\ \mathtt{H(this.m())}\Rightarrow\mathit{obj}(\mathtt{H},[\mathtt{h}\mapsto\mathbb{v}])\end{array}$$

Such an invariant does not hold for finite values, but it is verified by the unique infinite (but regular) value $\mathbb{v}$ satisfying the equation $\mathbb{v}=\mathit{obj}(\mathtt{H},[\mathtt{h}\mapsto\mathbb{v}])$.

Note that if rules are interpreted coinductively, but values can only be finite, then the claim proved in Section 4, (that is, any well-typed expression evaluates to a value) does not hold.

Case 3: There exist no values $\mathbb{v}$ s.t. $\emptyset\vdash e\Rightarrow\mathbb{v}$

The expression $\textbf{if}\mathtt{(new\ C().m())}\ \textbf{true}\mathtt{.m()}\ \textbf{else}\ \textbf{true}\mathtt{.m()}$ does not evaluate to any value; this is a direct consequence of the fact that no rules are applicable for the expression $\textbf{true}\mathtt{.m()}$ since $\textbf{true}$ does not evaluate to an object value. The main difference with the previous two cases is that here the expression to be evaluated cannot be typed in any type system insensitive to non termination. In the conventional nominal type system defined in Section 4 all previous examples except for the last one are well-typed.

Soundness of CBS w.r.t. ISS

We show that CBS as defined in Figure 3 is sound w.r.t. ISS as defined in Figure 2. More precisely, if $\emptyset\vdash e\Rightarrow\mathbb{v}$, then in ISS either e diverges (that is, e does not reduce to a normal form), or e reduces in zero or more steps to a value v s.t. $\emptyset\vdash v\Rightarrow\mathbb{v}$. In other words, we are guaranteed that the evaluation of an expression will never get stuck in ISS whenever CBS returns a value for it. CBS can be considered as a sound approximation (or, equivalently, as a sound abstraction) of ISS, therefore it plays the analogous role of a type system; hence, we use the standard technique that proves the progress and subject reduction properties. Clearly, CBS cannot play the role of a reference semantics for the language, because it is non deterministic for some non terminating expressions, and it is not defined for some ill-typed non terminating expressions. Yet, such a semantics is obtained for free by interpreting coinductively a standard big-step operational semantics, and it is useful for proving soundness of type systems, as shown in Section 4. Proving that CBS is sound w.r.t. ISS allows us to factor in two steps the proof that a given type system T is sound w.r.t. ISS: in step 1, T is proved to be sound w.r.t. CBS, while in step 2 CBS is proved to be sound w.r.t. ISS. Of course, step 2 needs to be proved only once, and then can be simply reused. The progress and subject reduction properties can be proved routinely [2], the former by induction on e, the latter by induction on the rules defining ISS. Proof by coinduction is only needed for the substitution lemma.

Theorem 3.1 (Progress). *If $\emptyset\vdash e\Rightarrow\mathbb{v}$, then either e is a value, or there exists e' s.t. $e\rightarrow e'$.*

Subject reduction relies on the following restricted form of substitution lemma which suffices for proving Theorem 3.2.

Lemma 3.1 (Substitution). *If $\overline{x}^n\mapsto\overline{\mathbb{v}}^n\vdash e\Rightarrow\mathbb{v}$, and for all $i=1..n$ $\emptyset\vdash v_i\Rightarrow\mathbb{v}_i$, then $\emptyset\vdash e[\overline{x}^n\mapsto\overline{v}^n]\Rightarrow\mathbb{v}$.*

Theorem 3.2 (Subject reduction). *If $\emptyset\vdash e\Rightarrow\mathbb{v}$, and $e\rightarrow e'$, then $\emptyset\vdash e'\Rightarrow\mathbb{v}$.*

Corollary 3.1. *If $\emptyset\vdash e\Rightarrow\mathbb{v}$, $e\overset{*}{\rightarrow}e'$, and e' is a normal form, then e' is a value, and $\emptyset\vdash e'\Rightarrow\mathbb{v}$.*

Proof. By induction on the number n of steps needed to reduce e to e'. If $n=0$, then $e=e'$, and trivially $\emptyset\vdash e'\Rightarrow\mathbb{v}$; furthermore, since e' is a normal form, by progress (Theorem 3.1) e' is a value. If $n>0$, then there exists e'' s.t. $e\rightarrow e''$, and e'' reduces to e' in $n-1$ steps. By subject reduction (Theorem 3.2) $\emptyset\vdash e''\Rightarrow\mathbb{v}$, then we conclude by inductive hypothesis. □

4. A NOMINAL TYPE SYSTEM

We define a standard nominal type system for our reference language, and prove that it is sound w.r.t. CBS. The complete proof with all necessary lemmas can be found in a companion paper [2].

Then we show that soundness of the type system w.r.t. ISS can be easily derived from the result of the previous Section as a simple corollary. Finally, we provide a general schema to be applied to a language equipped with both CBS and ISS, for proving soundness of a type systems w.r.t. ISS in terms of soundness w.r.t. CBS.

Besides functions *fields* and *meth*, already used for defining both ISS and CBS, the typing rules are based on the following auxiliary functions/operators, whose trivial definitions have been omitted for space limitation [2]. The standard subtyping relation $\le$ between nominal types; the predicate $\mathit{override}(c,m,\overline{\tau}^n,\tau)$ that holds iff $\mathit{meth}(c',m)$ is undefined or $\mathit{meth}(c',m)=\overline{\tau'}^n\ \overline{x}^n.e{:}\tau'$, $\overline{\tau'}^n\le\overline{\tau}^n$, and $\tau\le\tau'$, with c' direct superclass of c; the join operator $\vee$ which computes the least upper bound $\vee(\tau_1,\tau_2)$ of two types τ_1 and τ_2 (this is always defined since inheritance is single, and *bool* is a subtype of the top type `Object`).

The typing rules, which can be found in Figure 6, are quite standard. A type environment Γ is a finite sequence $\overline{x_i}^n{:}\overline{\tau}^n$,

$$\dfrac{\dfrac{}{\emptyset \vdash \textbf{new}\ \texttt{P()} \Rightarrow obj(\texttt{P},[\,])} \qquad \dfrac{\dfrac{\dfrac{}{\Pi \vdash \texttt{this} \Rightarrow obj(\texttt{P},[\,])} \quad \dfrac{\vdots}{\Pi \vdash \textbf{new}\ \texttt{H(this.m())} \Rightarrow \mathtt{v}}}{\Pi \vdash \texttt{this.m()} \Rightarrow \mathtt{v}}}{\Pi \vdash \textbf{new}\ \texttt{H(this.m())} \Rightarrow obj(\texttt{H},[\texttt{h}\mapsto \mathtt{v}])}}{\emptyset \vdash \textbf{new}\ \texttt{P().m()} \Rightarrow obj(\texttt{H},[\texttt{h}\mapsto\mathtt{v}])}$$

Figure 5: Proof of $\emptyset \vdash \textbf{new}\ \texttt{P().m()} \Rightarrow obj(\texttt{H},[\texttt{h}\mapsto\mathtt{v}])$, with $\mathtt{v} = obj(\texttt{H},[\texttt{h}\mapsto\mathtt{v}])$, $\Pi = \texttt{this} \mapsto obj(\texttt{P},[\,])$

$$(pro)\,\dfrac{\forall\, i = 1..n\ \vdash cd_i{:}\diamond \quad \emptyset \vdash e{:}\tau}{\vdash \overline{cd}^n\ e{:}\diamond} \qquad (cla)\,\dfrac{\forall\, i = 1..k\ c \vdash md_i{:}\diamond \quad \mathit{fields}(c)\ \text{defined}}{\vdash \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ \overline{fd}^n\ \overline{md}^k\ \}{:}\diamond}$$

$$(met)\,\dfrac{\texttt{this}{:}c, \overline{x}^n{:}\overline{\tau}^n \vdash e{:}\tau \quad \tau \leq \tau_0 \quad override(c, m, \overline{\tau}^n, \tau_0)}{c \vdash \tau_0\ m(\overline{\tau}^n\ \overline{x}^n)\ \{e\}{:}\diamond}$$

$$(var)\,\dfrac{}{\Gamma \vdash x{:}\tau}\ \Gamma(x) = \tau \qquad (fal)\,\dfrac{}{\Gamma \vdash \textbf{false}{:}bool} \qquad (tru)\,\dfrac{}{\Gamma \vdash \textbf{true}{:}bool}$$

$$(new)\,\dfrac{\forall\, i = 1..n\ \Gamma \vdash e_i{:}\tau_i \quad \mathit{fields}(c) = \overline{\tau'}^n\ \overline{f}^n \quad \forall\, i = 1..n\ \tau_i \leq \tau'_i}{\Gamma \vdash \textbf{new}\ c(\overline{e}^n){:}c} \qquad (fld)\,\dfrac{\Gamma \vdash e{:}c \quad \mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n \quad 1 \leq i \leq n}{\Gamma \vdash e.f_i{:}\tau_i}$$

$$(inv)\,\dfrac{\forall\, i = 0..n\ \Gamma \vdash e_i{:}\tau_i \quad meth(\tau_0, m) = \overline{\tau'}^n\ \overline{x}^n.e{:}\tau \quad \forall\, i = 1..n\ \tau_i \leq \tau'_i}{\Gamma \vdash e_0.m(\overline{e}^n){:}\tau} \qquad (if)\,\dfrac{\Gamma \vdash e{:}bool \quad \Gamma \vdash e_1{:}\tau_1 \quad \Gamma \vdash e_2{:}\tau_2}{\Gamma \vdash \textbf{if}\ (e)\ e_1\ \textbf{else}\ e_2{:}\vee(\tau_1, \tau_2)}$$

Figure 6: Nominal type system

where all variables $\overline{x_i}^n$ are distinct, denoting a finite function mapping variables to types ($\emptyset$ denotes the empty type environment, $dom(\Gamma)$ the domain of Γ). Rules (pro), (cla), and (met) define well-typed programs, classes, and methods, respectively. The other rules define well-typed expressions w.r.t. a given type environment.

To prove soundness of the type system w.r.t. CBS, we first define a relation $\mathtt{v} \in \tau$ between CBS values and nominal types: intuitively, such a relation defines the intended semantics of types as set of values [5, 6]. Such a relation is coinductively defined by the following rules:

$$(\text{TOP})\,\dfrac{}{\mathtt{v} \in \texttt{Object}} \qquad (\text{BOOL})\,\dfrac{\mathtt{v} = \mathit{false}\ \text{or}\ \mathtt{v} = \mathit{true}}{\mathtt{v} \in bool}$$

$$(\text{OBJ})\,\dfrac{\forall\, i = 1..n\ \mathtt{v}_i \in \tau_i \quad c \leq c' \quad \mathit{fields}(c) = \overline{\tau}^n\ \overline{f}^n}{obj(c, [\overline{f}^n \mapsto \overline{\mathtt{v}}^n]) \in c'}$$

The membership relation is easily extended to environments and type environments:

$$\Pi \in \Gamma \Leftrightarrow dom(\Gamma) \subseteq dom(\Pi)\ \text{and}\ \forall\, x \in dom(\Gamma)\ \Pi(x) \in \Gamma(x).$$

Proof of soundness.

In the sequel we assume the implicit hypothesis that all claims refer to a program where all classes are well-typed. All omitted lemmas and proofs can be found in a companion paper [2].

THEOREM 4.1. *If $\Gamma \vdash e{:}\tau$, and $\Pi \in \Gamma$, then there exists $\mathtt{v}$ s.t. $\Pi \vdash e \Rightarrow \mathtt{v}$ and $\mathtt{v} \in \tau$.*

The following corollary states the soundness of the type system w.r.t. ISS as a direct consequence of Theorem 4.1 and Corollary 3.1.

COROLLARY 4.1. *If $\emptyset \vdash e{:}\tau$, $e \xrightarrow{*} e'$, and e' is a normal form, then e' is a value.*

PROOF. By definition $\emptyset \in \emptyset$, therefore by Theorem 4.1 there exists $\mathtt{v}$ s.t. $\emptyset \vdash e \Rightarrow \mathtt{v}$. Finally, by Corollary 3.1 e' is a value. $\square$

Such a corollary is sufficient for guaranteeing the soundness of the type system w.r.t. ISS: a well-typed expression can never get stuck in ISS. However, by adding the following property (that can be proved easily), we can also deduce that the value e' is s.t. $\emptyset \vdash e'{:}\tau'$ with $\tau' \leq \tau$.

PROPOSITION 4.1. *If $\emptyset \vdash v \Rightarrow \mathtt{v}$, and $\mathtt{v} \in \tau$, then $\emptyset \vdash v{:}\tau'$, with $\tau' \leq \tau$.*

We can now prove the generalization of Corollary 4.1.

COROLLARY 4.2. *If $\emptyset \vdash e{:}\tau$, $e \xrightarrow{*} e'$, and e' is a normal form, then e' is a value and $\emptyset \vdash e{:}\tau'$ with $\tau' \leq \tau$.*

PROOF. The proof proceeds as for Corollary 4.1. By Theorem 4.1 we know also that $\mathtt{v} \in \tau$, and by Corollary 3.1 we know also that $\emptyset \vdash e' \Rightarrow \mathtt{v}$, hence we can conclude by Proposition 4.1. $\square$

Since the proofs of all corollaries are derived from properties expected to hold in general, we can provide a schema for proving soundness of type systems in terms of CBS.

ISS is defined by a reduction relation $e_1 \rightarrow e_2$, and a set of values v (which are a subset of expressions in normal form).

CBS is defined by a judgment $\Pi \vdash e \Rightarrow \mathtt{v}$, where Π is an environment associating variables with values, and $\mathtt{v}$ is a value (all definitions are expected to be coinductive).

The type system is defined by a judgment $\Gamma \vdash e{:}\tau$, where Γ is a type environment associating variables with types, and τ is a type.

Subtyping $\tau_1 \leq \tau_2$ and membership $\mathtt{v} \in \tau$ (which is easily extended to environments) are defined.

Primitive properties:

1. If $\emptyset \vdash e \Rightarrow \mathtt{v}$, then either e is a value, or there exists e' s.t. $e \rightarrow e'$.

2. If $\emptyset \vdash e \Rightarrow \mathtt{v}$, and $e \rightarrow e'$, then $\emptyset \vdash e' \Rightarrow \mathtt{v}$.
3. If $\Gamma \vdash e{:}\tau$, and $\Pi \in \Gamma$, then there exists $\mathtt{v}$ s.t. $\Pi \vdash e \Rightarrow \mathtt{v}$ and $\mathtt{v} \in \tau$.
4. If $\emptyset \vdash v \Rightarrow \mathtt{v}$, and $\mathtt{v} \in \tau$, then $\emptyset \vdash v{:}\tau'$, with $\tau' \leq \tau$.

We stress again that primitive properties 1 and 2 involve ISS and CBS only and, hence, can be proved once for all and reused for any type system.

Derived properties:

- If $\emptyset \vdash e{:}\tau$, $e \overset{*}{\rightarrow} e'$, and e' is a normal form, then e' is a value. Derivable from primitive properties 1,2, and 3.
- If $\emptyset \vdash e{:}\tau$, $e \overset{*}{\rightarrow} e'$, and e' is a normal form, then e' is a value and $\emptyset \vdash e{:}\tau'$ with $\tau' \leq \tau$. Derivable if primitive property 4 holds as well.

5. CONCLUSION

We have defined a coinductive big-step operational semantics of a simple Java-like language by interpreting coinductively its standard big-step operational semantics. With such a semantics it is possible to prove soundness of a standard nominal type system as shown in Section 4.

The pioneer work of Milner and Tofte [12] is one of the first where coinduction is used for proving consistency of the type system and the big-step semantics of a simple functional language; however rules are interpreted inductively, and the semantics does not capture diverging evaluations.

In their work Leroy and Grall [11] analyze two kinds of coinductive big-step operational semantics for the call-by-value λ-calculus, study their relationships with the small-step and denotational semantics, and their suitability for compiler correctness proofs. Besides the fact that here we consider a Java-like language, the main contribution of this paper w.r.t. Leroy and Grall's work is showing that by interpreting coinductively a standard big-step operational semantics, soundness of a standard nominal type system can be proved. We could prove such a result because (1) in our CBS not only evaluation rules are interpreted coinductively, but also the definition of values, and (2) the absence of first-class functions in our language makes the treatment simpler. Leroy and Grall show that a similar soundness claim does not hold in their setting; we conjecture that the only reason for that consists in the fact that in their coinductive semantics values are defined inductively, rather than coinductively. It would be interesting to investigate whether soundness holds for the λ-calculus when values are defined coinductively.

Kusmierek and Bono propose a different approach and prove type soundness w.r.t. an inductive big-step operational semantics; their proposal is centered on the idea of tracing the intermediate steps of a program execution with a partial derivation-search algorithm which deterministically computes the value and the proof tree of evaluation judgments. Similar approaches, although their corresponding semantics are not deterministic, are those of Ager [1] and Stoughton [16].

Nakata and Uustalu [14, 13] define a coinductive trace-based semantics, whose main aim, however, is formal verification of not terminating programs.

Ernst et al. [8] have proved soundness w.r.t. a big-step operational semantics with a coverage lemma ensuring that errors do not prevent expressions from evaluating to a result. To this aim extra rules have to be added for dealing with runtime errors generation and propagation, and finite evaluations.

6. REFERENCES

[1] M. S. Ager. From natural semantics to abstract machines. In *LOPSTR*, pages 245–261, 2004.

[2] D. Ancona. Coinductive big-step operational semantics for type soundness of Java-like languages (extended version). Technical report, DISI, University of Genova, June 2011.

[3] D. Ancona, A. Corradi, G. Lagorio, and F. Damiani. Abstract compilation of object-oriented languages into coinductive CLP(X): can type inference meet verification? In *FoVeOOS 2010*, volume 6528, 2011. Selected paper.

[4] D. Ancona and G. Lagorio. Coinductive type systems for object-oriented languages. In *ECOOP'09*, volume 5653, pages 2–26, 2009. Best paper prize.

[5] D. Ancona and G. Lagorio. Coinductive subtyping for abstract compilation of object-oriented languages into Horn formulas. In *GandALF 2010*, volume 25 of *Electronic Proceedings in Theoretical Computer Science*, pages 214–223, 2010.

[6] D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *FTFJP '10: Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, ACM Digital Library, 2010.

[7] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.

[8] E. Ernst, K. Ostermann, and W.R. Cook. A virtual class calculus. In *POPL*, pages 270–282, 2006.

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[10] J. D. M. Kusmierek and V. Bono. Big-step operational semantics revisited. *Fundam. Inform.*, 103(1-4):137–172, 2010.

[11] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207:284–304, 2009.

[12] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1990.

[13] K. Nakata and T. Uustalu. Trace-based coinductive operational semantics for while. In *TPHOLs 2009*, pages 375–390, 2009.

[14] K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of while. In *ESOP 2010*, pages 488–506, 2010.

[15] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *ICLP 2006*, pages 330–345, 2006.

[16] A. Stoughton. An operational semantics framework supporting the incremental construction of derivation trees. *Electr. Notes Theor. Comput. Sci.*, 10, 1997.