Accepting Blame for Safe Tunneled Exceptions

Yizhou Zhang* Guido Salvaneschi† Quinn Beightol*
Barbara Liskov‡ Andrew C. Myers*

*Cornell University, USA †TU Darmstadt, Germany ‡MIT, USA
yizhou@cs.cornell.edu salvaneschi@cs.tu-darmstadt.de qeb2@cornell.edu
liskov@csail.mit.edu andru@cs.cornell.edu

Abstract

Unhandled exceptions crash programs, so a compile-time check that exceptions are handled should in principle make software more reliable. But designers of some recent languages have argued that the benefits of statically checked exceptions are not worth the costs. We introduce a new statically checked exception mechanism that addresses the problems with existing checked-exception mechanisms. In particular, it interacts well with higher-order functions and other design patterns. The key insight is that whether an exception should be treated as a “checked” exception is not a property of its type but rather of the context in which the exception propagates. Statically checked exceptions can “tunnel” through code that is oblivious to their presence, but the type system nevertheless checks that these exceptions are handled. Further, exceptions can be tunneled without being accidentally caught, by expanding the space of exception identifiers to identify the exception-handling context. The resulting mechanism is expressive and syntactically light, and can be implemented efficiently. We demonstrate the expressiveness of the mechanism using significant codebases and evaluate its performance. We have implemented this new exception mechanism as part of the new Genus programming language, but the mechanism could equally well be applied to other programming languages.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Control structures

Keywords Exception tunneling; exception handling; Genus

1. Introduction

Exceptions make code more reliable by helping programmers handle abnormal or unusual run-time conditions. The core idea is to transfer control in a nonlocal way to handler code that can be factored out from common-case code. This separation of concerns simplifies code and prompts programmers not to forget about exceptional conditions.

There has been disagreement since the 1970’s about how or whether exceptions should be subject to static checking [19, 28]. This disagreement continues to the present day [17]. Some currently popular languages—Java [20] and Swift [46]—offer checked exceptions that the compiler statically ensures are handled. However, exceptions are not part of type checking in other popular languages such as C++ [44], C# [23], Scala [36], and Haskell [38].

Proponents of static checking argue that exceptions represent corner cases that are easy to forget. The evidence suggests they have a point. One study of a corpus of C# code [8] determined that 90% of the possible exceptions are undocumented. Undocumented exceptions make it hard to know whether all exceptions are handled, and these unhandled exceptions percolate up through abstraction layers, causing unexpected software failures. Statically checked exceptions help programmers build more robust code [48].

Opponents of static checking argue that the annotation burden of statically checked exceptions does not pay off—that statically checked exceptions are too rigid to support common design patterns and common ways in which software evolves [11, 51]. They, too, have a point. The problems with statically checked exceptions have become more apparent in recent years as object-oriented (OO) languages like C#, Scala, and Java have acquired lambda expressions and the use of higher-order functions has become more common. As a result, the promise of exceptions to help make software more reliable has been partly lost.

Studies of the effectiveness of exception mechanisms have concluded that existing mechanisms do not satisfy the appropriate design criteria [6, 7]. C# does not statically check exceptions because its designers did not know how to design such an exception mechanism well, saying “more thinking is needed before we put some kind of checked exceptions
mechanism in place” [22]. It seems the long-running conflict
between checked and unchecked exceptions can be resolved
only by a new exception mechanism.

This paper aims to provide a better exception mecha-
nism, one that combines the benefits of static checking with
the flexibility of unchecked exceptions. The new mechanism
gives programmers static, compile-time guidance to ensure
exceptions are handled, but works well with higher-order
functions and design patterns. It adds little programmer bur-
den and even reduces that burden. The run-time overhead
of the mechanism is low, because exception handling does
not require stack-trace collection in common use cases and
avoids the need to wrap checked exceptions inside unchecked
ones.

Two main insights underlie the new design. The first is that
the distinction between “checked” and “unchecked” should
not be a property of the type of the exception being raised, as
it is in Java, but rather a property of the context in which the
exception propagates. In contexts that are aware of an excep-
tion, the exception should be checked statically to ensure that
it is handled. To handle higher-order functions and design
patterns, however, some contexts must be oblivious to ex-
ceptions propagating through them; exceptions should tunnel
uncaught through oblivious contexts, effectively unchecked.

This principle implies that the same exception may be both
checked and unchecked at different points during its propa-
gation. To prevent oblivious code from accidentally catch-
ing exceptions, a second insight is needed: exceptions can be
distinguished by expanding the space of exception identifiers
with an additional label that describes the exception-aware
context in which this exception can be caught. These labels
can be viewed as an extension of the notion of blame labels
found in previous work on gradual typing [50]. Unlike with
gradual typing, this sort of “blame” is not a programmer er-
rnor; it is instead a way to indicate that exceptions should tun-
nel through the oblivious code until they arrive at the right
exception-aware context.

We start the rest of the paper by exploring requirements
for a good exception mechanism in Section 2. Sections 3–6
present our new exception mechanism informally in the con-
text of a Java-like language. Section 7 defines a core language
whose semantics show more precisely how the mechanism
works. Using this core language, we prove the key theorem
that all exceptions are handled explicitly. Section 8 describes
our implementation of the new exception mechanism in the
context of the Genus programming language [53]. The effec-
tiveness of the mechanism is evaluated in Section 9, using
code drawn from real-world codebases. Related work is ex-
plored more fully in Section 10; we conclude in Section 11.

2. Design Principles for Exceptions

The goal of an exception mechanism is to simplify and reg-
ularize the handling of exceptional events, making programs
more reliable. However, there are two quite different classes
of exceptional events, with different design goals:

- **Failures**. Some events cannot reasonably be expected to
  be handled correctly by the program—especially, events
  that arise because of programmer mistakes. Other events
  such as running out of memory also fall into this category.

- **Unusual conditions**. Other events arise during correct
  functioning of the program, in response to an unusual
  but planned-for state of the environment, or even just an
  unusual case of an algorithm.

These two classes place different requirements on the ex-
ception mechanism. For failures, efficiency is not a concern
because the program is not expected to recover. However,
programmers need the ability to debug the (stopped) program
to discover why the failure occurred, so it is important to
collect a stack trace. Furthermore, since failures imply vio-
lation of programmer assumptions, having to declare them
as part of method signatures or write handler code for them
is undesirable. Nonetheless, there are cases where the ability
to catch failure exceptions is useful, such as when building
frameworks for executing code that might fail.

For the second class of exceptions, unusual conditions, the
design goals are different. Now efficiency matters! Because
exceptions are slow in many common languages, program-
ners have learned to avoid using them. One insight is that
because unusual conditions are part of the correct function-
ing of the program, the overhead of collecting a stack trace is
unnecessary.

Unfortunately, existing languages tend not to support the
distinction between these two exception classes well. For ex-
ample, typical Java usage always leads to stack-trace collec-
tion, making exceptions very expensive. At the same time,
code is cluttered with handlers for impossible exceptions.

2.1 Higher-Order Functions and Exceptions

Current exception mechanisms do not work well in code that
uses higher-order functions. An example is an ML-style map
method that applies a function argument to each element
of a list, returning a new list. Callers of the higher-order
function may wish to provide as an argument a function
that produces exceptions to report an unusual condition that
was encountered, such as an I/O exception. Of course, we
want these exceptions to be handled. But the implementation
of map knows nothing about these exceptions, so if such
exceptions do occur, they should be handled by the caller of
map. It is unreasonable for map either to handle or to declare
them—its code should be oblivious to the exceptions.

This example illustrates why otherwise statically typed
functional programming languages such as ML and Haskell
do not try to type-check exceptions statically [25]. The
problem has also become more prominent in modern OO lan-
guages that have added lambda expressions and are increas-
ingly relying on libraries that use them (e.g., JavaFX, Apache
Spark). The problem is encountered even without lambda ex-
pressions, though; for example, Java’s Iterator interface declares no exceptions on its methods, which means that implementations of Iterator are not allowed to generate exceptions—unless they are made “unchecked”.

Our goal is to achieve the notational convenience of the functional programming languages along with the assurance that exceptions are handled, which is offered by languages such as Java, Swift [46], and Modula-3 [34]. We propose that a higher-order function like map should be implementable in a way that is oblivious to the exceptions possibly generated by its argument. The possible exceptions of the argument should not be declared in the signature of map; nor should the code of map have to say anything about these exceptions.

A subtle problem arises when a higher-order function like map uses exceptions inside its own implementation. If the exceptions of the argument and the internal exceptions collide, the map code could then accidentally catch exceptions that are not intended for it—an effect we call exception capture by analogy with the problem of variable capture in dynamically scoped languages [43]. For modularity, a way is needed to tunnel such exceptions through the intermediate calling context. In fact, accidentally-caught exceptions are a real source of serious bugs [10, 13].

An alternative to our oblivious-code approach that has been suggested previously [18, 24, 41, 47] is to parameterize higher-order code like map over the unknown exceptions of its argument. This exception polymorphism approach requires writing annotations on oblivious code yet still permits accidental exception capture.

2.2 Our Approach
Backed by common sense and some empirical evidence, we believe that code is more reliable when compile-time checking guides programmers to handle exceptional cases. It is disappointing that recent language designs such as C# and Scala have backed away from statically declared exceptions.

We propose a new statically checked exception mechanism that addresses the weaknesses of prior exception mechanisms:

- It supports static, local reasoning about exceptions. Local reasoning is efficient, but more importantly, it aids programmer understanding. A code context is required to handle only the exceptions it knows about statically.

- The mechanism is cheap when it needs to be: when exceptions are used for nonlocal control flow rather than failures.

- In the failure case, however, the mechanism collects the stack trace needed for debugging.

- It supports higher-order functions whose arguments are other functions that might throw exceptions to which the higher-order code is oblivious.

- It avoids the exception-capture problem both for higher-order functions and for failures.

3. The Exception Mechanism
We use Java as a starting point for our design because it is currently the most popular language with statically checked exceptions. Our design is presented as a version of the Genus language, a variant of Java with an expressive constrained genericity mechanism [53]. The essential ideas should apply equally well to other languages, such as Java, C# [31], Scala [36], and ML [32]. Since exception-oblivious code (like map) is often generic, it is important to study how exceptions interact with sophisticated generics.1

Previous languages have either had entirely “checked” or “unchecked” exceptions (in Java’s terminology), or, as in Java, have assigned exception types to one of these two categories. Our insight is that “checked” vs. “unchecked” is a property of the context of the exception rather than of its type. Any exception should be “checked” in a context that is not oblivious to the exception and can therefore handle it. But in a context that is oblivious to the exception, the exception should be treated as “unchecked”.

Genus requires that a method handle or explicitly propagate all exceptions it knows can arise from operations it uses or methods it calls. If the implementation of a method explicitly throws an unhandled exception whose type (or the supertype thereof) is not listed in the method’s header, the program is rejected.

Like in Java, exceptions can be handled using the try–catch construct, and a finally block that is always executed may be provided. (The try-with-resources statement of Java 7 is easily supported; it is orthogonal to the new features.)

A method that wishes merely to propagate an exception to its caller can simply place a throws clause in its signature. We say such an exception propagates in checked mode. Unlike in Java, exceptions in checked mode do not cause stack-trace collection.

3.1 Failures
Unlike Java, our mechanism makes it easy for the programmer to indicate that an exception should not happen. The programmer ordinarily does this by putting a fails clause in the method header. Any caller of the method is then oblivious to the exception, meaning that the exception will be treated as unchecked as it propagates further. When code fails because of an exception, the exception propagates in a special mode, the failure mode.

For example, a programmer who is certain that the Object class can be loaded successfully can write

```java
class loadObject() fails ClassNotFoundException {     
  return Class.forName("genus.lang.Object"); 
}
```

where the method forName in Class declares ClassNotFoundException in its throws clause. Note that a fails clause

---

1 Genus uses square brackets rather than angle brackets for generic type arguments: List[T] rather than List<T>. For brevity, we use Genus syntax and the Genus equivalents of Java core classes without further explanation.
is really part of the implementation rather than part of the method signature or specification. We write it in the header just for convenience.

Exceptions propagating in failure mode also differ in what happens at run time. Programmers need detailed information to debug the stopped program to discover how the failure occurred. Therefore, failure exceptions collect a stack trace. This is relatively slow (as slow as most Java exceptions!) but efficiency is not a concern for failures.

3.2 Avoiding Exception Capture
Since exceptions propagating in failure mode do not appear in method signatures, it is important to avoid catching them accidentally. For example, consider the following code that calls two functions \( g() \) and \( h() \):

```java
void f() {
    try {
        g(); // signature says "throws MyExc"
        h(); // signature doesn't say "throws MyExc"
    } catch (MyExc e) {...}
}
```

Suppose that because of a programmer mistake, the call to \( h() \) unexpectedly fails with exception \( MyExc \). If this exception were caught by the catch clause, \( f() \) would execute code intended to compensate for something that happened in \( g() \). We prevent this undesirable exception capture by ensuring that failure exceptions cannot be caught by any ordinary catch clauses: failure exceptions tunnel past all ordinary catch clauses.

Although exceptions in failure mode are not normally handled, there may be value in catching them at the top level of the program or at the boundary between components, to allow for more graceful exit. Genus supports this with a rarely used catch all construct that catches all exceptions of a given type, regardless of propagation mode. For example, if the try statement above were extended to include a second clause catch \( e \) (MyExc \( e \)) {...}, the first catch clause would catch the expected MyExcs thrown by \( g() \), and the second catch all clause would catch failure-mode MyExcs tunnelled through \( h() \).

3.3 Fail-by-Default Exceptions
Java has a commonly accepted set of exceptions that usually correspond to programmer mistakes: the built-in subclasses of RuntimeException or Error. To reduce annotation burden for the programmer, our mechanism does not ordinarily require writing a fails clause in order to convert such exceptions to failure mode. We say these exceptions fail by default.

Fail-by-default exceptions are different from Java’s unchecked exceptions. Unchecked exceptions confute failures with ordinary exceptions that are tunneling through oblivious code but that still ought to be subject to static checking.

In contrast, fail-by-default exceptions remain in checked mode until they reach a method-call boundary; they convert from checked mode to failure mode only if the current method does not declare the exception in its throws clause.

4. Higher-Order Abstractions and Tunneling
As discussed in Section 2.1, higher-order functions pose a problem for statically checked exception mechanisms. The same problem arises for many common object-oriented design patterns, which are essentially higher-order functions. Our solution is to tunnel exceptions through oblivious code.

For example, Genus allows the programmer to pass to the higher-order function map (Figure 1) an implementation of Function that throws exceptions, even though the signature of Function does not mention any exceptions. If the passed Function throws an exception, that exception is tunnelled through the exception-oblivious code of map to the caller of the exception-aware code that called map.

In the code of Figure 2, the method \( Class::forName \) in class \( Class \) is passed to map. This call is legal even though \( Class::forName \) is declared to throw an exception \( ClassNotFoundException \). Since map is oblivious to this exception, it cannot be expected to handle it. By contrast, the caller of map is aware that an object that throws exceptions is used at a type (Function) that does not declare any exceptions. Because it is aware of the exception, the caller is responsible for the exception.

If \( ClassNotFoundException \) arises at run time, it tunnels through the code of map and is caught by the catch clause. Alternatively, the caller could have explicitly converted the exception to a failure (via a fails clause) or explicitly allowed it to propagate (via a throws clause). In any case, exception handling is enforced statically.

4.1 Exception Tunneling is Safe and Light
In Java, the rigidity of checked exceptions has led to some verbose and dangerous idioms, especially when higher-order functions and design patterns are used. Exception tunneling helps avoid these undesirable programming practices.

In particular, Java programmers often abandon static checking of exceptions to make it possible for their excep-
interface Visitor {
    class IfTree implements Tree {
      void visit(IfTree t);
      void accept(Visitor v) ...
      { v.visit(this); ...
    }
}

class Pretty implements Visitor {
    void visit(IfTree t) {
      try {...} // pretty-print IfTree
      catch (IOException e) { throw new UncheckedIO(e); }
    }
  }

void printTree(Tree t, Pretty v) throws IOException {
  t.accept(v);
}

class UncheckedIO extends RuntimeException {
  ...
}

Figure 3: The pretty-printing visitor in javac (simplified).

Code for exception wrapping and unwrapping is highlighted.

weak interface Visitor {
  class IfTree implements Tree {
    void visit(IfTree t);
    void accept(Visitor v) ...
    { v.visit(this); ...
  }
}

class Pretty implements Visitor {
    void visit(IfTree t) throws IOException {...} // OK
}

1. void printTree(IfTree t, Pretty v) throws IOException {
2.  t.accept(v);
3. }

Figure 4: Exception tunneling in javac, ported to Genus.

4.2 Tunneling Checked Exceptions

Earlier we discussed two modes of exception propagation: checked mode and failure mode. Tunneling mode is a third mode of propagation. The relationships between the three modes are summarized in Figure 5. In tunneling mode, as in checked mode, static checking enforces handling of exceptions. As in failure mode, these exceptions do not need to be declared in signatures of the methods they tunnel through.

A given exception may propagate in more than one mode. Consider the alternative, slightly contrived implementation of map in Figure 6. It calls a helper method mapImpl, which then recursively calls itself to traverse the list. Suppose an exception arises when function f is applied to the sixth element in the list. Figure 6 shows a snapshot of the current call stack. Since f is the place where the exception is generated, the exception first propagates in checked mode within the function f. Because its caller mapImpl is oblivious to the exception, the exception then switches to tunneling mode and propagates through all the mapImpl stack frames. Finally, the caller of map knows about the exception and thus can handle it. The exception returns to checked mode when it reaches this caller. From there, it can be either caught, rethrown in checked mode, or turned into failure.

4.3 Tunneling, Exception Capture, and Blame

Tunneling avoids the phenomenon of exception capture discussed in Section 3. In OO languages like Java and C#, exception capture occurs because of an unexpected collision in
the space of exception identifiers; an exception identifier in these languages is simply the exception type. We avoid exception capture by augmenting the identity of a thrown exception to include a notion of “blame”.

To ensure that every exception is eventually either handled or treated as a failure, a method must discharge every exception it is aware of statically. There are three ways to discharge an exception: 1) handle it with a catch clause, 2) propagate it to its caller as a checked exception via a throws clause, or 3) convert it to a failure via a fails clause (which is implicit for fail-by-default exceptions).

Each of these three ways discharges exceptions from a set of program points that are statically known to give rise to these exceptions. We say these program points can be blamed for the exception. With each such program point, we associate a blame label that identifies where responsibility lies for the exception.

At run time, then, a thrown exception is identified both by its exception type, and by its blame label. An exception is discharged (e.g., caught by a catch clause, or converted to failure by a fails clause), only if the blame label of the exception lies within the lexical scope covered by that particular discharging point. Otherwise, the exception is one that the discharging point is oblivious to.

We use the word “blame” because this mechanism is related to the notion of blame used in work on behavioral contracts [14, 15] and gradual typing [50]. A compiler for a gradually typed language might label program points with unique identifiers wherever there is a mismatch between expected and actual types. When a cast failure happens at run time, blame can be attributed to the program point at fault.

Here, mismatch occurs analogously when the type of an actual argument declares exceptions not encompassed by those declared by the formal parameter type. Any exception mismatch in the parameters passed to a method call causes blame to be assigned to the program point of the method call. However, unlike in gradual typing, exceptions arising from a program point assigned blame do not imply mistakes, since the programmer must discharge the exceptions.

For example, in Figure 2, the program point where Class::forName is used at type Function is in the scope of the ensuing catch clause at line 3. Because it creates a mismatch with the signature of Function, this program point can be blamed for the exception. Similarly, in Figure 4, the shaded program point (line 2) where a Pretty object is used at type Visitor is in the scope of the clause throws IOException (line 1). Because there is a mismatch between Pretty and Visitor, the shaded program point can be blamed for the exception. Throughout the paper, we highlight program points that are assigned blame and their matching discharging points.

Findler et al. [15] use the term “blame” to mean “the programmer should be held accountable for shoddy craftsmanship”. At the risk of confusion, we reuse the term to mean there is an exception to be discharged in this context. 2

4.4 Weak Types
Some supertypes, usually interfaces, abstract across families of otherwise unrelated subtypes. Such interfaces often arise with design patterns like Iterator and Visitor. The intention of such types is to capture only a fragment of the behavior—a core set of methods—so that various implementations can have a common interface other software components can use.

Frequently there is utility in allowing subtypes of these interfaces to throw new exceptions. For example, suppose a lexer breaks input files into tokens; an Iterator might be used to deliver the tokens to code that consumes them. However, reading from a file can cause an IOException; the exception cannot reasonably be handled by the iterator, so it should be propagated to the client code. In Java, programmers cannot throw a checked exception like IOException in such an implementation; they must either resort to unchecked exceptions, abandoning static checking, or define their own interfaces that compose poorly with existing components.

Genus addresses this need for flexibility. Methods that allow their overridings to throw new exceptions are declared with the weak modifier. The weak modifier can also be applied to a type definition to conveniently indicate that all methods in that type are intended to be weak; see the definition of Iterator in Figure 7 for an example.

A subtype of a weak type can be inexact; for example, the Tokenizer class in Figure 7 is inexact with respect to its weak interface since its next method adds IOException. A Tokenizer can be used as an Iterator but this generates blame, forcing IOException to be handled, as in Figure 8.

Behavioral subtyping and conformance. Behavioral subtyping [27] is based on the idea that the allowed uses of an object should be known based on its apparent type. There-
5. Generics and Exceptions

We have also used Genus to explore the important interaction between exceptions and mechanisms for constrained parametric polymorphism. Various languages constrain generic type arguments in various ways: for example, Java and C# use subtyping constraints, whereas Haskell and Genus use the more flexible mechanism of type classes [49].

Genus provides constrained parametric polymorphism via constraints and models [53]. Like type classes, Genus constraints are predicates describing type features required by generic code. Genus models show how types can satisfy constraints, like type class instances in Haskell. Unlike Haskell instances, models are explicitly part of the instantiation of a generic abstraction. For example, the two instantiations Set[String] and Set[String with CaseInsensEq] are different types distinguished by the use of the model CaseInsensEq in place of default string equality. This distinction is helpful for precisely reasoning about exceptions.

As with interfaces, we would like the flexibility to instantiate generic abstractions with types whose operations throw additional exceptions not provided for by the constraint. Thus, similar to interfaces, Genus constraints can be weak; models may be inexact with respect to the weak constraints they witness.

The example in Figure 9 shows the utility of this feature, comparing Java and Genus code for an object pool abstraction. The left side of the figure shows an example adapted from the Apache Commons project [2]. The abstract factory type Factory defines a method make with a signature that declares “throws Exception”. This idiom is common in Java libraries, because it permits subtypes to refine the actual exceptions to be thrown.

However, such declarations are a source of frustration for Java programmers [51]. Consider that method make is called by the method borrow in ObjectPool to create a new object in case there is no idle one, so borrow must also declare “throws Exception”. The precise exception (in the example, SQLException) is therefore lost. Clients of ObjectPool<Connection> must handle Exception, which is no better, and perhaps worse, than having an unchecked exception. Further, client code that handles the overly general exception is more likely to suffer from exception capture.

The right side of Figure 9 shows a reasonable way to implement the example in Genus. A constraint Factory[T] is used to express the requirement that objects of type T can be made in some way; because it does not declare any exceptions, method borrow need not either. However, a model like ConnFactory can add its exceptions such as SQLException to the method make. Client code can then use the type ObjectPool<Connection with ConnFactory> to recycle Connection objects. Because the model is part of the type, it is statically apparent that the exception SQLException may be thrown; the client code will be required to handle this exception—but only this exception.

6. Exactness Analysis

The new exception mechanism poses new challenges for type checking. One challenge is that the identity of an exception includes a blame label, so blame should not be allowed to escape its scope. Otherwise, an exception might not be handled.

For example, consider the first definition of the PeekingIterator class in Figure 10. It decorates a wrapped Iterator in field inner to support one-element lookahead. Its next and peek methods call inner.next(). Per Section 4.4, it is possible to pass an object of some inexact subtype of Iterator to the constructor, to be assigned to inner. However, Iterator is a weak type, so the methods of the actual object being passed may throw exceptions not declared by Iterator. If the assignment to inner were allowed, the same exceptions would propagate when the next method of the PeekingIterator object were called. And this call could be delayed until outside the context that is aware of the mismatch with Iterator. In this case, the exceptions would not be guaranteed to be handled.

Therefore, we want to detect statically that the assignment to inner lets the blame from the inexact object escape. Storing an inexact object into a data structure at a weak type, or even returning an inexact object from a method, may permit such an escape of blame.

A second challenge for the new exception mechanism, and indeed for exception mechanisms generally, is that pro-
1 class PeekingIterator[E] implements Iterator[E] {
2     Iterator[E] inner;
3     PeekingIterator(Iterator[E] it) { inner = it; ... }
4     E peek() throws NoSuchElementException
5         { ... inner.next() ... }
6     ...
7 }
8 class PeekingIterator[I extends Iterator[E], E]
9     implements Iterator[E] {
10     I inner;
11     PeekingIterator(I it) { inner = it; ... }
12     E peek() throws NoSuchElementException
13         { ... inner.next() ... }
14     ...
15 }

Figure 10: Two definitions of PeekingIterator. The top one allows blame to escape, but a warning is issued for the constructor. The bottom one uses dependent exactness to soundly avoid the warning.

Programmers should not be forced to write handlers for program points where exceptions cannot happen. To address this challenge, the location of blame should be precise. Figure 8 offers an example of this problem. The method call iter.next() might appear (to the compiler) not to throw any exceptions because iter has type Iterator[Token], yet we know that it may throw a IOException because iter is initialized to a Tokenizer. A safe, conservative solution would be to require all of the code below this initialization to be wrapped in a try–catch. But this solution would make it difficult to continue the iteration after an IOException is raised.

Genus addresses these two challenges using an intraprocedural program analysis that assigns exactness to uses of weak types, with little annotation effort by the programmer.

### 6.1 Exactness Annotations and Exactness Defaults

**Indicating intolerance of inexactness at use sites.** When a formal parameter or local variable is assigned a weak type, the programmer can annotate the type with @exact to indicate that an exact subtype must be used. For example, if the programmer adds @exact to the formal parameter of the constructor at line 3 in Figure 10, it becomes a static error to pass an inexact implementation of Iterator to the constructor. So it is guaranteed that using inner will not generate unexpected exceptions. By contrast, without @exact, Genus issues a warning about the assignment to the escaping pointer inner at line 3. If the exception is actually thrown at run time, it is converted into failure and the usual stack trace is collected.

Although @exact might appear to add notational burden, our experience with porting existing Java code into Genus (Section 9) suggests that this escape hatch is rarely needed. We have not seen the need to use the @exact annotation to dismiss such warnings in existing code ported from Java. It seems that programmers use weak types differently from other types—weak types provide functionality rather than structure. Further, exactness defaults and exactness inference (Section 6.2) reduce annotation overhead, and exactness-dependent types (Section 6.3) provide more expressiveness.

**Exactness defaults.** Exactness defines what exceptions that are not declared by the weak type might nevertheless be generated by the term being typed. Exactness $E$ is formally a mapping from methods to sets of exceptions. These mappings form a lattice ordered as follows:

$$E_1 \leq E_2 \iff \text{dom}(E_1) \subseteq \text{dom}(E_2) \land \forall m. E_1(m) \subseteq E_2(m)$$

The bottom lattice element is strict exactness, denoted by $\emptyset$.

To avoid the need for programmers to write the annotation @exact for most uses of weak types, the compiler determines exactness using a combination of exactness defaults and automatic exactness inference. To see how these mechanisms work, consider the code in Figure 11.

1. Weak types used as return types or field types are exact, as these are the channels through which pointers can escape. For these types, we have $E = \emptyset$.
2. Methods and constructors are implicitly polymorphic with respect to exactness. That is, they can be viewed as parameterized by the exactness of their argument and receiver types.
3. Weak types in a local context are labeled by exactness variables: for example, $x$ and $y$ in Figure 11, at lines 2 and 11 respectively.

A unification-based inference engine solves for these variables, inferring exactness from the local variable uses (Section 6.2).

4. For a procedure call, an exactness variable is generated for each argument and/or receiver whose formal parameter type is weak. Exactness variables of this kind in Figure 11 are $z$ (line 9) and $w$ (line 12).

5. Recall that Genus supports constrained parametric polymorphism via type constraints [53]. Subtype constraints and where-clause constraints can also specify exactness. Their default exactness is deduced in ways similar to those listed above. For example, in Figure 11 the use of Runnable in $g$’s signature (line 1) constrains the type parameter $T$, so its exactness is resolved to a fresh name $e$, with respect to which $g$ is polymorphic.

### 6.2 Solving Exactness Constraints

At each assignment (including variable initialization and argument passing) to a variable with weak type $\tau$, inexact values must not escape into variables with exact types. Therefore, constraints of form $\tau(E_\tau) \leq \tau(E)$ are generated, where $E_\tau$ is the exactness of the left-hand side expected type $\tau$ in an assignment, resolved as described in Section 6.1, and $E$ is the exactness of the right-hand side actual type with respect
weak interface Runnable { void run(); }

1 void g[T extends Runnable] (List[T] l) {
2     Runnable(r) r0;
3     if (new Random().nextBoolean())
4         r0 = new Runnable()
5         { void run() throws IOException {...} };
6     else
7         r0 = new Runnable()
8         { void run() throws EOFException {...} };
9     try { r0.run(); } // Runnable(r)
10    catch (IOException ex) {...}
11    catch (NoSuchElementException ex) { break; }
12    r.run(); // Runnable(w)
13 }

Figure 11: Running example for exactness analysis. Uses of weak types are tagged by exactness variables, to be fed into the solver. (EOFException is a subtype of IOException.)

Figure 12 defines both a surface and a kernel syntax for CBC. Programs are written in the surface syntax, and rewritten to and evaluated in the kernel calculus. Applications in surface terms are tagged by unique blame labels (ranged over by ℓ) representing lexical positions where blame can arise. Rewriting propagates these labels from the surface language to the kernel language, for blame tracking during kernel evaluation.

The surface syntax assumes a fixed set of exception names ranged over by E. The kernel syntax allows them to be labeled to form new names; E and E^ℓ are different names.

Surface types (τ) include weak types and strong types. A strong type (σ) is either the base type B or a function type τ → σ[σ]_E^ℓ that does not allow mismatch against E. (An overline denotes a (possibly empty) set.) A weak type τ → σ[σ]_E^ℓ, on the other hand, tolerates mismatch. Notice that function return types must be strong to prevent blame from propagating from the return value of a method.
escaping. The same can be said about kernel types. Since there is an obvious injection (syntactic identity) from surface types to kernel types, we abuse notation by using \( \tau \) and \( \sigma \) in places where kernel types \( T \) and \( S \) are expected.

Environments \( \Gamma \) contain mappings from variables to their types and exactness. Exactness is represented by sets of exceptions. Strong types enforce strict exactness, so their exactness is represented by \( \emptyset \). The auxiliary function \( \mathcal{K}(\cdot) \) returns \( \perp \) for strong types and \( \top \) for weak types; values of strong types must not escape. \( \Gamma_{\perp} \) retains only the strongly-typed variables in \( \Gamma \).

7.2 Semantics

\textbf{Surface-to-kernel rewriting.} Figure 13 defines the rewriting rules. The judgment \( \Gamma; K \vdash f \rightsquigarrow e : [\tau]_E \) translates the surface term \( f \) to the kernel term \( e \), assigns \( f \) the type \( \tau \), and infers the exceptions \( E \) that evaluating \( e \) might raise.

Typing is dependent on \( K \), which indicates whether the term in question is guaranteed not to escape. For example, the left-hand-side term in an application is type-checked with \( K = \top \) as in R-App-E and R-App-I, while the body of an abstraction is type-checked with \( K = \perp \) as in R-Abs. R-Var-W denies first-class citizenship to weakly typed variables, and augments the return type if the environment indicates inexactness.

Exception mismatch is computed using the subtyping relation \( \preceq_T \), as in R-App-I and R-Let. But only with R-App-I can blame take effect, and its translation is consequently less obvious. To avoid exception capture, we need to give new names to exceptions involved in the mismatch. Therefore, the translation wraps the argument in an abstraction, which, when applied, catches any exceptions in the mismatch (\( E \)) and rethrows their labeled versions (\( E^\ell \)). The caller is aware of the exceptions \( E \), so it catches the labeled exceptions tunneled to its context and strips off their labels.

\textbf{Semantics of the kernel calculus.} The static and dynamic semantics of the kernel are omitted here for lack of space but can be found in the technical report [54]. The static semantics is largely similar to the typing induced by rewriting, except that the kernel need not worry about exception capture. Hence, exception propagation happens in the usual way.

7.3 Type Safety

The type system guarantees that if a program can be typed without exceptional effects, it cannot get stuck when evaluated, or terminate in an exception. This guarantee easily follows from two other standard results.

First of all, the kernel type system is sound:

\textbf{Theorem 1 (Kernel soundness: preservation and progress)}

\[ - \text{If } K \vdash e : [T]_\mathcal{P} \text{ and } e \rightarrow e' \text{ then } K \vdash e' : [T]_\mathcal{P}. \]

\[ - \text{If } K \vdash e : [T]_\mathcal{P} \text{ then either} \]

\[ 1. \exists v, e = v, \text{ or } \]

\[ 2. \exists U_0 \in \mathcal{U}. e = \text{throw } U_0, \text{ or } \]

\[ 3. e \rightarrow e'. \]

\textbf{Proof.} This is proved in the usual way, by induction on the kernel typing derivation. See the technical report for details of this proof and of other formal results in this paper.

Second, the translation from the surface language to the kernel language is type-preserving:

\textbf{Lemma 1 (Rewriting preserves types)}

\[ \text{If } \Gamma; K \vdash f \rightsquigarrow e : [\tau]_E \text{ then } \Gamma; K \vdash e : [\tau]_E. \]

\textbf{Proof.} By induction on the derivation of the translation.

The guarantee that well-typed programs handle their exceptions is a direct corollary of these two previous results:

\textbf{Corollary 1 (No uncaught exceptions)}

\[ \text{If } \emptyset; \perp \vdash f \rightsquigarrow e : [\tau]_E \text{ then } \exists v, e \rightarrow^* v. \]

8. Implementation

We have implemented the new exception mechanism for the Genus programming language. The implementation consists of about 5,800 lines of code, extending the compiler for the base Genus language [53]. Genus is implemented using Polyglot [35], so code generation works by translating to Java code, using a Java compiler as a back end.

The rest of this section focuses on the translation into Java. The translation is guided by two goals: 1) it should prevent accidental capturing of exceptions, and 2) it should add negligible performance overhead to normal control flow.

8.1 Representing Non-Checked Exceptions

Unlike exceptions in checked mode, exceptions traveling in tunneling mode or failure mode must acquire new identities to avoid accidental capturing. These identities are implemented by wrapping exceptions into objects of classes Blame and Failure. Both classes extend RuntimeException, but Blame does not collect a stack trace.
\[
\begin{align*}
\Gamma; K \vdash f \rightsquigarrow e : [\tau]_E & \quad [\text{R-Const}] & \Gamma; K \vdash b \rightsquigarrow b : [B]_B & \quad [\text{R-Vars}] & \Gamma; K \vdash x \rightsquigarrow x : [\sigma]_E & \quad [\text{R-Var-s}] \\
\Gamma; K \vdash x : \tau & \quad \vdash e : [\sigma]_E & \quad [\text{R-Abs}] & & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma; K \vdash \text{Iterator}\langle E \rangle & \quad [\text{R-Var-w}] & \Gamma; K \vdash x \rightsquigarrow x : [\tau]_E & \quad [\text{R-Let}] & \Gamma; K \vdash f \rightsquigarrow e : [\sigma]_E & \quad [\text{R-App-e}] & \Gamma; K \vdash \{ f, g \} & \rightsquigarrow e_1 : \tau_1 \quad \Gamma; K \vdash \text{try} f_1 \text{ catch } e \text{ throw } E & \quad [\text{R-Try-Catch}] & \Gamma; K \vdash f \rightsquigarrow e : [\sigma]_E & \quad [\text{R-Throw}] & \\
\end{align*}
\]

\[
\begin{align*}
\tau_2 \subseteq \tau_1 & \quad \text{SS-B} & \quad B \subseteq B & \quad \text{SS-S} & \tau_2 \subseteq \tau_1 & \quad \tau_1 \subseteq \tau_2 & \quad \text{SS-W} \\
\end{align*}
\]

Figure 13: Surface-to-kernel rewriting

### 8.2 Translating Exception-Oblivious Code

Methods with weakly typed parameters ignore extra exceptions generated by them. To ensure they are handled in the appropriate context, weakly typed parameters, including the receiver, are accompanied by an additional Blame argument that serves as the blame label. When the actual argument is exact, the Blame argument is null. For example, the weak type Iterator has the following translation:

```java
interface Iterator{E} {
    E next$Iterator(Blame b$) throws NoSuchElementException;
    ...
}
```

It receives a Blame object from its caller to accompany the weakly typed receiver. If an implementation of Iterator throws a mismatched exception, it is wrapped in this Blame object and tunneled through the code oblivious to it.

Exception-oblivious procedures are translated in a similar way. For example, the code generated for map (Figure 1) looks like the following:

```java
<T,R> List<R> map(Function<T,R> f, List<T> src, Blame b$) {
    ... f.apply$Function(t, b$) ...
}
```

The extra Blame argument b$ is intended for potential mismatch in the argument function f, and is passed down to the method call f.apply$Function(...) so that exceptions from f have the right blame label.

### 8.3 Translating Exception-Aware Code

The definition of Iterator's inexact subtype Tokenizer is exception-aware. Its translation is shown in the left of Figure 14. Per Java's type-checking rules, the overriding method in Tokenizer cannot throw extra checked exceptions. Instead, the overriding method next$Iterator(Blame) redirects to method next() that does the real work, possibly throwing an IOException, and then turns that exception into either a Blame or a Failure, which is unchecked. If the Blame argument is not null, there must be a program point ready to handle the exception. So the IOException is wrapped in the Blame object and is tunneled to that program point. If the Blame object is null, a Failure object wrapping the IOException is created and thrown. This might happen, for example, if the programmer chose to disregard the compiler warning reported for passing a Tokenizer (Figure 7) into the constructor of PeekingIterator (top of Figure 10).

The code in Figure 8 is also exception-aware, and Figure 14 (right) shows its translation. Instead of creating a new Blame object every time a mismatch happens, each thread maintains a Blame object pool that recycles Blame objects. A Blame object is borrowed from the pool at the blamable pro-
class Tokenizer implements Iterator<Token> {
    Token next$Iterator(Blame b$) throws NoSuchElementException {
        try {
            return next();
        } catch (IOException e) {
            if (b$ != null) { b$.inner = e; throw b$; }
            else throw new Failure(e);
        }
    }
    Token next() throws IOException, NoSuchElementException {...}
    ...
}

Blame b$ = null;
try {
    Iterator<Token> iter = new Tokenizer(reader);
    while (true) {
        Token t;
        try { t = iter.next$Iterator(b$=Thread.borrowB$()); }
        catch (NoSuchElementException e) {
            break;
        }
        catch (Blame bCaught$) {
            if (bCaught$ == b$) { log.write(...); continue; }
            else throw bCaught$;
        }
    }
} finally { Thread.handbackB$(b$); }

Figure 14: Translating exception-aware code (Tokenizer from Figure 7 and its using code from Figure 8) into Java

Aggressive interweaving of try–catch in method bodies might preclude certain compiler optimizations. Therefore, the translation uses a simple program analysis to identify existing enclosing try blocks onto which these new catch blocks can be attached.

8.4 Translating Failure Exceptions
A method body is wrapped in a try block whose corresponding catch block catches all exceptions that will switch to failure mode after exiting the method. A catch all block is translated into possibly multiple catch blocks to also catch compatible Failure and Blame objects.

9. Evaluation
The aim of our evaluation is to explore the expressiveness of the new exception mechanism, and its overhead with respect to both performance and notational burden.

9.1 Porting Java Code to Use Genus Exceptions
To evaluate the expressive power of the new exception mechanism, we ported various existing Java programs and libraries into Genus. Some of this code (ObjectPool and PeekingIterator) is described earlier, but we examined some larger code samples:

- We ported the Java Collections Framework and found that no @exact annotations were needed. In addition, the Genus compiler found unreachable code in AbstractSequentialList, thanks to fail-by-default exceptions propagating in checked mode (Section 3.3).
- We ported the javac visitor of Figure 3 into Genus, as mentioned earlier in Section 4.1. Conversion to the new exception mechanism allows more than 200 lines of code to be removed from class Pretty (~1,000 LOC), and more importantly, restores static checking.

• Using only checked exceptions, we managed to reimplement the EasyIO text parsing package that was developed for a Cornell programming course. This codebase (~1,000 LOC) uses exceptions heavily for backtracking.

9.2 Performance
The current Genus implementation targets Java. We explored its performance through several experiments. All data were collected using Java 8 on a 3.4GHz Intel Core i7 processor after warming up the HotSpot JVM.

Performance of normal-case code. Perhaps the most important performance consideration for an exception mechanism is whether it slows down normal-case code. To evaluate this, we ported Java code that only uses exceptions lightly—specifically, the JOlden benchmarks [9] and, representing larger applications, the SunFlow benchmark [45] from the DaCapo suite [4]. SunFlow is a ray tracer containing ~200 classes and ~21K LOC.

To evaluate the overhead of the new exception mechanism fairly, we compared the running times of this code between the extended Genus language (Section 8) and the base Genus language. Despite support for reified generics, the performance under base Genus is close to Java: compared to Java, it incurred a slowdown of 0.3%.

Figure 15: Performance of the exception mechanism on the JOlden benchmarks and SunFlow.

Figure 15:

3 www.cs.cornell.edu/courses/cs2112/2015fa/#Libraries
Table 1: Performance with EasyIO (s)

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Exception objects</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java exceptions</td>
<td>new instance</td>
<td>817.7</td>
</tr>
<tr>
<td></td>
<td>cached</td>
<td>124.7</td>
</tr>
<tr>
<td>Java unchecked wrappers</td>
<td>cached</td>
<td>826.0</td>
</tr>
<tr>
<td>Genus exceptions (tunneled)</td>
<td>new instance</td>
<td>139.8</td>
</tr>
<tr>
<td></td>
<td>cached</td>
<td>128.6</td>
</tr>
</tbody>
</table>

Table 2: Exception tunneling microbenchmarks

Figure 15 reports the results of this comparison. Each reported measurement averages at least 20 runs, with a standard error less than 1.2%. Benchmark parameters were set so that each run took more than 30 seconds.

Overall, the extended compiler generates slightly faster code than the original compiler. The average speedup is 2.4%, though performance varies by benchmark. The speedup is caused largely by the exception-based translation of enhanced for loops.

Performance of exception-heavy code. To evaluate the performance improvement that Genus obtains by avoiding stack-trace collection on realistic code, we measured the running times of pattern-matching regular expressions using the EasyIO package. It makes heavy use of exceptions to control search. The running times are shown in Table 1. Each number averages 10 runs, with 6.2M exceptions thrown in each run.

Stack-trace collection in Java causes more than 6× overhead compared to Genus exceptions. Genus targets Java, so it is not surprising that similar performance can be achieved with Java, if stack-trace collection is turned off and exception objects are cached.

Exception tunneling performance. We used a microbenchmark to explore the overhead of exception tunneling, the key difference between Genus and Java. The microbenchmark performs method calls on objects passed down from a higher-order function; the method call either throws an exception or returns immediately, and is prevented from being inlined. Since there is no other work done by the method, performance differences are magnified.

The results are shown in Table 2. We compare exception tunneling in Genus to two (unsafe) Java workarounds: typical Java exceptions and unchecked-exception wrappers. We also refine the comparison for typical Java exceptions and tunnelled Genus exceptions based on whether a new exception object is created for each throw; throwing a single cached instance is reasonable for non-failure exceptions carrying no extra information. Each number averages 20 runs, with a standard error less than 0.6% of the mean.

The rightmost column measures time to exit via an exception. Genus exceptions perform well because they do not collect a stack trace. The slowdown compared to the second row is mostly because exception mismatch requires borrow and handback calls (Section 8.3) in every loop iteration.

On the other hand, Genus exceptions significantly outperform unchecked wrappers, the safest way to tunnel in Java. Java’s performance is poor here because an unchecked-exception object is created for each raised exception, whereas the implementation of Genus recycles Blame objects.

The microbenchmark also measures the time to return from a method normally. The average cost of a call and return in Java was 6.0 ns. In the absence of mismatch, our translation adds the overhead of passing a null pointer to the normal return path, increasing the cost slightly to 6.3 ns. The results in Figure 15 suggest this increase is negligible in practice.

10. Related Work

Notable approaches to exceptions. PL/I was the first language with exceptions. It supports user-defined exceptions with exception handlers dynamically bound to exceptions [28]. Goodenough [19] and Liskov and Snyder [26] introduced statically scoped handlers. CLU [26] was the first language to support some static checking of exceptions. Exceptions are declared in function signatures, and thrown exceptions must appear in these signatures. If not explicitly resignaled, propagating exceptions automatically convert to failure. Mesa [33] supports both termination- and resumption-style exceptions but does not check them statically. Ada [1] attaches handlers to blocks, procedures, or packages. Unhandled exceptions propagate automatically, but exceptions are not declared. Eiffel [30] exceptions originate from the violation of assertions and are raised implicitly. Upon exceptions, programmers can retry the execution with different parameters; otherwise, exceptions implicitly propagate to callers. Modula-3 [34] introduced fully statically checked exceptions. Black [3] and Garcia et al. [16] present comparative studies of some exception mechanisms.

Empirical findings. An empirical study by Cabral and Marques [7] shows that in Java and .NET exception handlers are not specialized enough to allow effective handling, which we believe is partly attributable to a lack of documentation of exceptions in the .NET case [8] and the rigidity of checked exceptions in the Java case. Robillard and Murphy [40] identify the global reasoning required of programmers as a major reason why exceptions are hard to use.

Exception analysis. Function types in functional languages such as ML and Haskell do not include exceptions because they would interfere with the use of higher-order functions. Exception capture can be avoided in SML [32] because exception types are generative, but other variants of ML lack this feature. Leroy and Pessaux [25] observe that uncaught exceptions are the most common failure mode of large ML applications, motivating them and others [12] to develop program analyses to infer exceptions. Such analyses can be helpful, especially for usage studies [52], but they necessarily involve trading off performance and precision, and entail non-
local reasoning that does not aid programmers in reasoning about their code. A benefit of our mechanism is that it is likely to lead to more accurate, scalable static analyses, because precise exceptions largely remove the need to approximate exceptional control flow [5, 42].

**Exception polymorphism.** Some recent designs attempt to address the rigidity of checked exceptions through exception polymorphism: anchored exceptions [47] as a Java extension, polymorphic effects [41] as a Scala extension, row-polymorphic effect types in the Koka language [24], and the rethrows clause introduced in Swift 2 [46]. These approaches add annotation burden to exception-oblivious code; more fundamentally, they do not address exception capture.

**Blame tracking.** Our notion of blame is related to that introduced by work on contracts [14] and further developed by work on gradual typing [50], in which blame indicates where fault lies in the event of a contract violation (or cast failure). In our setting, an exception mismatch results in static blame being assigned that indicates where “fault” lies should an exception arise at run time, with the compiler statically checking that the “faulty” program point handles exceptions.

Blame has polarity [14, 50]; in our setting, exception mismatch at covariant (or contravariant) positions gives rise to positive (or negative) blame. Existing mechanisms for checked exceptions only consider positive blame; exceptions with negative blame are the missing piece. By contrast, in Genus both kinds of exceptions are subject to static checking, and our implementation and formalization manifest their difference: exceptions with negative blame acquire new identities to achieve safe tunneling.

**Type inference.** Our exactness inference bears resemblance to work on gradual type inference [39]. Both inference algorithms encode escape analysis “for free”, although the motivation for escape analysis differs. Our approach to default exactness (Section 6.1) is similar to default region annotations in a region-based type system [21].

11. Conclusions

Our new exception mechanism combines the benefits of static checking with the flexibility of unchecked exceptions. We were guided in the design of this mechanism by thinking carefully about the goals of an exception mechanism, by much previous work, and by many discussions found online. Our formal results and experience suggest that our approach improves assurance that exceptions are handled. The evaluation shows that the mechanism works well on real code. It adds negligible cost when exceptions are not being used; exception tunneling comes with a small performance penalty that appears to be more than offset in practice by avoiding the run-time overhead of wrapping exceptions. We hope this work helps programmers use exceptions in a principled way and gives language implementers an incentive to make exceptions more efficient.

**Acknowledgments**

We thank Owen Arden, Chinawat Isradisaikul, Jed Liu, Michael Hicks, and the anonymous reviewers for discussions about exception mechanisms and for suggestions about this paper. This research has been supported by MURI grant FA9550-12-1-0400, administered by the US Air Force, by NSF grant 1513797, and by the European Research Council, grant No. 321217. This paper does not necessarily reflect the views of any of its sponsors.

**References**
