An Implementation Mechanism for Tailorable Exceptional Flow

Steven te Brinke, Mark Laarakkers, Christoph Bockisch & Lodewijk Bergmans

{brinkes, m.laarakkers, c.m.bockisch, bergmans}@cs.utwente.nl
Motivation

- Managing software complexity
  - Separate concerns
  - Divide software in modules
  - Compose modules using composition operators
    - E.g. exception handlers

- New composition operators introduced
  - E.g. try-with-resources

- Composition operators should be flexible
  - Not built into the language
  - But provided as a library
def m():
  o()

def o():
  try {
    return
    p()
  } catch {
    return
  } finally {
    return
  }

def p():
  throw
  return
def m() -> None:
    o()

def o() -> None:
    try:
        p()
    except Exception as e:
        finally:
            return

def p() -> None:
    throw return

return
Continuations

- Store **execution state**
  - What is to be executed
  - Context bindings
    (e.g., local variables)

```python
def m():
    o()
    return()

def o():
    return()
```

continuation bound to ‘return’
Implementing Execution Orders with Continuations

- Implement composition mechanism as method
- Accept continuations as arguments
- Keywords become invocation of continuation or method
- Define control flow by passing continuations to invocations of continuations

```
def m()

    o()

    tryCatch({
        return()
        throw()
    },{
        return()
    })

return continuation
try continuation
catch continuation
```
def o()

  var out
  try {
    out = File.open(…)
    out.write(…)
  } catch {
  } finally {
    if (out != null)
      out.close()
  }

try (var out = File.open(…)) {
  out.write(…)
} catch {
}
Requirements

- Requirements for implementing tailored control flow:
  - Tailoring execution order through **continuations**
  - Tailoring scopes by making **scopes first-class**

- Provide a user-friendly language to tailor control flow
Implementing Control-Flow Composition

method tryFinally(var try, var finally) {
    try.enclosingScope = try.definingScope
    finally.enclosingScope = finally.definingScope
    try.return = { value →
        finally()
        return(value)
    }
    try.throw = { exception →
        finally()
        throw(exception)
    }
}
Implementing Control-Flow Composition

```java
method tryWithResources(var resources, var try) {
  resources.enclosingScope = resources.definingScope
  try.enclosingScope = resources.scope
  try.return = { value →
    for (resource defined in resources)
      resource.close()
    return(value)
  }
  try.throw = { exception →
    for (resource defined in resources)
      resource.close()
    throw(exception)
  }
} {
  resources()
  try()
  for (resource defined in resources)
    resource.close()
}
```

Meaning of “return” operation in context of “try”.

Meaning of “throw” operation in context of “try”.

Definition of scope hierarchy.
Related Work

- Delegates (C#)
  - Scopes bound at definition

- Delimited continuations (Scala)
  - Scopes bound at definition

- eval function (JavaScript)
  - Scopes bound during execution
Summary and Future Work

- Implementing tailorable control-flow mechanisms based on:
  - **Continuations** for tailoring execution order
  - **First-class scopes** for tailored scoping
- Showed an envisioned language to do so

Future Work

- Precise semantics of envisioned language

Steven te Brinke
brinkes@cs.utwente.nl