

# Translation and Optimization for a Core Calculus with Exceptions

Cristina David    Cristian Gherghina    Wei-Ngan Chin

Department of Computer Science, National University of Singapore  
{cristina,cristian,chinwn}@comp.nus.edu.sg

## Abstract

A requirement of any source language is to be rich in features and concise to use by the programmers. As a drawback, it is often too complex to analyse, causing research studies to omit some of the fancy features. For instance, exception handling is an important aspect of programming languages that is instrumental for building robust software with good error handling capability. However, exceptions are often omitted during the initial formulation on program analysis and optimization. Moreover, when considering the traditional approach of converting programs from high level languages to machine code, the target code is meant for the machine, being too cryptic (or low level) for program analysis. Our goal is to design an intermediate, minimal but expressive, core calculus which can be easily analysed and manipulated, and to show that this calculus can handle major language features by translating a significant imperative source language into it. The translation to the core calculus enables us to easily analyse and optimize the code, while not sacrificing the flexibility and rich characteristic of the source language.

**Categories and Subject Descriptors** D.3.3 [Programming Languages]: Language Construct and Features; D.3.4 [Programming Languages]: Processors; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Construct

**General Terms** Design, Languages, Theory

## 1. Introduction

Modern programming languages have many useful features to help the construction of software. Being meant for the development of applications, their main aim is to offer a high degree of flexibility and ease of use to the programmer. Consequently, they become complex and hard to analyse. For instance, one important feature is exception handling. This feature is used to handle unusual conditions that can lead to errors, unless remedial actions are suitably taken. However, exceptions may induce non-local control flows that could make programs harder to analyse statically. This worry is one reason why a number of past proposals (including our own work [21]) on calculi to facilitate formal reasoning [22, 8] have mostly ignored exceptions, in the name of simplicity. Some recent proposals [19, 17] have begun to consider a

core language with exceptions by adding both a `throw e` construct and a simplified `try e1 catch (c v) e2` construct from the Java language. However, this simplified feature was not able to *succinctly* handle more advanced features, such as `try-finally` nor `try-with-multiple-catches`. Another proposal [9] directly adds these advanced features in their core language, but this is done at the price of a more complex formalization.

Our proposal is to perform the analysis on an intermediate simplified core calculus, avoiding the complexity of the source language, but without restricting the flexibility expected by the programmers. The two crucial requirements for our calculus are to be easy to analyse (\*), and to be expressive enough as to allow the translation of more complex language constructs into it (\*\*). For achieving the first goal (\*), we endow our core calculus with a *unified view* of the control flow, which spans two dimensions:

- Unifies both normal and abnormal control flows. Unlike past works which simply add together the features of normal and abnormal control flows, our change is more fundamental since we provide a pair of unified language constructs (elaborated later) that would work for all kinds of control flows.
- Unifies the static control flow, where the syntax of the program directly determines which parts of the program may be executed next, and the dynamic control flow, where run-time values and inputs of the program are required to decide what to execute next. This is achieved by translating the `break`, `continue`, `return` constructs (specific to the static control flow), and the `try-catch` and `raise` constructs (specific to the dynamic control flow), into a unified control flow mechanism under our calculus.

An unexpected benefit is that our core calculus with exceptions is *as small as* the corresponding core calculus without exceptions. Designing analyses and optimizations for the core calculus is therefore much simpler than it would be for the source language! With regard to the current work's second goal (\*\*), we prove the expressivity of the core calculus by providing a set of rewrite rules for translating a medium-sized imperative source language into it.

We refer to our design as a *calculus* rather than a *language* since our intention is to support a broader range of formal reasoning activities, including analysis, language design, compilation, optimization and verification. The resulting core calculus will essentially contain a core language and a set of rules (including translation) that facilitate formal reasoning. Our goal is for a core calculus that is *syntactically minimal and expressively maximal*. We shall describe an application of our calculus, namely optimization. In order to prove the soundness of the optimization rules, we shall formalise the calculus by providing a big-step operational semantics.

### 1.1 Our Contributions

The main contributions of our paper are highlighted below:

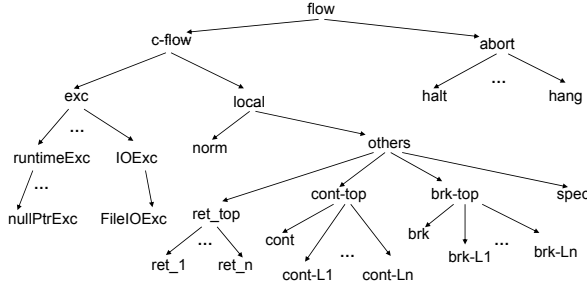
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.  
Copyright © 2009 ACM 978-1-60558-327-3/09/01...\$5.00

- **Unified Control Flows** : We propose a core calculus with a novel view of the control flows unifying both normal and exceptional executions. This new design is supported by a pair of unified constructs that are considerably more general than previous approaches. Due to this unification of the control flows, the core calculus is easier to analyse and optimize.
- **Translation Rules** : Though simple, our core calculus is expressive enough as to allow the translation of a Java-like imperative language into it. The translation is based on rewrite rules and illustrates how advanced language features, such as `try-finally` and multi-return functions, can be easily captured by our core calculus. Moreover, we prove two important properties of the translation, namely completeness and termination.
- **Optimization** : We provide a set of optimization rules for our calculus, designed to reduce the implementation overhead. These rules are specified at a high-level which facilitate both human understanding and the construction of correctness proofs. While the set of optimization rules is by no means exhaustive, these rules can help support better practical prospects for our core calculus. For all the rules we supply correctness proofs, which are also meant to illustrate the ease of designing optimizations and proving them correct in our core calculus.

## 2. Approach and Motivation

Our calculus is based on a unified view of control flows in which both normal and abnormal control flows are being handled in a uniform way. We shall organise these control flows into a tree hierarchy, as illustrated in Fig 1.



**Figure 1.** A Subtype Hierarchy on Control Flows

Each arrow  $c_2 \rightarrow c_1$  denotes a subtyping relation  $c_1 <: c_2$ . In this tree hierarchy, `exc` captures dynamic control flows due to exceptions, while `local` captures static control flows, such as break/continue for loops and return for methods. The control flow `norm` for normal execution is a special instance of this static control flow that will be transferred to the default next instruction for execution. A key feature of static control flows is that they can be efficiently implemented as local control transfers through either direct or indirect jumps, while dynamic control flows from exceptions would involve non-local transfer of control via catch handlers present in the function calling hierarchy at runtime. All control flows that can be ‘caught’ by our calculus are placed under the `c-flow` category, while the `abort` category denotes control flows that cannot be caught. This includes program termination by `halt`, and non-termination by `hang`. The latter could, in principle, be used by our calculus to reason about non-terminating behaviors but this aspect is not addressed in this paper.

Next, we will introduce the key constructs of our calculus. These constructs are meant to allow us to take full advantage of the complex control flow hierarchy introduced above.

In previous core languages with exceptions for Java, such as [19] and [17], a variable  $v$  would return a value with normal flow,

while `throw v` would invoke an exceptional flow based on the exception object in  $v$ . In our approach, we unify both these constructs with  $ft\#v$ , whereby normal flow is realised by `norm\#v` while exceptions may be thrown using `ty(v)\#v`. The function `ty(v)` returns the runtime type of an exception object pointed by  $v$ . In case  $v = \text{null}$ , it returns a special `nullPtrExc` flow type. This unified construct is a generalization of the exception mechanism used in Java since we allow each flow type to be unrelated to the type of the value being thrown. For example, we may use `exc\#13` to raise an exception with integer value 13. This is not *directly* expressible in Java, though it could be mimicked by a user-defined exception that embeds an integer value.

Another major construct of our calculus is a try-catch mechanism of the form `try e1 catch (((c@fv)\#v)) e2` which specifies a control flow  $c$  and two bound variables to capture a control flow type  $fv$  and its thrown value  $v$ , provided that  $fv <: c$ . This try-catch construct is more general than that used in Java since it can capture not only exceptional flow, but also normal flow and other abnormal control flows due to `break`, `continue` and `return` that can be translated to the corresponding control flows (see Sec 3 later). As a pleasant surprise, the usual sequential composition  $e_1; e_2$  is now a syntactic sugar for `try e1 catch (((norm@_)\#_)) e2` whereby each `_` denotes a distinct anonymous bound variable.

Our attempt to pair a control flow and its returned (or thrown) value using  $ft\#v$  can be *viewed* as injecting a sum type (or polymorphic variant) into the result of our computation. This *logical view* is used as a means to unify the various kinds of control flow under a single uniform mechanism in our calculus, to facilitate program analysis and optimization. In reality, our proposal never creates any value of sum type in its computed results, since this will merely add an extra layer of interpretive overheads.

As a final point, we will explain a couple of our design decisions. First, our use of a tree hierarchy, rather than a lattice, for our control flow is important for finite abstraction. A useful property of the tree hierarchy is that every two nodes of the tree, say  $c_1, c_2$ , are either *mutually-exclusive*, as denoted by  $\forall c. (c <: c_1 \implies \neg(c <: c_2))$ , or they *overlap*, as denoted by  $c_1 <: c_2 \vee c_2 <: c_1$ . This property is helpful for formal reasoning since we can statically determine disjointness of two flow types with the help of only its subtyping relation. This decision allows us to build finite set abstractions required to model multiple flows. While exceptions in Java were implicitly organised as a hierarchical tree, previous use of effects-based type system do not require this finitary abstraction property.

The second issue concerns efficiency. While the target of the above translations may appear inefficient due to an apparent need to unwind through a nested series of handlers, we emphasize that our primary goal is to make program codes easier to analyse. For actual execution, we could use compilation techniques to ensure that every static control flow is efficiently implemented by either a direct or indirect jump into its corresponding handler code. Moreover, we could also use a similar optimization to efficiently implement some of the dynamic control flows. Under suitable conditions, we can use an optimization rule, called *throw-catch linking*, that could directly link a throw operation for an exception with its intended handler through a parameterized jump (see Sec 5 later).

## 3. Towards a Core Calculus

We shall employ four key principles in the design of our core calculus with unified control flows. These principles, described below, are selected to allow our calculus to support a broad range of applications, from static analysis to efficient compilation.

- **Unified Constructs** : To minimise on language features, our calculus should unify together constructs that have similar functionality, where possible.

- *Syntactically Minimal* : To keep our calculus small, we shall aim for fewer and simpler constructs, where possible. This can make our calculus easier to formalise and analyse.
- *Expressively Maximal* : We strive to provide language constructs that are as general as possible, to allow them to be used in more scenarios. Our acid test is whether the calculus can succinctly encode more advanced language features, or not.
- *Computationally Positive* : The calculus should not hinder efficient compilation. Firstly, it supports a set of optimization rules. Secondly, intermediate steps used to make the calculus easier to analyse can be directly removed later by efficient compilation.

We shall now present our core calculus and also show how a given source language could be translated into it. The core calculus that we have designed is small but quite general. It is currently based on first-order imperative programming languages with monomorphic typing. We have originally intended for our core calculus to support only exception handling features. However, we were pleasantly surprised that other standard features, such as break/continue, and also more fancy features, such as multi-return functions, can be similarly supported by our core calculus without any change.

Though we shall support a class hierarchy for our data types, we shall avoid other object-oriented features such as instance methods and method overriding. To this end, we consider only static methods and support class types with single inheritance.

$P$	$::= \vec{D} ; \vec{M}$	program
$D$	$::= \text{class } c_1 \text{ extends } c_2$ $\{t f\}$	data declaration
$t$	$::= c \mid p$	user or prim. type
$M$	$::= t m [\text{with } n] (\vec{t} v) \{e\}$	method declaration
$w$	$::= v \mid v.f$	variable or field
$e$	$::= v$	variable
	$  k$	primitive constant
	$  \text{new } c$	new object
	$  v.f$	field access
	$  (c) e$	casting
	$  \text{throw } e$	throw exception
	$  \text{break } [L] \mid \text{ret-}i e$	break and return
	$  \text{continue } [L]$	loop continue
	$  w := e$	assignment
	$  e_1 ; e_2$	sequence
	$  (m \vec{v}) \text{ with } \lambda v. \vec{e}$	multi-return call
	$  \{t v ; e\}$	local var block
	$  \text{if } e \text{ then } e_1 \text{ else } e_2$	conditional
	$  \text{let } v = e_1 \text{ in } e_2$	local binding
	$  L : e$	labelled expression
	$  e_1 \text{ finally } e_2$	finally
	$  \text{try } e \text{ catch } (c_1 v_1) e_1$ $\quad [\text{catch}(c_i v_i) e_i]_{i=2}^n$	multiple catch handlers
	$  \text{do } e_1 \text{ while } e_2$	loop

Figure 2. Source Language : SrcLang

We consider the source language presented in Fig 2, which we call SrcLang. Take note that  $\vec{e}$  denotes  $e_1, \dots, e_n$ . At this point, we would like to emphasize that an important role of our calculus is to support language design, by allowing more advanced language constructs to be unambiguously expressed in terms of simpler constructs of our calculus. Therefore, we augment our source language with constructs that are challenging from the point of how control flow is transferred. Among these, we identify the try construct with multiple catch handlers, the finally construct,

the multi-return function call ([27]). The latter is represented in our language as  $(m \vec{v})$  with  $\lambda v. \vec{e}$ . Evaluating such a form involves evaluating the inner application,  $(m \vec{v})$ , in a context with  $n$  return points. The first return point is for the context of the call itself. The other  $n - 1$  return points are captured by return points of the form  $\lambda v_2. e_2, \dots, \lambda v_n. e_n$ . If the application eventually returns a value  $val$  to a return point of the form  $\lambda v_k. e_k$ , then  $v_k$  is bound to the value  $val$ , and expression  $e_k$  is evaluated in the caller's context. The return construct,  $\text{ret-}i e$ , specifies that the result of evaluating expression  $e$  is to be returned to the  $i$ -th return point of the caller.

Take note that, before the actual translation to the core language, we invoke a preprocessing phase for checking the validity of the local control flows. More specifically, we check that the `continue [L]` construct appears inside loops, `break [L]` and `continue [L]` mention only accessible labels, and, in the case of the multi-return function declaration with  $n$  return points, each of its return instruction must be of the form  $(\text{ret-}i e)$  such that  $i \leq n$ . Furthermore, each of the method's calls must also be invoked with  $n - 1$  return points.

Our core calculus with unified control flows, called Core-U, is given in Fig 3. We shall show its versatility by translating SrcLang source programs into it. We have added a special type  $(c, t)$  to capture the embedding of a control flow  $c$  with its value of type  $t$ . For simplicity, we have adopted a monomorphically typed language, though in principle our proposal should extend to polymorphically typed languages too.

$e$	$::= ft \# x$	
	$  v.f$	field access
	$  w := v$	assignment
	$  (m \vec{v})$	fct call
	$  \{t v ; e\}$	local var block
	$  \text{if } v \text{ then } e_1 \text{ else } e_2$	conditional
	$  \text{try } e_1 \text{ catch } ((c @ fv) \# v) e_2$	catch handler
	$  \text{do } e \text{ while } v$	do loop
$ft$	$::= c \mid \text{ty}(v) \mid fv$	flow
$t$	$::= c \mid p \mid (c, t)$	+ embedded type
$x$	$::= v \mid k \mid \text{new } c \mid (fv, v)$	value term
$w$	$::= v \mid v.f$	variable or field

Figure 3. Core Language : Core-U

Following the Stratego approach [29], we base the translation from SrcLang to Core-U on rewrite rules. The translation process is split into four independent steps such that, in each of the steps, there is no interference between the rules to be applied. In other words, each rule will neither trigger nor block the application of any other rule from the same rewrite set. Additionally, the rules from each phase will not trigger the application of any rule from a previous phase. As a consequence of this non-interference property, the order in which the rules are applied in each of the steps is irrelevant. The non-interference property is also important for guaranteeing the termination of the rewriting process.

### 3.1 Phase I: Preprocessing

In the first step, we transform those expressions that appear in a more complex form than the one allowed by our core calculus. For example, we may encounter `if e then e1 else e2`, while our core calculus can only accept a simpler `if v then e1 else e2`. This mismatch can be handled in a standard way with the help of the local `let` binding construct that always generates a fresh new variable.

$$\begin{aligned}
 (c) e &\Rightarrow_T \text{let } v = e \text{ in } (c) v \\
 &\quad \text{where } \text{notVar}(e) \\
 \text{if } e \text{ then } e_1 \text{ else } e_2 &\Rightarrow_T \text{let } v = e \text{ in } \text{if } v \text{ then } e_1 \text{ else } e_2 \\
 &\quad \text{where } \text{notVar}(e)
 \end{aligned}$$



its callers' sites. This is achieved by changing `ret-1` to the `norm` control flow, for each method declaration, as follows:

$$m[\text{with } n](\overline{t}v)\{e\} \Rightarrow_T m[\text{with } n](\overline{t}v)\{\text{try } e \text{ catch } (\text{ret-1}\#v) \text{ norm}\#v\}$$

- **translate the local binding construct:**

$$\text{let } v=e_1 \text{ in } e_2 \Rightarrow_T \text{try } e_1 \text{ catch } (\text{norm}\#v) e_2$$

- **handle abnormal controls due to breaks and continues for loops:**

$$\begin{aligned} L : \text{do } e_1 \text{ while } e_2 \\ \Rightarrow_T \text{try } \{ \text{bool } v; v:=\text{false}; \\ \quad \text{do } (\text{try } e_1 \text{ catch } (\text{cont}) v:=\text{true} \\ \quad \quad \text{catch } (\text{cont}-L) v:=\text{true}; \\ \quad \quad \text{if } \neg v \text{ then } v:=e_2 \text{ else norm}\#() \\ \quad \text{while } v \} \\ \text{catch } (\text{brk}) \text{norm}\#() \\ \text{catch } (\text{brk}-L) \text{norm}\#() \end{aligned}$$

The try-catch mechanism provides a uniform way to deal with non-local control flows, such as `break` and `continue` via exception handling. If these mechanisms had not been used, our calculus would resort to lower-level constructs, such as `goto` statements, for formal reasoning. Unstructured `goto` construct may be useful for efficient implementation, but has a lower abstraction level for formal reasoning.

- **handle abnormal controls due to breaks for labelled expressions:**

$$L : e \Rightarrow_T \text{try } e \text{ catch } (\text{brk}-L) \text{norm}\#()$$

### 3.3 Phase III: Wrapping up the translation

For this phase, the unified `try-catch` construct is used to replace the sequence operator  $e_1; e_2$ , and to simplify the RHS of assignment construct (to a variable), as shown below.

$$\begin{aligned} e_1; e_2 &\Rightarrow_T \text{try } e_1 \text{ catch } (\text{norm}) e_2 \\ w := e &\Rightarrow_T \text{try } e \text{ catch } (\text{norm}\#v) w := v \end{aligned}$$

The placement of these two rules in a separate phase is meant to maintain the non-interference property of the translation process. This is due to the fact that some of the translations rules from the previous phase trigger the application of the assignment and sequence translation rules.

### 3.4 Phase IV: Handling implicitly raised exceptions

To complete our translation, in the last rewriting step, we provide a set of rules to deal with constructs that may implicitly raise some exceptions, such as null dereferencing or memory overflow. These rules can help make all raised exceptions explicit.

$$\begin{aligned} v.f &\Rightarrow_T \text{if } v=\text{null} \text{ then nullPtrExc}\#v \\ &\quad \text{else } v.f \\ v.f := v_2 &\Rightarrow_T \text{if } v=\text{null} \text{ then nullPtrExc}\#v \\ &\quad \text{else } v.f := v_2 \\ ft\#\text{new } c &\Rightarrow_T \text{if enoughMem}(c) \text{ then } ft\#\text{new } c \\ &\quad \text{else OutofMemoryExc}\#\text{null} \\ \text{ty}(v)\#v &\Rightarrow_T \text{if } v=\text{null} \text{ then nullPtrExc}\#v \\ &\quad \text{else ty}(v)\#v \end{aligned}$$

Note that the translated codes can be more efficiently implemented than the original code, since they can be specialised as exception-free code. Furthermore, there is potential for a systematic optimization to eliminate some of the checks that have been explicitly inserted. For example, we could use a nullness analysis to help statically determine those nullness tests that are known to be redundant. Similarly, it is possible to aggregate a series of tests on memory sufficiency to be replaced by a bigger test at the beginning. These steps can lead to simpler (and more efficient) core programs.

It is possible to provide a semantics for `Core-U` and another semantics for `SrcLang`, before proving that the translation rules

are *fully abstract* [26] by preserving observational equivalence between the source and target programs under their respective semantics. This would have proven the correctness of the translation rules between the two languages. However, since the core language is at a relatively high level and can be viewed as a subset of the source language, we could also define the semantics of the `SrcLang` directly in terms of the `Core-U`. With this simplified approach, the translation rules would be correct by construction. This approach is not at all new. For example, the Haskell language [14] is largely defined in this way, by translating more complex language features into a simpler core.

Nevertheless, we will prove two important properties of our translation rules. Firstly, we shall show that given any arbitrary `SrcLang` expressions, we can always translate it into a `Core-U` counterpart. Secondly, we shall prove that the application of the translation rules always terminates.

**LEMMA 3.1 (Completeness of Translation).** *Consider any term  $e \in \text{SrcLang}$ . Repeated applications by our transformation rules via  $e \Rightarrow_T^* e'$  would eventually result in  $e' \in \text{Core-U}$ , when the transformation terminates.*

**Proof** [(sketch)] There is at least one transformation rule for each syntactic construct of `SrcLang`, except for local block  $\{t\ v; e\}$  which remains unchanged in `Core-U`. Furthermore, each target form in the RHS of the translation rules belongs to `Core-U` or can be transformed as so in a subsequent step. Hence, by induction on the syntactic structure of `SrcLang`, we can prove that the final expression belongs to `Core-U`, should the transformation terminate.  $\square$

**LEMMA 3.2 (Termination of Translation).** *The transformation  $e \Rightarrow_T^* e'$  always terminates.*

**Proof** [(sketch)] Let us first note that all the rewrite rules used in the translation process follow a common design, namely the RHS does not contain the pattern from the LHS that triggers the application of the rule. This property ensures that each application of a rewrite rule will eliminate one occurrence of a specific trigger. Moreover, none of the rules trigger the application of any other rule from the same set, nor the application of a rule from any of the previous steps. Consequently, the rewriting process, for each of the four steps, must terminate.  $\square$

## 4. Semantics

In the current section we provide a description of the operational semantics for our calculus. We introduce a big-step (evaluation) semantics which will be used when proving the soundness of the optimization rules (see Sec 5.1).

The machine configuration is represented by  $\langle e, h, s \rangle$  where  $e$  denotes the current program code,  $h$  denotes the current heap for mapping addresses to objects, and  $s$  denotes the current runtime stack for mapping variables to values. We assume sets  $Loc$  of locations (positive integer values),  $Val$  of values (either a constant, a location or a pair of control flow type and value),  $Var$  of variables (program variables and other meta variables), and  $ObjVal$  of object values stored in the heap, with  $c[f_1 \mapsto \nu_1, \dots, f_n \mapsto \nu_n]$  denoting an object value of class  $c$  where  $\nu_1, \dots, \nu_n$  are current values (from the domain  $Val$  such that  $Loc \subset Val$ ) of the corresponding fields  $f_1, \dots, f_n$ .

$$\begin{aligned} h &\in Heaps =_{df} Loc \rightarrow_{fm} ObjVal \\ s &\in Stacks =_{df} Var \rightarrow Val \end{aligned}$$

The semantics assumes unlimited stack and heap spaces.

For the dynamic semantics to follow through, we have introduced an additional final value. Its syntax is extended from the original expression syntax as shown next, where  $l \in Loc$ .

$$e ::= \dots \mid ft\#l \quad \text{flow and location}$$

Our evaluation judgment has the form  $\langle e, h, s \rangle \hookrightarrow_b \langle e_1, h_1, s_1 \rangle$ , specifying the entire transition from an initial configuration  $\langle e, h, s \rangle$

to a final configuration  $\langle e_1, h_1, s_1 \rangle$ . The interpretation is that  $e$  evaluates to the final value  $e_1$ . Each final value is of the form  $c\#a$  where  $c$  is a control flow type and  $a$  is either a value (constant or location) or a pair  $(c_1, a_1)$  to embed a control flow  $c_1$  with another value  $a_1$ . The type of each final value can be obtained by a semantic function  $type(a)$ . Syntactically  $a ::= k \mid l \mid (c, a)$ . While a pair of values may also be modelled by a data structure with two fields, we shall provide it explicitly to model this class of embedded operations more precisely.

The full set of transitions is given in Fig. 4. Take note that, following the translation to  $Core-U$ ,  $v.f$  is exception-free, meaning that if  $v$  was null, a `nullPtrExc` exception would have been previously raised. Consequently, our rules for  $v.f$  and  $v_1.f := v_2$  do not test for nullness. In the rules,  $s(v)$  retrieves the value of variable  $v$  present on the stack using `lookup(s, v)`. We also provide an overloaded function  $s(ft)$  for  $ft ::= c \mid ty(v) \mid fv$  which is defined as  $s(c) = c$  and  $s(ty(v)) = type(lookup(s, v))$  and  $s(fv) = lookup(s, fv)$ . A reminder that the sequence operation  $e_1; e_2$  that is used in the semantic rules for do-while construct is just a syntactic sugar for `try e1 catch (norm) e2`. Take note that the symbol  $\perp$ , appearing in the rules, stands for uninitialized.

In the rules for local declaration, method call and try catch, the function `newid()` returns a fresh identifier, while  $[u'/u]$  represents the substitution of  $u$  by  $u'$ . In order to avoid dynamic binding, every variable whose binding is added to the stack is substituted by a fresh identifier. For instance, in the case of the method call, after performing the renaming of the callee's formal parameters, the mappings for the fresh identifiers are temporarily added to the stack,  $s$ . These bindings will be removed after the evaluation of the callee's body.

## 5. Optimization Rules

We shall now provide a set of equivalence rules for our calculus that can be used for formal reasoning and optimization. One feature of our rules is that they are formulated at a relatively high-level based on our core calculus. We expect such high-level rules to be easier to understand and prove for correctness.

Our first rule is inspired by the associativity property of sequential composition, namely  $(e_1; e_2); e_3 \Leftrightarrow e_1; (e_2; e_3)$ . A corresponding rule, generalised to nested try-catch commands, is shown below:

### [Re-Ordering Rule]

$$\frac{c_2 <: c_1}{\text{try } (\text{try } e \text{ catch } (c_1) e_1) \text{ catch } (c_2) e_2 \Leftrightarrow \text{try } e \text{ catch } (c_1) (\text{try } e_1 \text{ catch } (c_2) e_2)}$$

This re-ordering may be used to help group a nested series of expressions with a similar property together, so that we may have a larger expression with the same property. For example, if expression  $e_1$  and  $e_2$  are free of dynamic control flows (namely exceptions), we may group them closer via the above rule. Grouping together expressions without exceptions can give us larger code blocks that can be better optimized.

The second optimization rule ensures the elimination of redundant catch handlers. This can occur if the try block never raises any of the control flows that are caught by a given handler. Take note that,  $throws(e)$  is intended to capture the entire set of exceptions (or control flows) that may escape from the expression  $e$ . The auxiliary function  $overlap(c_1, c_2)$  signifies the condition that  $c_1$  shares some common subtypes with  $c_2$ ,  $overlap(c_1, c_2) = c_1 <: c_2 \vee c_2 <: c_1$ .

### [Catch Elimination Rule]

$$\frac{\forall c \in throws(e) \cdot \neg overlap(c, c_1)}{\text{try } e \text{ catch } (c_1 @ fv_1 \# v_1) e_1 \Rightarrow e}$$

Our next rule is intended to optimize the stack unwinding mechanism that is typically used to propagate a raised exception to its

handler. Occasionally, it is possible to determine that the invocation of a particular exception (or control flow) will always be caught by a given handler. If this scenario is detected, we can transform a raised exception into a direct jump to its handler's code. Such a jump feature shall be added directly to the target compiled language. As our core calculus also supports the capture of both a control flow and its thrown value, we will have to pass these items to the handler during such a jump. Argument passing can be implemented with the help of a parameterized jump. A `jump(L(\vec{v}))` command is said to be parameterized by  $\vec{v}$  if it carries a list of arguments for its labelled location. Correspondingly, a labelled location `jump(L(\vec{v}) : e)` is said to be parameterized with  $\vec{v}$ , if variables  $\vec{v}$  in code  $e$  can be initialized by its corresponding jump. Our parameterized label shall always be used in the context of a catch handler of the form `try e catch (c@fv#v) L(fv, v):e2`, which shall be abbreviated as `try e catch (c) L(fv, v):e2`.

In real languages, the jump instruction would be a machine primitive that changes the current program counter to the address of the labelled instruction. However, for convenience, we shall model the jump construct (in the same way as `break` construct) with an abnormal control flow that would be caught by its labelled handler. Using this interpretation, the jump construct, `jump L(ft, x)`, is essentially modelled using `jump-L#(ft, x)`, where each flow type `jump-L` will only be captured by its handler at the labelled location  $L$ . We provide in Fig. 5 the big-step semantics for the newly introduced jump construct. Take note that the formalised semantic mirrors the state transition relation for the actual machine.

We shall now consider the optimization itself. As a simple example, consider the code fragment:

```
try try (try exc#3 catch (nullPtrExc) e1)
    catch (exc#v) e2
    catch (norm) e3
```

An exception `exc#3` is being thrown that will be caught by the second catch handler. We may directly link this throw with its corresponding catch handler by the following transformed code:

```
try try (try jump L1(exc, 3) catch (nullPtrExc) e1)
    catch (exc) L1(_, v) : e2
    catch (norm) e3
```

This is a desirable optimization since a jump command can usually be implemented much more efficiently. The rule for linking a throw with its corresponding catch handler can be formally expressed, as follows:

### [Throw-Catch Linking Rule]

$$\frac{escape(C[], fc) \quad \Gamma(ft) = fc \quad fc <: c}{\text{try } C[ft\#x] \text{ catch } (c@fv\#v) e \Rightarrow \text{try } C[\text{jump } L(ft, x)] \text{ catch } (c) L(fv, v) : e}$$

Note that we assume a prior type inference algorithm. Consequently,  $\Gamma$  denotes the type environment, with its corresponding runtime stack capturing both values and control flows. For each flow,  $ft$ ,  $\Gamma(ft)$  will capture the flow type. We use a context notation  $C[]$  to denote an expression with a single hole  $[]$ , and  $C[e]$  to denote the replacement of the hole by  $e$  for the given context. Formally, the context notation is defined as:

$$C[] ::= [] \mid \{t v; C[]\} \mid \text{if } v \text{ then } C[] \text{ else } e \mid \text{if } v \text{ then } e \text{ else } C[] \mid \text{try } C[] \text{ catch } (@fv\#v) e \mid \text{try } e \text{ catch } (c@fv\#v) C[] \mid \text{do } C[] \text{ while } v$$

We also provide an operator  $escape(C[], fc)$  that can determine if a control flow  $fc$  will escape its given context  $C[]$ . This operation is defined, as follows:

$$\begin{aligned} escape([], fc) &= \text{true} \\ escape(\text{try } C[] \text{ catch } (c) e, fc) \mid overlap(fc, c) &= \text{false} \\ escape(\text{try } C[] \text{ catch } (c) e, fc) \mid \neg overlap(fc, c) &= escape(C[], fc) \end{aligned}$$

$$\begin{array}{c}
\langle ft\#v, h, s \rangle \hookrightarrow_b \langle s(ft)\#s(v), h, s \rangle \quad \langle ft\#(fv, v), h, s \rangle \hookrightarrow_b \langle s(ft)\#(s(fv), s(v)), h, s \rangle \quad \langle ft\#k, h, s \rangle \hookrightarrow_b \langle s(ft)\#k, h, s \rangle \\
\\
\langle v.f, h, s \rangle \hookrightarrow_b \langle \mathbf{norm}\#(h(s(v)).f), h, s \rangle \quad \frac{l = \mathbf{fresh}()}{\langle ft\#\mathbf{new} \ c, h, s \rangle \hookrightarrow_b \langle s(ft)\#l, h + [l \mapsto c(\perp)], s \rangle} \\
\langle v_1 := v_2, h, s \rangle \hookrightarrow_b \langle \mathbf{norm}\#(), h, s[v_1 \mapsto s(v_2)] \rangle \quad \langle v_1.f := v_2, h, s \rangle \hookrightarrow_b \langle \mathbf{norm}\#(), h[s(v_1).f \mapsto s(v_2)], s \rangle \\
\\
\frac{\langle e, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle \quad c_1 <: c \quad u, fu = \mathbf{newid}()}{\langle e_2[fu/fv, u/v], h_1, s_1 + [fu \mapsto c_1, u \mapsto a_1] \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 \rangle} \quad \frac{\langle e, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle \quad \neg(c_1 <: c)}{\langle \mathbf{try} \ e \ \mathbf{catch} \ (c @ fv\#v) \ e_2, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle} \\
\langle \mathbf{try} \ e \ \mathbf{catch} \ (c @ fv\#v) \ e_2, h, s \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 - \{fu, u\} \rangle \quad \langle \mathbf{try} \ e \ \mathbf{catch} \ (c @ fv\#v) \ e_2, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle \\
\\
\frac{s(v) = \mathbf{true} \quad \langle e_1, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle}{\langle \mathbf{if} \ v \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle} \quad \frac{s(v) = \mathbf{false} \quad \langle e_2, h, s \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 \rangle}{\langle \mathbf{if} \ v \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2, h, s \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 \rangle} \\
\\
\frac{u = \mathbf{newid}() \quad \langle e[u/v], h, s + [u \mapsto \perp] \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle}{\langle \{t \ v; e\}, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 - \{u\} \rangle} \quad \frac{\langle e; \mathbf{if} \ v \ \mathbf{then} \ (\mathbf{do} \ e \ \mathbf{while} \ v) \ \mathbf{else} \ (), h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle}{\langle \mathbf{do} \ e \ \mathbf{while} \ v, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle} \\
\\
\frac{t_0 \ m \ [\mathbf{with} \ n] \ (\overrightarrow{t \ u}) \ \{e\} \quad u' = \mathbf{newid}() \quad \langle e[u'/u], h, s + [u' \mapsto s(v)] \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle}{\langle (m \ \overrightarrow{v}), h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 - \{\overrightarrow{u'}\} \rangle}
\end{array}$$

Figure 4. Big-Step Semantics

$$\begin{array}{c}
\frac{val = (s(ft), s(x))}{\langle \mathbf{jump} \ L(ft, x), h, s \rangle \hookrightarrow_b \langle \mathbf{jump} - L\#val, h, s \rangle} \quad \frac{\langle e_1, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle \quad \neg(c_1 <: c)}{\langle \mathbf{try} \ e_1 \ \mathbf{catch} \ (c) \ L(fv, v):e_2, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle} \\
\\
\frac{\langle e_1, h, s \rangle \hookrightarrow_b \langle \mathbf{jump} - L\#(c_1, a_1), h_1, s_1 \rangle \quad \langle e_2, h_1, s_1 + [fv \mapsto c_1, v \mapsto a_1] \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 \rangle}{\langle \mathbf{try} \ e_1 \ \mathbf{catch} \ (c) \ L(fv, v):e_2, h, s \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 - \{fv, v\} \rangle} \quad \frac{\langle e_1, h, s \rangle \hookrightarrow_b \langle c_1\#a_1, h_1, s_1 \rangle \quad c_1 <: c \quad \langle e_2, h_1, s_1 + [fv \mapsto c_1, v \mapsto a_1] \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 \rangle}{\langle \mathbf{try} \ e_1 \ \mathbf{catch} \ (c) \ L(fv, v):e_2, h, s \rangle \hookrightarrow_b \langle c_2\#a_2, h_2, s_2 - \{fv, v\} \rangle}
\end{array}$$

Figure 5. Big-Step Semantics for the Jump Construct

$escape(E, fc) = escape(C[], fc)$   
where  $E ::= \mathbf{if} \ v \ \mathbf{then} \ C[] \ \mathbf{else} \ e \mid \mathbf{if} \ v \ \mathbf{then} \ e \ \mathbf{else} \ C[]$   
 $\mid \mathbf{do} \ C[] \ \mathbf{while} \ v \mid \{t \ v; C[]\} \mid \mathbf{try} \ e \ \mathbf{catch} \ (c) \ C[]$

After some throw-catch linkings, it may be possible to obtain code involving a series of consecutive jumps. In order to shortcut such a series of jumps, we consider the scenario under which a parameterized jump can be inlined. This can occur if control flows from  $e$  are never caught by the context  $(\mathbf{try} \ C[] \ \mathbf{catch} \ (c) \ L(fv, v) : e)$ , as defined in the rule below:

$$\frac{\forall c_1 \in \mathbf{throws}(e) \cdot (escape(C[], c_1) \wedge \neg \mathbf{overlap}(c, c_1))}{\mathbf{try} \ C[\mathbf{jump} \ L(ft, x)] \ \mathbf{catch} \ (c) \ L(fv, v) : e} \\
\Rightarrow \mathbf{try} \ C[[fv \mapsto ft, v \mapsto x]e] \ \mathbf{catch} \ (c) \ L(fv, v) : e$$

To avoid name capture during this non-local inlining, we have to ensure that the variables from  $(\mathbf{free}(e) - \{fv, v\})$  do not clash with the bound variables from the context  $C[]$ . This can be ensured by uniquely renaming the bound variables in  $C[]$ .

### 5.1 Soundness of Optimization Rules

**DEFINITION 5.1 (Semantics Preserving Transformation).** An expression  $e'$  is said to be a semantics preserving transformation of  $e$  if, whenever the evaluation of  $e$  terminates and

$$\langle e, h, s \rangle \hookrightarrow_b \langle c\#a, h_1, s_1 \rangle,$$

then the evaluation of  $e'$  terminates and

$$\langle e', h, s \rangle \hookrightarrow_b \langle c\#a, h_1, s_1 \rangle.$$

**DEFINITION 5.2 (Sound Optimization Rule).** An optimization rule rewriting an expression  $exp$  into an expression  $exp'$  is said to be sound if  $exp'$  is a semantics preserving transformation of  $exp$ .

Def. 5.1 and Def. 5.2 are used to state the soundness of the optimization rules. When sketching the proofs of the soundness lemmas, we will make use of some notational conventions. The initial

expression, before optimization, will be denoted by  $exp$ , while the optimized one, obtained after rewriting, will be denoted by  $exp'$ . Additionally,  $h$  will stand for the current heap, and  $s$  for the current stack.

**LEMMA 5.1 (Soundness of Re-Ordering Rule ( $\Rightarrow$  direction)).** If  $c_2 <: c_1$  then the following expression:

$$\mathbf{try} \ e \ \mathbf{catch} \ (c_1 @ fv_1\#v_1) \ (\mathbf{try} \ e_1 \ \mathbf{catch} \ (c_2 @ fv_2\#v_2) \ e_2)$$

is a semantics preserving transformation of:

$$\mathbf{try} \ (\mathbf{try} \ e \ \mathbf{catch} \ (c_1 @ fv_1\#v_1) \ e_1) \ \mathbf{catch} \ (c_2 @ fv_2\#v_2) \ e_2$$

**Proof:** According to the hypothesis that the evaluation of  $exp$  terminates and to the big-step semantics of the try catch construct, we can assume that  $e$  gets evaluated to  $c\#a$ . Consequently, there are two possibilities:

- $c <: c_1$ . In this case, the evaluation of  $exp$  is reduced to the evaluation of the handler  $e_1$  in a  $\mathbf{try} \ e_1 \ \mathbf{catch} \ (c_2) \ e_2$  construct. On the other hand, when considering the expression  $exp'$  obtained after optimization,  $e$  gets evaluated in a similar way and, assuming that  $c <: c_1$ , everything is reduced again to  $\mathbf{try} \ e_1 \ \mathbf{catch} \ (c_2) \ e_2$ . Conclusion follows from here.
- $\neg(c <: c_1)$ . Conform to the big-step semantics, the handler  $e_1$  is not reachable in  $exp$ . Moreover, due to the assumption that  $c_2 <: c_1$ , the handler  $e_2$  is also unreachable. Therefore,  $exp$  is evaluated to  $c\#a$ . On the other hand, in  $exp'$ , the handler  $\mathbf{try} \ e_1 \ \mathbf{catch} \ (c_2 @ fv_2\#v_2) \ e_2$  is also unreachable. Hence,  $exp'$  also evaluates to  $c\#a$ .  $\square$

Take note that the **[Re-Ordering Rule]** rule claims the equivalence of the two expressions,  $exp$  and  $exp'$ . Therefore, we need another lemma for proving soundness when the rule is applied from right to left.

**LEMMA 5.2 (Soundness of Re-Ordering Rule ( $\Leftarrow$  direction)).**

If  $c_2 <: c_1$  then the following expression:

$$\text{try } (\text{try } e \text{ catch } (c_1 @ f v_1 \# v_1) e_1) \text{ catch } (c_2 @ f v_2 \# v_2) e_2$$

is a semantics preserving transformation of:

$$\text{try } e \text{ catch } (c_1 @ f v_1 \# v_1) (\text{try } e_1 \text{ catch } (c_2 @ f v_2 \# v_2) e_2)$$

**Proof:** Similar to the proof for Lemma 5.1.  $\square$

**LEMMA 5.3 (Soundness of Catch Elimination Rule).** Assuming that  $\forall c \in \text{throws}(e) \cdot \neg \text{overlap}(c, c_1)$  then expression  $e$  is a semantics preserving transformation of  $\text{try } e \text{ catch } (c_1 @ f v_1 \# v_1) e_1$ .

**Proof:** We show that  $\text{exp}'$  is a semantics preserving transformation of  $\text{exp}$  by proving that the handler  $e_1$  of  $\text{exp}$  is unreachable. Let us assume that  $e$  evaluates to  $c \# a$ . According to the assumption that all the exceptions are checked (any exception escaping from a method's body must be declared in its throws list), to the big-step semantics for try catch, and the hypothesis that all the control flows of  $e$  escape from  $\text{exp}$ , the handler  $e_1$  is unreachable.  $\square$

**LEMMA 5.4. Soundness of Throw-Catch Linking Rule:** If the following conditions hold:

$$\text{escape}(C[], fc) \text{ and } \Gamma(ft) = fc \text{ and } fc <: c$$

then the expression:  $\text{try } C[\text{jump } L(ft, x)] \text{ catch } (c) L(fv, v) : e$  is a semantics preserving transformation of the expression

$$\text{try } C[ft \# x] \text{ catch } (c @ f v \# v) e.$$

**Proof:** By structural induction on the context of the try block  $C[]$ .

- Case  $[]$ . Straightforward.
- Case  $\text{try } C[] \text{ catch } (c) e$ . Conform to the big-step semantics, the evaluation of both  $\text{exp}$  and  $\text{exp}'$  implies first the evaluation of the inner try block. According to the hypothesis that  $\Gamma(ft) = fc$  and that the control flow  $fc$  escapes the context  $C[]$  being caught by the external handler, both expressions will reduce to the following machine configuration:  $\langle e, h_1, s_1 + [fv \mapsto ft, v \mapsto x] \rangle$ , where  $s_1$  and  $h_1$  denote the stack and heap after the evaluation of the try block. Conclusion is immediate.
- Case  $\text{do } C[] \text{ while } v$ . The initial expression  $\text{exp}$  reduces to:
$$\text{try } C[ft \# x]; \text{ if } v \text{ then } (\text{do } C[ft \# x] \text{ while } v) \text{ else } () \text{ catch } (c @ f v \# v) e$$

which is syntactic sugar for:

$$\text{try } (\text{try } C[ft \# x] \text{ catch } (norm) \text{ if } v \text{ then } (\text{do } C[ft \# x] \text{ while } v) \text{ else } ()) \text{ catch } (c @ f v \# v) e$$

On the other hand,  $\text{exp}'$  reduces to:

$$\text{try } (\text{try } C[\text{jump } L(ft, x)] \text{ catch } (norm) \text{ if } v \text{ then } (\text{do } C[\text{jump } L(ft, x)] \text{ while } v) \text{ else } ()) \text{ catch } (c) L(fv, v) : e$$

The conclusion follows from the hypothesis that  $\text{escape}(C[], fc)$  and  $\Gamma(ft) = fc$  and  $fc <: c$  and from the dynamic semantics.

- Case  $t v; C[]$ . We conclude immediately from the big-step semantics and the hypothesis that  $\Gamma(ft) = fc$  and the control flow  $fc$  escapes the context  $C[]$  and it is caught by the external handler. Both  $\text{exp}$  and  $\text{exp}'$  reduce to  $\langle e, h_1, s_1 + [fv \mapsto ft, v \mapsto x] \rangle$ , where  $s_1$  and  $h_1$  denote the stack and heap after the evaluation of the try block, respectively.
- Case  $\text{if } v \text{ then } C[] \text{ else } e_1$ . According to the big-step semantics for the if construct, there are two cases:
  - $s(v) = \text{true}$ . The two expressions,  $\text{exp}$  and  $\text{exp}'$ , reduce respectively to:
$$\text{try } C[ft \# x] \text{ catch } (c @ f v \# v) e$$
and
$$\text{try } C[\text{jump } L(ft, x)] \text{ catch } (c) L(fv, v) : e$$
Conclusion follows immediately.

- $s(v) = \text{false}$  According to the big-step semantics:

- $\text{exp}$  reduces to:  $\text{try } e_1 \text{ catch } (c @ f v \# v) e$
- $\text{exp}'$  reduces to:  $\text{try } e_1 \text{ catch } (c) L(fv, v) : e$

Follows from the big-step semantics, as  $e_1$  does not contain any jump construct.

- Case  $\text{if } v \text{ then } e_1 \text{ else } C[]$ . Similar to the previous case.
- Case  $\text{try } e_1 \text{ catch } ((c_1 @ f v_1) \# v_1) C[]$ . According to the big-step semantics, there are two cases:
  - the evaluation of  $e_1$  generates a control flow  $c'$  and  $c' <: c_1$ . From the dynamic semantics, the handlers  $C[ft \# x]$  and  $C[\text{jump } L(ft, x)]$  are to be evaluated, respectively. Conclusion follows from the induction hypothesis.
  - the evaluation of  $e_1$  generates a control flow  $c'$  and  $\neg(c' <: c_1)$ . Conclusion follows from the fact that  $e_1$  does not contain any jump construct.  $\square$

**LEMMA 5.5 (Soundness of Jump Inlining Rule:).** Assuming that the following conditions hold:

$$\forall c_1 \in \text{throws}(e) \cdot (\text{escape}(C[], c_1) \wedge \neg \text{overlap}(c, c_1))$$

then expression

$$\text{try } C[[fv \mapsto ft, v \mapsto x]e] \text{ catch } (c) L(fv, v) : e$$

is a semantics preserving transformation of the expression

$$\text{try } C[\text{jump } L(ft, x)] \text{ catch } (c) L(fv, v) : e.$$

**Proof:** By structural induction on the context of the try block  $C[]$ .

- Case  $[]$ . Straightforward.
- Case  $\text{try } C[] \text{ catch } (c) e$ . Conform to the big-step semantics, the evaluation of both  $\text{exp}$  and  $\text{exp}'$  implies first the evaluation of the inner try block. The conclusion comes from the assumption that all the control flows from  $e$  escape the context  $C[]$  and from the induction hypothesis.
- Case  $\text{do } C[] \text{ while } v$ . The initial expression  $\text{exp}$  reduces to:
$$\text{try } (C[\text{jump } L(ft, x)]; \text{ if } v \text{ then } (\text{do } C[\text{jump } L(ft, x)] \text{ while } v) \text{ else } ()) \text{ catch } (c) L(fv, v) : e$$

which is syntactic sugar for:

$$\text{try } (\text{try } C[\text{jump } L(ft, x)] \text{ catch } (norm) \text{ if } v \text{ then } (\text{do } C[\text{jump } L(ft, x)] \text{ while } v) \text{ else } ()) \text{ catch } (c) L(fv, v) : e$$

On the other hand,  $\text{exp}'$  reduces to:

$$\text{try } (\text{try } C[[fv \mapsto ft, v \mapsto x]e] \text{ catch } (norm) \text{ if } v \text{ then } (\text{do } C[[fv \mapsto ft, v \mapsto x]e] \text{ while } v) \text{ else } ()) \text{ catch } (c) L(fv, v) : e$$

From the big-step semantics and the assumption that all the control flows from  $e$  escape both the context  $C[]$  and the external try catch, the two expressions will reduce to  $[fv \mapsto ft, v \mapsto x]e$ .

- Case  $t v_1; C[]$ . The two expressions,  $\text{exp}$  and  $\text{exp}'$ , reduce to
$$\text{try } (t v_1; C[\text{jump } L(ft, x)]) \text{ catch } (c) L(fv, v) : e$$
and
$$\text{try } (t v_1; C[[fv \mapsto ft, v \mapsto x]e]) \text{ catch } (c) L(fv, v) : e,$$
respectively. According to the big-step semantics and the assumption that all the control flows from  $e$  escape both the context  $C[]$  and the external try catch construct, the two expressions will evaluate  $e$ . Note that, for the latter case, the stack will also contain the local variable  $v_1$ . In order to avoid name clashing, we uniquely rename the free variables of  $e$ . Consequently, the result of the evaluation is the same regardless of whether or not  $v_1$  is on the stack.
- Case  $\text{if } v \text{ then } C[] \text{ else } e_1$ . According to the big-step semantics for the if construct, there are two possibilities:

- $s(v)=\text{true}$ . The conclusion follows immediately from the induction hypothesis as the two expressions,  $exp$  and  $exp'$ , reduce respectively to:

$\text{try } C[\text{jump } L(ft, x)] \text{ catch } (c) L(fv, v) : e$

and  $\text{try } C[[fv \mapsto ft, v \mapsto x]e] \text{ catch } (c) L(fv, v) : e$ .

- $s(v)=\text{false}$ . Both expressions reduce to:  
 $\text{try } e_1 \text{ catch } (c) L(fv, v) : e$ .

- Case  $\text{if } v \text{ then } e_1 \text{ else } C[]$ . Similar to the previous case.
- Case  $\text{try } e_1 \text{ catch } ((c_1 @ fv_1) \# v_1) C[]$ . According to the big-step semantics, there are two possibilities:
  - the evaluation of  $e_1$  generates a control flow  $c'$  with the property  $c' <: c_1$ . From the dynamic semantics, the handlers  $C[\text{jump } L(ft, x)]$  and  $C[[fv \mapsto ft, v \mapsto x]e]$  are to be evaluated, respectively. Conclusion follows from the induction hypothesis.
  - the evaluation of  $e_1$  generates a control flow  $c'$  and  $\neg(c' <: c_1)$ . Both expressions reduce to  $\text{try } e_1 \text{ catch } (c) L(fv, v) : e$ .

□

## 6. Related Work

Exceptions and abnormal control flows are undoubtedly important parts of programming language systems that should be adequately handled during language design, program analysis, implementation, optimization and also program verification. In recent years, there have been a flurry of activities in the above tasks to do better justice for these somewhat neglected language features.

At a foundational level, Ancona et al [1] have added a limited form of exceptions into Featherweight Java [16] to formally study the interaction between inheritance and exceptions. Others [19, 9, 4] have provided a richer core language with exceptions but have not taken the penultimate<sup>1</sup> step in unifying the myriad of control flows, including that for normal execution. These proposals have formalised type systems that have been proven sound, though [19] skipped the tracking of uncaught exceptions.

Another important dimension is for the inference of uncaught exceptions [17, 11, 24]. The outcome of such an analysis can be used to verify the user-supplied set of uncaught exceptions or to automatically determine the set of uncaught exceptions, for each method. For example, [17] employs two analyses of different granularity, at the expression and method levels, to collect constraints over the set of exception types handled by each Java program. By solving the constraint system using techniques from [11], they can obtain a fairly precise set of uncaught exceptions.

Exceptions have also been used in functional languages [10, 18], which originated from early Lisp/ML implementations. A key challenge arises with higher-order functions, but techniques for uncaught exceptions based on effects type system can be used in this setting. In the case of purely functional Haskell language [18], Peyton-Jones et al. used IO monads to confine the effects of exceptions and also to deal with lazy evaluation. They also argue that imprecise exceptions are natural for lazy semantics and that non-determinism introduced by exceptions is acceptable. Another step towards supporting exceptions for the entire non-strict functional language, without being constrained to just the monadic fragment, was taken by Glynn et al. [13]. The authors added non-determinism into the underlying semantics. This generalization is helpful for non-strict languages, but is achieved at the expense of a more complex semantics. A recent approach towards exception handling in a higher-order setting, is taken in [3]. Exceptions are represented as sums and exception handlers are assigned polymorphic, extensible

row types. Furthermore, following a translation to an internal language, each exception handler is viewed as an alternative continuation whose domain is the sum of all exceptions that could arise at a given program point. In contrast to our work, the above proposals in the functional setting do not provide a hierarchy on the exceptions that is typically expected in the object-oriented paradigm. Moreover, there is no attempt to unify the control flows, with exceptions being treated separately from normal execution. This can make language extensions harder. However, our current formulation is in a first-order setting, but we plan to extend it in the near future to the higher-order polymorphic setting.

A construct similar in nature with exceptions is the multi-return function call ([27]). A method is said to have multiple returns if each of its calls is allowed to choose from a number of return addresses for its method's exit. Each such return can be implemented efficiently as a local transfer of control. In comparison, exceptions are more general (but also more expensive) than multi-return function calls as they can be used to implement non-local control transfers. When an exception occurs, control can be transferred to a handler further back in the control chain. By providing a hierarchy of control flows, our approach can unify (and distinguish) both the dynamic and the static control flows within a single setting. This has helped to support a smaller core calculus that can facilitate both program analysis and efficient implementation.

Exception-based programs are generally harder to analyse, since they introduce extra control flows to the programs. Various techniques have been proposed to handle this challenge, but most of them have been carried out at a lower-level of abstraction. For example, [28] describes the effects of exceptions on a control-flow, data-flow and control dependence analysis and suggested using a control flow graph that will contain exceptional flows. For this approach to be concise, a prior type inference for exception types must also be done. Along the same line, [6] introduced a form of a compact control flow graph that includes exceptional control flow and demonstrated that it could be used to construct an improved interprocedural analysis. Weimer and Necula [30] use a data flow analysis to detect broken specifications for resource usage. Even in the verification setting, Barnett and Leino [2] resorted to lower-level unstructured programs (with global variables) to model and analyse exception-based programs. Though lower-level constructs can be made to work, they are typically harder to formulate and prove correct. Our proposal to unify both normal and abnormal control flows in a core calculus is aimed at providing a higher-level of abstraction for reasoning about exception-based programs, including those needed by advance program analyses.

It is important to implement exception-based programs efficiently. [25] looked at different approaches to exception handling in the framework of an intermediate language, called C--, using stack cutting and methods with multiple returns. To reduce the overheads of stack unwinding, [7] proposed to cache frequent unwinding paths. They also look at delaying the creation of exceptions, and hence the stack unwinding, until it is really being used. Ogaraware et al [23] investigated a new technique, called Exception Directed Optimization, that has no penalty on normal path but optimizes frequently executed exception handling paths. Two techniques used are *linking* and *inlining*. Linking is used if an exception raised is always caught by a handler in the same method. *Inlining* is applied to non-recursive methods to aggressively link exception throwing points with its catch handler in a parent method, where desirable. However, no correctness proofs are given for these optimizations. In comparison, our throw-catch linking optimization is formulated at a higher-level of abstraction and has been proven correct. Moreover, it is possible to perform inter-procedural optimization (without inlining) through methods with multiple returns.

<sup>1</sup>We leave open what could be the ultimate step.

To support deeper reasoning of exception-based programs, Huisman et al. [15] proposed an extension for Hoare-style verification to handle exceptions and abnormal control flows. Separate logical rules were formulated to support reasoning on total and partial correctness, and for handling each kind of abnormal control flows. However, the myriad of logical rules used can be complex to implement. At the specification level, both SPEC#[20] and ESC/Java[12] have provided means to specify program states for exceptional scenarios via a specification construct, called *exsures*. Similarly [5] also employed weakest precondition calculation for soundly verifying Java bytecode annotated with pre and post conditions, and exceptional specifications. In the case of our core calculus, we would be able to provide a unified specification mechanism for the outcomes of both normal and abnormal executions, without having to treat exceptions in a special way.

## 7. Conclusion

We have presented a new core calculus with unified control flows that is compact and expressive. Our original motivation for this work was to develop a verifier that could support formal reasoning of exception-based programs. During the implementation process, we have found it beneficial to make a fundamental language re-design to uniformly handle all kinds of control flows, for both normal and exceptional executions. This decision has allowed us to elegantly formalise a core calculus with its big-step semantics and optimization rules. Our calculus will be useful for a wide range of applications, including compilation, analysis, optimization, verification and language design. On the last application, we expect language designers to be able to formulate new language features by succinct translations to our calculus. By providing a smaller calculus with generalised constructs, we expect these tasks to be carried at a high-level of abstraction with lower cost, due to the minimalist but expressive nature of our calculus.

## Acknowledgments

We thank Florin Craciun and Stefan Andrei for their help in an earlier project. We also thank Luke Ong for his insightful comments on this work, including some suggestions for a set of clearer terminologies. Thanks also to Tobias Nipkow and Kwangkeun Yi for answering some queries on their papers, and to the PEPM'09 reviewers for their consistent comments.

## References

- [1] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In *OOPSLA*, pages 16–30, 2001.
- [2] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
- [3] Matthias Blume, Umut A. Acar, and Wonseok Chae. Exception handlers as extensible cases. In *APLAS*, 2008.
- [4] Egon Börger and Wolfram Schulte. A practical method for specification and analysis of exception handling - a Java/Jvm case study. *IEEE Trans. Software Eng.*, 26(9):872–887, 2000.
- [5] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. JVer: A Java verifier. In *CAV*, pages 144–147, 2005.
- [6] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *SIGSOFT Softw. Eng. Notes*, 24(5):21–31, 1999.
- [7] M. Cierniak, G. Lueh, and JM. Stihnoth. Practicing JUDO : Java under dynamic optimization. In *PLDI*, Vancouver, Canada, 2000.
- [8] Sophia Drossopoulou and Susan Eisenbach. Java is type safe - probably. In *ECOOP*, pages 389–418, 1997.
- [9] Sophia Drossopoulou and Tanya Valkevych. Java exceptions throw no surprises. Technical report, March 2000.
- [10] M. Fähndrich, J.S. Foster, A. Aiken, and J. Cu. Tracking down exceptions in Standard ML programs. In *Report UCB/CSD 98-996*, February 1999.
- [11] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *SAS*, pages 114–126, London, UK, 1997. Springer-Verlag.
- [12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, New York, NY, USA, 2002. ACM.
- [13] K. Glynn, P. Stuckey, M. Sulzmann, and H Sondergaard. Exception analysis for non-strict languages. In *ICFP*. ACM Press, 2002.
- [14] P. Hudak and et al. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992.
- [15] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *FASE*, pages 284–303, 2000.
- [16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, Denver, Colorado, November 1999.
- [17] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72(1):59–69, 2004.
- [18] S. Peyton Jones, A. Reid, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI*, 1999.
- [19] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
- [20] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM*, pages 218–227, Washington, DC, USA, 2004. IEEE Computer Society.
- [21] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, Nice, France, January 2007.
- [22] Tobias Nipkow and David von Oheimb. Java<sub>light</sub> is type-safe - definitely. In *POPL*, pages 161–170, 1998.
- [23] T. Ogarawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in Java. In *OOPSLA*, Tampa Bay, Florida, October 2001.
- [24] Francois Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *POPL*, pages 276–290, 1999.
- [25] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *PLDI*, 2000.
- [26] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *AOSD*, pages 135–148, New York, NY, USA, 2007. ACM.
- [27] Olin Shivers and David Fisher. Multi-return function call. In *ICFP*, pages 79–89, New York, NY, USA, 2004.
- [28] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Software Eng.*, 26(9):849–871, 2000.
- [29] Eelco Visser. Program transformation with Stratego/XT. Rules, strategies, tools, and systems in Stratego/XT 0.9. Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004.
- [30] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2):1–51, 2008.