Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 212 (2008) 177–189
www.elsevier.com/locate/entcs

Reasoning About Multi-Lingual Exception Handling Using RIPLS

John Ridgway[1]

Computer Science Department
Trinity College
Hartford, USA

Jack C. Wileden[2]

Computer Science Department
University of Massachussets
Amherst, USA

Abstract

Building multi-lingual software is a practical necessity. At present, with object-oriented programming the dominant paradigm, it is common to assemble software systems comprising components written in at least two different object-oriented languages. Modern object-oriented languages provide exception handling mechanisms as a means of enriching the signatures of methods with a specification of what to do if the method "fails", i.e., cannot carry out its intended (normal) function for some reason. Indeed, Java and C++ (and many other object-oriented languages, including C#) have remarkably similar exception handling mechanisms. As we demonstrate, however, those exception handling mechanisms do not necessarily interoperate smoothly when used in multi-lingual software systems.
We believe that our long-term goal of maximally effortless and error-free multi-lingual programming requires automated tools that are based on solid formal foundations. Toward that end, we have developed a formal language, which we call RIPLS, that can be used to rigorously study properties of multilingual software. In this paper, we demonstrate RIPLS and our approach by using it to study exception handling in multi-lingual object-oriented systems, and show how use of our methods can identify problems that standard techniques cannot. We then exhibit a correctly-working version of multi-lingual exception-handling and use our methods to confirm its correctness. Finally we discuss how experience with these RIPLS-based methods has informed our designs for automated tools that will implement correctly-working multi-lingual exception handling.
This work makes a significant contribution by demonstrating that formal, theoretical foundations can be used to solve practical problems in multi-lingual software development.

Keywords: Exception Handling, Formal Methods and Software, Multi-lingual Programming

[1] Email: John.Ridgway@trincoll.edu
[2] Email: wileden@cs.umass.edu

doi:10.1016/j.entcs.2008.04.061

1 Introduction

Building multi-lingual software is a practical necessity. At present, with object-oriented programming the dominant paradigm, it is common to assemble software systems comprising components written in at least two different object-oriented languages. For example, a Java Graphical User Interface (GUI) front-end is often coupled with a C++ computational program. Because current implementations of Java tend to be slow, and the GUI front-ends for C++ are less portable than those for Java, the multi-lingual version offers the best combination of portability and efficiency. Our long-term research program is aimed at making this kind of multi-lingual programming as effortless and error-free as possible.

Modern object-oriented languages provide exception handling mechanisms as a means of enriching the signatures of methods with a specification of what to do if the method "fails", i.e., cannot carry out its intended (normal) function for some reason. Indeed, Java and C++ (and many other object-oriented languages, including C#) have remarkably similar exception handling mechanisms. As we demonstrate in the next section, however, those exception handling mechanisms do not necessarily interoperate smoothly when used in multi-lingual software systems.

We believe that our long-term goal of maximally effortless and error-free multi-lingual programming requires automated tools that are based on solid formal foundations. Toward that end, we have developed a core language, which we call RIPLS (Reasoning about Interoperating Programming Language Systems), that can be used to rigorously study properties of multi-lingual software. Various extensions to RIPLS enable us to study specific properties or combinations of properties, and RIPLS and its extensions are used to inform, and eventually will be used to produce, automated tools to ease the development of error-free multi-lingual software.

Our approach to developing the tools that we desire is to use a formal language to reason about the problems that can occur in multi-lingual interoperation. Having understood the problems we can then develop tools that are easy for the programmer to use, that ensure that errors cannot arise from the mismatch between the language systems, and that provide a guarantee that no errors will be added. In keeping with the philosophy that our tools should have formal underpinnings and strong guarantees we also feel that our formal work should have strong guarantees; consequently all of our formal proofs are machine-checked using the Isabelle proof assistant[11].

In this paper, we demonstrate RIPLS and our approach to uncovering problems by presenting a version of RIPLS suitable for studying exception handling in multi-lingual systems. We begin in Section 2 by describing a motivating example: a dramatic failure of interoperability due to the incompatibility of the exception-handling mechanisms of Java and C++. We then define the relevant parts of this version of RIPLS, and discuss its soundness, in Section 3. Section 4 outlines how RIPLS can be used in both formal and practical ways to address the interoperability problem illustrated by the motivating example of Section 2. The paper concludes with a consideration of related work and a summary of the present status and

future directions of our efforts toward reducing the effort and eliminating the errors resulting from current approaches to developing multi-lingual software.

2 A Motivating Example

Assume that we have a system that consists of a user-friendly front-end coupled with a computationally-intensive back-end. In the following C++ listing the functions `e` and `g` are assumed to be in the front-end and `f` and `h` are assumed to be in the back-end.

```
#include "Afull.hh"
int e(int i) { return f(i); }
int g(int i) { return h(i); }
int f(int i) { try { return g(i); }
               catch (IntException x) { return x.getValue(); } }
int h(int i) { throw *(new IntException(3)); }
```

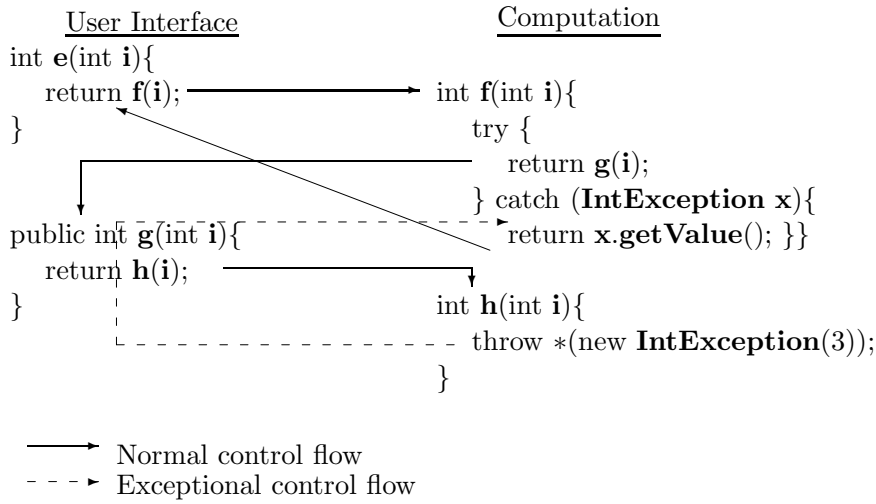

Fig. 1. Control flow through the example program.

If we call `e` we would expect it to return 3, and indeed it does, with the control flow through this code shown in Figure 1.

Note that procedure `f` calls procedure `g` inside a `try` ... `catch` block. This C++ exception-handling construct specifies how to handle an exception thrown by any code within the scope of the `try` ... portion of the construct. When `h` throws the exception it immediately terminates the computation of `h` and of `g` and control is transferred to the `catch` ... portion of the construct, with `x` receiving the exception value created by `h`. `f` handles the exception by returning 3, the integer value associated with the exception when it was created.

Note that almost the same code (leave out the * and the #`include`) might be a fragment of a Java program, since Java has an essentially identical exception

mechanism. If we call `e` in the all-Java version we would expect it to exhibit the same control flow and return 3, and indeed it does.

Now suppose that we split up the code so that the front-end is written in Java, and the back-end in C++. We will do this in the standard way, using the Java Native Interface (JNI) and the automated support provided by the J2SE Development Kit (JDK), namely the `javah` tool. Here is the Java front-end code, and the corresponding back-end code in C++:

```
public class A {
  int e(int i) { return f(i); }
  int g(int i) { return h(i); }
  native int f(int i);
  native int h(int i);
}
```

```
#include "Adylib.hh"
int f(int i) { try { return g(i); }
               catch (IntException x) {
                 return x.getValue(); }}
int h(int i) {
  throw *(new IntException(3)); }
```

The last two lines of the Java class are declarations that methods `f` and `h` have the signatures noted and that they are *native*, i.e., they are not written in Java, but are C or C++ procedures to be found in some shared library.

This code is very similar to the code of the original example, the only real difference being the addition of the `#include` directive (and the addition of a $*$). The `#include` directive tells the C++ compiler to load the indicated file, which the programmer must supply, containing function prototypes for `f` and `h`, and the definition of the `IntException` class.

Since the programmer is going to use the JNI to assemble these two pieces, the next step is to run `javah`, the JDK tool supporting JNI use. Given the `A` class file as input, `javah` will produce a file called `A.hh` containing prototype declarations for the native methods of Java class `A`. These prototypes specify two procedures, `Java_A_f` and `Java_A_h`, which are the names that the JNI expects for the implementations of the `f` and `h` native methods of the class `A`.

Finally, it is left to the programmer to take the JNI-produced prototypes and manually produce the required "glue code" that will connect all these pieces together. This "glue code" (see the snippet in Section 4) implements `Java_A_f` and `Java_A_h` as wrapper procedures that simply call the actual method implementations of `f` and `h`, respectively, and return the results of the calls. It also includes code for a third procedure `g` that invokes the Java method `g` from C++ using the JNI method invocation interface.

Even this extremely simplified example of Java-C++ interoperability has required quite a bit of effort to assemble.[3] Moreover, the JDK automates relatively little of it, leaving a fair amount of complicated and tedious coding to the programmer – a situation rife with opportunities for error. Nevertheless, we now have a complete collection of code. This can be assembled into a program that can be run using a Java Virtual Machine (JVM) and the JNI. One would hope that the

[3] Full details and all the code can be found in [12].

resultant system would give the same result as the C++-only or Java-only versions. Unfortunately the actual result is not well-defined. As a consequence, when run on one sample system, the entire process dies with a SIGABORT.[4] Although this is actually "correct" behavior, it is certainly not what the programmer would have expected or intended. The problem is a subtle one: while both C++ and Java have exception mechanisms, the intermediate glue code is written in C, which does not have an exception mechanism and C++ specifies some abnormal termination behaviors when an exception is returned to C. Current approaches to multi-lingual programming make it the responsibility of the programmer to be aware that this problem (and other similarly subtle problems) can arise and to take appropriate action to avoid such problems. The central goal of our research is to reduce the demands on the programmer by making such problems easier to discover and making the required remedial actions easier to carry out. The remainder of this paper presents one facet of our efforts in this direction.

3 Definition of RIPLS

The RIPLS language was inspired by the $\mathcal{R}$ language[14], which extends the simply-typed λ-calculus by adding mechanism[5] and effect annotations. With these annotations it was possible to reason about interoperation among multiple programming languages using the same programming language system (PLS). We distinguish between programming languages, such as Java, and PLSs, such as the JVM, which can potentially support multiple programming languages. The major contribution of RIPLS is the ability to reason about multiple PLSs, e.g., a JVM and a C++ runtime system.

RIPLS is defined in the usual way with an abstract syntax, typing rules, an abstract machine and a set of transition rules (an operational semantics). The language has been proven (with mechanically-checked proofs) to be type-safe and to make progress. In this section we will introduce the type system of the language and show the portions of the definition that are interesting or unusual. The full language is given elsewhere[12]. In the next section we will discuss how the language is used.

The RIPLS language uses effects and mechanism annotations to allow us to reason about distinct PLSs. In this paper the only effects we are interested in are exception effects; these are created when an expression raises (throws in Java and C++ terminology) an exception, and are retracted by exception handlers. This is an unusual use of effects; the authors are unaware of other systems that retract effects, and yet it seems the most natural approach for our purposes. If an exception is *always* handled inside a particular expression, does it really have any effect outside that expression? We use mechanism annotations to deal with the related issue of whether a particular expression is *able* to raise an exception. We distinguish

[4] On another sample system, we are informed, the JVM terminates with a note that it received an unexpected exception. This is preferable to just dying, but is still not the desired behavior.

[5] $\mathcal{R}$ called these resource annotations. We avoid the term resource because of its other connotations.

between mechanism annotations as a form of prerequisite description, and effects as a part of the results of an execution.

Mechanisms are used to describe the facilities available in a given PLS. Any PLS must have some continuation mechanism, i.e., a way of deciding what to do after it has finished evaluating the current expression. While such mechanisms must exist in each PLS, they may not be the same. For instance, C++ uses a continuation mechanism that allocates frames on the system stack, while a JVM will have an internal stack-based allocation mechanism. Similarly, both Java and C++ have exception-handling mechanisms, yet, as the example earlier demonstrated, they need not be the same mechanism. Figure 2 gives the abstract syntax of RIPLS. The set of abstract mechanisms is richer than it need be for the purposes of this

Abstract Mechanisms

$$\begin{aligned} \mathit{StackCont} &\ni a_{\mathsf{S}} \\ \mathit{ContMech} \ni a &::= a_{\mathsf{S}} \\ \mathit{ExnMech} &\ni cx \\ \mathit{CtrlMech} \ni c &::= a \mid cx \\ \mathit{PrimMech} \ni r &::= c \end{aligned}$$

Mechanism Descriptors

$$\begin{aligned} \mathit{PrimMechSet} \ni rs &::= 2^{\mathit{PrimMech}} \\ \mathit{MechanismDesc} \ni \rho &::= \langle a, rs \rangle \end{aligned}$$

Effects

$$\begin{aligned} \mathit{PrimEff} \ni f &::= \texttt{exception}(cx, \tau) \\ \mathit{Effects} \ni \varepsilon &::= 2^{\mathit{PrimEff}} \\ \mathit{EffectList} \ni \varepsilon &::= [2^{\mathit{PrimEff}}] \end{aligned}$$

Types

$$\begin{aligned} \mathit{BasicTyp} &\ni b \\ \mathit{Typ} \ni \tau &::= b \mid \tau \rightarrow^{\rho}_{\varepsilon} \tau' \end{aligned}$$

Values

$$\begin{aligned} \mathit{Const} &\ni d \\ \mathit{Val} \ni v &::= d \mid \lambda^{\rho} x{:}\tau \;.\; e \end{aligned}$$

Terms

$$\begin{aligned} \mathit{Var} &\ni x \\ \mathit{Exp} \ni e &::= x_1 \cdot x_2 \\ &\mid \lfloor v \rfloor \\ &\mid \lfloor x \rfloor \\ &\mid \texttt{let}\ x : \tau \leftarrow e_1\ \texttt{in}\ e_2 \\ &\mid \texttt{add}(r)e \\ &\mid \texttt{block}(r)e \\ &\mid \texttt{redirect}(a)e \\ &\mid \texttt{raise}[\tau]\ x \\ &\mid e_1\ \texttt{handle}\ x : \tau \;.\; e_2 \end{aligned}$$

Fig. 2. Abstract syntax of RIPLS.

paper, but leaves room for future extensions. We assume that there are a set of stack continuation mechanisms of interest, but do not specify them; similarly, we assume an unspecified set of exception-handling mechanisms. These *abstract mechanisms* serve as tags; indicators that at run-time an actual mechanism of the appropriate variety will be available. In a sense, they are like types for values. Continuation mechanisms are always stack-based in this paper. We say that a mechanism is a *control* mechanism if it is a continuation mechanism or an exception-handling mechanism; these are the only mechanisms that can have an effect on the control flow of the program. Finally we talk about *primitive* mechanisms. Currently the only primitive mechanisms are control mechanisms; in the future we will add memory mechanisms, and possibly others.

A *mechanism descriptor* is a pair of a continuation mechanism and a set of

primitive mechanisms. These are taken to be an indication of which mechanisms will be available at run-time, together with an indication of what continuation mechanism to use. Thus $\langle S, \{S, X\}\rangle$ might indicate that we were using the particular stack-based continuation mechanism S and that there was an exception-handling mechanism X available to use. The mechanism descriptors essentially serve as an abstraction of the PLS that a piece of code runs in.

Each continuation mechanism has a related exception-handling mechanism. The function $relatedExceptionHandler : ContMech \mapsto ExnMech$ (and abbreviated xh), specifies the exception-handling mechanism related to a given continuation mechanism. This function need not be 1 : 1, which will allow us, in the future, to work with multiple languages on a common PLS that share an exception-handling mechanism, but not a continuation mechanism.

In the current version of RIPLS there is only one primitive effect of interest, the `exception` effect, which is parameterized by the related abstract mechanism and by the type of the exception. If an exception is raised by a particular mechanism only that mechanism can handle it; thus we need to know which mechanism raised it. Parameterizing the exception effect with its type allows us to capture the notion, common to C++ and Java exceptions, that an exception handler only handles exceptions of certain types; thus we need the type of the exception to determine whether the exception is handled or propagates. Since an expression can produce multiple types of exceptions we refer to the set of primitive effects that it can produce as its *Effects* (see Figure 2).

Types are almost straightforward. We have the usual base types and function types, where base types, typically denoted by b, are members of the set $BasicTyp$, and function types are composed from other types. The difference between function types in the simply-typed λ-calculus and those in RIPLS is the presence of mechanism and effects annotations. The mechanism annotation states that any function of this type requires that exactly the specified set of mechanisms be available when the function is evaluated. The effects annotation gives a set of *possible* effects of this function. A simplistic view of the annotations is that the mechanism annotation specifies the language in which the function was written, and thus the PLS that it requires, and that, in this paper, the effects annotation specifies the exceptions that the function can throw, similar to a `throws` clause on a Java method or a `throw` clause on a C++ function or method.

There are values to match the types; constant values, usually denoted d, for basic types, and procedure values for function types. Procedures are unusual in that they have a mechanism annotation, which defines the required mechanism set, and an effects annotation which specifies which effects the procedure might produce.

The terms of the language are of three kinds. The first kind are standard: application, use of a value, extraction of the value associated with a variable, and a `let` expression with the usual semantics. The second kind are related to mechanism sets: `add` allows for the addition of a mechanism to the current mechanism set; `block` removes a mechanism from the current set of mechanisms, and `redirect` changes which continuation mechanism is current. Taken together, these three terms allow

for arbitrary manipulation of the mechanism set;[6] which provides the abstraction of the idea of transferring control from one PLS to another. Finally, we have exception-related terms: the `raise` expression raises the exception specified by the value of variable x, and the `handle` expression handles exceptions. For technical reasons the `raise` expression also specifies a type. The `handle` expression evaluates e_1, and if no exception is raised in that evaluation it returns the value of e_1. If an exception of type τ is raised (and not handled) within e_1, then the value of the exception is bound to x and e_2 is evaluated. If an exception of any other type is raised (and not handled) within e_1, then the exception propagates out of the term entirely.

Figure 3 gives the typing rules for effects, types, and values. We have some

Effects

$$\text{(Eff-empty)}\ \frac{a \in rs}{\vdash_\varepsilon \emptyset \leq \langle a, rs\rangle} \qquad \text{(Eff-exn)}\ \frac{cx = xh(a) \quad a \in rs \quad cx \in rs \quad \vdash_\varepsilon \varepsilon \leq \langle a, rs\rangle}{\vdash_\varepsilon \texttt{exception}(cx, \tau) \cup \varepsilon \leq \langle a, rs\rangle}$$

Types

$$\text{(Typ-basic)}\ \frac{b \in \mathit{BasicTyp}}{\vdash_\tau b} \qquad \text{(Typ-fun)}\ \frac{\vdash_\tau \tau_1 \quad \vdash_\tau \tau_2 \quad \vdash_\varepsilon \varepsilon \leq \rho}{\vdash_\tau \tau_1 \to^\rho_\varepsilon \tau_2}$$

Type Environment Formation

$$\text{(Env-typ-empty)}\ \vdash_\Gamma [] \qquad \text{(Env-typ-ext)}\ \frac{\vdash_\Gamma \Gamma \quad \vdash_\tau \tau}{\vdash_\Gamma \Gamma, x \mapsto \tau}$$

Values

$$\text{(Val-const)}\ \frac{\vdash_\Gamma \Gamma}{\Gamma \vdash_v d : \theta(d)} \qquad \text{(Val-abs)}\ \frac{\vdash_\tau \tau \quad \rho; \Gamma, x \mapsto \tau \vdash_e e : \tau'; \varepsilon \quad \vdash_\varepsilon \varepsilon \leq \rho}{\Gamma \vdash_v \lambda^\rho x{:}\tau \ . \ e : \tau \to^\rho_\varepsilon \tau'}$$

Fig. 3. Typing rules for effects, types, and values in RIPLS.

unusual judgments here. The judgment $\vdash_\varepsilon \varepsilon \leq \rho$, indicates that the set of effects given by ε is supported by the mechanisms given by ρ. The Eff-exn rule states that an exception effect is only supported by a mechanism descriptor if the exception-handling mechanism is the one related to the current continuation mechanism, and if both of those mechanisms are in the set of mechanisms in the descriptor. The judgment $\vdash_\tau \tau$ states that τ is a valid type. Basic types are always valid, while a function type is only valid if the effects that any procedure of that type can produce are supported by the specified mechanisms.

The value typing judgment, $\vdash_v$ is straightforward. Constants have their intrinsic types and procedures have an extended type that includes the mechanisms required to run the procedure (ρ) and the effects that the procedure may produce during execution (ε).

The typing rules for terms are given in Figure 4, and use an extended typing

[6] We could have used a simpler construct here, a $\texttt{use}(\rho)\ e$ expression, as in $\mathcal{R}$ but we chose to use separate terms in order to make evaluation simple and deterministic, without imposing some arbitrary ordering on the mechanisms.

TERMS

(Exp-app)
$$\frac{\vdash_\Gamma \Gamma \quad \Gamma(x_1) = \tau_2 \to^\rho_\varepsilon \tau_1 \quad \Gamma(x_2) = \tau_2 \quad \vdash_\varepsilon \varepsilon \leq \rho}{\rho; \Gamma \vdash_e x_1 \cdot x_2 : \tau_1; \varepsilon}$$

(Exp-val)
$$\frac{\Gamma \vdash_v v : \tau}{\rho; \Gamma \vdash_e \lfloor v \rfloor : \tau; \emptyset}$$

(Exp-var)
$$\frac{\vdash_\Gamma \Gamma \quad \Gamma(x) = \tau}{\rho; \Gamma \vdash_e \lfloor x \rfloor : \tau; \emptyset}$$

(Exp-let)
$$\frac{\rho; \Gamma \vdash_e e_1 : \tau_1; \varepsilon_1 \quad \rho; \Gamma, x \mapsto \tau_1 \vdash_e e_2 : \tau_2; \varepsilon_2 \quad \vdash_\varepsilon \varepsilon_1 \cup \varepsilon_2 \leq \rho}{\rho; \Gamma \vdash_e \texttt{let}\ x : \tau_1 \leftarrow e_1\ \texttt{in}\ e_2 : \tau_2; \varepsilon_1 \cup \varepsilon_2}$$

(Exp-add)
$$\frac{\langle a, rs \cup \{r\}\rangle; \Gamma \vdash_e e : \tau; \varepsilon \quad \vdash_\varepsilon \varepsilon \leq \langle a, rs \cup \{r\}\rangle \quad r \notin rs}{\langle a, rs\rangle; \Gamma \vdash_e \texttt{add}(r)e : \tau; \varepsilon}$$

(Exp-block)
$$\frac{\begin{array}{l}\langle a, rs - \{r\}\rangle; \Gamma \vdash_e e : \tau; \varepsilon \quad \vdash_\varepsilon \varepsilon \leq \langle a, rs - \{r\}\rangle \quad r \in rs \quad a \neq r \\ \forall cx.\forall \tau.(\texttt{exception}(cx, \tau) \in \varepsilon \longrightarrow (cx \in rs \wedge r \neq cx \wedge cx = xh(a)))\end{array}}{\langle a, rs\rangle; \Gamma \vdash_e \texttt{block}(r)e : \tau; \varepsilon}$$

(Exp-redirect)
$$\frac{\langle a', rs\rangle; \Gamma \vdash_e e : \tau; \varepsilon \quad \vdash_\varepsilon \varepsilon \leq \langle a', rs\rangle \quad a' \in rs \quad a \neq a' \quad \forall cx.\forall \tau.\texttt{exception}(cx, \tau) \notin \varepsilon}{\langle a, rs\rangle; \Gamma \vdash_e \texttt{redirect}(a')e : \tau; \varepsilon}$$

(Exp-raise)
$$\frac{\vdash_\Gamma \Gamma \quad \Gamma(x) = \tau' \quad \vdash_\tau \tau \quad \vdash_\tau \tau' \quad \vdash_\varepsilon \texttt{exception}(xh(a), \tau') \leq \langle a, rs\rangle}{\langle a, rs\rangle; \Gamma \vdash_e \texttt{raise}[\tau]\ x : \tau; \texttt{exception}(xh(a), \tau')}$$

(Exp-handle)
$$\frac{\begin{array}{c}\langle a, rs\rangle; \Gamma \vdash_e e_1 : \tau; \varepsilon_1 \quad \langle a, rs\rangle; \Gamma, x \mapsto \tau' \vdash_e e_2 : \tau; \varepsilon_2 \\ \vdash_\varepsilon \texttt{exception}(xh(a), \tau') \cup \varepsilon_1 \cup \varepsilon_2 \leq \langle a, rs\rangle\end{array}}{\langle a, rs\rangle; \Gamma \vdash_e e_1\ \texttt{handle}\ x : \tau'\ .\ e_2 : \tau; \varepsilon_1 - \texttt{exception}(xh(a), \tau') \cup \varepsilon_2}$$

(Env-top)
$$\Gamma, x \mapsto \tau(x) = \tau$$

(Env-other)
$$\frac{\Gamma(x') = \tau' \quad x' \neq x}{\Gamma, x \mapsto \tau(x') = \tau'}$$

Fig. 4. Typing rules for the terms of RIPLS.

judgment. Instead of the normal $\Gamma \vdash e : \tau$, we have $\rho, \Gamma \vdash_e e : \tau; \varepsilon$, i.e., we have added a mechanism descriptor to the preconditions and the effects to the result. Thus the judgment indicates that, given a specific mechanism descriptor and type environment, an expression has a particular type and can only produce a specific set of effects. For the most part effects simply accumulate, in keeping with their normal behavior; but we deviate from this in the definition of the Exp-handle rule. Our reasoning is that a handled exception is not really propagated beyond the handler. We then argue that the effects of the expression e_1 `handle` $x : \tau_1$. e_2 are the effects that could be produced by e_1 that are not handled by this exception-handler together with the effects of e_2.

4 Usage

In this section we revisit the example given in Section 2. We show that RIPLS is strong enough to determine the problem with that example. We will consider the Java class `A` and the C++ implementation class `Adylib`.

Translating the interesting parts of these classes into RIPLS yields:

```
let three : int ←⌊3⌋ in
  let h : (int →^{C++}_{exn(XC,Int)} int) ←(λ^{C++} i:int . raise[int] three) in
    let g : (int →^{Java}_{exn(XJ,Int)} int) ←(λ^{Java} i:int . h ·i) in
      let f : (int →^{C++}_{exn(XC,Int)} int) ←
          (λ^{C++} i:int .
```

```
        g ·i handle j : int . ⌊j⌋)
  let e : (int →^Java_{exn(XJ,Int)} int) ←(λ^Java i:int . f ·i) in e ·one
```

where C++ is an abbreviation for the mechanism descriptor $\langle \mathtt{SC}, \{\mathtt{SC}, \mathtt{XC}\}\rangle$ and Java is an abbreviation for the mechanism descriptor $\langle \mathtt{SJ}, \{\mathtt{SJ}, \mathtt{XJ}\}\rangle$.

Unfortunately this code does not type-check under the RIPLS type system for reasons unrelated to the exceptions issue. Consider procedure g, which requires the Java mechanisms to evaluate and which calls procedure h, which requires the C++ mechanisms. This fails to type-check, implying that we cannot make calls from Java into C++, or vice-versa, yet this interaction happens regularly when the programmer uses the JNI. The key to this is that the JNI implicitly does certain conversions; the declaration native int h(int i) automatically creates an intermediary function, expressed in RIPLS as:

```
let hJNI : (int →^Java_{exn(XJ,Int)} int) ←
  (λ^Java i:int . add(SC) add(XC) redirect(SC) block(SJ) block(XJ) Java_A_h ·i
      )
in ...
```

A similar procedure is created for the f method. (The names hJNI and fJNI are arbitrary and do not represent anything real.) These procedures actually do the transition between Java and C++ and ensure that the required mechanisms (and no others) are available. There are similar implicit procedures for methods e and g allowing them to be called from C++.[7] It turns out that the RIPLS code given above for the Java class A and the C++ code is somewhat inaccurate. Basically we need to replace the call to h with a call to hJNI, and similarly with g, and f. Making these substitutions, the resulting code would be well-typed under the RIPLS type system, but for the problems with exceptions.

Note the form of the names of the function called. They begin with the string Java_ followed by the name of the class, then by another _, then by the name of the method. If the method were overloaded there would be another part to the name indicating the type of the arguments. The result of this style of name, which is chosen by the JNI, is that programmers end up writing glue code, an extremely simplified snippet of which follows:

```
JNIEXPORT jint JNICALL
    Java_A_h(JNIEnv *env, jobject o, jint arg) {
  int rc = h(arg);
  return (jint)rc;
}
```

Now examine the RIPLS equivalent code for Java_A_h.

[7] These procedures are totally invisible, and are implicitly called when using a Call<Type>Method (or other) library call to invoke a method.

$\texttt{let Java\_A\_h} : (\texttt{int} \rightarrow^{\texttt{C++}}_{\{\texttt{exn(XC,Int)}\}} \texttt{int}) \leftarrow (\lambda^{\texttt{C++}}\ \texttt{arg:int}\ .\ \texttt{h}\ \cdot \texttt{i})$

Note that if `g` calls `hJNI`, and `hJNI` calls `Java_A_h`, which in turn calls `h` the RIPLS code will not type-check. Here the problem is that the `add(XC)...` expression in `hJNI` fails, because the expression in the body of the `add` expression has a C++ exception that is not supported by a set of mechanisms that does not include `XC`, and the typing rule for the `add` expression requires that the effects of the expression all be supported by the mechanisms available to the `add` expression, not including the mechanism that the expression itself adds.

As we saw in Section 2, the Java and C++ code shown does compile and will run, but will misbehave. When we translate that code into RIPLS and attempt to type-check it, the type-checking fails, indicating that there is, in fact, a problem with the combined Java and C++ code. This problem is undetected by the Java and C++ compilers, which only analyze their respective pieces of the multi-lingual system. We conclude that our methodology of translating the components into a common formal language and analyzing the resultant program will expose problems that are not exposed by conventional tools.

A type-correct version of the example would require that each snippet of glue code from Java to C++ be modified to include a `try...catch` block to catch any exceptions that the C++ code might throw and do something reasonable with them. It would also require that glue code from C++ to Java include a `throw(...)` clause to indicate what C++ exceptions could be thrown and an appropriate set of calls to the JNI Application Programming Interface (API) to catch and clear the caught exception. This code would also have to do something reasonable with the exception. Space does not allow us to present this in any more detail.

Of course, our formal foundations will be most valuable to practicing programmers if those foundations can serve as a basis for practical methods or automated tools to assist in the development of correct systems comprising interoperating components written in different languages. We have begun working on such methods and tools and have demonstrated some initial prototypes, improving on our earlier Exu and JExu tools, that are informed by the formal foundations described here and that automate the generation of correctly interoperating Java and C++ classes even when those classes use exceptions [12]. These preliminary results reinforce our belief in the importance of solid formal foundations to our long-term goal of maximally effortless and error-free multi-lingual programming.

5 Related and Future Work

The objective of the research reported in this paper was to establish foundations for multi-lingual exception handling in object-oriented languages. Our approach involves applying rigorous and automated techniques for the study of formal languages to develop suitable formal foundations, then using those foundations as a basis for practical techniques and tools to support multi-lingual programming.

While there has been quite a lot of previous work on pragmatic approaches to

multi-lingual interoperability there has been much less work on formal foundations. Our own previous work on pragmatic approaches includes the PolySPIN framework [5,6], and implementations of it called PolySPINner [3,2]. This work suffered from a lack of formal underpinnings and a concomitant failure to adequately address fundamental typing issues. These problems were compounded by the fact that the work was based on C++ and CLOS. We therefore produced Exu [4] in an attempt to demonstrate the feasibility of applying the PolySPIN approach to C++ and Java, but this work also lacked any formal foundations. Subsequent efforts added exception handling support to Exu [1] and to JExu [7], a companion tool to Exu. Others have recently taken conceptually similar pragmatic approaches to supporting integration of Java with .Net [9]. Some commercial tools for interoperating between Java and C++ include "JunC++ion", [8], which has similarities to Exu and JExu, JNBridge, Borland's Janeva, and Intrinsyc's Ja.Net. Like PolySPIN, Exu and JExu, these all lack formal underpinnings and are therefore subject to similar shortcomings.

The only previous research on formal foundations for multi-lingual programming of which we are aware is that of Trifonov and Shao [14], Sullivan et al.[13], and Matthews and Findler [10]. We have emulated the approach used in the first of these, namely describing a common formal language into which surface languages are translated in order that the interoperability issues can be explored. Whereas that work grew out of the FLINT project, and consequently restricted itself to a single run-time platform, our approach expressly addresses issues surrounding multiple PLSs, not just multiple programming languages. Similarly, Sullivan's work focused on formalizing a single run-time platform, specifically Microsoft's Component Object Model (COM), in order to demonstrate inconsistencies in its specification[13]. The formalization done by Matthews and Findler describes interesting approaches to interoperation between ML and Scheme, but does not address either object-orientation or issues arising from a multiplicity of run-time platforms.

Given the promising results reported here, where use of our RIPLS formal foundation enables rigorous analysis of an important aspect of multi-lingual object-oriented programming, supports identification of a problem that occurs in real uses of exception handling and informs the design of automated tools for multi-lingual programming, we are pursuing future development of both the formal foundations and pragmatic tools. We plan to produce additional extensions to RIPLS that will focus on other aspects of multi-lingual object-oriented programming, with the goal of providing a complete treatment of interoperation in the C++ and Java environment. At a minimum, the set of such extensions should include method dispatching (single-dispatching) and parametric polymorphism. In addition it ought to be possible to extend RIPLS to support other things such as differences in memory models: garbage collection versus explicit memory management.

We also plan to continue working on practical methods and tools for interoperability based on our formal foundations. In particular, we would like to write translators (compilers) for C++ and Java to RIPLS. The intent of this would not be to actually use these translators for any execution, but to have full RIPLS code

that we could check for type-correctness, and thus be in a position to certify that a polylingual program had no added type errors. This would serve to automate analyses like the one that we carried out manually in Section 4.

Our long term goal is to improve the effectiveness of programmers through tools based on solid formal foundations. The research presented in this paper represents progress towards providing seamless multi-lingual interoperability among object-oriented languages, and thus is an important step toward our long term goal.

References

[1] Altidor, J., "Interoperability Between Java and C++ using Exu and JExu," Bachelor's honors project report, University of Massachusetts Amherst (2006).

[2] Barrett, D. J., "Polylingual Systems: An Approach to Seamless Interoperability," Ph.D. dissertation, University of Massachusetts, Amherst, MA (1998).

[3] Barrett, D. J., A. Kaplan and J. C. Wileden, *Automated support for seamless interoperability in polylingual software systems*, in: D. Garlan, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering* (1996), pp. 147–155, to appear.

[4] Bubba, J., A. Kaplan and J. C. Wileden, *The Exu approach to safe, transparent and lightweight interoperability*, in: *Proceedings of the 25th IEEE Computer Software and Applications Conference (COMPSAC)*, Chicago, 2001, pp. 393–400.

[5] Kaplan, A. and J. C. Wileden, *Toward painless polylingual persistence*, in: R. Connor and S. Nettles, editors, *Persistent Object Systems, Principles and Practice, Proc. Seventh International Workshop on Persistent Object Systems* (1996), pp. 11–22.

[6] Kaplan, A. and J. C. Wileden, *Foundations for transparent data exchange and integration*, in: *Contributed Papers Collection for the Conference on Scientific and Technical Data Exchange and Integration*, 1997.

[7] Kielbasinski, A., "Exception Handling Interoperability Between C++ and Java," Bachelor's honors project report, University of Massachusetts Amherst (2004).

[8] Krapf, A. R., *JunC++ion r2.1 product description* (2003).
URL http://www.codemesh.com/en/JunctionCurrentRelease.html

[9] Krapf, A. R., *Integrating Java into the .NET environment*, .NET Developer's Journal **2** (2004).

[10] Matthews, J. and R. B. Findler, *Operational semantics for multi-language programs*, in: *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2007), pp. 3–10, general Chair-Martin Hofmann and Program Chair-Matthias Felleisen.

[11] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL — A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer, 2002.

[12] Ridgway, J. V. E., "Foundations for Polylingual Systems," Ph.D. thesis, University of Massachusetts Amherst (2004).

[13] Sullivan, K. J., M. Murchukov and D. Socha, *Analysis of a conflict between interface negotiation and aggregation in Microsoft's component object model*, IEEE Transactions on Software Engineering **25** (1999), pp. 584–599.

[14] Trifonov, V. and Z. Shao, *Safe and principled language interoperation*, in: S. D. Swierstra, editor, *Programming Languages and Systems: 8th European Symposium on Programming, ESOP'99, Proceedings*, Lecture Notes in Computer Science **1576** (1999), pp. 128–146, held as part of the Joint European Conference on Theory and Practice of Software, ETAPS'99.