A DSL for Compensable and Interruptible Executions

Hiroaki Inoue
Graduate School of Informatics
Kyoto University
Japan
hinoue@kuis.kyoto-u.ac.jp

Tomoyuki Aotani
School of Computing
Tokyo Institute of Technology
Japan
aotani@c.titech.ac.jp

Atsushi Igarashi
Graduate School of Informatics
Kyoto University
Japan
igarashi@kuis.kyoto-u.ac.jp

Abstract

Context-awareness is getting more and more important in software applications. Such an application runs depending on the time-varying status of the surrounding environment such as network connection, battery/energy charge and heat. Interruptions, or asynchronous exceptions, are useful to achieve context-awareness: if the environment changes, the execution of the application is interrupted reactively to stop and/or recover the internal state for adapting to the new environment. It is, however, difficult to program with interruptions modularly in most programming languages because their support is too basic and is based on synchronous exception handling mechanism such as `try-catch`.

We propose a domain-specific language *ContextWorkflow* for modular interruptible programs as a solution to the problem. An interruptible program is basically a workflow, i.e., a sequence of atomic computations with compensations. The uniqueness of ContextWorkflow is that, during its execution, a workflow watches the context, which is represented as a reactive value in functional reactive programming and instructs how the execution reflects the status of the surrounding environment.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; **Error handling and recovery**; *Procedures, functions and subroutines*;

Keywords Asynchronous Exception, Interruptible Execution, Monads, Transaction, Workflow

ACM Reference Format:
Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. 2017. A DSL for Compensable and Interruptible Executions. In *Proceedings of 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3141858.3141860

1 Introduction

As mobile computing devices are spread, recent applications tend to depend on external information that is time-varying, such as time, heat, battery level, asynchronous commands and availability of some functions such as GPS modules. As examples, we will consider the following two applications.

Application 1: autonomous robot. Consider a robot which has a battery and moves around to do some task. In case the battery is running out before accomplishing the given task, it should suspend what it is doing, return to a refueling point, and resume the task.

Application 2: energy-aware computing. Energy-aware computing [11, 14] is a research area that explores a programming model to control trade-off between energy consumption and accuracy of computation. For example, in Eco [14], energy-aware computations change their accuracy dynamically according to the amounts of remaining computations and available energy. The accuracy is high and low if enough energy is available or not, respectively. Suppose that a mobile device running a computation with low energy is plugged. It would be then desirable that the computation aborts immediately and restarts to get a more accurate result.

In order to realize such applications, programs must be *interruptible*: an ability to suspend or abort the execution by promptly reacting to changes of external information and to compensate an incompleted execution properly—e.g., a robot moves back to the refueling point.

1.1 Problem

One common but naive way to realize interruption is to use exception handling constructs such as `try-catch` and `throw`. However, it is cumbersome to write interruptible programs using exceptions because programmers have to take the following three concerns into account:

- *Atomicity.* Programmers need to divide an interruptible program into regions where interruption can occur.
- *Propagation of Exception.* If `try-catch` is nested, some methods often have to rethrow exceptions in order to propagate them to outer handlers.

- *Progress.* It is important to know how much progress an interruptible program has made in order to clean up incomplete executions. For example, whether a program `a();b();...` is interrupted just after `a()` or just after `b()` may influence how clean-up code should be executed. However, a single `try-catch` does not care at which program point a program is interrupted, so programmers have to manage some kind of states or use nested `try`-blocks (causing the propagation problem above) as in the code below[1]:

```
// state management
try{
  state = beforeA; a(); state = beforeB; b();
} catch {
  if(state == beforeB) { comp_b(); } comp_a(); }

// nested try-catch
try{ a();
  try{ b(); } catch {
    comp_b(); ex_rethrow();
  } } catch { comp_a(); }
```

  where `comp_a()` and `comp_b()` are the recovering code for `a()` and `b()`, respectively.

We show how these concerns can be addressed in thread programming using polling and asynchronous exceptions.

Polling. In Java, a thread can be interrupted by calling `Thread.interrupt()` on a `Thread` object. Then, `InterruptedException` is thrown in the thread if the thread is blocked by `sleep()`, `wait()` or `join()`; otherwise, an interrupted flag is turned on and the thread has to manually check the flag using `Thread.isInterrupted()` and throw an exception manually.

```
if(Thread.currentThread().isInterrupted())
  throw new InterruptedException();
```

Therefore, atomicity is controlled mostly by manual polling of the interruption flag. Propagation must be done manually by rethrowing `InterruptedException`. Progress has to be managed manually, too.

Asynchronous exceptions. A few programming languages such as Ruby and Haskell support asynchronous exception. In Haskell [8], an interruption occurs when another thread calls `throwTo`. For example, timeout is implemented as follows: one thread creates another thread, namely `th`, for a time-consuming task using `forkIO`, sleeps for a few minutes using `threadDelay` and throws `Timeout` to `th`.

```
do th <- forkIO $ timeConsuming
   threadDelay someFixedTime
   throwTo th Timeout
```

[1]In this hypothetical language, all exceptions thrown are caught and the block after `catch` will be executed.

Atomicity is controlled using `mask` and `interruptible` rather than cumbersome manual polling as follows[2].

```
timeConsuming =
  mask $ do
  preprocess
  b <- catch(interruptible mainTask)
       (\(e :: Timeout) -> do {recovery; throw e});
  postprocess
```

`mask` makes `timeConsuming` atomic, while `interruptible` allows asynchronous exceptions to occur within `mainTask`.

The exception is, however, still handled using `catch`, which takes a computation and a handler just as `try-catch`. If `Timeout` occurs, the handler rethrows the exception. Therefore, the progress and propagation concerns should be manually programmed.

1.2 Our Work

In this paper, we propose a language *ContextWorkflow* to make interruptible programs easier to write, composable, and more understandable. It supports easy handling of atomicity, automatic propagation of interruption exceptions and automatic progress management.

The language is inspired by workflow (or long-running transaction) [4, 5]. A workflow is a sequence of *atomic action* (or inner transactions) and its execution results in all or nothing, similarly to ordinary transactions. Each atomic action is accompanied by a compensation. When an interruption or failure takes place—it can only occur when one atomic action is finished (before the next atomic action starts)—the whole workflow is aborted by running the compensations of the already completed transactions in a reverse order.

The main idea of our ContextWorkflow is that a workflow is executed under some *context*, which changes over time asynchronously and indicates how the execution of a workflow proceeds, such as *continue*, *abort* and *restart*. Asynchronous context change is similar to asynchronous exceptions in the sense that it interrupts the execution of the workflow (at a program point between atomic actions). In the continue context, normal actions are executed with their compensations recorded. In the abort context, the execution of a normal action is stopped and the recorded compensations are executed. The restart context is a combination of the abort and continue contexts, i.e., the normal execution is stopped, the compensations are run, and the whole workflow is reexecuted from the beginning.

Contexts are represented as signals in Reactive Programming [10] and (implicitly) checked at the beginning of each atomic action in order to realize asynchronous interruption. This makes it easy to consider the three concerns above.

[2]Strictly speaking, we have to use `mask_` instead of `mask`.

Atomicity is achieved by checking the context only at program points between atomic actions. Propagation of exception and progress are addressed automatically by the nature of workflows.

The paper is organized as follows: Section 2 describes more details of ContextWorkflow, Section 3 shows examples, Section 4 discusses related work, and Section 5 concludes this paper with future work. The implementation is available at https://github.com/h-inoue/ContextWorkflow.

2 ContextWorkflow

ContextWorkflow is implemented as a domain-specific language embedded in Scala using the Monad interface in functional programming library scalaz [1]. This section explains the basic constructs of ContextWorkflow.

2.1 Atomic Action and Workflow

An atomic action consists of a normal action (i.e., a Scala expression) and a compensation action, and a workflow is a sequence of atomic actions. Both of them are represented as objects of type `Workflow`, which is basically a compensation monad [9], which is a combination of the continuation and exception monads and defined below. [3]

```scala
trait Workflow[A] { self =>
  def execute[B](g: A => Try[B]): Try[B]
  // bind
  def flatMap[C](f: A => Workflow[C]): Workflow[C] =
    new Workflow[C] {
      def execute[D](g: C => Try[D]): Try[D] =
        self.execute(a => f(a).execute(g))
    }
}
// return
def unit[A](a: => A):Workflow[A] = atom(a, v => ())
def atom[A](proc: => A)(comp:A => Unit)
:Workflow[A] = new Workflow[A]{
  def execute[B](g: A => Try[B]): Try[B] =
    Try(proc) match {
      case Failure(e) => Failure(e)
      case Success(a) => g(a) match {
        case Failure(e) => {comp(a); Failure(e)}
        case Success(x) => Success(x)
      } } }
```

Method `execute` takes a continuation as the argument and runs the workflow similarly to the continuation monad. Method `flatMap` combines two workflows. Method `unit` injects an expression returning a value of type A into a workflow. The type "`=> A`" of the argument denotes that the expression passed as the argument is evaluated lazily, i.e., evaluated only when the value is necessary.

[3]Readers familiar with monads in Haskell should consider `flatMap` and `unit` as `bind` and `return`.

Method `atom` creates an atomic action from a normal action `proc` of type `=> A` and a compensation action `comp` of type `A => Unit`. A compensation action takes the result of the corresponding normal action—which has been finished—as an argument. More specifically, `execute` of the created workflow takes a continuation and executes the normal action. If the normal action fails due to an interruption, it propagates the failure. If the normal action is evaluated successfully, the given continuation `g` is applied to the result. If the continuation fails, the workflow executes the compensation action with the result of the normal action and propagates the failure. Otherwise the workflow returns the result of the continuation. For example, `atom(i += 1)(_ => i -= 1)` is an atomic action. Its normal action adds one to the variable `i` and returns the value of the type of `Unit` (equals the unique value `()`). The compensation action does its inverse.

The compensation action is not necessarily the inverse of the normal action. The purpose of the compensation action is to ensure the "state" (e.g., the state of a file handler) is acceptable even if an interruption occurs and the program stops or rolls back.

The atomic actions created by `atom` do not check the context and thus never fail. We will extend `atom` and `execute` in the later section so that an interruption is detected by checking the context at the beginning of a normal action.

Combining two or more workflows is easy thanks to the composability of monads and the `for`-comprehension in Scala. For example, we can get a workflow by sequencing two atomic actions `t1` and `t2` and returning `()` as follows:

```scala
val t3:Workflow[Unit] = for{
  _ <- t1
  _ <- t2
} yield ()
```

To run a workflow, we invoke `execute` with an identity continuation. We provide `run` to ease this task.

```scala
def run[A](wf: Workflow[A]):Try[A] =
  wf.execute[A]((x: A) => Success(x))
```

Note: Though not appeared in the paper, current implementation is more complex since it uses `Trampoline` in order to avoid `StackOverflowError`.

2.2 Interruption and Context

Interruptions occur only between atomic actions. Our implementation scheme is (1) to represent time-varying contexts and (2) to check the context associated to a workflow at the beginning of each atomic action in it. This section explains our signal-based representation of the context and another constructor of atomic actions with context checking namely `catom`. This section also explains how a workflow is executed when an interruption occurs.

Signal of Context. The context steers the execution of a workflow. It is either `Continue`, `Restart` or `Abort`.

```
trait Context
object Continue extends Context
object Abort extends Context
object Restart extends Context
```

`Continue` means that the execution should continue; `Abort` means that the execution should be stopped after executing the recorded compensation actions; `Restart` is a combination of `Abort` and `Continue`, i.e., it means that the workflow should be executed after executing the recorded compensation actions.

We represent variation of a context over time by a signal in functional reactive programming and use existing library REScala [10]. A signal is intuitively a function from time to some value. It can depend on other signals, and a value change of a signal is propagated to other signals which are dependent on the signal. Therefore, the library helps us implement time-varying contexts easily.

For example, we can represent the interruption due to timeout as a signal of `Context` as follows:

```
def timeout:Signal[Context] = {
  val seconds:Signal[Int] = /* updated every second */
  return Signal { if(seconds() > 10) Abort else Continue } }
```

Signal `seconds` gives the elapsed time in seconds. Method `timeout` creates a new signal of context, of which value is `Continue` for 10 seconds and `Abort` after that.

Context Polling. To check the context at the beginning of every atomic action, it is necessary to make the context available within `execute`. We modify `Workflow` so that `execute` takes a signal of `Context` as an additional argument.

```
def execute[B](ctx: Signal[Context])(g: A => Try[B]): Try[B]
```

It is also necessary to extend the constructor of atomic actions `atom` so that `execute` of the created `Workflow` object (1) takes the context as an argument and (2) checks the current value of the context before executing the normal action. The extended constructor `catom` is defined as follows:

```
def catom[A](proc: => A)(comp: A => Unit)
:Workflow[A] =
  new Workflow[Unit] {
    def execute[B](ctx: Signal[Context])(g: Unit => Try[B]):
         Try[B] =
      ctx.now match {
        case Abort => Failure(AbortEx)
        case Restart => Failure(RestartEx)
        case Continue => g(()) }
  }.flatMap((_:Unit) => atom(proc,comp))
```

For readability, we provide an alternative form `expr1 %% expr2`, which is equivalent to `catom(expr1,expr2)`.

Note that method `unit` is unchanged. One may wonder if it should be rewritten using `catom`. However, `catom` is *not* the unit operator of the `Workflow` monad. (One of the monad laws would be broken.)

Finally we extend `run` to take a factory function (like `timeout` above) to generate a signal of `Context` that is passed to `execute` so that the whole workflow is executed under the context. Restart of a workflow is also realized by `run`.

```
def run[A](wf: Workflow[A],
           createSC: () => Signal[Context]):Try[A] =
  wf.execute[A](createSC())((x: A) => Success(x)) match {
    case Failure(RestartEx) => run(t,createSC)
    case e => e }

run(t3, timeout _) // workflow execution
```

2.3 Larger Workflows

This section explains how we can build larger workflows and how they are executed.

Larger workflows using functions over monads. Not only the `for`-comprehension but also utility functions over monads are available to build workflows. For example, it is easy to build workflows that processes lists using `foldLeftM` defined in `FoldableOps[A]` (e.g., lists) of scalaz:

```
def foldLeftM[G[_], B]
      (z: B)(f: (B, A) => G[B])(implicit M: Monad[G]): G[B]
```

If we pass a function that creates the workflow as `f`, we will obtain a workflow that processes over `A`'s.

Below is an example that adds the integer values from 1 to 100 to variable `i` and prints the value of `i` unless it is not interrupted.

```
1  var i = 0
2  val lst = (1 to 100).toList
3  val longproc = for{
4    _ <- () %% (_ => println("compensated"))
5    _ <- lst.foldLeftM(()){
6      (_,n) => (i += n)%%(_ => i -= n)}
7    _ <- println(i) %% (_ => ())
8  } yield ()
```

If the workflow is interrupted, the program subtracts numbers already added in LIFO order. The atomic action in line 4 merely adds the compensation action. Lines 5 and 6 create a workflow that processes the list of the integers from 1 to 100. The normal action is to increment `i` by an integer in the list while the compensation action is to decrement `i`.

The execution of the workflow `longproc` depends on the context. Suppose that the workflow execution is interrupted after adding 43. Then the execution proceeds as follows:

```
(i += 1) → (i += 2) → ... → (i += 43)
 → (*interruption*) → (i -= 43) → (i -= 42) → ...
 → (i -= 1) → (println("compensated"))
 → (Failure e)
```

Sub-workflow. We would sometimes like to thread a part of a workflow so that the compensation actions of the part are ignored if the part is completed successfully. We call such a part *sub-workflow* and provide a construct sub that makes a part of workflow a sub-workflow:

```
def sub[A](t: Workflow[A]): Workflow[A]
```

Below is a simple example that uses sub.

```
1  for{
2    _ <- procA
3    _ <- sub{ procB }
4    _ <- procC
5    _ <- println("done") %% (_ => ())
6  } yield ()
```

Suppose that an interruption occurs just before the atomic action in line 5 is executed. Only the compensation actions of procC and procA are executed. Without sub around procB, the compensation actions for procB will be executed (between those of procC and procA even if procB has finished successfully. We will see a use case of sub in the next section.

3 Case Study

In this section, we look two case studies, which are based on applications described in Section 1, in order to describe expressiveness of ContextWorkflow.

3.1 Application 1: Maze Search Robot

Consider programming a robot that searches a maze. One main issue here is that the robot has limited energy, so he must get back to the start point before exhausting its energy. Other problem settings are:

- A maze is given as a set of Nodes, each of which represents floor and has its coordinate information.
- There is no goal. The robot is supposed to search every path in the maze as long as he can.

The method visit below is the core of the search, which is depth-first.

```
1  def visit(n: Node, maze: Set[Node]):Workflow[Unit] ={
2    for{
3      _ <- visited(n) %% (_ => unknown(n))
4      _ <- neighbors(n,maze).foldLeftM(()){
5        (_,neighbor) =>
6        if(isVisited(neighbor)){
7          sub {
8            for {
9              _ <- moveFromTo(n,neighbor)
10             _ <- visit(neighbor,maze)
11             _ <- move(n) %% (_ => ()) // get back to n
12           } yield () }
13       } else catom(())() // empty catom
14     } } yield ()
15 }
16 def neighbors(n: Node, maze: Set[Node]): List[Node] =
17   ... // getting neighbor nodes of the node n
18 def move(n: Node): Unit = ... // moving the robot to n
19 def moveFromTo(from: Node, to: Node): Workflow[Unit] =
20   move(to) %% (_ => move(from))
```

The method visit takes two parameters: the node n of the start point of the search and the maze, which is a set of nodes. An instance of Node has a flag to represent whether it has been visited or not. If visit is called, the program (1) sets the flag of the node n (by calling visited), (2) for each neighbor of the node whose flag has not been set, moves the robot to the neighbor n and recursively calls visit. If the search from the neighbors is done, it moves the robot back to the start node n. If interruption occurs, incomplete visit should be compensated: moving back to the start node (line 9) and turn off the flag of the start node (calling unknown on line 3).

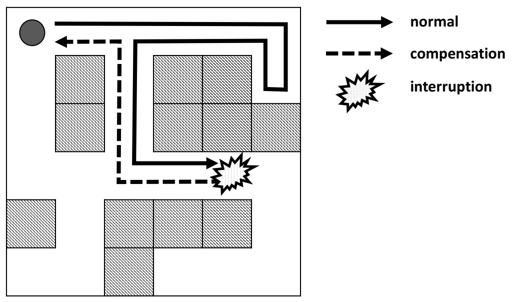

Figure 1. Maze search simulation

We can run visit with conditions involving reactive values such as timeout described above.

```
run(visit(startNode, maze), timeout _)
```

If the running time is beyond the given threshold, the execution will stop, compensating its actions, so the robot will come back to the start point. Figure 1 illustrates a search, being interrupted halfway.

An interesting point here is that a workflow produced by the method moveFromTo is a reusable unit, which we can use in another visit algorithm.

The reason why a sub-workflow is used in lines 7–12 is that we would like the robot to take a shortcut at compensation time. If we did not use sub-workflow in Figure 1, the robot would trace back all the route along which it came.

In this example, move() is treated as atomic and so an interruption does not occur during the move (even though it may take a long time for a robot to make a physical move). If one would like to put a checkpoint more frequently, he or she should subdivide the move function into smaller atomic actions and define move so as to return a Workflow[Unit].

3.2 Application 2: Energy-Aware Computing

Here, we consider a kind of energy-aware computing. Suppose that there are three battery levels: plugged, high-level and low-level. Suppose also that our energy-aware computations have conditional branches to change accuracy of the computation according to the battery level: the accuracy in plugged is higher than in high-level, which is higher than in low-level. If the battery level changes to high-level from low-level or to plugged from the others, the system aborts the computation, discards less accurate results, and restarts the computation for higher accuracy.

Let's make the idea concrete in code. Suppose that `BatLevel` represents the battery level and is inherited by objects `High`, `Low` and `Plugged`. We can write a signal of context using transitions of battery levels.

```
1  val bat:Signal[BatLevel] = ...
2                         // signal of Plugged | High | Low
3  def batctx():Signal[Context] = Signal {
4    (bat.delay(1).apply(), bat()) match {
5      case (Low,High) => Restart
6      case (Low,Plugged) => Restart
7      case (High,Plugged) => Restart
8      case _ => Continue } }
```

The expression `(bat.delay(1).apply(), bat())` represents the pair of the previous battery level and the current battery level, so the pair represents the transition of the battery level. The patterns in lines 5–7 represents when the battery level becomes higher, in which case, the context becomes `Restart`. Otherwise, the context becomes `Continue`.

It would be more interesting to introduce new contexts to suspend and resume a workflow, since we could add a rule like "transition from high to low triggers suspend." Implementation of suspend and resume is left for future work.

4 Related Work

There are several studies on context-oriented programming focusing on changing the behavior of the program in response to asynchronous context changes [6, 12]. The most related among them is Flute [2]. It supports interruptible context-dependent execution. Interruptions occur when the context changes and the context is represented as a reactive value. If the execution of the program is interrupted, it is suspended and another execution that reflect the changed context starts. The main difference is that ContextWorkflow focuses on how to interrupt execution and recover incomplete execution, while Flute puts emphasis on changing program behavior according to context changes.

Workflow or long-running transaction [4, 5] is a well-known error-recovery technique especially in distributed systems. Ramalingam and Vaswani [9] propose a monadic realization of workflows, used to develop a service tolerate to duplicate requests. We use the compensation monad, given as an extension of the workflow, as an integral part of the language. We think, though, combination with contexts and automated polling are new.

Modularization of exception handling code has been a major concern in Aspect-Oriented Programming [3, 7] because separation of exception handling code from normal code enhances re-usability of each modules. Our approach rather regards a pair of a normal code and a compensation as a unit of reuse.

Compensation actions can be seen as weak manual inversions of normal actions. In reversible programming languages [13], any programs run forward and backward and it is ensured that each direction is the exact inverse of the other. In other words, if programmers write a normal action in reversible programming languages, its compensation action is automatically defined. Manually specifying compensation actions have an advantage and a disadvantage compared to the reversible programming approach. The advantage is that we can avoid visiting unnecessary nodes to go back to the start node in the maze search example in Section 3. The disadvantage is that we have to specify the inversions of normal actions manually as in `longproc` in Section 2.3.

5 Conclusions

We have proposed how an interruptible program is written by ContextWorkflow, which is a domain specific language to write interruptible programs compositionally. ContextWorkflow combines the ideas of long-running transactions and reactive values. This approach provides modularity since an atomic computation with a compensation also becomes a workflow.

Future Work. This is work in progress and there is a lot to explore.

There are still several points to be decided in the design space of the atomicity and propagation concerns. For example, for atomicity, we should consider how a higher order workflow like `catom(catom(...) (...)) (...)` behaves; for propagation of exception, how a context change occurring at compensation time is treated. Formalization of the semantics is also left for future work.

Support for "suspension" is another big part of our future work. It allows us to suspend a workflow execution and to resume it later. This feature would be useful especially in distributed computing; a long-running program which is incompleted can be transferred to another machine and resume there.

We also would like to support partial restart, which stops propagating exceptions at a certain point in a workflow and restart the workflow from the point. This enables us to avoid unnecessary recomputations. Suppose that the battery level is initially high, changes to low and then changes to high in the example in Section 3.2. When the battery level comes back to high, the workflow is restarted completely even though some initial part of the workflow is executed in high level. Partial restart allows us to reuse the result of the part for efficiency.

In Java thread, interruptions occur when a thread is blocked by `sleep()`, `wait()` and `join()`. ContextWorkflow must take care of such thread blocking; even if the thread is blocked, it should be interruptible and take care of context changes. This is difficult in the current approach because the thread itself does not check the context.

Acknowledgments

This work was supported in part by Kyoto University Design School (Inoue).

References

[1] [n. d.]. scalaz. ([n. d.]). https://github.com/scalaz/scalaz

[2] Engineer Bainomugisha, Jorge Vallejos, Coen De Roover, Andoni Lombide Carreton, and Wolfgang De Meuter. 2012. Interruptible context-dependent executions: a fresh look at programming context-aware applications. In *Proc. of ACM Onward! 2012*. ACM, 67–84.

[3] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. 2008. EJFlow: Taming Exceptional Control Flows in Aspect-oriented Programming. In *Proc. of AOSD'08*. ACM, New York, NY, USA, 72–83.

[4] Christian Colombo and Gordon J. Pace. 2013. Recovery Within Long-running Transactions. *ACM Comput. Surv.* 45, 3, Article 28 (July 2013), 35 pages.

[5] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proc. of ACM SIGMOD*. ACM, New York, NY, USA, 249–259.

[6] Hiroaki Inoue and Atsushi Igarashi. 2016. A Library-based Approach to Context-dependent Computation with Reactive Values: Suppressing Reactions of Context-dependent Functions Using Dynamic Binding. In *Companion Proc. of the 15th Intl. Conf. on Modularity*. ACM, New York, NY, USA, 50–54.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. *Aspect-oriented programming*. Springer, 220–242.

[8] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. 2001. Asynchronous Exceptions in Haskell. In *Proc. of ACM PLDI*. ACM, New York, NY, USA, 274–285.

[9] Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. In *Proc. of ACM POPL (POPL '13)*. ACM, New York, NY, USA, 249–262.

[10] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proc. of Intl. Conf. on Modularity*. ACM, 25–36.

[11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proc. of ACM PLDI (PLDI '11)*. ACM, New York, NY, USA, 164–174.

[12] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-oriented Programming: Beyond Layers. In *Proc. of Intl. Conf. on Dynamic Languages*. ACM, New York, NY, USA, 143–156.

[13] Tetsuo Yokoyama and Robert Glück. 2007. A Reversible Programming Language and Its Invertible Self-interpreter. In *Proc. ACM PEPM (PEPM '07)*. ACM, New York, NY, USA, 144–153.

[14] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. 2015. A Programming Model for Sustainable Software. In *Proc. of ACM ICSE*. IEEE Press, Piscataway, NJ, USA, 767–777.