Reconciling Method Overloading and Dynamically Typed Scripting Languages

Alexandre Bergel

Pleiad Group, Computer Science Department (DCC), University of Chile, Santiago, Chile
www.bergel.eu

Abstract

The Java Virtual Machine (JVM) has been adopted as the executing platform by a large number of dynamically typed programming languages. For example, Scheme, Ruby, Javascript, Lisp, and Basic have been successfully implemented on the JVM and each is supported by a large community. Interoperability with Java is one important requirement shared by all these languages.
We claim that the lack of type annotation in interpreted dynamic languages makes this interoperability either flawed or incomplete in presence of method overloading. We studied 17 popular dynamically typed languages for JVM and .Net, none of them were able to properly handle the complexity of method overloading.
We present *dynamic type tag*, an elegant solution for dynamic language interpreters to properly interact with Java objects in presence of overloaded methods. The idea is to embody a type annotation in a Java object reference. Java references may be annotated in order to properly determine the signature of methods to invoke. We demonstrate its applicability in the JSmall language and provide the *pellucid embedding*, a formalization of our approach.

Keywords: Multi-language system, interoperability, SmalltalkLite, JavaLite, dynamic languages

1. Introduction

Probably due to its robustness and its availability for hundreds of different hardware platforms, the Java Virtual Machine (JVM) has been adopted as the executing platform by a large number of interpreters for dynamically typed programming languages (commonly shortened as "dynamic languages"). More than 200 different language interpreters are known to run on the JVM[1]. A number of them have a dynamically typed system (also called "latently typed

[1] http://www.robert-tolksdorf.de/vmlanguages.html

system"). Such a type system implies that it may not be possible to statically establish an assumption regarding the type of an expression.

The benefits of using a dynamically typed language as a way to script and compose Java objects are multiple. Simplicity, clarity and conciseness of the dynamic language over Java are a strong but not exclusive motivation. The lack of a static type system brings such a flexibility that dynamic adaption and incremental compilation are almost unconstrained in a dynamically typed language.

An embedded language becomes more interesting when it is able to interact with its environment. Being able to interoperate with Java significantly raises the expressiveness of such a language. The interoperability between Java and an embedded dynamically typed language interpreter is essentially based on the ability to send messages to Java objects from the hosted interpreter.

Java classes should be freely instantiated and objects should be manipulated within the dynamic language interpreter. In particular, sending messages to Java objects from the dynamic language is an essential requirement for useful interoperability.

However, such interoperability is hard to achieve because of the static nature of Java. In the 13 dynamically typed languages for JVM and the 4 dynamic languages for .Net we studied, some of them behave either wrongly or in an unpredictable way in presence of overloaded Java methods, while the remaining simply raise runtime exceptions when no method can be selected. None of them offer the flexibility of Java regarding calling overloaded methods. This situation becomes critical when considering the number of overridden methods present in the JDK. For example, Swing[2] has 9648 methods and 2303 of them are overloaded[3]. Scripting graphical user interfaces (GUIs) is the traditional application domain where benefits of dynamically typed languages over Java to build a GUI are apparent.

As we will see later, the limitation of dynamic languages to properly handle overloading of Java methods comes from the fact that only the class of Java objects is used to resolve Java method signatures at runtime. As a consequence, overloaded methods may be inaccessible from the embedded interpreter since overloaded Java methods having more specific parameters types will "hide" the more generic ones. Most of the time, this is not really a problem since the method that is called is the one which has the greatest possible amount of knowledge about its actual arguments. However, this is less flexible and more restrictive than what Java permit: it would be a bug if a scripting language using a Java library follows different invocation rules than in Java.

Our analysis, problem formulation and solution are equally applicable to the .Net platform. But since many more dynamic languages are available in Java than in .Net, this paper is Java-centric. The .Net case is related in Section 6.

[2]version 6 of the Java Platform, Standard Edition

[3]To illustrate the metric we use, there are 2 overloaded methods in `class C{void m(){} void m(C c){} void n(){}}`.

This paper is about fixing the flaw in the interoperability with Java against overloaded methods, issue shared by most dynamically typed languages. Handling Java overloaded methods within an embedded dynamic language interpreter is a well known and difficult problem recognized by most dynamic language communities. Several solutions have been proposed, but as far as we are aware of, none of them are complete.

We propose a generic technique called *dynamic type tag* for dynamically typed programming languages. The idea is to make a Java object wrapper embed a type intended to be used when an overloaded method has to be resolved at runtime. Each of the references to a particular Java object may embed a dynamic type tag. This embedded type comes in addition to the dynamic type of the wrapped Java object. Dynamic type tag is implemented in JSmall, a Smalltalk interpreter running on the Java virtual machine. This paper makes the following contributions and innovations:

- it highlights an important deficiency in the way common dynamically typed languages interoperate with Java;
- it presents the pellucid embedding, a simple and generic solution easily embeddable in a language for which a static type system is missing;
- it demonstrates type soundness and highlights type tag propagation along the control flow.

This paper is structured as follows. In Section 2 we present the limitations of dynamically typed programming languages in the way they interact with Java. In Section 3 we solve these issues with the dynamic type tag. We use the JSmall Smalltalk language as an informal support. In Section 4 we discuss about some benefits of the static type annotation. In Section 5 we formally describe the dynamic type tag with the pellucid embedding. In Section 6 we provide an overview of the related work. In Section 7 we conclude by summarizing the presented work.

2. Static and dynamic typing

This section presents a number of scenarios illustrating limitation of dynamic languages in presence of method overloading. Each of the 3 scenarios shows methods that are either inaccessible or wrongly selected upon sending messages to a Java object. As an illustration, consider the contrived but compact Java class definitions intended to write XML transcriptions of some values:

```
// Java code
public class XMLWriter {
  public XMLWriter() { ... }

  public void write(int e) { ... }
  public void write(Integer e) { ... }
  public void write(XMLElement e) { ... }
  public void write(StructuredXMLElement e) { ... }
  public void write(Serializable e) { ... }
}

public class XMLElement extends Object { ... }

public class StructuredXMLElement
  extends XMLElement implements Serializable { ... }
```

We will use this example throughout this paper. For the sake of clarity, we will use only one language, JSmall, to illustrate these limitations. We will *not use* the dynamic type tag in this section since this linguistic construct aims at addressing these limitations. We will add appropriate notes in case of divergence with other languages.

Scenario 1: Method overloading. A method called on a Java object in JSmall is translated into an invocation of a Java method on this object. The signature of the Java method that has to be invoked is resolved from the name of the method call, the number of parameters and the dynamic type of the parameters. The fact that the message name and the number of arguments are statically determined gives a good indication about the targeted Java method signature (note that we do not consider the recent introduction in Java of variable number of parameters).

The Java method to invoke is easily resolved when one method only in the inheritance chain of the receiver's Java class matches the name and number of parameters of the call. Resolving the Java method to invoke remains easy in presence of several Java methods having the same name, each having an arity (*i.e.,* number of arguments) different from the other methods: the number of arguments contained in the call is sufficient to discern the right Java method.

The situation gets far more complex when several methods have the same name and arity. The only way in Jython[4], JRuby[5], and Rhino[6] to resolve the Java method signature to invoke is to use the runtime type of arguments. The method for which its signature matches or is "close" enough to the method call in the dynamically typed language is elected for invocation by the Java object wrapper (usually based on the class of the Java object). Consider the following example:

[4]http://www.jython.org
[5]http://jruby.codehaus.org
[6]http://www.mozilla.org/rhino/ScriptingJava.html

```
"JSmall code"
w := 'XMLWriter' asJavaClass new.
e1 := 'XMLElement' asJavaClass new.
e2 := 'StructuredXMLElement' asJavaClass new.
w write: e1.    "call #1"
w write: e2.    "call #2"
```

Call #1 uniquely matches write(XMLElement) since (i) the name and the number of arguments contained in the JSmall call match and (ii) the dynamic type of the e1 variable is XMLElement.

Call #2 is more problematic because two methods are equally eligible for an invocation (write(StructuredXMLElement) and write(Serializable)) since StructuredXMLElement implements Serializable. A decision has to be made by the dynamic language interpreter. JSmall and Rhino will throw an exception saying that this call is ambiguous and JRuby will select write(Serializable). A similar code in Jython will invoke write(StructuredXMLElement). The algorithm of selection used by Java object wrappers may favor subtyping of interfaces, or classes, or may use the first method given by the virtual machine that is close enough when enumerating methods in order to resolve the call. No consensus among different communities has been reached.

Scenario 2: Primitive and reference values. When a Java method is called by a dynamic scripting language, numerical values provided as arguments are automatically boxed into their corresponding reference value, *i.e.,* the JSmall integer 10 is converted into an instance of java.lang.Integer when used to call a method on a Java object. This new type is then used to resolve the Java method to invoke. For example, consider the following excerpt of JSmall and JRuby code:

```
"JSmall code"                             # JRuby code
w := 'XMLWriter' asJavaClass new.         include Java
w write: 10.                              include_class "XMLWriter"
                                          w = XMLWriter.new
                                          w.write(10)
```

The write: 10 message is sent to an instance of the class XMLWriter. In JSmall, this method call raises an exception[7] and in JRuby XMLWriter.write(Integer) is executed instead. In both cases, two methods may be equally invoked (write(int) and write(Integer)). However, only one method is accessible from the client. In JSmall, the fact that write(...) is overloaded completely hides the write(Integer) method, whereas in JRuby write(int) is hidden. Programmers in JSmall and JRuby cannot indicate which write(...) they are referring to. It is reasonable to expect write(Integer) and write(int) to have the same behavior in most cases. However, this cannot stand as a motivation from preventing a programmer to select a particular one, especially since nothing enforces programming consistency and in some case where serialization is involved (as in our situation), the

[7]Similarly than in Rhino, JSmall raises an exception in case of ambiguity. The dynamic type tag introduced later removes the ambiguity

difference may matter.

Scenario 3: The empty value. As we described above, the dynamic type of a wrapped Java object is used to resolve the Java method when receiving a message sent within JRuby. With such a strategy, the empty value will inevitably be problematic.

In JSmall, the expression `w write: nil` raises a runtime error since no version of `write(...)` can be reasonably chosen. In JRuby, the expression `w.write(nil)` picks the last method declared in the Java class. Jython does not have any preference and picks any method. Rhino raises a runtime error.

From what we can see in large and popular API, it is perfectly conceivable to explicitly provide the empty value when calling a method. For example, in the JDK, the Swing method `JComponent.setComponentPopupMenu(JPopupMenu)` may accept a null value to delegate a popup menu to a parent object.

2.1. Reflection is not acceptable

Reflection may be employed to retrieve a particular method, and then to invoke it independently from the type of the parameter. Use of reflection is the only way to circumvent the limitations described above.

For example, the following JRuby code invokes `write(Object)` and uses an instance of `StructuredXMLElement` as the parameter:

```
# JRuby code
cls = Java::JavaClass.for_name("XMLWriter")
w = cls.constructor().new_instance()
cls.java_method(:write, "java.io.Serializable").
  invoke(w, StructuredXMLElement.new)
```

This kind of writing goes against the primary aim of dynamically typed scripting languages, which is to provide a more concise and expressive language than Java, the underlying hosting language. In that respect, reflection appears to be an ad hoc and verbose solution. Moreover using reflection completely goes against pillars of object orientation since the responsibility of objects to understand messages has dramatically shifted to the caller side. As a consequence, this kind of method invocation do not benefit from polymorphism since no lookup along a class inheritance happens.

3. Dynamic Type Tag

This section presents a natural and concise solution to the problems described in the previous section by introducing dynamic type tags for foreign objects in a dynamically typed language. The dynamic type tag is a generic solution to cope with the Java type system for embedded dynamic languages.

3.1. Dynamic Type Tag in JSmall

Each object in JSmall understands the message type: aType, where aType is a character string that represents the dynamic type tag. Legal strings are the ones that correspond to Java type names. This annotation is contained in the reference and is used to resolve the Java method signature when sending messages to Java objects. The formulation *exp* type: aType has the following semantics: (i) *exp* is evaluated, (ii) the resulting value is turned into a Java object if not already, and (iii) the reference of this object is tagged with aType.

The value aType provided as argument should be equal or be a supertype of the dynamic type of the tagged Java value. An error is raised if it is a subtype or unrelated. This is similar to a failed Java downcast: one cannot downcast a Java value with a subtype of the dynamic type of the value.

Note that type: does not perform any side effect: it behaves as a function that returns a new reference. Naturally, the referenced alien object remains untouched. In the forthcoming Section 5 we give the formal semantics of type:. The tag is always associated with a wrapper that serves as a value in the host dynamic language.

In JSmall, the XMLWriter class described above may be accessed within JSmall as follows:

```
"JSmall code"
"Instantiation of the XMLWriter Java class"
w := 'XMLWriter' asJavaClass new.

"The value 10 is converted into a Java object,
and its type is set to 'int' "
i := 10 type: 'int'.

"Instantiation of StructuredXMLElement and
set its type to 'java.io.Serializable'"
obj := 'StructuredXMLElement' asJavaClass
  new type: 'java.io.Serializable'.

w write: i.              "Invocation of write(int)"
w write: obj.            "Invocation of write(Serializable)"

"Invocation of write(StructuredXMLElement)"
w write: (obj type: 'StructuredXMLElement').
w write: obj.            "Invocation of write(Serializable)"
```

In this short excerpt, type: is used three times. The expression 10 type: 'int' means that the JSmall value 10 is converted into a Java object, returning a JSmall wrapper for the Java integer 10, then the static type tag is set to the primitive type int. When the variable i is used as a parameter when sending write:, the resolved Java method is write(int), which results into the invocation of XMLWriter.write(int).

Similarly for the second use of type:, an instance of the Java class StructuredXMLElement is first created, then the static type of this new object is set to java.io.Serializable. When this structured XML element is passed to write:, the corresponding Java method signature is write(Serializable).

Finally, the expression obj type: 'StructuredXMLElement' returns a new reference of the Java object pointed by obj for which its static type is StructuredXMLElement. Note that the static type of obj remains unchanged, this is why w write: obj sends the Java message write(Serializable).

When no static type is set, the dynamic type of the object is used to resolve the Java method signature. In case of an abuse of the type: message (*e.g.,*'Object' asJavaClass new type: 'int'), errors will be signaled at runtime. The expression type: allows for downcasting and upcasting only.

3.2. Evaluation of JSmall's Dynamic Type Tag

We presented three severe limitations of dynamically typed languages in the way overloaded methods are invoked. This subsection will review the different scenarios exposed in Section 2 against JSmall's static annotation.

Scenario 1: Method overloading. Annotating each message parameter with a static type enables overloaded methods to be called. Method selection is then based on the static type of the parameters instead of the dynamic type. The following code excerpt illustrates this situation:

```
"JSmall code"
w := 'XMLWriter' asJavaClass new.
e := 'StructuredXMLElement' asJavaClass new.
w write: e.                                   "Exception raised"
w write: (e type: 'Serializable').
w write: (e type: 'StructuredXMLElement').
```

The argument of the first call write(e) does not have any static type. By not annotating the provided argument, we fall in an ambiguous case. In that case, a runtime exception is raised, similarly to Rhino.

The second call invokes write(Serializable) since the dynamic type tag of the argument is Serializable. An exception is thrown in case of no matching method. The third call to write(...) matches write(StructuredXMLElement).

One could argue that e may have the dynamic tag StructuredXMLElement. However, that would breach the distinction we are making between the dynamic and static type of an object. The dynamic type tag must always be set to resolve conflicting situation.

Scenario 2: Primitive and reference values. The type: keyword may be employed to assign a primitive type to a numerical value. Java objects are wrapped pretty much the same way than with JRuby, Jython and Rhino: numerical values are kept as references when wrapped (*i.e.,* as an instance of Integer, Float, ...). However, the static type reference contained in a Java wrapper is used instead of the dynamic type when resolving the method for which the numerical value was provided. Each wrapper in JSmall has a type field and this field may be set with type:. Consider the following example:

```
"JSmall code"
w := 'XMLWriter' asJavaClass new.
w write: 10.                        "Exception raised"
w write: (10 type: 'int').          "call write(int)"
w write: (10 type: 'Integer').      "call write(Integer)"
```

The first call write: 10 raises an exception, similarly to Rhino. The two subsequent calls of write: use the dynamic type tag to resolve the Java method signature to invoke.

Scenario 3: The empty value. When the empty value of JSmall, nil, is an argument when calling a Java method, nil is converted into Java's null. This value may also be annotated, as for any Java object value. As in the situation previously described, the static type is used to resolve the Java method. The following code shows different combinations:

```
"JSmall code"
w := 'XMLWriter' asJavaClass new.
w write: (nil type: 'XMLElement').     "call write(XMLElement)"
w write: (nil type: 'Serializable').   "call write(Serializable)"
w write: (nil type: 'Integer').        "call write(Integer)"
```

The interpretation of this code follows the rules given previously. The call write(nil type: 'Integer') invokes write(Integer) with the nil value as argument.

Since the dynamic type tag is attached to a reference, the nil value, as any object, may be tagged multiple times.

3.3. From Java to JSmall

The previous section essentially focuses on passing annotated objects from JSmall to Java. The same mechanism applies in the other way around. Values returned to JSmall from Java are automatically annotated with the return type of the called Java method. The aim of this automatic annotation is not to lose the type when results from calling Java methods have to be used as arguments when calling another Java method.

Note that the return type declared in the Java method is used to tag the returned value, and not the dynamic type of the value.

Consider the following XML reader:

```
// Java code
public class XMLReader {
  public int readInt() { ... }
  public Integer readInteger() { ... }
  public XMLElement readXMLElement() { ... }
  public StructuredXMLElement readStructuredXMLElement() {...}
}
```

An example in JSmall that replicates some elements contained in an XML file may be:

```
"JSmall code"
r := 'XMLReader' asJavaClass new: 'Data.xml'.
w := 'XMLWriter' asJavaClass new: 'ReplicatedData.xml'.
anInteger := r readInt.
w write: anInteger.
anElement := r readStructuredXMLElement.
w write: anElement.
```

The two variables anInteger and anElement refer to two Java objects, returned by readInt and readStructuredXMLElement, respectively. These objects are annotated with the static types int and StructuredXMLElement, respectively, since these types correspond to the return types of the invoked methods.

4. Discussion

A few points points are worth discussing.

Using types on the caller side. The dynamic type tag embeds a type in a Java object reference. An alternative design would be to put the annotation on the method calls, as opposed to attached to the values to those method calls. For example, this annotation could be specified using < ... > as in the expression w write<StructuredXMLElement>: anElement. The general method invocation could then be *exp* name1<Type1>: *exp1* name2<Type2>: *exp2* ... in case of multiple argument invocation.

However, this model would be suboptimal compared with the dynamic type tag since it assumes that a Java class user *must* associate parameter values and returned values with the type information contained in the signature of methods involved in the computation. Another point that would be missed when specifying the type on the caller side is the ability to pass objects around along with their dynamic type tag, which means that they can preserve their static type for further invocations (later in time or through multiple calls on the JSmall side).

Let us consider the example of the previous section to illustrate this important point. Using this alternative model, combining the XML reader and writer could be written as follows:

```
"JSmall code"
r := 'XMLReader' asJavaClass new: ...
w := 'XMLWriter' asJavaClass new: ...
w write<StructuredXMLElement>: (r readStructuredXMLElement).
```

This short piece of code assumes that a user of XMLWriter has the knowledge about the return type of XMLReader.readStructuredXMLElement() in order to select XMLWriter.write (StructuredXMLElement) for invocation. As described in Section 3.3, embedding the type in the object reference relieves the programmer from having to associate return values with return types of called methods.

Bijection between method names. We assume a bijection between JSmall and Java method names. Although we provide a converting schema, this issue is out of the scope of this paper.

This bijection is immediate: a JSmall method may always be named in Java, and a public Java method may always be named in JSmall. However, argument passing from Smalltalk to Java need some care since arguments in Smalltalk are inserted within the method name itself. For example, one way to create a geometrical rectangle is to send the message `center: centerPoint extent: extentPoint` to the class `Rectangle`. The variables `centerPoint` and `extentPoint` are values determining the size of the rectangle where as the name of the invoked method is `center:extent:`.

A Java message send within JSmall must be expressed using the JSmall syntax. For exemple, the Java method `JButton.setInputMap(int condition, InputMap map)` may be written in JSmall as `jButton set: condition InputMap: map` or `jButton setInputMap: (condition, map)`. Java method names that cannot be nicely written using this style, may be written with arguments separated with commas. For example, the method `JButton.reshape(int x, int y, int w, int h)` may be invoked using `jButton reshape: (x, y, w, h)`. In that case, the message comma ("," is a method name in JSmall) is sent to the value `x`, taking the argument `y`. Java messages sent from JSmall may always use either style.

5. The Pellucid Embedding

This section formalizes the dynamic type tag described in the previous section. The purpose of this formalization is multiple. Firstly, it demonstrates that the dynamic type tag described previously is not tied to any particular programming language. Secondly, it precisely exhibits the inter-languages message passing mechanism. Thirdly, our formalization shows that adding dynamic type tags does not break the type soundness of Java.

The strategy we adopted for this aim can be summarized as follows: We first provide a formal model for JSmall and a second one for Java. We then tie these models by extending them with the lump embedding, a technique proposed by Matthews and Findler [MF07]. This technique is used to represent JSmall values in Java and the other way around without allowing methods to be called. For that purpose, the pellucid embedding is a further extension of the lump embedding to enable messages to be sent to objects issued from a different language. A set of properties will be then formalized.

The lump embedding was originally conceived to express the Scheme embedding in ML. A further contribution of this section is the application of the lump embedding in an object-oriented setting.

The structure of our formalism and outline of this section may be schematically depicted (Figure 1). Gray boxes are piece of work taken from other work, and put here for sake of completeness. White boxes are novel and should be considered as paper innovations.

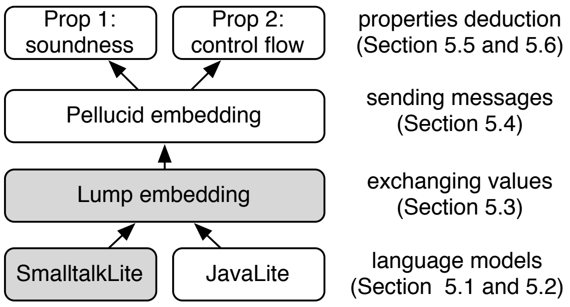

Prop 1: soundness
Prop 2: control flow
properties deduction (Section 5.5 and 5.6)
Pellucid embedding
sending messages (Section 5.4)
Lump embedding
exchanging values (Section 5.3)
SmalltalkLite
JavaLite
language models (Section 5.1 and 5.2)

Figure 1: Formalism edification (grey units do not belongs to the contributions of this paper).

To help the reader to dissociate Smalltalk code from Java code, we use a textual convention: non terminal terms written in blue roman font designate Smalltalk terms, where those **in red bold face font** belong to Java.

5.1. SmalltalkLite

SmalltalkLite is a Smalltalk-like dynamically typed language[8] featuring single inheritance, message-passing, field access and update, and self message send.

SmalltalkLite has been first presented in a previous work [BDNW08]. Its syntax and semantics rules are given again here for sake of completeness. Super-call and temporary variable declaration have been elided since those are not directly related to the point of this section, which is about migration of type between JSmall and Smalltalk and cross-languages method calls.

The syntax of SmalltalkLite is presented in Figure 2. A program in SmalltalkLite is a set of class definitions and an expression. This expression is the "starting point" of a program and this expression is evaluated when a program has to be executed. Note that the SmalltalkLite syntax is different from the real Smalltalk one. It has been "javaized" to reduce differences between Smalltalk and Java to its essence, its type system. We use a star * to designate a list of elements.

For now, a program in SmalltalkLite is confined and no interaction with Java is possible. We will introduce later new syntactic elements in this syntax to enable such interaction.

Before we introduce the reduction semantics rules, we need to introduce how syntactic elements (*cf.*, Figure 2) may be translated into redex elements, final stage before evaluation. This translation is performed at runtime by the

[8]The essential difference between the Smalltalk and Java formal model we are making here is about the preciseness of typing information, and not whether type checking occurs at run-time only.

$$
\begin{array}{rcl}
\mathrm{P} & = & \mathrm{defn}^{*}\epsilon \\
\mathrm{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^{*}\mathrm{meth}^{*}\ \} \\
\epsilon & = & \textbf{new}\ c \ \mid\ x\ \mid\ \textbf{self}\ \mid\ \mathsf{nil} \\
 & \mid & f\ \mid\ f{=}\epsilon\ \mid\ \epsilon.m(\epsilon^{*}) \\
\mathrm{meth} & = & m(x^{*})\ \{\ \epsilon\ \} \\
c & = & \text{a class name}\ \mid\ \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}
$$

Figure 2: SmalltalkLite syntax

$$
\begin{array}{rcl}
o[\![\textbf{new}\ c]\!] & = & \textbf{new}\ c \\
o[\![x]\!] & = & x \\
o[\![\textbf{self}]\!] & = & o \\
o[\![\mathsf{nil}]\!] & = & \mathsf{nil}
\end{array}
\qquad
\begin{array}{rcl}
o[\![f]\!] & = & \underline{o.}f \\
o[\![f{=}\mathrm{e}]\!] & = & \underline{o.}f{=}o[\![\mathrm{e}]\!] \\
o[\![\mathrm{e}.m(\mathrm{e_i}^{*})]\!] & = & o[\![\mathrm{e}]\!].m(o[\![\mathrm{e_i}]\!]^{*})
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{e} & = & v\ \mid\ \textbf{new}\ c\ \mid\ x\ \mid\ \mathsf{nil}\ \mid\ \mathrm{e}.f\ \mid\ \mathrm{e}.f{=}\mathrm{e} \\
 & \mid & \mathrm{e}.m(\mathrm{e}^{*})\ \mid\ \mathrm{error}\ \mathit{string} \\
\mathrm{E} & = & [\ ]\ \mid\ \underline{o.}f{=}E\ \mid\ E.m(\mathrm{e}^{*})\ \mid\ o.m(v^{*}\ E\ \mathrm{e}^{*}) \\
v & = & \mathsf{nil}\ \mid\ o
\end{array}
$$

Figure 3: Translating expressions to redexes before evaluation

operator $o[\![\ ]\!]$ before evaluating a method body. It annotates variable accesses with the object in which the variable has to be looked up. Privacy of variables is achieved in SmalltalkLite by using an implicit object (**self**, the current object) as the environment in which this variable has to be looked up. The redex translation annotates a field access with an object o. This object also replaces occurrence of **self**.

The redex syntax is shown in the lower part of Figure 3. It defines the syntax used in the resulting translation performed by $o[\![\ ]\!]$. This translation occurs before evaluating a method body as it will be shown later in Figure 5. A SmalltalkLite value is either nil or an object reference (o). Value of o is determined by the reduction rules (Figure 5).

Our reduction semantics is inspired from ClassicJava's [FKF99] to specify operational semantics for our systems. In the figure, we define an evaluation context E for SmalltalkLite. A value in our language can either be nil or an object reference. Underlined phrases are inserted by the transformation to redexes and are not part of the surface syntax.

The type elaboration rules for SmalltalkLite are defined by the following

$$\frac{P \vdash_S^T f_j : TST \;\; \text{for each} \;\; j \in [1,n] \qquad P,c \vdash_S^T \mathit{meth}_k \Rightarrow \mathit{meth'}_k \;\; \text{for each} \; k \in [1,p]}{P \vdash_S^T \begin{array}{c} \textbf{class}\ c\ \textbf{extends}\ c' \, \{f_1 \;\ldots\; f_n \;\; \mathit{meth}_1 \;\ldots\; \mathit{meth}_p\} \;\Rightarrow \\ \textbf{class}\ c\ \textbf{extends}\ c' \, \{f_1 : TST \;\ldots\; f_n : TST \;\; \mathit{meth'}_1 \;\ldots\; \mathit{meth'}_p\} \end{array}} \textbf{[defn]}$$

$$\frac{P \vdash_S^T x_j : TST \;\; \text{for each} \; j{\in}[1{,}n] \qquad P,[\textbf{self} : t,\ x_1 : TST,\ \ldots,\ x_n : TST] \vdash_S^T \mathrm{e} \Rightarrow \mathrm{e}' : TST}{P,t \vdash_S^T m(x_1,\ \ldots,x_n)\{\mathrm{e}\} \Rightarrow m(x_1,\ \ldots,\ x_n)\{\mathrm{e}'\}} \textbf{[meth]}$$

$$\frac{P \vdash_S^T c}{P \vdash_S^T \textbf{new}\ c \Rightarrow \textbf{new}\ c : TST} \textbf{[new]}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{P{,}\Gamma \vdash_S^T x \Rightarrow x : TST} \textbf{[var]}$$

$$\frac{f \in_P \Gamma(\textbf{self})}{P{,}\Gamma \vdash_S^T f \Rightarrow f : TST} \textbf{[get]}$$

$$\frac{f \in_P \Gamma(\textbf{self}) \qquad P{,}\Gamma \vdash_S^T \mathrm{e_v} \Rightarrow \mathrm{e'_v} : TST}{P{,}\Gamma \vdash_S^T f{=}\mathrm{e_v} \Rightarrow f{=}\mathrm{e'_v} : TST} \textbf{[set]}$$

$$\frac{}{P{,}\Gamma \vdash_S^T \mathsf{nil} \Rightarrow \mathsf{nil} : TST} \textbf{[null]}$$

$$\frac{P, \Gamma \vdash_S^T \mathrm{e} \Rightarrow \mathrm{e}' : TST \qquad P, \Gamma \vdash_S^T \mathrm{e_j} \Rightarrow \mathrm{e'_j} : TST \;\; \text{for each} \;\; j \in [1, n]}{P{,}\Gamma \vdash_S^T \mathrm{e}.m(\mathrm{e_1},\ \ldots,\ \mathrm{e_n}) \Rightarrow \mathrm{e}'.m(\mathrm{e'_1},\ \ldots,\ \mathrm{e'_n}) : TST} \textbf{[send]}$$

Figure 4: SmalltalkLite typing rules

$$
\begin{array}{lll}
P \vdash_S & \langle E[\textbf{new } c], \mathcal{S}\rangle \hookrightarrow \langle E[o], \mathcal{S}[o \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P c\}\rangle]\rangle & [\textit{new}] \\
 & \text{where } o \notin \mathrm{dom}(\mathcal{S}) & \\
P \vdash_S & \langle E[\underline{o}.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle & [\textit{get}] \\
 & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = v & \\
P \vdash_S & \langle E[\underline{o}.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle & [\textit{set}] \\
 & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle & \\
P \vdash_S & \langle E[o.m(v_1, \ \ldots, \ v_n)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![\epsilon[v_1/x_1, \ \ldots, \ v_n/x_n]]\!]], \mathcal{S}\rangle & [\textit{send}] \\
 & \text{where } \mathcal{S}[o] = \langle c, \mathcal{F}\rangle \text{ and } \langle m, x^*, \epsilon\rangle \in_P c & \\
 & & \\
P \vdash_S & \langle E[\underline{o}.f], \mathcal{S}\rangle \hookrightarrow \langle E[\textit{error} \ \textit{``eval''}], \mathcal{S}\rangle & [\textit{get-err}] \\
 & \text{where } o = \mathsf{nil} \quad \text{or} \quad \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = \bot & \\
P \vdash_S & \langle E[\underline{o}.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[\textit{error} \ \textit{``eval''}], \mathcal{S}\rangle & [\textit{set-err}] \\
 & \text{where } o = \mathsf{nil} \quad \text{or} \quad \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(f) = \bot & \\
P \vdash_S & \langle E[o.m(v_1, \ \ldots, \ v_n)], \mathcal{S}\rangle \hookrightarrow \langle E[\textit{error} \ \textit{``eval''}], \mathcal{S}\rangle & [\textit{send-err}] \\
 & \text{where } o = \mathsf{nil} \quad \text{or} \quad \mathcal{S}[o] = \langle c, \mathcal{F}\rangle \text{ and } \langle m, x^*, \epsilon\rangle \notin_P c &
\end{array}
$$

Figure 5: Reductions for SmalltalkLite (the S in $\vdash_S$ is for Smalltalk)

judgments:

$$
\begin{array}{rlll}
P & \vdash^T_S & \mathrm{defn} \Rightarrow \mathrm{defn}' & \mathrm{defn} \text{ elaborates to } \mathrm{defn}' \\
P, c & \vdash^T_S & \mathrm{meth} \Rightarrow \mathrm{meth}' & \mathrm{meth} \text{ in c elaborates to } \mathrm{meth}' \\
P, \Gamma & \vdash^T_S & \mathrm{e} \Rightarrow \mathrm{e}' : \mathit{TST} & \mathrm{e} \text{ elaborates to } \mathrm{e}' \\
 & & & \text{with type } \mathit{TST} \text{ in } \Gamma
\end{array}
$$

Context-sensitive checks and type elaboration rules for SmalltalkLite are given in Figure 4. This type system gives to all closed terms the type TST ("the Smalltalk type"). Every SmalltalkLite expression has a rule that gives type TST if its subparts have type TST.

The list of reductions rules is given in Figure 5. These rules are pretty standard. The [*send*] rule translates an expression body e into redexes before evaluating it. The relation $\in_P$ predicates the existence of a field or a method on a class. The relation $\leq_P$ asserts the subclass relationship. Mapping between field names and fields value is denoted with $\mathcal{F} = \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\}$. $\mathcal{F}$ is a function that takes as argument the field name to lookup and returns its associated value. $\mathcal{F}[f \mapsto v]$ designates the replacement of a field value or the addition a new field.

Accessing a field or sending a message may fail since no type information prevents such situation. [*get-err*], [*set-err*] and [*send-err*] take care of this.

$$\begin{array}{rcl}
P & = & \textbf{defn}^*\epsilon \\
\textbf{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ \textbf{field}^*\textbf{meth}^*\ \} \\
\textbf{field} & = & t\ \ f \\
\epsilon & = & \textbf{new}\ c\ \ |\ x\ |\ \textbf{this}\ |\ \mathsf{nil} \\
 & | & \epsilon.f\ \ |\ \epsilon.f{=}\epsilon\ \ |\ \epsilon.m(\epsilon^*) \\
\textbf{meth} & = & t\ \ m(\text{arg}^*)\ \{\ \epsilon\ \} \\
\textbf{arg} & = & t\ \ x \\
t & = & c \\
c & = & \text{a class name}\ \ |\ \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}$$

Figure 6: JavaLite syntax

5.2. JavaLite

JavaLite is a formalization of Java that captures the essence of its typing system, including method overloading. Java has been reduced to focus on method resolution, an essential feature for our purpose.

In JavaLite fields are public and their accesses must be explicitly preceded by the object in which this field has to be looked up; methods are public; methods may be overloaded; and object interaction solely uses message-passing. Objects are the only values of our language. For sake of clarity and conciseness, we do not model int and Integer since this language constructs are not essential for our purpose. Primitive types are a particularity of Java that is orthogonal to our model.

Syntax and semantics rules of JavaLite are very similar to SmalltalkLite. The main difference is the privacy of variables and the typing rules.

Before we go on the description of the language, it might be worth pinpointing differences with other Java formalisms. ClassicJava [FKF99] and FeatherweightJava [IPW01] are two different minimal models for Java. ClassicJava supports interfaces and field assignments but no genericity, whereas FeatherweightJava may support genericity but does not have variable assignments and interfaces. These two models do not support method overloading. This is the primary reason why we did not adopt one of these for our purpose. Other calculus such as MJ [BPP03] and Core-Java [CGPC06] are available. These two languages may be seen as contenders for a minimal imperative core calculus. Imperative feature and ability to handle concurrency (for Core-Java) are outside the scope of the paper. Principal typing [AZ04] for Java supports method overloading, but their focus is more on separate compilation and high-level representation of bytecode. An extension of FeatherweightJava has been proposed [BCV08] to support dynamic and static method overloading. However, the fact that this extension has been made for expressing

$$
\begin{array}{rcl}
\mathbf{e} & = & \mathbf{v} \mid \textbf{new}\ c \mid x \mid \mathbf{e}\underline{\ :\ c}.f \mid \mathbf{e}\underline{\ :\ c}.f{=}\mathbf{e} \\
 & \mid & \mathbf{e}\underline{\ :\ c}.m(\mathbf{e}^*) \\
\mathbf{E} & = & [\,] \mid \mathbf{E}\underline{\ :\ c}.f \mid \mathbf{E}\underline{\ :\ c}.f{=}\mathbf{e} \mid o\underline{\ :\ c}.f{=}\mathbf{E} \\
 & \mid & \mathbf{E}.m(\mathbf{e}^*) \mid o.m(\mathbf{v}^*\ \mathbf{E}\ \mathbf{e}^*) \\
\mathbf{v} & = & \mathsf{nil} \mid o
\end{array}
$$

Figure 7: Redex syntax for JavaLite; underlined phrases are inserted by elaboration and are not part of the surface syntax

multi-inheritance, the tiny bit that would be useful for our purpose comes with a large set of unnecessary artifacts. Instead, the idea of JavaLite is to have a minimal calculus that extends SmalltalkLite with a Java-like type system. Figure 6 describes the syntax of JavaLite.

The differences with SmalltalkLite are public fields (*i.e.,* accesses are prefixed by an expression), method parameters and variable declaration are annotated with a type, and **self** has been replaced by **this**.

The redex syntax is presented in Figure 7. Similarly than in Smalltalk-Lite, the **this** pseudo variable does not belong to this syntax since it is replaced by the current object when a method body has to be evaluated.

The big piece of this formalization is the typing rules that embody method overloading.

The type elaboration for JavaLite is defined by the following judgments:

	$\vdash_J^T$	$P \Rightarrow P' : t$	P elaborates to P' with type t
P	$\vdash_J^T$	$\mathbf{defn} \Rightarrow \mathbf{defn}'$	**defn** elaborates to **defn**′
P, c	$\vdash_J^T$	$\mathbf{meth} \Rightarrow \mathbf{meth}'$	**meth** in c elaborates to **meth**′
P, Γ	$\vdash_J^T$	$\mathbf{e} \Rightarrow \mathbf{e}' : t$	**e** elaborates to **e**′ with type t in Γ
P	$\vdash_J^T$	t	t exists

Figure 8 presents the JavaLite typing rules. The type elaboration verifies that a JavaLite program defines a static tree of classes. In our calculus, a type is a class since interfaces and generics are not supported. Each type is annotated with its collection of fields and methods, including those inherited from its ancestors. Underlined phrases are inserted by the typing elaboration and are not part of the surface syntax

The **[prog]** rule says the program elaborates only if its expression **e** and its class definitions elaborates for a program P.

[defn] says that a class definition elaborates only if each field's type exists and each method **meth** elaborates into **meth**′. Note that at this stage we assume that we do not have cycle in the hierarchy of classes. To keep our

$$\frac{P,[]\ \vdash^T_J\ \mathbf{e} \Rightarrow \mathbf{e}' : t \qquad P\ \vdash^T_J\ defn_j \Rightarrow defn'_j \text{ for } j\in[1,n] \text{ where } P{=}defn_1 \ldots defn_n\ \mathbf{e}}{\vdash^T_J\ defn_1 \ldots defn_n\ \mathbf{e} \Rightarrow defn'_1 \ldots defn'_n\ \mathbf{e}' : t}\,\textbf{[prog]}$$

$$\frac{P\ \vdash^T_J\ t_j \quad \text{for } j\in[1,n] \qquad P,c\ \vdash^T_J\ \mathbf{meth}_k \Rightarrow \mathbf{meth}'_k \quad \text{for each } k \in [1,p]}{P\ \vdash^T_J\ \begin{array}{c}\textbf{class}\ c\ \textbf{extends}\ c'\ \{t_1\ f_1\ \ldots\ t_n\ f_n\ \mathbf{meth}_1\ \ldots\ \mathbf{meth}_p\} \Rightarrow \\ \textbf{class}\ c\ \textbf{extends}\ c'\ \{t_1\ f_1\ \ldots\ t_n\ f_n\ \mathbf{meth}'_1\ \ldots\ \mathbf{meth}'_p\}\end{array}}\,\textbf{[defn]}$$

$$\frac{P\ \vdash^T_J\ t_j \quad \text{for each } j \in [1,n] \qquad P,[\mathbf{this} : t_0,\ x_1 : t_1, \ldots, x_n : t_n]\ \vdash^T_J\ \mathbf{e} \Rightarrow \mathbf{e}' : t' \qquad t' \leq_P t}{P,t_0\ \vdash^T_J\ t\ \ m(t_1\ \ x_1, \ldots, t_n\ \ x_n)\{\mathbf{e}\} \Rightarrow t\ \ m(t_1\ \ x_1, \ldots, t_n\ \ x_n)\{\mathbf{e}'\}}\,\textbf{[meth]}$$

$$\frac{P\ \vdash^T_J\ c}{P,\Gamma\ \vdash^T_J\ \mathbf{new}\ c \Rightarrow \mathbf{new}\ c : c}\,\textbf{[new]}$$

$$\frac{x \in \text{dom}(\Gamma)}{P,\Gamma\ \vdash^T_J\ x \Rightarrow x : \Gamma(x)}\,\textbf{[var]}$$

$$\frac{P,\Gamma\ \vdash^T_J\ \mathbf{e} \Rightarrow \mathbf{e}' : t' \qquad \langle c.f,\ t\rangle \in_P t'}{P,\Gamma\ \vdash^T_J\ \mathbf{e}.f \Rightarrow \mathbf{e}'\underline{\ :\ c}.f : t}\,\textbf{[get]}$$

$$\frac{P,\Gamma\ \vdash^T_J\ \mathbf{e} \Rightarrow \mathbf{e}' : t' \qquad \langle c.f,\ t\rangle \in_P t' \qquad P,\Gamma\ \vdash^T_J\ \mathbf{e_v} \Rightarrow \mathbf{e}'_\mathbf{v} : t'' \qquad t'' \leq_P t}{P,\Gamma\ \vdash^T_J\ \mathbf{e}.f{=}\mathbf{e_v} \Rightarrow \mathbf{e}'\underline{\ :\ c}.f{=}\mathbf{e}'_\mathbf{v} : t}\,\textbf{[set]}$$

$$\frac{P\ \vdash^T_J\ t}{P,\Gamma\ \vdash^T_J\ \mathsf{nil} \Rightarrow \mathsf{nil} : t}\,\textbf{[nil]}$$

$$\frac{\begin{array}{c}P,\Gamma\ \vdash^T_J\ \mathbf{e} \Rightarrow \mathbf{e}' : t' \qquad\qquad P,\Gamma\ \vdash^T_J\ \mathbf{e_j} \Rightarrow \mathbf{e}'_\mathbf{j} : t'_j \\ t'_j \leq_P t_j \quad \text{for each} \quad j \in [1,n] \\ \langle m, (t_1, \ldots, t_n \rightarrow t), (x_1, \ldots, x_n), \mathbf{e_b}\rangle \in_P t' \\ \textsc{MostSpecific}(t', \langle m, (t_1, \ldots, t_n \rightarrow t), (x_1, \ldots, x_n), \mathbf{e_b}\rangle)\end{array}}{P,\Gamma\ \vdash^T_J\ \mathbf{e}.m(\mathbf{e_1}, \ldots, \mathbf{e_n}) \Rightarrow \mathbf{e}'\underline{\ :\ t'}.m(\mathbf{e}'_\mathbf{1}\underline{\ :\ t_1}, \ldots, \mathbf{e}'_\mathbf{n}\underline{\ :\ t_n}) : t}\,\textbf{[send]}$$

Figure 8: JavaLite typing rules (in $\vdash^T_J$, the T is for type and J is for Java)

calculus concise, we do not enforce this restriction, however our calculus may be easily extended to satisfy this restriction [FKF99].

The **[meth]** rule elaborates a method definition that belongs to a class t_0 only if each parameter's type exists and if the method body $\mathbf{e}$ elaborates and has a type t' that is a subtype of the method return type t. The environment Γ used to elaborate each expression phrase is set in the **[meth]** rule. Γ maps **this** and each method parameters to its corresponding type.

Type elaboration for class instantiation is realized with **[new]**. The expression **new** c elaborates only if the class c exists in a program P. As given in **[var]**, the type of a variable is given by the environment Γ.

In **[get]**, a variable assignment $\mathbf{e}.f$ elaborates into $\mathbf{e}'\underline{\ :\ c}.f\ : t$ only if $\mathbf{e}$ elaborates to $\mathbf{e}'$ with a type t' and f exists in t'. In order to be looked up in t', the field f needs to be prefixed with c that designates the class from which the field f originates and has the type t.

The **[set]** rule is very similar to **[get]**. Elaboration of the assigned value $\mathbf{e_v}$

$$\begin{array}{lll}
P \vdash_J & \langle \mathbf{E}[\mathbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[o], \mathcal{S}[o \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P c\}\rangle]\rangle & [\mathit{new}] \\
 & \text{where } o \notin \mathrm{dom}(\mathcal{S}) & \\
P \vdash_J & \langle \mathbf{E}[o\underline{\;:\;c'}.f], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[v], \mathcal{S}\rangle & [\mathit{get}] \\
 & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \ \text{ and } \mathcal{F}(c'.f) = v & \\
P \vdash_J & \langle \mathbf{E}[o\underline{\;:\;c'}.f = v], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[c'.f \mapsto v]\rangle]\rangle & [\mathit{set}] \\
 & \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle & \\
P \vdash_J & \langle \mathbf{E}[o\underline{\;:\;t}.m(v_1\underline{\;:\;t_1}, \ldots, v_n\underline{\;:\;t_n})], \mathcal{S}\rangle \hookrightarrow & \\
 & \langle \mathbf{E}[\mathbf{e}[o/\mathbf{this},\ v_1/x_1,\ \ldots,\ v_n/x_n]], \mathcal{S}\rangle & [\mathit{send}] \\
 & \quad \text{where } \mathcal{S}(o) = \langle c, \mathcal{F}\rangle \text{ and} & \\
 & \quad \langle m,\ (t_1, \ldots, t_n \to t'),\ (x_1,\ \ldots,\ x_n),\ \mathbf{e}\rangle\ \in_P\ c & \\
P \vdash_J & \langle \mathbf{E}[\mathsf{nil}\underline{\;:\;c'}.f], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[\mathit{error}\ \text{“}\mathit{eval}\text{”}], \mathcal{S}\rangle & [\mathit{get\text{-}err}] \\
P \vdash_J & \langle \mathbf{E}[\mathsf{nil}\underline{\;:\;c'}.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[\mathit{error}\ \text{“}\mathit{eval}\text{”}], \mathcal{S}\rangle & [\mathit{set\text{-}err}] \\
P \vdash_J & \langle \mathbf{E}[\mathsf{nil}\underline{\;:\;t}.m(v_1,\ \ldots,\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathbf{E}[\mathit{error}\ \text{“}\mathit{eval}\text{”}], \mathcal{S}\rangle & [\mathit{send\text{-}err}]
\end{array}$$

Figure 9: Reductions for JavaLite (the J in $\vdash_J$ is for Java)

should have the type t'', a subtype of t.

As given in **[nil]**, the nil pseudo variable may reduce into any type t.

For a given method call, the **[send]** typing rule annotates each argument with the static type of the declared argument in the method definition. This annotation will be used to lookup the method implementation at runtime. This is essential to handle method overloading.

When typing a method call, we guarantee the arguments have the type the method demands, and assume the result will have the type the method promises.

Java allows for method invocation type conversion. In other words this means that an exact match in the type of a parameter and the corresponding argument is not enforced. We say it is a match if the type of argument may be converted to the type of the corresponding parameter by method invocation conversion. When resolving the method signature, several methods may be applicable in presence of overloaded methods. In presence of overloaded methods, Java (and therefore JavaLite) selects the methods that "fits" best, called *most specific* [TT01].

Informally, *meth* is more specific than *meth'* if any invocation handled by *meth* can also be handled by *meth'*. More precisely, it means that for two methods *meth* and *meth'* having the same arity such that $meth = \langle m, (t_1,\ \ldots,\ t_n \to t), x^*, e\rangle$ and $meth' = \langle m, (t'_1,\ \ldots,\ t'_n \to t'), x^*, e'\rangle$, each t_j can be converted to t'_j for $j \in [1, n]$. Since we do not support interfaces and primitive types, t may be converted into t' if $t \leq_P t'$. The predicate $\textsc{MoreSpecific}((t_1, \ldots, t_n), (t'_1, \ldots, t'_n))$ is true if *meth* is more specific than *meth'*. The predicate is defined as:

$$\begin{array}{l}\textsc{MoreSpecific}((t_1,\ldots,t_n),(t'_1,\ldots,t'_n)) \iff \\ \forall\ j \in [1,n]\ \ t_j \leq_P t'_j\end{array}$$

A method *meth* is strictly more specific than another method *meth'* if and only if *meth* is more specific than *meth'* and *meth'* is not more specific than *meth*:

$$\begin{array}{l}\textsc{StrictlyMoreSpecific}((t_1,\ldots,t_n),(t'_1,\ldots,t'_n)) \iff \\ \ \textsc{MoreSpecific}((t_1,\ldots,t_n),(t'_1,\ldots,t'_n)) \\ \ \wedge\ \neg\textsc{MoreSpecific}((t'_1,\ldots,t'_n),(t_1,\ldots,t_n))\end{array}$$

A method is said to be maximally specific for a method invocation if it is applicable and there is no other applicable method that is strictly more specific. We are now ready to provide a definition for the most specific method. The predicate $\textsc{MostSpecific}(c, \langle m,(t_1,\ldots,t_n \to t), x^*, \mathbf{e}\rangle)$ is true if the method belongs to c and the method is the only strictly more specific method in c.

$$\begin{array}{l}\textsc{MostSpecific}(c, \langle m,(t_1,\ldots,t_n \to t), x^*, \mathbf{e}\rangle) \iff \\ \langle m,(t_1,\ldots,t_n \to t), x^*, \mathbf{e}\rangle \in_P c \quad \wedge \\ \forall\ \langle m,(t'_1,\ldots,t'_n) \to t'), x'^*, \mathbf{e}'\rangle\ \in_P\ c\ \ ,\ \ t_j \neq t'_j\ \ j \in [1,n] \\ \textsc{StrictlyMoreSpecific}((t_1,\ldots,t_n),(t'_1,\ldots,t'_n))\end{array}$$

The source declaration of any field or method in a class is computed with $\min_P$ by finding the minimum (*i.e.,* farthest from the root) superclass that declares the field or method.

The MostSpecific predicate ensures that only one method is the most specific. If the most specific method cannot be found (or said in another way: if no method is more specific that the other ones), the predicate is not true and the type elaboration ends.

The list of reduction rules for JavaLite is given in Figure 9. Those rules are pretty standard and are very similar to SmalltalkLite's set of rules. The type annotations $\underline{t_1}$, …, $\underline{t_n}$ contained in the message send are used to lookup the method.

5.3. The lump embedding

The lump embedding is a simple method for giving operational semantics to multi-language systems [MF07]. It has been designed to be expressive enough to support a wide variety of embedding strategies. This method is based on simple constructs called *boundaries*, cross-language casts that regulate both control flow and value conversion between languages.

We extend the two calculi given above with syntactic boundaries between JavaLite and SmalltalkLite, a kind of cross-language cast that indicates a switch of languages: Java values can appear in JSmall and JSmall values can appear in Java. The extensions is shown in Figure 10.

Syntax. First of all, a program should permit SmalltalkLite and JavaLite definitions to coexist. The notion of program is refined accordingly. The

$$P = \mathbf{defn}^* \; \mathrm{defn}^* \; \epsilon$$

$$\begin{array}{rcl} \mathbf{e} & = & \ldots \mid \; [\![\mathrm{e}]\!]^{t}_{SJ} \\ \mathbf{v} & = & \ldots \mid \; [\![\mathrm{v}]\!]^{\mathbf{L}}_{SJ} \\ t & = & \ldots \mid \; \mathbf{L} \\ \mathbf{E} & = & \ldots \mid \; [\![\mathrm{E}]\!]^{t}_{SJ} \end{array} \qquad \begin{array}{rcl} \mathrm{e} & = & \ldots \mid \; [\![\mathbf{e}]\!]^{t}_{JS} \\ \mathrm{v} & = & \ldots \mid \; [\![\mathbf{v}]\!]^{t}_{JS} \quad \text{where } t \neq \mathbf{L} \\ \mathrm{E} & = & \ldots \mid \; [\![\mathbf{E}]\!]^{t}_{JS} \end{array}$$

$$\frac{P,\Gamma \vdash^{T}_{J} \mathbf{e} : t' \qquad t' \leq_P t}{P,\Gamma \vdash^{T}_{S} [\![\mathbf{e}]\!]^{t}_{JS} \Rightarrow [\![\mathbf{e}]\!]^{t}_{JS} : \mathit{TST}} \qquad \frac{P,\Gamma \vdash^{T}_{S} \mathrm{e} : \mathit{TST}}{P,\Gamma \vdash^{T}_{J} [\![\mathrm{e}]\!]^{t}_{SJ} \Rightarrow [\![\mathrm{e}]\!]^{t}_{SJ} : t}$$

$$P \vdash_J \langle \mathbf{E}\left[\left[\!\!\left[[\![\mathbf{v}]\!]^{t}_{JS} \right]\!\!\right]^{t}_{SJ}\right], \mathcal{S}\rangle \;\hookrightarrow\; P \vdash_J \langle \mathbf{E}[\mathbf{v}], \mathcal{S}\rangle$$

$$P \vdash_J \langle \mathbf{E}\left[[\![\mathrm{v}]\!]^{t}_{SJ}\right], \mathcal{S}\rangle \;\hookrightarrow\; \langle \mathbf{E}\left[[\![\mathrm{error}\ \text{“value”}]\!]^{t}_{SJ}\right], \mathcal{S}\rangle$$
$$\text{if } \mathrm{v} \neq [\![\mathbf{v}']\!]^{t}_{JS} \text{ and } t \neq \mathbf{L}$$

$$P \vdash_S \langle \mathrm{E}\left[\left[\!\!\left[[\![\mathrm{v}]\!]^{t}_{SJ} \right]\!\!\right]^{t}_{JS}\right], \mathcal{S}\rangle \;\hookrightarrow\; P \vdash_S \langle \mathrm{E}[\mathrm{v}], \mathcal{S}\rangle$$

Figure 10: Extensions of SmalltalkLite and JavaLite to form the lump embedding (TST is The Smalltalk Type)

expression part of a program (the program “starting point”) is a JavaLite expression. This enforces the embedding of the Smalltalk calculus in the Java one.

Then, we add boundaries as a new kind of expression in each language. We extended e to produce $[\![\mathbf{e}]\!]^{t}_{JS}$ and extended $\mathbf{e}$ to produce $[\![\mathrm{e}]\!]^{t}_{SJ}$. The type t indicates the type JavaLite will consider the expression on its side of the boundary. The SJ subscript means “Smalltalk inside, Java outside” and JS means “Java inside, Smalltalk outside”.

A boundary is a reference to an alien object (*i.e.,* object which lives in a different language). An alien object may be referenced several times, each reference expressed with a boundary.

Note that in the remaining of the formalization we extended the color and font convention to terminal and non-terminal elements that are contained in a boundary. This will hopefully ease the reading of this section.

Typing rules. In our lump embedding extension, we add a new type $\mathbf{L}$ (for “lump”), a direct subtype of `Object`, to JavaLite and we add two new typing rules, one for each new syntactic form. The new SmalltalkLite judgment says that an $[\![\mathbf{e}]\!]^{t}_{JS}$ boundary is well-typed if the JavaLite type system proves that $\mathbf{e}$ has type t' with $t' \leq_P t$: a SmalltalkLite program type-checks if it is closed and all its JavaLite subterms have a subtype the program claims they have. The new JavaLite judgement says that $[\![\mathrm{e}]\!]^{t}_{SJ}$ has type t if it is closed and e type-checks under SmalltalkLite's typing system.

In both case, t can be any type. If $t = \mathbf{L}$ a native SmalltalkLite value crosses the boundary, which will be considered as a lump in JavaLite; if $t \neq \mathbf{L}$ a JavaLite value crosses the boundary, which will be a lump in SmalltalkLite.

JavaLite's typing rules guarantee that values that appear inside $[\![\mathbf{v}]\!]^{\mathbf{L}}_{JS}$ expressions will in fact be lump values. SmalltalkLite offers no such guarantee, so the rule for eliminating a $[\![\ ]\!]^{t}_{SJ}$ boundary must apply whenever the Smalltalk expression is a value at all. This is why $\mathbf{E}\left[[\![\mathrm{v}]\!]^{t}_{SJ}\right]$ may lead to an error.

Note that nil may be tagged. Since nil is a value, it may be enclosed within a boundary, meaning that a reference to nil may be tagged.

Operational semantics. To allow SmalltalkLite expression to evaluate inside JavaLite expressions and vice versa, we define evaluation contexts mutually recursive at boundaries. We extended $\mathbf{E}$ with $[\![\mathrm{E}]\!]^{t}_{SJ}$ to allow SmalltalkLite expressions to be evaluated in JavaLite and E with $[\![\mathbf{E}]\!]^{t}_{JS}$ to evaluate JavaLite expressions in SmalltalkLite.

A typing error may occur during the evaluation when an embedded SmalltalkLite expression is mistyped. SJ boundaries with a non-lump type that contain Smalltalk values and JS boundaries of type $\mathbf{L}$ that contain Java values should reduce, since they represent foreign values returning to a native context. This is done by canceling matching boundaries, as the reductions rules in Figure 10 show.

5.4. The pellucid embedding

In this subsection we extend the lump embedding with the necessary typing and evaluating rules to handle message sends and to cast boundaries. This new embedding is called the pellucid embedding.

The first rule given in Figure 11 describes the evaluation of a message sent in JavaLite to a boundary. Since no assumption can be made on the return type of the method invocation, the boundary type is $\mathbf{L}$.

The second rule describes a message sent in SmalltalkLite to a boundary. This rule elaborates a call into a Java call. This call has to be annotated with some types to properly define the method signature to lookup. By paying a close attention, one should see that the way this call is annotated is the same one performed by the Java typing rules. The only difference is that this annotation is performed at runtime. It might therefore fail. In that case, an error is raised.

In each of these two extra-boundary calls, values provided as message parameters must be issued by the same language that the object receiver is, *e.g.*, sending a message to $[\![\mathbf{o}]\!]^{t}_{JS}$ requires arguments to be boundaries $[\![\mathbf{v_k}]\!]^{t_k}_{JS}$.

In this second rule, it may be tempting to say that $t_j = t'_j$. Such assumption cannot be made since t'_j are arbitrary set when by the dynamic type tag one wants to give. This is why we need to retrieve the minimal (most specific) method.

$$\text{[J-SEND]}$$
$$P \vdash_J \langle \mathbf{E}\left[[\![\mathrm{o}]\!]^{t}_{SJ}.m([\![\mathrm{v_1}]\!]^{t_1}_{SJ}, \ \ldots, \ [\![\mathrm{v_n}]\!]^{t_n}_{SJ})\right], \mathcal{S}\rangle \quad \hookrightarrow \quad \langle \mathbf{E}\left[[\![\mathrm{o.m}(\mathrm{v_1}, \ \ldots, \ \mathrm{v_n})]\!]^{\mathbf{L}}_{SJ}\right], \mathcal{S}\rangle$$

$$\text{[S-SEND]}$$
$$P \vdash_S \langle \mathrm{E}\left[[\![\mathbf{o}]\!]^{t'}_{JS}.m([\![\mathbf{v_1}]\!]^{t'_1}_{JS}, \ \ldots, \ [\![\mathbf{v_n}]\!]^{t'_n}_{JS})\right], \mathcal{S}\rangle \quad \hookrightarrow$$
$$\begin{cases} \langle \mathrm{E}\left[[\![\mathbf{o\underline{\ :\ t'}.m(v_1\underline{\ :\ t_1}, \ \ldots, \ v_n\underline{\ :\ t_n})}]\!]^{t}_{JS}\right], \mathcal{S}\rangle \\ \text{where} \\ \quad t_j = \min_P\{t'' \mid t'_j \leq_P t''\} \ \text{ for each } \ j \in [1,n] \\ \quad \langle m, (t_1, \ \ldots, \ t_n \ \to \ t), (x_1, \ \ldots, \ x_n), \mathbf{e_b}\rangle \ \in_P \ t' \\ \quad \textsc{MostSpecific}(t', \langle m, (t_1, \ \ldots, \ t_n \ \to \ t), (x_1, \ \ldots, \ x_n), \mathbf{e_b}\rangle) \\ \\ \langle \mathrm{E}[\mathrm{error} \ \text{“call”}], \mathcal{S}\rangle \quad \text{else} \end{cases}$$

$$\text{[S-POP]}$$
$$P \vdash_S \langle \mathrm{E}\left[\left\lfloor [\![\mathbf{v}]\!]^{t}_{JS} \right\rfloor^{t'}\right], \mathcal{S}\rangle \quad \hookrightarrow \quad \langle \mathrm{E}\left[[\![\mathbf{v}]\!]^{t'}_{JS}\right], \mathcal{S}\rangle$$
$$\text{where } \ t \leq_P t' \ \vee \ t' \leq_P t \quad \text{and } \ c \leq_P t' \ \text{ for } \ \mathcal{S}(\mathbf{v}) = \langle c, \_\rangle$$

$$\frac{P,\Gamma \vdash^T_J t \qquad P,\Gamma \vdash^T_J t' \qquad t \leq_P t' \ \vee \ t' \leq_P t}{P,\Gamma \vdash^T_S \left\lfloor [\![\mathbf{v}]\!]^{t}_{JS} \right\rfloor^{t'} \Rightarrow \left\lfloor [\![\mathbf{v}]\!]^{t}_{JS} \right\rfloor^{t'} : TST} \qquad \begin{array}{lcl} \mathbf{E} & = & \ldots \mid \ \lfloor \mathbf{E} \rfloor^{t} \\ \mathrm{E} & = & \ldots \mid \ \lfloor \mathrm{E} \rfloor^{t} \end{array}$$

Figure 11: The pellucid embedding

We introduce the operator $\lfloor \ \rfloor^{t}$ to upcast and downcast boundaries. By making this operator accessible in JSmall (the `type:` keyword), the link between the pellucid embedding and the informal description given in the previous section should now be clearer.

The cast performed by $\lfloor \ \rfloor^{t}$ is checked when the cast is interpreted. The new static type should be either a supertype or a subtype ($t \leq_P t' \ \vee \ t' \leq_P t$) of the type of the object. Furthermore, a value cannot be downcasted with a type that is a subtype of the real type of the value ($c \leq_P t'$).

Originally conceived to make Scheme and ML interoperable, the lump embedding is applied in this paper in a different setting with Java and Smalltalk. This embedding has been extended to enable message passing from one language to the second one. This new embedding is a natural calculus extension that does not rely on any particular feature expect that the two considered languages should be able to send messages toward objects. This embedding may be successfully applied to different languages (e.g., Python and C#).

The last execution rule ($\hookrightarrow$) enforces the coherence of the provided dynamic type tag. The tag must be coherent with the wrapped Java value. It could be an upcast or downcast (if an upcast has been previously made). However the type tag cannot be a subtype of the real type. This is like downcasting a Java value with a type below the real type. It is important to notice that these constraints are not mandatory to get all the benefits of dynamic type tags. In order to ease the implementation, one may want to leave them aside when implementing dynamic type tag. Improper type tag will be signaled with an error upon method call.

Matthews and Findler's method supports higher-order functions, which can represent various data structures and operations. This means that an operation such as function application to be performed in the foreign language is definable as a lambda term in the host language. JavaLite and SmalltalkLite do not have explicit higher-order terms[9]. As a consequence, for each operation to be performed in the foreign language, the operational semantics must be extended with a dedicated reduction rule that takes embedded foreign terms as arguments and generates the corresponding foreign term. Figure 11 only show this for method invocation and type cast.

5.5. Properties 1: type soundness

Type soundness says that a term may either reduce, or is a value or stops with a specific errors generated by reductions rules. The type-soundness theorem is proved by using the standard technique of subject reduction and progress theorems [WF94].

Note that because of the way we have combined the two languages, type soundness entails that both languages are type-sound with respect to *their own* type systems – in other words, that both single-language type soundness proofs are special cases of the soundness theorem for the entire system. Type soundness of the lump embedding has been demonstrated [MF07]. To conserve space, we do not demonstrate the type soundness of JavaLite, which does not present any major issue. The demonstration for SmalltalkLite is trivial since there is one unique type, not presented therefore. We will rather focus on the extension made on them.

We refer to the empty set with []. To prove this theorem, we will use the classical conserve, subject reduction and progress lemmas. Since the rewriting rules reduce annotated terms, we derive two new type judgements, $\vdash^{T}_{J}$ and $\vdash^{T}_{S}$, that relate annotated terms to show that reduction preserves type correctness. These new rules perform the same checks as the rule it is derived from without adding annotation. $\vdash^{T}_{S}$ performs trivial checks: all elements must have the type TST. For **[set]**, **[get]** and **[send]** that annotate the program being type checked, $\vdash^{T}_{J}$ performs the same check than $\vdash^{T}_{J}$ without modifying the program. Since the type checking rules for the lump embedding (Figure 10) and

[9]One could argue that an object is a higher-order term since it can carry behavior and it is not known by the client which behavior it is.

the pellucid embedding (Figure 11) do not annotate the program, no particular treatment is required.

Lemma 1. (Conserve).
If $\vdash^T_J\ P \Rightarrow P' : t$ and $P' = defn_1 \dots defn_n\ \mathbf{e}$, then $P', [\,]\ \vdash^T_J\ \mathbf{e} : t$

Proof. If $\vdash^T_J\ P \Rightarrow P' : t$ and $P' = defn_1 \dots defn_n\ \mathbf{e}$, then $P, [\,]\ \vdash^T_J\ \mathbf{e} : t$ per definition, thus $P', [\,]\ \vdash^T_J\ \mathbf{e} : t$ $\square$

The subject reduction lemma states that each evaluation step preserves the type correctness of the expression-store pair $\langle e, \mathcal{S}\rangle$, e being either $\mathbf{e}$ or e. Said in another word, for a given configuration on the left-hand side of an evaluation step, it exists a type environment that establishes the expression's type. This environment must be consistent with the store. This consistency is given by the judgment $P, \Gamma\ \vdash_\sigma \mathcal{S}$: it is true when all objects contained in $\mathcal{S}$ have a binding in Γ and all instances variables are consistent with the class of the object in which it the variable is defined.

Lemma 2. (Subject Reduction). Reduction preserve types.

- If $P, \Gamma\ \vdash^T_J\ e : t$ and $P, \Gamma\ \vdash_\sigma \mathcal{S}$ and $P\ \vdash_J\ \langle \mathbf{e}, \mathcal{S}\rangle \hookrightarrow \langle \mathbf{e}', \mathcal{S}'\rangle$, then $\mathbf{e}'$ is an error or $\exists\Gamma'$ such that $P, \Gamma'\ \vdash^T_J\ e' : t'$ and $P, \Gamma'\ \vdash_\sigma \mathcal{S}'$ and $t' \leq_P t$

- If $P, \Gamma\ \vdash^T_S\ e : \mathit{TST}$ and $P, \Gamma\ \vdash_\sigma \mathcal{S}$ and $P\ \vdash_S\ \langle \mathrm{e}, \mathcal{S}\rangle \hookrightarrow \langle \mathrm{e}', \mathcal{S}'\rangle$, then e' is an error or $\exists\Gamma'$ such that $P, \Gamma'\ \vdash^T_S\ e' : \mathit{TST}$ and $P, \Gamma'\ \vdash_\sigma \mathcal{S}'$

Proof. The proof examines reduction steps for $\vdash_J$ and $\vdash_S$. If the execution has not halted, then for the new environment Γ' we constructed, we show that the two related consequents of the theorem are satisfied, relative to the new expression, store and environment.

J-SEND: If $\mathbf{e} = [\![\mathrm{o}]\!]^{t''}_{SJ}.m([\![\mathrm{v_1}]\!]^{t''_1}_{SJ}, \ \dots, \ [\![\mathrm{v_n}]\!]^{t''_n}_{SJ})$ and $P, \Gamma\ \vdash^T_J\ e : t$ then $\mathbf{e}' = [\![\mathrm{o.m(v_1,\ \dots,\ v_n)}]\!]^{\mathbf{L}}_{SJ}$ according to the reduction rule, $\Gamma = \Gamma'$ and $P, \Gamma\ \vdash^T_J\ e' : t'$ and $t' \leq_P t$ according to the typing rule in the lump embedding.

S-SEND: if $\mathrm{e} = [\![\mathbf{o}]\!]^{t'}_{JS}.m([\![\mathbf{v_1}]\!]^{t'_1}_{JS}, \ \dots, \ [\![\mathbf{v_n}]\!]^{t'_n}_{JS})$ and $P, \Gamma\ \vdash^T_S\ e : t$ then $\mathrm{e}' = [\![\mathbf{o\underline{\ :\ t'}.m(v_1\underline{\ :\ t_1},\ \dots,\ v_n\underline{\ :\ t_n})}]\!]^{t}_{JS}$ according to the reduction rule and $\Gamma = \Gamma'$ according to the typing rule in the lump embedding.

S-POP: if $\mathrm{e} = \left\lfloor [\![\mathbf{v}]\!]^{t}_{JS} \right\rfloor^{t'}$ and $P, \Gamma\ \vdash^T_S\ e : t$ then $\mathrm{e}' = [\![\mathbf{v}]\!]^{t'}_{JS}$ according to the reduction rule and $\Gamma = \Gamma'$ according to the typing rule in the lump embedding.

Other rules of the lump embedding are proven [MF07]. JAVALITE and SMALLTALKLITE rules are proven assuming their type soundness. $\square$

Lemma 3. (Progress). For all Java expression $\mathbf{e}$, JSmall expression e, both of the following hold:

- if $P, \Gamma \vdash_J^T \mathbf{e} : t$, then either $\mathbf{e}$ is a Java value, or there exists an $\mathbf{e}'$ such that $P \vdash_J \langle \mathbf{e}, \mathcal{S} \rangle \hookrightarrow \langle \mathbf{e}', \mathcal{S}' \rangle$, or $\langle \mathbf{e}, \mathcal{S} \rangle$ reduces to an error.
- if $P, \Gamma \vdash_S^T \mathrm{e} : TST$, then either e is a JSmall value, or there exists an e' such that $P \vdash_S \langle \mathrm{e}, \mathcal{S} \rangle \hookrightarrow \langle \mathrm{e}', \mathcal{S}' \rangle$, or $\langle \mathrm{e}, \mathcal{S} \rangle$ reduces to an error.

Proof. By simultaneously analyzing all the possible cases for the current redex in $\mathbf{e}$ and e (in the case that they are not a value).

J-SEND, S-SEND and S-POP satisfy the hypothesis by directly applying the reduction rules of the pellucid embedding.

Other rules of the lump embedding are proven [MF07]. JAVALITE and SMALLTALKLITE rules are proven assuming their type soundness. □

The type soundness property can be formulated as the following theorem:

Theorem. (Pellucid soundness theorem).
Given *Lemma 1*, *Lemma 2* and *Lemma 3*, if $\vdash_J^T P \Rightarrow P' : t$ and $P = \mathbf{defn}^* \ \mathrm{defn}^* \ \mathbf{e}$ and the execution of P terminates, then either:

- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathbf{o}, \mathcal{S} \rangle$ and $\mathcal{S}(\mathbf{o}) = \langle t', \mathcal{F} \rangle$ and $t' \leq_P t$; or
- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathsf{nil}, \mathcal{S} \rangle$; or
- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathit{error\ “call”}, \mathcal{S} \rangle$; or
- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathit{error\ “value”}, \mathcal{S} \rangle$; or
- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathit{error\ “eval”}, \mathcal{S} \rangle$; or
- $P \vdash_J \langle \mathbf{e}, [\,] \rangle \hookrightarrow^* \langle \mathit{error\ “eval”}, \mathcal{S} \rangle$

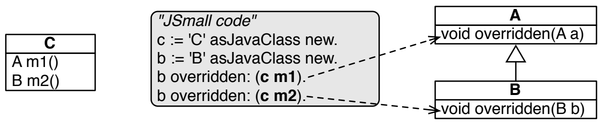
The theorem says that any state reachable from the original is either a value, one of the specified errors, or available for another step.

The typing rules of JAVALITE and SMALLTALKLITE do not explicitly deal with errors. However, this is not necessary in our case. If an JAVALITE and SMALLTALKLITE expression is well typed, then it cannot be stuck.

5.6. Properties 2: determining the execution flow

Ideally, dynamic type tags should be automatically set when possible without requiring a manual intervention. This means that a programmer does not need to use the `type:` message (written $\lfloor\ \rfloor^t$ in the formal model) for values returned from a Java method call. Intuitively, if a Java method has a return type t, then values returned from invocations of the method must be statically annotated with t.

Consider the three Java classes `A`, `B`, and `C` described in Figure 12. The class `C` contains two methods, `m1()` and `m2()`, having a return type `A`, `B`, respectively.



```
"JSmall code"
c := 'C' asJavaClass new.
b := 'B' asJavaClass new.
b overridden: (c m1).
b overridden: (c m2).
```

Figure 12: Type of return value impacts the control flow (A, B, C are Java classes).

Invoking one of these methods returns a value (instance or null) statically annotated with A or B. This annotation is used to resolve future invocations, like when calling overridden(...).

The pellucid embedding reflects this as illustrated by the following reduction steps:

$$\begin{array}{rlcl}
[\![\textbf{new B()}]\!]^{B}_{JS}. & \mathit{overridden}(& [\![\textbf{new C()}]\!]^{C}_{JS}.\mathit{m1}() &) \hookrightarrow^{*} \\
[\![\boldsymbol{\nu_1}]\!]^{B}_{JS}. & \mathit{overridden}(& [\![\boldsymbol{\nu_2}]\!]^{C}_{JS}.\mathit{m1}() &) \hookrightarrow \\
[\![\boldsymbol{\nu_1}]\!]^{B}_{JS}. & \mathit{overridden}(& [\![\boldsymbol{\nu_2}\ \underline{:\ \textbf{B}}\textbf{.m1()}]\!]^{A}_{JS} &) \hookrightarrow \\
[\![\boldsymbol{\nu_1}]\!]^{B}_{JS}. & \mathit{overridden}(& [\![\boldsymbol{\nu_3}]\!]^{A}_{JS} &)
\end{array}$$

Consequently, the method overriden(A) will be invoked. This does not come as a surprise since the pellucid embedding (Figure 11) takes care of using the Java method return type to set the type of $[\![...]\!]^{t}_{JS}$. As a consequence, Java objects handed over to JSmall are annotated with the type specified in the Java API. JSmall programmers are therefore relieved from manually setting these annotations on Java objects.

6. Related Work

This section relates the work presented in this paper using two different perspectives. The first one (Section 6.1) reviews all the dynamically typed scripting languages that we are aware of on Java and .Net, and compare them against the dynamic type tag presented in this paper. Then, secondly (Section 6.2), we review more theoretical approaches against the pellucid embedding.

6.1. Dynamically typed languages

Clojure. Clojure[10] is a dialect of Lisp and is predominantly a functional programming language which features a rich set of immutable, persistent data structures. As most dynamic languages running on a JVM, Clojure is designed to be a hosted language, sharing the JVM type system, GC, thread, etc. Clojure offers the *dot-target-member* notation for Java calls. A Java method call in Clojure has the following pattern: (. *expression* (instanceMethodName *args**)) which calls the method instanceMethodName on expression with args* as arguments. The

[10] http://clojure.sourceforge.net

dot-target-member uses the dynamic type of argument values to resolve overloaded methods.

Jython. Jython[11], an implementation of Python in Java, behaves in a way similar to JRuby. The dynamic type of Java objects are used to resolve the signature of the Java method to invoke upon message send. Java objects in Jython are instances of the Java class PyJavaInstance, a subclass of PyInstance[12], itself a subclass of PyObject.

JScheme. JScheme is a dialect of Scheme for JVM with a very simple interface to Java. The *Java Dot Notation*[13] provides JScheme with an access to most Java constructors, methods, and fields for all Java classes. The idea is to annotate Java calls with a dot notation to indicate which Java elements have to be invoked. Let us consider the following JScheme code obtained from the JScheme webpage and its corresponding Java version:

```
; JScheme code
(define win (Frame. "Hello"))
(.resize win 200 300)
(.Container.resize win 200 300)
(.Component.resize win 200 300)

// Java code
Frame win = new Frame("Hello");
win.resize(200, 300);
((Container) win).resize(200,300);
((Component) win).resize(200,300);
```

The Java dot notation enables a particular overloaded method to be called if the set of overloaded methods are spread over a type hierarchy. Upcasting the Java object receiver may expose overloaded methods. Let us assume the two classes:

```
class XMLElement {
  void printOn(OutputStream o) {}}

class StructuredXMLElement
        extends XMLElement {
  void printOn(ObjectOutputStream o) {}}
```

JScheme's Java Dot Notation allows each printOn to be called on an instance of StructuredXMLElement with an instance of ObjectOutputStream as parameter. For example (.XMLElement.printOn (StructuredXMLElement.) (ObjectOutputStream.)) invokes the Java method printOn (OutputStream) on an instance of StructuredXMLElement. The receiver is dynamically upcasted to XMLElement, which reveals the method printOn(OutputStream). In contrary to the dynamic

[11] http://www.jython.org/Project/userguide.html
[12] http://jython.svn.sourceforge.net/viewvc/jython/trunk/jython/src/org/python/core/PyInstance.java?view=markup
[13] http://jscheme.sourceforge.net/jscheme/doc/javaprimitives.html

type tag, the Java Dot Notation is not able to select a particular overloaded methods when they are defined in the same class.

LiveConnect. LifeConnect is a feature of Web browsers that allows Java and JavaScript software to intercommunicate within a Web page. In an earlier version of LiveConnect, the way to invoke overloaded Java methods was to find the first applicable method that is enumerated by the Java VM. "Applicable" means that the method name and the number of arguments match and that each of the JavaScript arguments can be concerted to the corresponding Java type listed in the method's signature.

The enumeration of methods by the Netscape JVM always reflected the order of methods in the class file. Rearrangement of methods in the Java source files was often required to invoke the desired method. This was a source of pain since source code was not often available and because the static nature of this method resolution algorithm sometimes made it impossible to choose a different method at each invocation site[14].

Version 3 of LiveConnect provides a more intelligent way to deal resolving overridden methods when calling Java methods. This version of LiveConnect is implemented in Rhino.

Javascript in Java. Rhino is an open-source implementation of JavaScript written in Java. Java objects may be instantiated and messages may be sent to them[15]. Rhino selects an overloaded method at runtime based on the type of the arguments in the same fashion than Jython. An error is raised upon ambiguous call.

Sixx. Sixx[16] is a Scheme based interpreter intended to keep its memory footprint low (around 20KB). Java methods are made first class entities in Sixx. The (method *className methodName argTypes**) special form returns a reification of the Java method named methodName having a signature that exactly matches argTypes*. It may be employed as follows:

```
; Sixx code
; call a static method
(define cos (method "java.lang.Math" "cos" "double"))
(print (cos 1.0))
```

Java methods are accessed through reflection. It therefore falls into the drawback already mentioned earlier (Section 2.1): methods are not looked up but directly evaluated. Polymorphism is not supported therefore.

Other dynamic languages for the JVM. Each language in the list given above addresses Java interoperability against method overloading. Other dy-

[14]http://www.mozilla.org/js/liveconnect/lc3_method_overloading.html
[15]http://www.mozilla.org/rhino/ScriptingJava.html
[16]http://dgym.homeunix.net/projects/sixx

namically typed languages are available for the JVM: Bex[17], Tea[18], Judo-Script[19], ObjectScript[20], Kanaputs[21], Groovy[22], JPiccola[23], Agora [GWDD06]. All those languages suffer from the same problems regarding overloading of Java methods.

IronPython. A new implementation Python has been implemented on the .Net platform. IronPython[24] gained a major attention recently. It supports an interactive console with fully dynamic compilation and is well integrated with the rest of the .NET Framework. All .NET libraries are available to Python programmers, while maintaining full compatibility with the Python language. IronPython has an `Overloads` property on all methods that will allow you to select a particular signature if needed. For example, the following `o.foo.Overloads[A](b)` will invoke the method `foo(A)`, independently of the dynamic type of `b`. This strategy falls into the problem cited earlier (Section 4).

Other dynamic languages for the .Net platform. A number of dynamically typed scripting language exist on the .Net platform. We reviewed Iron-Lisp[25], IronScheme[26], and Nua[27]. Unfortunately, these works have not reached a sufficient stage (implementation and documentation) to address the problem presented in this paper.

6.2. Combining dynamic and static typing

Recently, a number of researchers have suggested different ways to integrate static and dynamic typing into a single framework. Dynamic type tag presented in this paper is evaluated against these frameworks.

Contract and mirror. Gray *et al* [GFF05] proposed a fine-grained interoperability between a statically typed, object-oriented language and a dynamically typed, functional language. They conducted an experiment over Java and Scheme. These two languages have been extended by making functions available in Java, and objects available in Scheme. A notion of dynamically typed expressions is added to Java that makes Java more compatible with Scheme.

The pellucid embedding does not add new construct to the two languages it applies on. Instead, it add a new syntactic construct that allow for typing foreign objects. Moreover, the goal is slightly different since Gray *et al* focussed

[17]http://bexscript.sourceforge.net
[18]http://www.pdmfc.com/tea
[19]http://www.judoscript.com
[20]http://objectscript.sourceforge.net
[21]http://www.kanaputs.org
[22]http://groovy.codehaus.org
[23]http://www.iam.unibe.ch/ scg/Research/Piccola
[24]http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython
[25]http://www.codeplex.com/IronLisp
[26]http://www.codeplex.com/IronScheme
[27]Lua for the DLR: http://www.codeplex.com/Nua

on potentially delayed checked and coercions of data, whereas our embedding target accessing overloaded methods in a dynamic setting.

Gradual typing. Gradual typing provides the benefits of both static and dynamic checking by allowing a programmer to specify the portion of the program to be type checked at compile time or at run-time [ST07]. This is achieved by adding or removing type annotations on variables.

Similarly to gradual typing, the pellucid embedding promotes an early type checking when possible. However, gradual typing operates on one single language.

Safe unknown type. Lagorio and Zucca [LZ07] propose an extension of the Java type checker that allows programmer to not type in strategic places of their code. They introduced an unknown type wherever they do not want to commit to a certain type. Methods having parameters or result of the unknown type are translated to have unknown type replaced with a know type. The latter being statically computed with a sophisticated type system. Method overloading has not been considered so far.

Hybrid type checking. An extension of traditional static types is proposed with hybrid type checking [Fla06] to support precise specifications while preserving the ability to detect simple, syntactic errors at compile time. Hybrid type checking is inspired from prior work on soft typing [CF91] by extending it to reject ill-typed programs according.

A different perspective has been adopted in this paper. The pellucid embedding is an extension of classical type checker supporting method overloading. It annotates foreign object with a type annotation, and allows this type annotation to be manually set.

Blame typing. Wadler and Findler [WF07] have introduced a notion of blame (from contracts [FF02]) to a type system with casts. Programmers using this type system may add contracts to evolve dynamically typed program into statically typed programs (as with gradual types) or to evolve statically typed programs into programs with refinement types (as with hybrid types). Walder and Findler demonstrated that their blame typing is a flexible framework in which a number of type systems, including gradual typing and hybrid typing, may be expressed in.

A cast from source type S to target type T is written $(T \Leftarrow S)s$, where subterm s has type S and the whole term has type T. Blame typing adds a blame to a type cast: $(T \Leftarrow S)^{pn} s$.

A blame may either be positive (p) or negative (n). A positive blame is allocated if the term contained in the cast fails to satisfy the contract implied by the cast. A negative blame is allocated if the context containing the cast fails to satisfy the contract. The lump embedding (Section 5.3) may be expressed in blame typing since we defined cross-language casts between Java and Smalltalk. The pellucid embedding cannot be directly expressed in blame typing, since rewriting is necessary to achieve it. However, as a future work, we plan to use blame typing to provide a finer feedback in case of type failure.

Cross-language inheritance. Some dynamically typed scripting languages support inheritance across languages. A class defined in a language A may be subclassed in a different language B. Gray [Gra08] provides an approach to enable a Java class to be subclassed in Scheme and the other way around. Her compilation technique provides safe interoperability by expanding the source language to insert wrappers that transfer values between typed and untyped expressions. She uses two kind of wrappers: one that converts typed-values into untyped expression (a guard); and another that converts untyped values into typed expressions (a mimic).

Gray provides a safe approach for cross language lookup method mechanism. Overridden methods are supported. However, overloading, a characteristic widely supported by statically typed languages, has been left outside her work. How to deal with overloading is exactly the point of dynamic type tag.

7. Conclusion

The omnipresence of the Java virtual machine and the rigidity of statically typed Java application has greatly contributed to spreading the use of dynamically typed languages on this platform. It is primordial for those "high-level languages" to interact with the ambient Java environment. However, differences in the type system of Java and an embedded dynamic language make a complete interaction hard to obtain.

This paper presents an elegant solution for enabling an embedded dynamic language to call Java overloaded methods. This problem has been around for years and hasn't been properly addressed. Our idea is to augment the reference to a Java object with a type. This type is then used as a dynamic type tag when methods have to be called on Java objects from a dynamically typed language. A theoretical foundation conveying the essence of this mechanism is also presented. Although dynamic type tag is presented in the context of JSmall, we argue that this mechanism is applicable to other dynamic languages.

Embedding a radically different language is a challenging task which comes with numerous problems. Differences in the type system is one important issue which is tackled in this paper. Preserving the identify of converted object is also an issue shared by a large range of languages. In the future we plan to investigate on this. Another future work is to consider the generic case. Generics were deliberately left outside of this paper, however combining generic programming with dynamic type tag looks promising.

Acknowledgments. We gracefully thank Lorenzo Bettini, Stéphane Ducasse, Matthew Flatt, Oscar Nierstrasz, Hellen Silva, Eric Tanter, Laurence Tratt for their precious review and comments on an early draft. We also thank Jim Hugunin for his help with IronPython.

References

[AZ04] Davide Ancona and Elena Zucca. Principal typings for java-like languages. *SIGPLAN Not.*, 39(1):306–317, 2004.

[BCV08] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight Java with Dynamic and Static Overloading. *Science of Computer Programming*, 2008. To appear.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[BPP03] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge Computer Laboratory, J.J. Thomson Avenue, Cambridge. CB3 0FD. UK, 2003.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292, New York, NY, USA, 1991. ACM.

[CGPC06] Florin Craciun, Hong Yaw Goh, Corneliu Popeea, and Wei-Ngan Chin. Core-java: an expression-oriented java. In *Proceedings of OOPSLA '06, Companion*, pages 639–640. ACM.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, 2002.

[FH92] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[FKF99] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. Technical Report TR 97-293, Rice University, 1999.

[Fla06] Cormac Flanagan. Hybrid type checking. In *Proceedings of POPL '06*, pages 245–256, New York, NY, USA, 2006. ACM.

[GFF05] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. *SIGPLAN Not.*, 40(10):231–245, 2005.

[Gra08] Kathryn E. Gray. Safe cross-language inheritance. In *Proceedings of ECOOP'08*, volume 5142 of *LNCS*, pages 52–75.

[GWDD06] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection — a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[LZ07] Giovanni Lagorio and Elena Zucca. Just: safe unknown types in java-like languages. *Journal of Object Technology*, 6(2):71 – 100, February 2007.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *SIGPLAN Not.*, 42(1):3–10, 2007.

[Pie02] Benjamin Pierce. *Types and Programming Languages.* The MIT Press, 2002.

[ST07] Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proceedings of ECOOP'07*, volume 4609 of *LNCS*, pages 151–175.

[TT01] Satyam Tyagi and Paul Tarau. A most specific method finding algorithm for reflection based dynamic prolog-to-java interfaces. In *Proceedings of PADL '01*, pages 322–336, London, UK, 2001. Springer-Verlag.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[WF07] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 15–26, 2007.