Abstraction-Safe Effect Handlers via Tunneling

Yizhou Zhang
Cornell University
Exception handling

```java
try {
    ... // normal-case code
} catch (SomeException e) {
    ... // exception-handling code
}
```

Separation of concerns
Programs **crash** when programmers forget to handle exceptions

- Statically checking exceptions
To check, or not to check, that is the question…

I quite like checked exceptions. They’re actually one of the few things that keeps me writing java for more projects instead of switching more of my time to golang.

Checked exceptions are great, but unchecked exceptions are a huge disaster only mitigated by

Merely that there's a case for checked exceptions, especially if you are actually trying to write "reliable" software.

4. Unchecked exceptions are a production run-time nightmare. My anecdotal but extensive experience is that

---

**Checked exceptions I love you, but you have to go**
39 points | fogus | 9 years ago | 35 comments

55 points | tosh | 5 months ago | 86 comments | (https://blog.phillipauer.de/chec)

Java's [checked exceptions were a mistake](https://www.researchgate.net/profile/alexk94/publication/267489585_Checked_exceptions_were_a_mistake/links/0f355b6e0cf7f8c67a000000/checked-exceptions-were-a-mistake.pdf) (2003)
2 points | alexk | 9 years ago | 0 comments

**Checked exceptions are indeed obnoxious and a major language design failure**, which is why pretty much every language has [optional checks](https://www.researchgate.net/profile/alexk94/publication/267489585_Checked_exceptions_were_a_mistake/links/0f355b6e0cf7f8c67a000000/checked-exceptions-were-a-mistake.pdf), which is why pretty much every language has [optional checks](https://www.researchgate.net/profile/alexk94/publication/267489585_Checked_exceptions_were_a_mistake/links/0f355b6e0cf7f8c67a000000/checked-exceptions-were-a-mistake.pdf). The big problem with Java checked exceptions was that the language was simply not flexible enough in other areas to handle higher-order stuff. For
To check, or not to check, that is the question…

Checked exceptions maketh software more **reliable**!

I quite like **checked exceptions**. They're actually one of the few things that keeps me writing java for more projects instead of switching more of my time to golang.

**Checked exceptions are great**, but **unchecked exceptions are a huge disaster** only mitigated by

Merely that there's a **case for checked exceptions**, especially if you are actually trying to write "**reliable**" software.

4. **Unchecked exceptions are a production run-time nightmare**. My anecdotal but extensive experience is that

---

The big problem with that the language in other areas
To check, or not to check, that is the question…

Checked exceptions are too **rigid** to work with higher-order programming!

The big problem with Java checked exceptions was that the language was simply **not flexible enough** in other areas to handle higher-order stuff. For
To check, or not to check, that is the question…

Checked exceptions are too **rigid** to work with higher-order programming!

---

**odersky**

Re: Re: Add compiler warning for checked exception

The problem with checked exceptions is best demonstrated by the map method on lists:

```scala
def map[B](f: A => B): List[B]
```

How to annotate `map` with `@throws`? If `map` does not get a `@throws` annotation itself then presumably you cannot pass it any function that has a `@throws`. That would introduce cumbersome restrictions and distinctions for the ways in which `map` can be used. Things would be...
To check, or not to check, that is the question...

Checked exceptions are too **rigid** to work with higher-order programming!

A higher-order function in Java

```
<X,Y> List<Y> map(List<X> l, X → Y f) {...}
```

Raises `IOException` if the file does not exist

```
Tree parseFile(File file) throws IOException {...}
```
Checked exceptions are too rigid

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> l, X→Y f) {...}
```

Raises IOException if the file does not exist

```java
Tree parseFile(File file) throws IOException {...}
```

Client code

```java
List<File> files = ...;
List<Tree> trees;
try {
    trees = map(files, f->parseFile(f));
} catch (IOException e) {
    ...
}
```

Compile-time error
Checked exceptions are too rigid

A checked exception class in Java

```java
class IOException extends Exception {...}
```

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> l, X→Y f) {...}
```

Raises IOException if the file does not exist

```java
Tree parseFile(File file) throws IOException {...}
```

Client code

```java
List<File> files = ...;  
List<Tree> trees;  
try {
    trees = map(files, f→parseFile(f));
    ...
} catch (IOException e) {
    ...
}
```

Compile-time error
Checked exceptions are too rigid

A checked exception class in Java

```java
class IOException extends Exception {...}
```

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> l, X → Y f)
```

Raises IOException if the file does not exist

```java
Tree parseFile(File file) throws IOException {...}
```

Client code

```java
List<File> files = ...;
List<Tree> trees;
try {
    trees = map(files, f->parseFile(f));
    ...
} catch (IOException e) {
    ...
}
```

Compile-time error: exception mismatch

Lambda-expression has type

```
File → Tree throws IOException
```

but a function of type

```
File → Tree
```

is expected.
Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```java
class UncheckedIOException extends RuntimeException {...}
```

A higher-order function is introduced

```java
<X,Y> List<Y> map(List<X> l, X → Y f) {...}
```

 Raises IOException if the file does not exist

```java
Tree parseFile(File file) throws IOException {...}
```

Client code

```java
List<File> files = ...;
List<Tree> trees;
try {
    trees = map(files, f->parseFile(f));
    ...
} catch (IOException e) {
    ...
}
```

**Compile-time error: exception mismatch**

Lambda-expression has type

File → Tree throws IOException

but a function of type

File → Tree

is expected.
Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```java
class UncheckedIOException extends RuntimeException {...}
```

```java
List<File> files = ...;
List<Tree> trees;
trees = map(files, f->parseFile(f));
...```

List<File> files = ...;
List<Tree> trees;
trees = map(files, f->{
    try {
        return parseFile(f); }
    catch (IOException e) {
        throw new UncheckedIOException(e);
    }
});
Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

```java
class UncheckedIOException extends RuntimeException {
    //...
}
```

```java
List<File> files = ...;
List<Tree> trees;
try {
    trees = map(files, f->{
        try {
            return parseFile(f);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        }
    });
    ...
} catch (UncheckedIOException u) {
    IOException e = u.getCause();
    ...
}
```
Pattern: Use unchecked exception wrappers

An unchecked wrapper class introduced in Java 8

class UncheckedIOException extends RuntimeException {...}

List<File> files = ...;
List<Tree> trees;
try {
    trees = map(files, f->{
        try {
            return parseFile(f);
        } catch (IOException e) {
            throw new UncheckedIOException(e);
        };
    });
}

... catch (UncheckedIOException u) {
    IOException e = u.getCause();
    ...
}
List<File> files = ...
List<Tree> trees;
try {
trees = map(files, f->{
    try {
        return parseFile(f);
    }
    catch (IOException e) {
        throw new UncheckedIOException(e);
    }
});
...} catch (UncheckedIOException u) {
    IOException e = u.getCause();
    ...
}
The tragedy of exceptions
The tragedy of exceptions

“More thinking is needed before we put some kind of checked exceptions mechanism in place.”
—Anders Hejlsberg—

D’oh!
The tragedy of exceptions

“More thinking is needed before we put some kind of checked exceptions mechanism in place.”
—Anders Hejlsberg—

“Trying is the first step toward failure.”
The tragedy of exceptions

Unchecked exceptions lead to uncaught exceptions.

Checked exceptions do not work with higher-order abstractions.

Use patterns!

Anti-pattern subverts static checking.

Unchecked exceptions can be handled by accident.
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NoSuchElementException extends RuntimeException {...}
```

Raises NoSuchElementException if there is no next element

```java
interface Iterator<X> {
    X next() throws NoSuchElementException;
    boolean hasNext();
}
```
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

Raises NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    boolean hasNext();
}
```
Programming task:
Transform a `List<Iterator<Int>>` into `List<Int>` by fetching the first `Int` from each `Iterator<Int>`.

Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

Raises NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    boolean hasNext();
}
```

Unchecked exceptions can be handled by accident
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

A higher-order function

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {...}
```

Raised NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

Programming task:

Transform a `List<Iterator<Int>>` into `List<Int>` by fetching the first `Int` from each `Iterator<Int>`. 
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

A higher-order function

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {...}
```

Raised NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}

Unchecked exceptions can be handled by accident
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

A higher-order function

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {...}
```

List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}

Raises NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

Unchecked exceptions can be handled by accident
Unchecked exceptions can be handled by accident

An unchecked exception

```
class NSE extends RuntimeException {...}
```

A higher-order function

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {...}
```

List<Iterator<Int>> iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {...}

Raises NSE if there is no next element

```
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

```
try … catch
  map
  it->it.next()
```

Unchecked exceptions can be handled by accident
Unchecked exceptions can be handled by accident

An unchecked exception

```java
class NSE extends RuntimeException {...}
```

A higher-order function

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {...}
```

List<Iterator<Int>> iters = ...;
try {
   var ints = map(iters, it->it.next());
   ...
} catch (NSE e) {...}

Raises NSE if there is no next element

```java
interface Iterator<X> {
   X next() throws NSE;
   ...
}
```

`<X,Y> List<Y> map(List<X> xs, X→Y f)` raises NSE if there is no next element.

Unchecked exceptions can be handled by accident.
Unchecked exceptions can be handled **by accident**

A higher-order function in Java

```
public static <X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        ys.add(f(it.next()));
    }
    return ys;
}
```

Client code

```
List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

Unchecked exceptions can be handled **by accident**.
Unchecked exceptions can be handled by accident

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {
  List<Y> ys = new ArrayList<Y>();
  Iterator<X> it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); } 
    catch (NSE e) { break; } 
  }
  return ys;
}
```

Client code

```java
List<Iterator<Int>> iters = ...;
try {
  var ints = map(iters, it->it.next());
  ...
} catch (NSE e) {...}
```

Raises NSE if there is no next element
```
interface Iterator<X> {
  X next() throws NSE;
  ...
}
```

Unchecked exceptions can be handled by accident
```java
it->it.next()
```
Unchecked exceptions can be handled by accident

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) {
            break;
        }
    }
    return ys;
}
```

Client code

```java
List<Iterator<Int>> iters = ...
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```

Raises NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

NOT TRUE

Unchecked exceptions can be handled by accident.
Unchecked exceptions can be handled by accident

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```java
List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it→it.next());
    ...;
} catch (NSE e) {...}
```
Unchecked exceptions can be handled by accident

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```
List<Iterator<Int>> iters = …;
try {
    var ints = map(iters, it→it.next());
    …
} catch (NSE e) {…}
```
A higher-order function in Java

```
<X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next())); }  
        catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```
List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```

Raises NSE if there is no next element

```
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

Unchecked exceptions can be handled by accident

Handler intercepts exceptions raised by f!
Accidentally handled exceptions are real

An `IndexOutOfBoundsException` is accidentally handled by `java.util.AbstractList`
To check, or not to check, that is the question...

Checked exception types

IOException
SQLException
DataFormatException
 ...

Checked exceptions are too rigid.
Unchecked exceptions lead to uncaught and accidentally caught exceptions.

Checked exceptions are too rigid.
Unchecked exceptions lead to uncaught and accidentally caught exceptions.

Checked exceptions types
- IOException
- SQLException
- DataFormatException

Unchecked exception types
- UncheckedIOException
- NoSuchElementException
- NumberFormatException

To check, or not to check, that is the question...

Checked exceptions are too rigid.
To check, or not to check, that depends on the context the exception is passing through.
To check, or not to check, that depends on the context the exception is passing through.

Exceptions are checked in contexts aware of them (e.g., the caller of `map`).

Exceptions are “unchecked” in contexts oblivious to them (e.g., the definition of `map`).
To check, or not to check, that depends on the context the exception is passing through.

Exceptions are checked in contexts aware of them (e.g., the caller of `map`).

Exceptions are “unchecked” in contexts oblivious to them (e.g., the definition of `map`).

Guarantees

Exceptions are always caught, but never accidentally caught.
An object-oriented language with powerful generics [Zhang & al. 2015]

Syntax is close to Java, but uses [] instead of <>

The new exception mechanism can be applied to many other languages

Guarantees

Exceptions are always caught, but never accidentally caught.
Recall this Java program doesn’t type-check…

A higher-order function in Java

\[
<X, Y> \text{ List}\langle Y \rangle \text{ map(List}\langle X \rangle \text{ l, } X \rightarrow Y \text{ f) } \{\ldots\}
\]

Raises IOException if the file does not exist

```
Tree parseFile(String filename)
throws IOException {\ldots}
```

Client code

```
List<File> files = \ldots;
List<Tree> trees;
try {
    trees = map(files, f->parseFile(f)),
    \ldots
} catch (IOException e) {\ldots}
```

Compile-time error: exception mismatch

Lambda-expression has type

`File \rightarrow Tree throws IOException`

but a function of type

`File \rightarrow Tree` is expected.
This Genus program type-checks!

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```java
Tree parseFile(String filename) throws IOException {...}
```

Client code

```java
List[File] files = ...;
List[Tree] trees;
try {
    trees = map(files, f->parseFile(f));
    ...
} catch (IOException e) {...}
```
This Genus program type-checks!

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```
Tree parseFile(String filename)
throws IOException {...}
```

Client code

```
List[File] files = ...;
List[Tree] trees;
try {
    trees = map(files, f→parseFile(f));
    ...
} catch (IOException e) {...}
```
This Genus program type-checks!

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```java
Tree parseFile(String filename)
throws IOException {...}
```

Client code

```java
List[File] files = ...;
List[Tree] trees;
try {
    trees = map(files, f->parseFile(f));
    ...
} catch (IOException e) {...}
```

Exception mismatch is allowed

`aware` of exceptions raised by the lambda expression
This Genus program type-checks!

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

`oblivious` to exceptions raised by `f`

Raises `IOException` if the file does not exist

```java
Tree parseFile(String filename)
throws IOException {...}
```

Client code

```java
List[File] files = ...;
List[Tree] trees;
try {
    trees = map(files, f->parseFile(f));
    ...
} catch (IOException e) {...}
```

`Exception mismatch is allowed`

`aware` of exceptions raised by the lambda expression
Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

 Raises IOException if the file does not exist

```java
Tree parseFile(String filename) throws IOException {...}
```

Client code

```java
List[File] files = …;
List[Tree] trees;
try {
    trees = map(files, f->parseFile(f));
    …
} catch (IOException e) {...}
```
Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```
Tree parseFile(String filename)
throws IOException {...}
```

Client code

```
List[File] files = ...;
List[Tree] trees;
trees = map(files, f->parseFile(f));
...
```
Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

\[
\text{List}[Y] \map{X,Y}(\text{List}[X] \; l, \; X \rightarrow Y \; f) \{\ldots\}
\]

Raises IOException if the file does not exist

\[
\text{Tree} \; \text{parseFile}(\text{String} \; \text{filename}) \quad \text{throws} \; \text{IOException} \{\ldots\}
\]

Client code

\[
\text{List}[\text{File}] \; \text{files} = \ldots; \\
\text{List}[\text{Tree}] \; \text{trees}; \\
\text{trees} = \map{\text{files}, \; f \rightarrow \text{parseFile}(f)}; \\
\ldots
\]
Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```
Tree parseFile(String filename) throws IOException {...}
```

Client code

```
List[File] files = ...;
List[Tree] trees;
trees = map(files, f->parseFile(f));
... 
```
Exceptions tunnel through contexts oblivious to them

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Raises IOException if the file does not exist

```java
Tree parseFile(String filename)
throws IOException {...}
```

Client code

```java
List[File] files = ...;
List[Tree] trees;
trees = map(files, f->parseFile(f));
...
```
Recall this Java program handles exceptions **by accident**…

A higher-order function in Java

```java
<X,Y> List<Y> map(List<X> xs, X→Y f) {
    List<Y> ys = new ArrayList<Y>();
    Iterator<X> it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```java
List<Iterator<Int>> iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```

Raises NSE if there is no next element

```java
interface Iterator<X> {
    X next() throws NSE;
    ...
}
```

Handler intercepts exceptions raised by f!
Genus prevents accidental handling

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) {
            break;
        }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```
Genus prevents accidental handling

A higher-order function in Genus

List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE e) {
            break;
        }
    }
    return ys;
}

Client code

List[It] iters = …;
try {
    var ints = map(iters, it->it.next());
    ...;
} catch (NSE e) {...}

Tunneling in Genus

Accidental handling in Java

Identifier of an exception: type

Genus prevents accidental handling

Raises NSE if there is no next element

Tunneling in Genus

Accidental handling in Java

Identifier of an exception: type
Genus prevents accidental handling

A higher-order function in Genus

List[X] → new ArrayList[Y]()
Iterator[X]
while (true)
try { ys.add(f(it.next())); }
catch (NSE e) { break; }
return ys;

List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  ……
} catch (NSE e) { …}

Client code

Tunneling in Genus

Identifier of an exception: type and blame label

Accidental handling in Java

Identifier of an exception: type

Genus prevents accidental handling

 Raises NSE if there is no next element
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
interface Iterator[X] {
  X next() throws NSE;
  // ...
}
```

Raises NSE if there is no next element

```java
List[Y] map[X,Y](List[X] xs, X→ Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try {
      ys.add(f(it.next()));
    } catch (NSE e) {
      break;
    }
  }
  return ys;
}
```

Program

```
List[Iterator[int]] iters = ...;
var ints = map(iters, it->it.next());
... 
```

Exception mismatch

Compiler

```
interface Iterator[X] {
  X next() throws NSE;
  // ...
}
```

Blame label created
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
interface Iterator<X> {
  X next() throws NSE;
  …
}
```

Raises NSE if there is no next element

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Client code

```java
List[Iterator[int]] iters = …;
var ints = map(iters, it->it.next());
…
```

Program

Exception mismatch

Compiler

Blame label created
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); } 
    catch (NSE e) { break; } 
  }
  return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...
try {
  var ints = map(iters, it->it.next());
  ...
} catch (NSE·B1 e) {...}
```

Raises NSE if there is no next element

```java
interface Iterator[X] {
  X next() throws NSE;
  ...
}
```

Program Exception mismatch

Compiler Blame label created
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next())); }
        catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE·B1 e) {...}
```
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        ys.add(f(it.next()));
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE•B1 e) {...}
```

Raises NSE if there is no next element

```java
interface Iterator[X] {
    X next() throws NSE;
    …
}
```

Program  Exception mismatch

Compiler  Blame label created

Runtime  Exceptions tunneled to contexts at fault

Augmenting exception identifiers with blame labels
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    ys.add(f(it.next()));
  }
  return ys;
}
```

Client code

```
List[Iterator[int]] iters = ...;
try {
  var ints = map(iters, it->it.next());
  ...
} catch (NSE·B1 e) {...}
```

Program

```
interface Iterator[X] {
  X next() throws NSE;
  ……
}
```

Compiler

```
B2
```

Exception mismatch

Runtime

```
B1
```

Blame label created

Exceptions tunneled to contexts at fault

B2

Raises NSE if there is no next element

B1

 Raises NSE if there is no next element
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]() ;
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE•B2 e) {
            break;
        }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE•B1 e) {...}
```

Program  Exception mismatch
Compiler  Blame label created
Runtime  Exceptions tunneled to contexts at fault

Raises NSE if there is no next element

A higher-order function in Genus

```java
interface Iterator[X] {
    X next() throws NSE;
    …
}
```

List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]() ;
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE•B2 e) {
            break;
        }
    }
    return ys;
}

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE•B1 e) {...}
```

Program  Exception mismatch
Compiler  Blame label created
Runtime  Exceptions tunneled to contexts at fault

Raises NSE if there is no next element
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next())); } catch (NSE·B2 e) { break; }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE·B1 e) {...}
```

Program

```
interface Iterator[X] {
    X next() throws NSE;
    ...
}
```

Compiler

Exception mismatch

Blame label created

Runtime

Exceptions tunneled to contexts at fault
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next())); } 
        catch (NSE·B2 e) { break; } 
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE·B1 e) {...}
```

 Raises NSE if there is no next element

```java
interface Iterator[X] {
    X next(b) throws NSE·b;
    ...
}
```

Program: Exception mismatch

Compiler: Blame label created

Runtime: Exceptions tunneled to contexts at fault
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next(B2))); }
    catch (NSE·B2 e) { break; }
  }
  return ys;
}
```

Client code

```
List[Iterator[int]] iters = ...;
try {
  var ints = map(iters, it->it.next());
  ...
} catch (NSE·B1 e) {...}
```

Raises NSE if there is no next element

```
interface Iterator[X] {
  X next(b) throws NSE·b;
  ...
}
```

Program  Exception mismatch

Compiler  Blame label created

Runtime  Exceptions tunneled to contexts at fault
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next(B2))); }
        catch (NSE•B2 e) { break; }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next(B1));
    ...
} catch (NSE•B1 e) {...}
```

Program

```
interface Iterator[X] {
    X next(b) throws NSE•b;
    ...
}
```

Compiler

Exception mismatch

Runtime

Blame label created

Exceptions tunneled to contexts at fault
Augmenting exception identifiers with blame labels

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try { ys.add(f(it.next())); }
        catch (NSE e) { break; }
    }
    return ys;
}
```

Client code

```java
List[Iterator[int]] iters = ...;
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE e) {...}
```
No need to write blame labels out

A higher-order function in Genus

```java
List[Y] map[X,Y](List[X] xs, X→Y f) {
  List[Y] ys = new ArrayList[Y]();
  Iterator[X] it = xs.iterator();
  while (true) {
    try { ys.add(f(it.next())); }
    catch (NSE e) { break; }
  }
  return ys;
}
```

Client code

```java
List[Iterator[int]] iters = …;
try {
  var ints = map(iters, it->it.next());
  …
} catch (NSE e) {…}
```
Blame labels as a notion of context

Program

Gradual typing
Type mismatch

Compiler

Blame label created

Exc. tunneling
Exception mismatch

Runtime

Cast failures attributed to context at fault

Exception mismatch
Exceptions tunneled to context at fault
Blame labels as a notion of context

Gradual typing

Program: Type mismatch
Compiler: Blame label created
Runtime: Cast failures attributed to context at fault

Exc. tunneling

Program: Exception mismatch
Compiler: Blame label created
Runtime: Exceptions tunneled to context at fault

Blame labels impose static obligations on blameful contexts to handle exceptions so labeled.
A theorem

Theorem

Well-typed programs handle their exceptions.

Type system checks that exception mismatches do not escape.
A theorem

**Theorem**

Well-typed programs handle their exceptions.

Type system checks that exception mismatches do not escape.

```
List[Y] map[X,Y](List[X] l, X→Y f) {...}
```

Function f should not escape map.
A theorem, and a compiler too

**Theorem**
Well-typed programs handle their exceptions.

**Compiler**
An extension of the base Genus compiler
39,000 LoC + 6,000 LoC
Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates \(~200\) (out of \(~1,000\) LoC in a visitor class in javac
Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.
Genus version restores static checking of exceptions
Genus version eliminates $\sim 200$ (out of $\sim 1,000$) LoC in a visitor class in javac

```java
public void visitReturn(JCReturn tree) {
    try {
        print("return");
        if (tree.expr != null) {
            print(" ");
            printExpr(tree.expr);
        }
        print(";");
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
}
```

exception unwrapping in javac
Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.

Genus version restores static checking of exceptions

Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

Porting the Java Collections Framework (all general-purpose implementations)

Eliminated risks of accidental handling
Experience: cleaner yet safer code

Porting Java code that uses anti-patterns

Examples come from various sources: the javac compiler, Apache Commons, etc.
Genus version restores static checking of exceptions
Genus version eliminates ~200 (out of ~1,000) LoC in a visitor class in javac

Porting the Java Collections Framework (all general-purpose implementations)
Eliminated risks of accidental handling

Lists.transform() throws a NoSuchElementException if an IndexOutOfBoundsException is raised #1606

Interestingly enough, the issue doesn't appear to come from Guava code -- it's java.util.AbstractList's implementation of iterator that catches I/O/BE and turns it into an NSE.
Compilation

Translation reifies blame labels so that exceptions can acquire new identities.
Compilation

Translation reifies blame labels so that exceptions can acquire new identities.

It imposes a cost on normal control flow even when exceptions are not raised.
Performance of exception-infrequent code

y-axis = \frac{\text{running time of Genus using Java’s exception mechanism}}{\text{running time of Genus using the new exception mechanism}}

0.03% slowdown across benchmarks
0.87% slowdown in the worst case

(Benchmarks come from JOlden and DaCapo suites)
Using NSE to speed up for-each loops

Performance of exception-infrequent code

Better ↓

y-axis = \frac{\text{running time of Genus using Java's exception mechanism}}{\text{running time of Genus using the new exception mechanism and using NSE to speed up for-each loops}}

2.4% speedup across benchmarks
0.5% slowdown in the worst case

(Benchmarks come from JOlden and DaCapo suites)
Using **NSE** to speed up for-each loops

```java
for (T x : c) {
    // loop body
}
```

**conventional translation**

```java
Iterator<T> it = c.iterator();
while (it.hasNext()) {
    T x = it.next();
    // loop body
}
```

**N+1 calls to hasNext()**
Using **NSE** to speed up for-each loops

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
    T x = it.next();
    ...
    // loop body
}
```

**N+1 calls to hasNext()**

```
for (T x : c) {
    ...
    // loop body
}
```

**conventional translation**

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
    T x = it.next();
    ...
    // loop body
}
```

**exception-based translation**

```
Iterator[T] it = c.iterator();
while (true) {
    try {
        T x = it.next();
        ...
        // loop body
    }
    catch (NSE e) {
        break;
    }
}
```
Exception-based translation is frowned upon...

```java
// Do not use this hideous code for iteration over a collection!
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next();
        ...
    }
} catch (NoSuchElementException e) { }
```

Exception-based translation is frowned upon…

- N+1 calls to hasNext()
The NSE is guaranteed to be handled

Optimization is cost-effective

```
Iterator<T> it = c.iterator();
while (it.hasNext()) {
    T x = it.next();
    ... // loop body
}
```

```
for (T x : c) {
    ... // loop body
}
```

```
Iterator<T> it = c.iterator();
while (true) {
    try {
        T x = it.next();
        ... // loop body
    }
    catch (NSE e) {
        break;
    }
}
```

`$N+1$ calls to hasNext()`

one `throw-and-catch`
The **NSE** is guaranteed not to be accidentally handled

Optimization is correct

```java
for (T x : c) {
    ... // loop body
}
```

conventional translation

```java
Iterator<T> it = c.iterator();
while (it.hasNext()) {
    T x = it.next();
    ... // loop body
}
```

N+1 calls to `hasNext()`

exception-based translation

```java
Iterator<T> it = c.iterator();
while (true) {
    try {
        T x = it.next();
        ... // loop body
    }
    catch (NSE e) {
        break;
    }
}
```

one throw-and-catch

The NSE is guaranteed not to be accidentally handled. Optimization is correct.
Algebraic effects subsume exceptions

effect NSE {
    void throw();
}


Algebraic effects subsume exceptions

Effects can be accidentally handled if not checked!

```
effect NSE {
  void throw();
}
```
Effect polymorphism

```
List[Y] map[X,Y](List[X] xs, X→Y f) {...}
```
Effect polymorphism

List[Y] map[X,Y,E](List[X] xs, X→Y throws E f) throws E {...}
Effect polymorphism

List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {...}
Effect polymorphism

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {...}
```
Effect polymorphism

Statically checked effects can be accidentally handled!

```java
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {...}
```
Effect polymorphism

Statically checked effects can be accidentally handled!

An effect-polymorphic, higher-order function

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch NSE { void throw() { break; } }
    }
    return ys;
}
```

Client code

```
try {
    var ints = map(iters, it->it.next());
    ...
} catch NSE { void throw() { ... } }
```
Effect polymorphism, but with tunneling

Accidental handling is prevented

An effect-polymorphic, higher-order function

```java
List[Y]/E map[X,Y,E](List[X] xs, X → Y/E f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE) {
            void throw() {
                break;
            }
        }
    }
    return ys;
}
```

Client code

```java
try {
    var ints = map(iter, it->it.next());
}
```
Effect polymorphism, but with tunneling

Accidental handling is prevented

An effect-polymorphic, higher-order function

```java
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE) { void throw() { break; } }
    }
    return ys;
}
```

Client code

```java
try {
    var ints = map(iters, it->it.next());
    ...
} catch (NSE) { void throw() { ... } }
```
Effect polymorphism, but with tunneling

Accidental handling is prevented

Effects tunnel through contexts polymorphic to them
(cf. exception-oblivious contexts in Genus)

Compiler (or programmer) assigns unique names to effect handlers
(cf. unique blame labels in Genus)

Handlers have lexical regions
(cf. local escape analysis in Genus)
The theorem we proved

Theorem: Well-typed programs handle their exceptions.
One more theorem to prove

**Theorem**

Well-typed programs handle their exceptions.

**Theorem**

No unhandled exceptions

No accidentally handled exceptions
One more theorem to prove

A theorem about type safety

Well-typed programs handle their exceptions.

No unhandled exceptions

Theorem

No accidentally handled exceptions
A theorem about type safety

Well-typed programs handle their exceptions.

No unhandled exceptions

A theorem beyond type safety

No accidentally handled exceptions

?
One more theorem to prove

A theorem beyond type safety

No accidentally handled exceptions

Accidental handling causes implementation details to leak through abstraction boundaries.
One more theorem to prove

A theorem beyond type safety

No accidentally handled exceptions

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.
One more theorem to prove

A theorem about abstraction safety

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

No accidentally handled exceptions
One more theorem to prove

A theorem about abstraction safety

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

No accidentally handled exceptions

[Zhang & Myers 2019]
A theorem about abstraction safety

Accidental handling causes implementation details to leak through abstraction boundaries.

The new tunneling semantics fixes this leakage and preserves abstraction.

No accidentally handled algebraic effects

[Zhang & Myers 2019]
Accidental handling is a violation of abstraction

An effect-polymorphic, higher-order abstraction

\[
\text{List}[Y]/E \text{ map}[X,Y,E](\text{List}[X] \text{ xs, } X \rightarrow Y/E \text{ f})
\]
Accidental handling is a violation of abstraction

```java
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (it.hasNext()) {
        ys.add(f(it.next()));
    }
    return ys;
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    List[Y] ys = new ArrayList[Y]();
    Iterator[X] it = xs.iterator();
    while (true) {
        try {
            ys.add(f(it.next()));
        } catch (NSE) {
            void throw() { break; }
        }
    }
    return ys;
}
```
Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

```java
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    ...
    // N+1 calls to hasNext()
}
```

```java
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    ...
    // one throw-and-catch
}
```
Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    ... // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
    ... // one throw-and-catch
}
```

Client makes distinct observations

```
List[Iterator[int]] iters = ...
try {
    ints = map(iters, it->it.next());
    ...
} catch NSE { void throw() { ... } }
```

iters contains an empty iterator
Accidental handling is a violation of abstraction

Two supposedly equivalent implementations

List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
... // N+1 calls to hasNext()
}

List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
... // one throw-and-catch
}

Client makes distinct observations

map raises NSE

map returns normally

iters contains an empty iterator

List[Iterator[int]] iters = ...
try {
    ints = map(iters, it->it.next());
    ...
} catch NSE { void throw() { ... } }
Accidental handling causes implementation details to leak through abstraction boundaries.

Two supposedly equivalent implementations

```haskell
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // N+1 calls to hasNext()
}
```

```haskell
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // one throw-and-catch
}
```

Client makes distinct observations

map raises NSE

map returns normally
Two supposedly equivalent implementations

```scala
List[Y]/E map[X,Y,E](List[X] xs, X→ Y/E f) {
  ... // N+1 calls to hasNext()
}
```

```scala
List[Y]/E map[X,Y,E](List[X] xs, X→ Y/E f) {
  ... // one throw-and-catch
}
```

Client makes distinct observations

map raises NSE if iters contains an empty iterator

map returns normally if iters contains an empty iterator

The new tunneling semantics fixes the leakage and preserves abstraction.
The new tunneling semantics fixes the leakage and preserves abstraction.

**Two equivalent implementations**

```
List[Y]/E map[X,Y,E](List[X] xs, X → Y/E f) {
    ...
    // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X → Y/E f) {
    ...
    // one throw-and-catch
}
```

Client makes distinct observations

- map raises NSE if iters contains an empty iterator
- map returns normally if iters contains an empty iterator
Two equivalent implementations

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ...
  // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ...
  // one throw-and-catch
}
```

No client can distinguish between the two implementations

The new tunneling semantics *fixes* the leakage and preserves abstraction.

contextual equivalence [Morris 1968]
No client can distinguish between the two implementations

contextual equivalence [Morris 1968]

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit
Definition of the logical relation

Algebraic effects render the core language Turing-complete

→ Make the logical relation step-indexed [Appel and McAllester 2001]

Reduction is dependent on surrounding evaluation contexts

→ Make the logical relation biorthogonal [Pitts and Stark 1998]

Runtime allocates fresh names

→ Make the logical relation indexed by (degenerate) possible worlds

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Abstraction Theorem for type polymorphism
[Reynolds 1983]

Abstraction Theorem for effect polymorphism
[Zhang & Myers 2019]
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

1. Abstraction Theorem for type polymorphism
   [Reynolds 1983]
   
   \[ \text{List}[Y] \text{ map}[X,Y](\text{List}[X] \ l, X \to Y \ f) \] { ... } 

   Should not make decisions based on the run-time instantiations of type variables \( X \) and \( Y \)

2. Abstraction Theorem for effect polymorphism
   [Zhang & Myers 2019]
   
   \[ \text{List}[Y]/E \text{ map}[X,Y,E](\text{List}[X] \ l, X \to Y/E \ f) \] { ... }
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence

Abstraction Theorem for type polymorphism
[Reynolds 1983]

\[
\text{List}[Y] \; \text{map}[X,Y](\text{List}[X] \; l, \; X \to Y \; f) \; \{ \; \}
\]

Should **not** make decisions based on the run-time instantiations of type variables \(X\) and \(Y\)

Abstraction Theorem for effect polymorphism
[Zhang & Myers 2019]

\[
\text{List}[Y/E] \; \text{map}[X,Y,E](\text{List}[X] \; l, \; X \to Y/E \; f) \; \{ \; \}
\]

Should **not** make decisions based on the run-time instantiation of effect variable \(E\)
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

Abstraction Theorem for effect polymorphism

\[ \text{List}[Y]/E \ \text{map}[X,Y,E](\text{List}[X] \ l, \ X \rightarrow Y/E \ f) \] { … }

Should not make decisions based on the run-time instantiation of effect variable E

Handling the run-time instantiation of effect variable E breaks abstraction!

The abstraction story, in abstract

1 Define contextual equivalence
2 Define the logical relation
3 Show that logical relatedness implies contextual equivalence
4 Profit

The abstraction theorem (aka Parametricity)

Handling the run-time instantiation of effect variable E breaks abstraction!
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

Sound w.r.t. contextual equivalence

Coq mechanization

https://github.com/yizhouzhang/abseff-coq

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit
Properties of the logical relation

Abstraction Theorem (aka Parametricity)

Sound w.r.t. contextual equivalence

The abstraction story, in abstract

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

Coq mechanization

https://github.com/yizhouzhang/abseff-coq

Results apply to the core language of Genus too.
Deriving equivalence results

Abstraction Theorem (aka Parametricity)

Sound w.r.t. contextual equivalence

No accidentally handled algebraic effects

Theorem

1 Define contextual equivalence
2 Define the logical relation
3 Show that logical relatedness implies contextual equivalence
4 Profit
Deriving equivalence results

The abstraction story, in abstract

1 Define contextual equivalence
2 Define the logical relation
3 Show that logical relatedness implies contextual equivalence
4 Profit

Theorem

Abstraction Theorem (aka Parametricity)

Sound w.r.t. contextual equivalence

No accidentally handled algebraic effects
Deriving equivalence results

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // N+1 calls to hasNext()
}
```

```
List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // one throw-and-catch
}
```

**The abstraction story, in abstract**

1. Define contextual equivalence
2. Define the logical relation
3. Show that logical relatedness implies contextual equivalence
4. Profit

**Theorem**

- Abstraction Theorem (aka Parametricity)
- Sound w.r.t. contextual equivalence

No accidentally handled algebraic effects
Deriving equivalence results

List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // N+1 calls to hasNext()
}

compiler optimization

List[Y]/E map[X,Y,E](List[X] xs, X→Y/E f) {
  ... // one throw-and-catch
}

The abstraction story, in abstract

1 Define contextual equivalence
2 Define the logical relation
3 Show that logical relatedness implies contextual equivalence
4 Profit

Theorem

- Abstraction Theorem (aka Parametricity)
- Sound w.r.t. contextual equivalence

No accidentally handled algebraic effects
Abstraction-Safe Effect Handlers via Tunneling

Yizhou Zhang
Cornell University