ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions

Hiroaki Inoue[1]
Graduate School of Informatics, Kyoto University, Kyoto, Japan
hinoue@fos.kuis.kyoto-u.ac.jp

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

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

Abstract

Context-aware applications, whose behavior reactively depends on the time-varying status of the surrounding environment – such as network connection, battery level, and sensors – are getting more and more pervasive and important. The term "context-awareness" usually suggests prompt reactions to context changes: as the context change signals that the current execution cannot be continued, the application should immediately abort its execution, possibly does some clean-up tasks, and suspend until the context allows it to restart. Interruptions, or asynchronous exceptions, are useful to achieve context-awareness. It is, however, difficult to program with interruptions in a compositional way in most programming languages because their support is too primitive, relying on synchronous exception handling mechanism such as `try–catch`.

We propose a new domain-specific language *ContextWorkflow* for interruptible programs as a solution to the problem. A basic unit of an interruptible program is a workflow, i.e., a sequence of atomic computations accompanied with compensation actions. The uniqueness of ContextWorkflow is that, during its execution, a workflow keeps watching the context between atomic actions and decides if the computation should be continued, aborted, or suspended. Our contribution of this paper is as follows; (1) the design of a workflow-like language with asynchronous interruption, checkpointing, sub-workflows and suspension; (2) a formal semantics of the core language; (3) a monadic interpreter corresponding to the semantics; and (4) its concrete implementation as an *embedded domain-specific language* in Scala.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases workflow, asynchronous exception, checkpoint, monad, embedded domain specific language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.2

Supplement Material ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.4.3.4`

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

Acknowledgements We thank Hidehiko Masuhara and anonymous reviewers for valuable comments.

---

[1] The current affiliation is Mitsubishi Electric Corporation.

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 2; pp. 2:1–2:33
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

As mobile computing devices have spread, recent applications tend to depend on external information (called *context*) that is time-varying, such as battery level, heat, human input, network connection, and availability of external modules; these applications are so-called *context-aware applications* [3, 7]. Context-aware applications are usually required to promptly react to context changes; hence, they have to be *interruptible* or support *asynchronous interruption*.

An example is a package manager application that updates packages in an operating system or a software development environment. It tends to be running for a long time because, even if only one package is selected by the user for the update, it is necessary to resolve package dependency, download archive files, unpack them, and more: the whole task takes a considerable amount of time. Examples of the interruptions are network disconnection and a press of the "cancel" button. Another example is a battery-powered robot that moves around to do some task such as cleaning rooms. Examples of the interruptions are a low battery level and sensor malfunction.

Reactions to interruptions cannot be simple. In the package manager, for example, it is not desirable just to abort the package manager promptly in response to a press of the "cancel" button because the package dependency may be broken, i.e., packages may be partially updated/installed. A desirable package manager must ensure the consistency of packages by performing some recovery actions, e.g., reverting the update by re-installing the previous versions of the packages. It may also be preferable in the case of network disconnection to suspend the execution until the connection comes back. In the robot example, a desirable reaction to a low battery level is stopping the task and returning to a base for charge.

The two examples show that, if an interruption occurs, it is necessary for context-aware applications to be able to promptly (1) abort with reverting the "effects" that comes from uncompleted tasks (file replacement in the first example and robot movement in the second example) or (2) suspend the program until we can run the continuations or reversions.

Developing context-aware applications in existing mainstream languages is difficult because of the following two problems. First, as Bainomugisha et al. indicated [7], the languages lack constructs for promptly reacting to context changes. Inserting the code for the context checks manually is not desirable from a modularity perspective. *Asynchronous exceptions* [40], which enable us to throw exceptions to other threads, could be a solution to the point. It is, however, still weak for context-aware applications because the context usually depends on multiple time-varying data and asynchronous exceptions themselves are not helpful for tidying them up.

Second, support for recovery from asynchronous interruption in the existing languages is weak. Although today's standard approach to handling interruptions is to use the exception handling constructs such as `try-catch-finally`, they are not useful for reversion and suspension; in particular, reversion is similar to resource handling with exceptions, which is hard with the constructs [60]. A more complicated and difficult reaction is *partial abort* [25], which is a combination of reversion and suspension and is realized by using *checkpoints* [48, 62, 17]. Checkpoints are useful to make applications robust [41] and avoid wasteful recomputation [14].

Our solution to the problems is based on the ideas of *Flute* [7] and *workflow* [22, 12]. Flute is a programming language originally proposed to solve the first problem. To represent the context depending on multiple time-varying data, Flute uses *functional reactive programming* (FRP) [19, 6] that represents time-varying values as streams and provides operations over them, which are useful to unify multiple sensory data into one stream. Flute also supports suspending the program execution.

Workflow [22, 12] represents a long-running interruptible transaction that consists of several atomic transactions. The typical applications are web applications and business process management, and recently workflow is adapted to context-aware applications [44, 4, 54]. One import idea of workflow for us is *compensation* [60], where each action of a program is accompanied by a compensation action, meaning a recovery action; and program execution takes account of its progress and automatically constructs its recovery action.

1.1 Contributions

In this paper, we propose a language *ContextWorkflow* as a solution to the two problems. ContextWorkflow is a workflow-based language that supports compensation, asynchronous interruption, suspension, and checkpoints. It also provides *sub-workflows* and *programmable compensations* [9, 12] that ignore and replace the compensations of completed portions of workflow, respectively.

Our approach to implementing ContextWorkflow is to embed it in other "host" languages [31]. The benefit of the approach is that the language itself remains small but can be powerful because any features of the host language are still available.

Our technical contributions are (1) a design of the workflow-based programming language with asynchronous interruption, (2) a formalization of the language, including the big-step operational semantics, (3) monadic interpreters corresponding to the semantics, and (4) an implementation of ContextWorkflow by embedding into Scala. The details are as follows.

Asynchronous Interruption in Workflow. Our approach to asynchronous interruption uses signals of FRP [19, 6] and polling [20], and our novel finding is that the idea of workflow and compensation fit with the approach. A workflow in ContextWorkflow is executed under some *context*, which changes over time asynchronously and indicates how the execution of workflow proceeds. An asynchronous interruption is detected by checking the context. We suppose that each atomic transactions should not be interrupted asynchronously; and we regard atomic transactions as a primitive construct of our language. The context is checked at the beginning of each atomic transaction similarly to transactions in database [24] and software transactional memory [53]. The difference between our workflow and the transactions is with regard to the time when a check runs. In the transactions, a check runs at the end. We also introduce constructs for blocking interruptions as in Concurrent Haskell [40] for avoiding unnecessary context checks.

Formalizing ContextWorkflow. We develop a big-step operational semantics that models the essential constructs of ContextWorkflow, that is, workflow, compensation, asynchronous interruption, sub-workflows, programmable compensations, checkpoints, and suspension. The semantics is inspired by Bruni et al.'s formalization [9] of Sagas [22], which is a foundation of workflow. Our main contribution is to add checkpoints and suspension to the existing semantics, especially considering sub-workflows. We also provide and prove basic properties of the new calculus and describe small extensions. In addition, we discuss whether the polling code should be inserted before or after an atomic transaction using the core calculus.

Monadic Interpreter. We develop two monadic interpreters in lazy and eager languages that closely correspond with the big-step operational semantics. We define two `CW` monads using the reader, exception monads and free monad transformers that represent the abstract syntax trees of ContextWorkflow programs. One could define the `CW` monad based on the free monad [5] over the compensation functor [47] that consists of the exception and continuation monads. Such a definition is, however, not desirable because it is hard to support sub-workflows, programmable compensations, and checkpoints while keeping correspondence with the big-step operational semantics straightforward. We instead use the free monad transformers to define the `CW` monads. Note that the functions that collapse, or fold, free monad transformers are different between eager and lazy languages due to efficiency and stack safety [56]. Two monads and monadic interpreters are therefore necessary.

Embedding in Scala. We carefully embed ContextWorkflow in Scala based on the monadic interpreter. In our embedding, one can throw Scala exceptions using `throw` in atomic actions and handle them using Scala's standard exception handling mechanism. We use the macro system in Scala to make the ContextWorkflow program syntax look more natural.

The rest of this paper is organized as follows. In Section 2, we informally introduce ContextWorkflow with a running example of a maze search robot. Section 3 provides a formal calculus of the core ContextWorkflow. In Section 4, we construct a monadic interpreter and show further implementation techniques in Scala. Section 5 presents related work, followed by future work and conclusion.

2 ContextWorkflow Constructs.

In this section, we look at the basic constructs of ContextWorkflow using a maze search program as a running example. Here, the notation is based on our implementation, which is an EDSL in Scala.

A program in ContextWorkflow is a workflow that is a sequence of *primitive workflows* (similar to atomic transactions). When an interruption takes place – it can only occur between primitive workflows – the whole workflow is aborted after running the compensations of the already completed primitive workflows in the reverse order, or is suspended (and the rest of the computation is returned).

2.1 Example: Explorer Robot

As a running example, we consider a battery-powered robot that explores a (physical) maze. Our goal is to program the following context-dependent behavior:

1. The robot must get back to the start or a special point equipped with a battery charger, at which the robot can recharge its battery. (We call such a special point simply a charger.)
2. When it starts to rain, the robot should suspend its exploration.

Our basic exploration strategy is to visit every place in the maze in the depth-first search (DFS) manner. We assume that the maze is represented by a graph; the graph is represented as a set of nodes, which consist of two-dimensional coordinates of integers. A node is connected to another node if and only if the distance between the two nodes is one, e.g., (1,0) and (1,1) are connected, but (1,0) and (1,2) are not. This means that if a pair of coordinates is not in the node set, there is a wall at that position. We define the class `Node` for nodes and functions as follows.

```
case class Node(loc:(Int,Int), var visited:Boolean)
def neighbors(n: Node, maze: Set[Node]): List[Node] = // getting the neighbors of n
def visited(n: Node): Unit = {n.visited = true} // setting the visited flag of n on
def unknown(n: Node): Unit = {n.visited = false} // setting the visited flag off
def move(n: Node): Unit = /* actually moving the robot to n */
def visit(n: Node, maze: Set[Node]): Unit = { // main search program
```

```
visited(n);
neighbors(n, maze).foldLeft(()){(_, neighbor) =>
if(!neighbor.visited){ move(neighbor); visit(neighbor, maze); move(n); }
} }
```

A `Node` has coordinate information `loc` and a flag `visited` that is used to remember whether the node has been visited or not. The function `neighbors` returns the neighboring nodes of a given node `n`. The functions `visited` and `unknown` mark the given node `n` as visited and unvisited, respectively. The function `move` takes a node as an argument and moves the robot to the position it represents. It works only if the robot is currently at its neighbor or the node itself. The function `visit` is the main function that must be refined as our development proceeds; it takes a node `n` and a graph `maze`, and just visits every node in `maze` from `n` recursively in a DFS manner without allowing any interruptions.

In the rest of this section, we revise `visit` using the features of ContextWorkflow. We use compensations to move the robot back; suspension to stop the robot when it starts to rain; nested workflow to skip some compensation actions; blocking constructs `atomic` and `nonatomic` to avoid redundant/unnecessary context checks; and checkpoints to stop the robot at a charger while it is getting back.

2.2 Interruptible and Compensable Workflow

To make `visit` interruptible and compensable, we change it to a sequence of primitive workflows. We write a primitive workflow, which consists of a normal action `n` and a compensation action `c`, as `n /+ c` in ContextWorkflow. Normal and compensation actions can be any Scala code (of certain types).

Each function call to `visited`, `move`, and `visit` should be lifted to a primitive workflow because it changes the “state,” i.e., the flags of nodes and the position of the robot. If an interruption occurs, the changes have to be reverted by compensations. The compensation action of each function call is basically its inverse in our example.[2] For example, we define a primitive workflow `moveFromTo` for `move` with its reverse as follows:

```
def moveFromTo(from: Node, to: Node): CW[Unit] = move(to) /+ (_ => move(from))
```

The normal action `move(to)` is of the type `Unit`, and the compensation `_ => move(from)` is of the type `Unit => Unit`; a compensation action takes the result of the corresponding normal action – which has been finished – as an argument. The whole primitive workflow is of the type `CW[Unit]` where `CW` is the class representing a workflow and means the workflow returns a value of `Unit` after its successful execution. A workflow, which is an instance of `CW[T]`, can be run by invoking `exec`, which will be explained shortly.

ContextWorkflow provides the `workflow` block and the operator `!!` to combine two or more (primitive) workflows. The `workflow` block is used to build a long workflow, and the `!!` operator is used to sequence workflows in the `workflow` block.

```
def workflow[T](body: T): CW[T]
def !![T](m: CW[T]): T
```

For example, we write like `workflow{ val x = !!(m); !!(f(x)); ...}`, where `x` becomes the result of the workflow `m`. If unnecessary, `val * =` can be omitted. This notation is

---

[2] The compensation action is not necessarily the inverse of the normal action in general. The purpose of the compensation action is to ensure the “state” is acceptable even if an interruption occurs and the program stops or rolls back.

almost the syntactic sugar of for-comprehension in Scala; e.g., the foregoing code is equal to `m.flatMap(x => f(x).flatMap(_ => ...))`. We can also use ordinary `if` for branching and `fold` (called `foldCW`) for iteration in ContextWorkflow.

```
def foldCW[A,B](l: List[A])(z: B)(f: (B,A) => CW[B]): CW[B] // fold for CW
```

Then, the interruptible version of `visit` is as follows.

```
def visit(n: Node, maze: Set[Node]): CW[Unit] = workflow {
 !!(visited(n) /+ (_ => unknown(n))) // reversible visited
 !!{foldCW(neighbors(n, maze))(()){(_, neighbor) =>
 if(!neighbor.visited) workflow{
 !!(moveFromTo(n, neighbor)) // the robot moves to the neighbor
 !!(visit(neighbor, maze))
 !!(moveFromTo(neighbor, n)) // the robot gets back to the original node n
 }
 else () /+ () } } }
```

Note that compensation actions are inverses of their corresponding normal actions.

To execute a workflow, we invoke the method `exec` of `CW` class:

```
def exec(...): \/[Option[CW[A]], A]
```

where `\/` introduces disjunctions of two types whose constructors are `-\/(l)` (meaning the left value) and `\/-(r)` (meaning the right value); `Option` is the type of optional values consisting of `Some(a)` and `None`. The type `\/[Option[CW[A]], A]` represents that the result may be abort `-\/(None)`, suspended workflow `-\/(Some(cw))` or successful execution `\/-(a)`. The argument of `exec` is optional and will be explained in detail later.

2.3 Interruption and Context

We need contexts to interrupt execution of the workflow in ContextWorkflow. A context signals how the execution of a workflow proceeds and changes over time asynchronously.

The context is represented by a stream of values of type `Context`, which can be any of `Continue`, `Abort`, `PAbort`, or `Suspend`. Their meanings are as follows:

- `Continue` continues the execution; normal actions are executed with their compensations recorded.
- `Abort` aborts the execution after executing the recorded compensations.
- `PAbort` means partial abort, which is similar to `Abort` but sensitive to checkpoints: it rolls back by executing the recorded compensations until the checkpoint most recently passed and returns the continuation at the checkpoint.
- `Suspend` suspends the execution and returns the rest of the workflow.

The current context is checked periodically (similarly to polling). More concretely, this periodic checking, called *context checking*, takes place before the execution of the normal action of each primitive workflow; if the current context is not equal to `Continue`, it is interrupted immediately.

To create a stream of `Context`s, we use a signal in the FRP library REScala [51]. For example, we can represent an interruption due to a low battery level as a signal of `Context` as follows, assuming that there is another signal `battery` indicating the battery level.

```
val battery: Signal[Int] = /* a signal indicating the battery level */
val lowbattery: Signal[Context] = Signal{ if(battery() < 20) Abort else Continue }
```

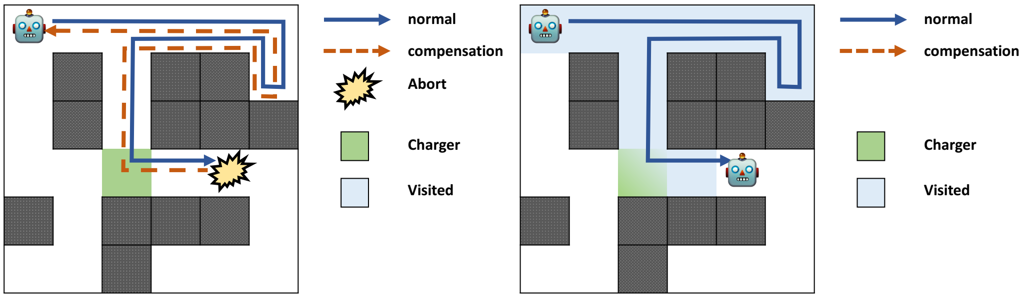

Figure 1 Maze search simulation: Abort (left) and Suspend (right).

The signal `lowbattery` is of the type `Signal[Context]`, whose value is `Continue` while the battery level is higher than 20% and `Abort` otherwise.

The context may depend on multiple sensory data. Such a context is easy to represent, owing to the expressiveness of REScala. For example, to suspend the robot when the rain starts, we need another sensory data that reflects the weather condition. It is achieved by creating another signal that relates to both the battery level and the weather.

```
val weather: Signal[Context] = Signal{ if(/* badWeather */) Suspend else Continue }
val mazectx: Signal[Context] = Signal { (lowbattery(), weather()) match {
 case (Continue, Continue) => Continue
 case (Abort, _) => Abort
 case _ => Suspend } }
```

The signal `mazectx` depends on not only `lowbattery` but also `weather`, which is another context related to the weather. Notice that we also give precedence between the two contexts here: `Abort` from `lowbattery` supersedes `Suspend` from `weather`.

To make our workflow depend on `mazectx`, we need to give it as the argument to `exec`:

```
visit(...).exec(mazectx)
```

Fig. 1 illustrates an execution of `visit`, where it is aborted (left) or suspended (right) halfway. Currently, a partial abort at the same place results in the same trace as the aborted case, since chargers (checkpoints) are not set yet.

A suspended workflow is also a workflow and we can start it by writing as follows:

```
val r = visit(...).exec(...)
sleep(/*until it is ready to resume the program*/);
r match { case -\/(Some(s)) => s.exec(...) } // restart if suspended
```

Here, `s` is the suspended workflow and its type is `CW[Unit]`.

2.4 Nested Workflow and Programmable Compensations

Sometimes we would like to skip some compensation actions. In our example, the behavior of the aborted case is not desirable because the robot follows exactly the path in which it came to the aborted point and does not come back straight to the start. A better compensation would be to take a shortcut to the start node as shown in Fig. 2.

This can be achieved by delimiting a part of a workflow and ignoring the compensation actions of the delimited part 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:

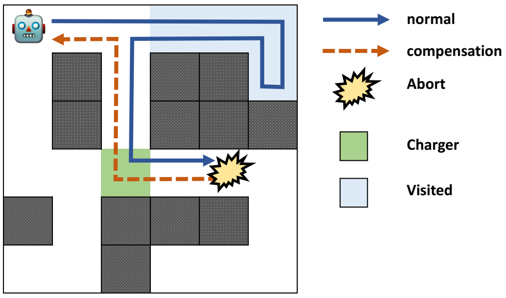

Figure 2 Maze search simulation: Abort (refined).

```
def sub[A](cw: CW[A]): CW[A]
```

We revise `visit` by using `sub` to skip undesirable compensation actions as follows:

```
def visit(n: Node, maze: Set[Node]):CW[Unit] = workflow {
 ...
 if(!neighbor.visited)
 sub{ workflow{
 !!(moveFromTo(n, neighbor))
 !!(visit(neighbor, maze))
 !!(moveFromTo(neighbor, n))
 } }  ... }
```

If a partial search from the `neighbor` is complete, compensations for it will be skipped.

It is possible to perform another compensation action instead of just skipping the compensation actions within sub-workflows by writing something like `sub(...)/+ comp`, which is the so-called programmable compensation [9, 12]. For example, we can add a log:

```
sub{ ... } /+ (_ => println("skipping compensations"))
```

2.5 Checkpoint

Using the above constructs, we still cannot realize the behavior of the robot so that it gets back to a charger. What we have to do is to let the robot *partially* roll back its move and suspend at the charger. For this purpose, we use checkpoints. A checkpoint saves the current execution state when it is passed. If a workflow is partially aborted, it executes only compensations until the checkpoint most recently passed and then suspends.

Let `Node` have another flag `hasCharger` that represents whether the node has a charger or not. We just add a `checkpoint`, which is a construct provided by ContextWorkflow, into the method `visit` as follows:

```
class Node(..., hasCharger:Boolean)
def visit(n: Node, maze: Set[Node]):CW[Unit] = workflow {
 !!(visited(n) /+ (_ => unknown(n)))
 if(n.hasCharger) !!(checkpoint) // checkpoint setting
 !!{foldCW(...)(...){...} }
}
```

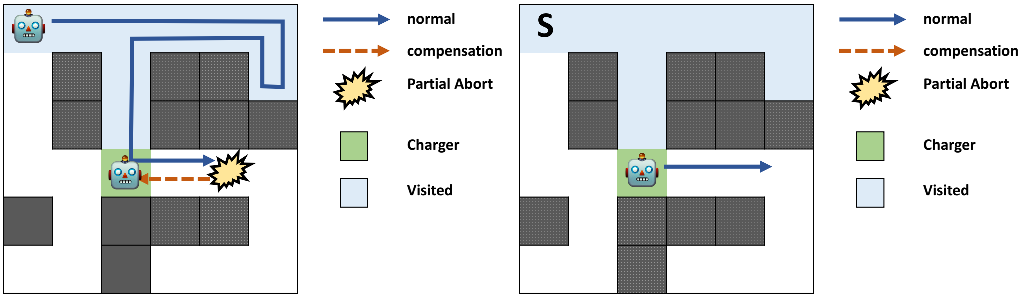

Figure 3 Maze search simulation: Partial abort (left) and its restart (right).

The left side of Fig. 3 illustrates a search being partially aborted and suspended at the checkpoint (charger). If `exec` on the suspended workflow, returned by the partial abort, is invoked, then the robot moves again from the charger (the right side of Fig. 3).

2.6 Blocking Context Checking

We would like to avoid redundant/unnecessary context checks from an efficiency perspective. In our example, it is not necessary to check the context at the beginning of (1) marking the node as visited and (2) skipping (i.e., `()/+()`) because they take little time. ContextWorkflow provides `atomic` and `nonatomic` blocks to activate and deactivate context checks.

```
def atomic[A](cw: CW[A]): CW[A]
def nonatomic[A](cw: CW[A]): CW[A]
```

An `atomic` block restrains context checking inside it, and a `nonatomic` block enforces context checking inside it. If they are nested inside each other, the innermost block takes effect.

Then, we refine the method `visit` as follows:

```
1 def visit(n: Node, maze: Set[Node]):CW[Unit] =
2  atomic{ workflow {
3  !!(visited(n) /+ (_ => unknown(n))); ...
4  !!{foldCW(neighbors(n, maze))(()){(_, neighbor) =>
5  if(!neighbor.visited)
6  nonatomic{ sub{ workflow{ ... visit(...); ... } } }
7  else () /+ () } } } }
```

By enclosing the whole workflow (except the sub-workflow) by `atomic`, context checks will not be performed on lines 3 and 7.

The purpose of `atomic` and `nonatomic` blocks is not only to improve the efficiency but also to control the atomicity of interruption. Such constructs are also very common in the languages supporting asynchronous exceptions and/or transactions; for example, Concurrent Haskell [40] has blocking constructs of asynchronous exceptions `block` and `unblock`.

3 Operational Semantics of Core ContextWorkflow

In this section, we describe the operational semantics of ContextWorkflow by formalizing a core calculus, which models compensation, checkpoints, sub-workflow, programmable compensations, and context-checking. Our calculus is inspired by Bruni et.al's formalization of Sagas [9]. Our main contribution is how to treat suspension and checkpointing considering sub-workflows, in the context of workflow languages.

$$\begin{array}{lcll}
\mathtt{t} & ::= & \mathtt{A/C} \mid \mathtt{check} \mid \mathtt{cp} \mid \mathtt{cp\#E} \mid \mathtt{sub(t)/C} \mid \mathtt{t;t} & \textit{(workflows)} \\
\mathtt{A,C} & ::= & \epsilon \mid \ldots & \textit{(atomic actions)} \\
\mathtt{c} & ::= & \mathtt{C} \mid \mathtt{sub} \mid \mathtt{ccp\#E} & \textit{(compensations)} \\
\mathtt{E} & ::= & \mathtt{[]} \mid \mathtt{E[[];t]} & \textit{(evaluation context)}
\end{array}$$

Figure 4 Syntax of core ContextWorkflow.

3.1 Syntax

We show the syntax of our calculus in Fig. 4. Meta-variable `t` ranges over context workflows; `s` ranges over contexts; `c` ranges over compensations; `A` and `C` range over atomic actions, which are commands from the underlying programming language and so not specified. (We assume only that the empty atomic action ϵ is included.) We use `A` for normal and `C` for compensation actions.

`A/C` is a primitive workflow consisting of a pair of a normal action `A` and a compensation action `C`. `sub(t)/C` is a sub-workflow with a programmable compensation; if `/C` is omitted, the empty action will be assumed. `check` is the context checking code that asks the current execution status. The reason why `check` is explicit in the syntax is to point out where context checking occurs; actually, whether `check` appears before or after a primitive workflow is significant – see Section 3.4 for discussions. `cp` is a checkpoint declaration and `cp#E`, which does not appear in the source program, is an automatically created checkpoint declaration that records an evaluation context `E`, and `cp` is replaced by `cp#E` at run time.

In compensations, `sub` is the marker that indicates the start point of a sub-workflow; `ccp#E` is a checkpoint automatically installed into a compensation sequence, where `E` is the evaluation context that is going to be executed when this checkpoint is executed.

3.2 Big-Step Semantics

In this section, we present a big-step semantics. We use overlines to denote sequences (with appropriate delimiters). For example, $\overline{\mathtt{c}}$ stands for a possibly empty sequence $\mathtt{c}_1\mathtt{;}\cdots\mathtt{;c}_n$. We also use $\overline{\mathtt{c}} \setminus \mathtt{c}$ to represent the sequence obtained by removing `c` from $\overline{\mathtt{c}}$, and similarly for other metavariables. Moreover, we use $\overrightarrow{\mathtt{A}}$ for a sequence of atomic actions excluding ϵ, e.g., $\mathtt{A}_1, \cdots, \mathtt{A}_n, \mathtt{C}_n, \cdots, \mathtt{C}_1$.

The following relations give our semantics of core ContextWorkflow:

$\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle \overline{\mathtt{c}}' \rangle$	workflow success
$\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}} \rangle \Uparrow^{\overrightarrow{\mathtt{A}}}_{A\mid P\mid S} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$	workflow interruption
$\langle \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$	compensations success
$\langle \mathtt{t}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle \rangle$	program commit
$\langle \mathtt{t}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}}_{A} \langle \rangle$	program abort
$\langle \mathtt{t}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}}_{P} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$	program partial abort
$\langle \mathtt{t}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A}}}_{S} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$	program suspend

where "$A|P|S$" means that one of these symbols (A for abort, P for partial abort, and S for suspend) comes at this position and E_s is an evaluation context. These judgments basically mean that, if the left side of $\Downarrow^{\overrightarrow{\mathtt{A}}}_{\_}$ or $\Uparrow^{\overrightarrow{\mathtt{A}}}_{\_}$ is executed, it terminates after executing $\overrightarrow{\mathtt{A}}$ and returns the right side, which is a sequence of compensation actions $\overline{\mathtt{c}}'$ possibly with a suspended computation $\mathtt{E}_s$. The first two relations are for the execution of `t` under evaluation context $\mathtt{E}$ with compensation actions $\overline{\mathtt{c}}$ recorded by past commands; the first relation is for successful execution and the second relation is for interrupted execution, where $\mathtt{E}_s$ is empty ($\mathtt{[]}$) in the case of abort A or partial abort P. The third relation is for the execution of compensation actions that are returned when a workflow is aborted or partially aborted. The last four relations are the main relations for execution of a program, which is $\mathtt{t}$ and compensation actions $\overline{\mathtt{c}}$, which are in many cases empty. If the program is committed or aborted, it returns nothing; if the program is partially aborted or suspended, then it returns compensations $\overline{\mathtt{c}}'$ and the evaluation context $\mathtt{E}_s$. The reason why a compensation sequence is also returned is that it is used when the suspended workflow restarts; in other words, if $\langle\overline{\mathtt{c}},\mathtt{E}_s\rangle$ is returned by suspension, a restart of the suspended computation can be expressed by running a program $\langle\mathtt{E}_s\mathtt{[check]},\overline{\mathtt{c}}\rangle$ – `check` means that the restart should check the context first to check if the context allows the restart.

The semantics is defined by the rules in Fig. 5; the auxiliary function *rmsub1* to forget compensations in the nearest sub-workflow is defined as follows.

$$\begin{aligned} \mathit{rmsub1}(\bullet) &= \bullet \\ \mathit{rmsub1}(\mathtt{c;}\overline{\mathtt{c}}) &= \text{if } \mathtt{c} = \mathtt{sub}\text{, then } \overline{\mathtt{c}} \text{ else } \mathit{rmsub1}(\overline{\mathtt{c}}) \end{aligned}$$

The rule CW-PW is for the primitive workflow that performs normal action `A` and adds compensation `C`. The rules CW-CHECK-* are for `check` and one of them is chosen non-deterministically. The rule CW-CHECKPOINT is for a checkpoint, which records the current continuation `E` (with symbol `ccp`) to the list of compensation actions. The hole in the evaluation context is filled with `[]; cp#E`, which means that, when the recorded continuation is executed under a different context, the original continuation is recorded (CW-CHECKPOINT-REVISIT). The rule CW-SUB is for a successful sub-workflow execution, which replaces compensations in the sub-workflow with `c`; CW-SUB-INT is for interrupted sub-workflow execution. Both rules also add `(sub [])/C` onto the stack of frames (that is, the evaluation context) before executing `t`. The rules CW-SEQ-* are for sequences, which push $\mathtt{t}_2$ on the stack of frames. The rules CW-PROGRAM-* are for program execution, where CW-PROGRAM-ABORT is to run compensations except `ccp` (represented by $\overline{\mathtt{c}}' \setminus \mathtt{ccp}$), meaning that checkpoints are simply ignored. CW-PROGRAM-PABORT performs compensations – if they include `ccp`, compensation will stop at the first `ccp` and return the evaluation context recorded there (see CW-COMP-CCP). The rules CW-COMP* are for the execution of compensations.

An example of workflow execution is shown as follows. The derivation tree for this relation is given in Appendix A.2.

$$\begin{gathered} \langle\mathtt{sub\{sub\{t_1;cp;sub(t_2)/C}_a\mathtt{;check\}/C}_b\mathtt{;t_3\};t_4},\bullet\rangle \Downarrow_P^{\mathtt{A_1,A_2,C}_a} \\ \langle\mathtt{C_1;sub;sub},\mathtt{sub(sub([];cp\#E_1;sub(t_2)/C}_a\mathtt{;check)/C}_b\mathtt{;t_3);t_4}\rangle \end{gathered}$$

where $\mathtt{t}_k = \mathtt{A}_k\mathtt{/C}_k$ $(1 \le k \le 4)$ and $\mathtt{E_1} = \mathtt{sub(sub([];sub(t_2)/C}_a\mathtt{;check)/C}_b\mathtt{;t_3);t_4}$.

This is an example of partial abort at the `check`; hence, an evaluation context and compensations are returned. If we would like to restart the suspended workflow, we give `check` (or ϵ/ϵ, if the initial check can be omitted) to the evaluation context. Then, restarting it may perform normal actions $\mathtt{A}_2$, $\mathtt{A}_3$, and $\mathtt{A}_4$ and terminate. In other words, the relation below can be derived.

$$\langle\mathtt{sub(sub(check;cp\#E_1;sub(t_2)/C}_a\mathtt{;check)/C}_b\mathtt{;t_3);t_4},\mathtt{C_1;sub;sub}\rangle \Downarrow^{\mathtt{A_2,A_3,A_4}} \langle\rangle$$

$$\frac{}{\langle\texttt{A/C},\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\texttt{A}} \langle\texttt{C;}\overline{\texttt{c}}\rangle}\qquad\text{(CW-PW)}$$

$$\frac{}{\langle\texttt{check},\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\epsilon} \langle\overline{\texttt{c}}\rangle}\qquad\text{(CW-Check-Cont)}$$

$$\frac{}{\langle\texttt{check},\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\epsilon}_{S} \langle\overline{\texttt{c}},\texttt{E}\rangle}\qquad\text{(CW-Check-Suspend)}$$

$$\frac{}{\langle\texttt{check},\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\epsilon}_{A} \langle\overline{\texttt{c}},\texttt{[]}\rangle}\qquad\text{(CW-Check-Abort)}$$

$$\frac{}{\langle\texttt{check},\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\epsilon}_{P} \langle\overline{\texttt{c}},\texttt{[]}\rangle}\qquad\text{(CW-Check-PAbort)}$$

$$\frac{}{\langle\texttt{cp},\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\epsilon} \langle\texttt{ccp\#E[[];cp\#E];}\overline{\texttt{c}}\rangle}\qquad\text{(CW-Checkpoint)}$$

$$\frac{}{\langle\texttt{cp\#E}_0,\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\epsilon} \langle\texttt{ccp\#E}_0\texttt{[[];cp\#E}_0\texttt{];}\overline{\texttt{c}}\rangle}\qquad\text{(CW-Checkpoint-Revisit)}$$

$$\frac{\langle\texttt{t},\texttt{E[(sub []\,)/C]},\texttt{sub;}\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}}} \langle\overline{\texttt{c}}'\rangle}{\langle\texttt{sub(t)/C},\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}}} \langle\texttt{C;}\mathit{rmsub1}(\overline{\texttt{c}}')\rangle}\qquad\text{(CW-Sub)}$$

$$\frac{\langle\texttt{t},\texttt{E[(sub []\,)/C]},\texttt{sub;}\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{*} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}{\langle\texttt{sub(t)/C},\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{*} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}\qquad\text{(CW-Sub-Int)}$$

$$\frac{\langle\texttt{t}_1,\texttt{E[[];t}_2\texttt{]},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}_1}} \langle\overline{\texttt{c}}'\rangle \qquad \langle\texttt{t}_2,\texttt{E},\overline{\texttt{c}}'\rangle \Downarrow^{\overrightarrow{\texttt{A}_2}} \langle\overline{\texttt{c}}''\rangle}{\langle\texttt{t}_1\texttt{;t}_2,\texttt{E},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}_1};\overrightarrow{\texttt{A}_2}} \langle\overline{\texttt{c}}''\rangle}\qquad\text{(CW-Seq)}$$

$$\frac{\langle\texttt{t}_1,\texttt{E[[];t}_2\texttt{]},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{*} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}{\langle\texttt{t}_1\texttt{;t}_2,\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{*} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}\qquad\text{(CW-Seq-Int1)}$$

$$\frac{\langle\texttt{t}_1,\texttt{E[[];t}_2\texttt{]},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}_1}} \langle\overline{\texttt{c}}'\rangle \qquad \langle\texttt{t}_2,\texttt{E},\overline{\texttt{c}}'\rangle \Uparrow^{\overrightarrow{\texttt{A}_2}}_{*} \langle\overline{\texttt{c}}'',\texttt{E}_s\rangle}{\langle\texttt{t}_1\texttt{;t}_2,\texttt{E},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}_1};\overrightarrow{\texttt{A}_2}}_{*} \langle\overline{\texttt{c}}'',\texttt{E}_s\rangle}\qquad\text{(CW-Seq-Int2)}$$

$$\frac{\langle\texttt{t},\texttt{[]},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}}} \langle\bullet\rangle}{\langle\texttt{t},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}}} \langle\rangle}\qquad\text{(CW-Program-Commit)}$$

$$\frac{\langle\texttt{t},\texttt{[]},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{A} \langle\overline{\texttt{c}}',\texttt{[]}\rangle \qquad \langle\overline{\texttt{c}}'\setminus\texttt{ccp}\rangle \Downarrow^{\overrightarrow{\texttt{C}}} \langle\bullet,\texttt{[]}\rangle}{\langle\texttt{t},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}};\overrightarrow{\texttt{C}}}_{A} \langle\rangle}\qquad\text{(CW-Program-Abort)}$$

$$\frac{\langle\texttt{t},\texttt{[]},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{P} \langle\overline{\texttt{c}}',\texttt{[]}\rangle \qquad \langle\overline{\texttt{c}}'\rangle \Downarrow^{\overrightarrow{\texttt{C}}} \langle\overline{\texttt{c}}'',\texttt{E}_s\rangle}{\langle\texttt{t},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}};\overrightarrow{\texttt{C}}}_{P} \langle\overline{\texttt{c}}'',\texttt{E}_s\rangle}\qquad\text{(CW-Program-PAbort)}$$

$$\frac{\langle\texttt{t},\texttt{[]},\overline{\texttt{c}}\rangle \Uparrow^{\overrightarrow{\texttt{A}}}_{S} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}{\langle\texttt{t},\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{A}}}_{S} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}\qquad\text{(CW-Program-Suspend)}$$

$$\boxed{\text{compensation}}$$

$$\frac{}{\langle\texttt{C}\rangle \Downarrow^{\texttt{C}} \langle\bullet,\texttt{[]}\rangle}\qquad\text{(CW-Comp-Action)}$$

$$\frac{}{\langle\texttt{sub}\rangle \Downarrow^{\epsilon} \langle\bullet,\texttt{[]}\rangle}\qquad\text{(CW-Comp-Sub)}$$

$$\frac{\langle\texttt{c}\rangle \Downarrow^{\overrightarrow{\texttt{C}}} \langle\bullet,\texttt{[]}\rangle \qquad \langle\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{C}'}} \langle\bullet,\texttt{[]}\rangle}{\langle\texttt{c;}\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{C}};\overrightarrow{\texttt{C}'}} \langle\bullet,\texttt{[]}\rangle}\qquad\text{(CW-Comp-Seq)}$$

$$\frac{}{\langle\texttt{ccp\#E}\rangle \Downarrow^{\epsilon} \langle\bullet,\texttt{E}\rangle}\qquad\text{(CW-Comp-Ccp)}$$

$$\frac{\texttt{E}_s \neq \texttt{[]} \qquad \langle\texttt{c}\rangle \Downarrow^{\overrightarrow{\texttt{C}}} \langle\bullet,\texttt{E}_s\rangle}{\langle\texttt{c;}\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{C}}} \langle\overline{\texttt{c}},\texttt{E}_s\rangle}\qquad\text{(CW-Comp-Seq-PAbort1)}$$

$$\frac{E_s \neq \texttt{[]} \qquad \langle\texttt{c}\rangle \Downarrow^{\overrightarrow{\texttt{C}_1}} \langle\bullet,\texttt{[]}\rangle \qquad \langle\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{C}_2}} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}{\langle\texttt{c;}\overline{\texttt{c}}\rangle \Downarrow^{\overrightarrow{\texttt{C}_1};\overrightarrow{\texttt{C}_2}} \langle\overline{\texttt{c}}',\texttt{E}_s\rangle}\qquad\text{(CW-Comp-Seq-PAbort2)}$$

Figure 5 Big step semantics of core ContextWorkflow.

3.3 Properties

Here, we state some properties that hold of the semantics. The main aim of this section is to rigorously give the specification to the language. In particular, giving specifications about suspension and partial aborts (checkpoints) is important since these are unusual in the context of workflow languages.

In the following theorems, let $\mathtt{p}_k = \mathtt{A}_k/\mathtt{C}_k$ for some k, and we define a function $\flat(\mathtt{t})$ and predicates *includes* and *nosub* as follows.

- Let $\flat(\mathtt{t})$ be a workflow obtained from `t` by removing `sub`, `check`, `cp` and `cp#E` from `t`.
- $\mathit{includes}(\mathtt{t}, m, n)$ iff $\flat(\mathtt{t}) = \mathtt{p}_m\,\mathtt{;}\cdots\mathtt{;}\mathtt{p}_n$ and $m \leq n$; or `t` has no primitive workflows and $m \not\leq n$.
- $\mathit{includes}(\mathtt{E}, m, n) = \mathit{includes}(\mathtt{E[check]}, m, n)$.
- $\mathit{includes}(\overline{\mathtt{c}}, m, n)$ iff $\overline{\mathtt{c}} \setminus \{\mathtt{sub}, \mathtt{ccp\#E}\} = \mathtt{C}_m\,\mathtt{;}\ldots\mathtt{;}\mathtt{C}_n$ and $m \geq n$; or $\overline{\mathtt{c}}$ has no atomic actions $\mathtt{C}_*$ and $m \not\leq n$.
- $\mathit{nosub}(\mathtt{t}, m, n)$ iff $\mathit{includes}(t, m, n)$ and `t` has no `sub`-workflow.

Theorems 1 and 2 state about the behaviors under contexts `Continue` and `Abort`. These are the basic properties of Sagas [9].

▶ **Theorem 1** (Workflow commits). *If* $\mathit{includes}(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \bullet \rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle\rangle$ *and* $m \leq n$*, then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots \mathtt{A}_n$.

▶ **Theorem 2** (Workflow aborts (Successful Compensation)). *If* $\mathit{nosub}(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \bullet \rangle \Downarrow^{\overrightarrow{\mathtt{A}}}_{A} \langle\rangle$ *and* $m \leq n$*, then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_m$ *for some* i $(m \leq i \leq n)$*, or* $\overrightarrow{\mathtt{A}} = \epsilon$.

Theorem 3 states that, even though a workflow is suspended in the middle by `Suspend`, the resulting normal actions after its final commit are always the same. Therefore, it ensures that a suspended workflow actually continues from the suspension point.

▶ **Theorem 3** (Restarted suspended workflow commits). *If* $\langle \mathtt{t}, \bullet \rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle\rangle$ *and* $\langle \mathtt{t}, \bullet \rangle \Downarrow^{\overrightarrow{\mathtt{A'}}}_{S} \langle \overline{\mathtt{c}}, \mathtt{E} \rangle$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A''}}} \langle\rangle$*, then* $\overrightarrow{\mathtt{A}} = \overrightarrow{\mathtt{A'}}, \overrightarrow{\mathtt{A''}}$.

Theorem 4 states that if a workflow suspends at a checkpoint by `PAbort`, it surely did compensations corresponding to completed normal actions successive to the checkpoint; moreover, the suspended workflow actually points to the continuation from the checkpoint.

▶ **Theorem 4** (Workflow partially aborts). *If* $\mathit{nosub}(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \bullet \rangle \Downarrow^{\overrightarrow{\mathtt{A}}}_{P} \langle \overline{\mathtt{c}}, \mathtt{E} \rangle$ *and* $m \leq n$*, then either of the followings hold.*

- $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_i, \mathtt{C}_i, \mathtt{C}_{i-1}, \cdots, \mathtt{C}_j$ *and* $\mathit{includes}(\mathtt{E}, j, n)$ *and* $\mathit{includes}(\overline{\mathtt{c}}, j-1, m)$ *for some* i *and* j $(m \leq j \leq i \leq n)$*.*
- $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_n$ *and* $\mathit{includes}(\mathtt{E}, 1, 0)$ *and* $\mathit{includes}(\overline{\mathtt{c}}, n, m)$*.*

Moreover, the following conditions hold.

1. *(Suspended workflow commits) If* $\langle \mathtt{E[check]}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A'}}} \langle\rangle$*, then* $\overrightarrow{\mathtt{A'}} = \mathtt{A}_j, \cdots, \mathtt{A}_n$*, or* $\overrightarrow{\mathtt{A'}} = \epsilon$ *(if* $\mathit{includes}(\mathtt{E}, 1, 0)$*).*
2. *(Suspended workflow aborts) If* $\langle \mathtt{E[check]}, \overline{\mathtt{c}} \rangle \Downarrow^{\overrightarrow{\mathtt{A'}}}_{A} \langle\rangle$*, then* $\overrightarrow{\mathtt{A'}} = \mathtt{A}_j, \cdots \mathtt{A}_k, \mathtt{C}_k, \cdots, \mathtt{C}_m$ *for some* k $(j \leq k \leq n)$ *or* $\overrightarrow{\mathtt{A'}} = \mathtt{C}_{j-1}, \cdots, \mathtt{C}_m$*.*

Theorem 5 provides the properties about a complex workflow including a sub-workflow, checkpoints and `PAbort`; it describes that a completed sub-workflow is skipped at the compensation time and a suspended workflow remembers the original program structure including checkpoints and the sub-workflow.

▶ **Theorem 5** (Partial abort, checkpoint and nested workflow). *Suppose that includes(*`t`*,1,n) and* `t` *without* `check` *is*
$\mathtt{p}_1\mathtt{;}\ldots\mathtt{;cp;p}_k\mathtt{;}\ldots\mathtt{;p}_m\mathtt{;sub(p}_{m+1}\mathtt{;}\ldots\mathtt{;cp;p}_j\mathtt{;}\ldots\mathtt{;p}_l\mathtt{)/C}_a\mathtt{;p}_{l+1}\mathtt{;}\ldots\mathtt{;p}_n$ *and* $\langle \mathtt{t}, \bullet\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}}} \langle \overline{\mathtt{c}}, \mathtt{E}\rangle$.

1. *(Partial abort skips compensations of complete sub-workflows) If* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}}\}$, *then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_1, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_{l+1}, \mathtt{C}_a, \mathtt{C}_m, \cdots, \mathtt{C}_k$ *for some* $i > l$.
2. *(A suspended workflow remembers checkpoints in complete sub-workflows) If* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}}\}$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}'\rangle$ *and* $\mathtt{A}_j \in \{\overrightarrow{\mathtt{A}'}\} \wedge \mathtt{A}_l \notin \{\overrightarrow{\mathtt{A}'}\}$, *then* $\overrightarrow{\mathtt{A}'} = \mathtt{A}_k, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_j$ *for some* i *such that* $(j \le i \le l)$.
3. *(A suspended workflow remembers checkpoints before a sub-workflow) If* $\mathtt{C}_j \in \{\overrightarrow{\mathtt{A}}\}$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}'\rangle$ *and* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}'}\}$, *then* $\overrightarrow{\mathtt{A}'} = \mathtt{A}_j, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_{l+1}, \mathtt{C}_a, \mathtt{C}_m, \cdots, \mathtt{C}_k$ *for some* $i > l$.

For the robot example, the first and the third items of Theorem 5 are significant; otherwise, the robot would move back to the whole path at the compensation time, and forget checkpoints. The second item is important in an example that needs re-calculation of a complete sub-workflow.

3.4 Discussion

Design Choice of Primitive Workflow with Context Checking. Although `A/C` is the primitive workflow in the calculus, it does not appear explicitly in the DSL. We regard `A/C` preceded by `check` as a primitive workflow and give another notation `A /+ C` in the DSL, representing asynchronous interruption. Actually, another interpretation of `A /+ C` would be to put `check` *after* `A/C`. The difference between these interpretations becomes clear when executing a sub-workflow. Let $\mathtt{t}_k = \mathtt{A}_k\mathtt{/+C}_k$ for $k = 1, 2$. Then, when we execute $\mathtt{sub(t}_1\mathtt{;t}_2\mathtt{)}$, is it possible that the resulting action sequence $\mathtt{A}_1\mathtt{,A}_2\mathtt{,C}_2\mathtt{,C}_1$ appears? In the former choice (where `A/+C` is `check; A/C`), such a result never occurs – possible sequences of actions are only •, "$\mathtt{A}_1\mathtt{,C}_1$", or "$\mathtt{A}_1\mathtt{,A}_2$" – while it may in the latter.

The former choice is looser than the latter in the sense that the whole execution may commit after the execution though context checking actually occurs during the execution of an atomic action. Such a behavior is critical in cases where an atomic action must be performed in the `Continue` context. For example, suppose that a workflow contains an atomic action to download something and the context relates to network availability; then the atomic action must commit only at the time when it is executed in the `Continue` context; otherwise, the downloaded file would be incomplete. Therefore, we can regard the latter choice as transactions.

Since we suppose that many context-aware applications such as robots are not strict, in our implementation, we adopt the former choice by default. Fortunately, we can switch between both semantics easily.

Atomic and nonatomic blocks. It is easy to extend with `atomic` and `nonatomic`. Their semantics is similar to sub-workflows and they basically control non-determinism in `check`.

Abnormal termination. We can consider *abnormal termination* [9], a stronger notion of abort that occurs when an atomic action (or a compensation action) fails without even performing any compensation. Though we do not include abnormal termination here, it is not difficult to add it; it is enough to add nondeterminism to rules CW-PW and CW-COMP-ACTION and the other relation for the abnormal signal. Later, we implement abnormal termination in the E-DSL, by using exceptions in Scala.

Differences with respect to the calculus [9]. Here, we describe the differences with respect to the existing calculus [9], by which ours are inspired.

- Ours adds the notions of checkpoint, partial abort, and suspension. Technically, our semantics introduces evaluation contexts in order to capture continuations of workflow executions.
- Ours omits abnormal termination and does not model parallelism.
- In ours, an abort inside a nested workflow results in an abort of the parent workflow. Although this design choice is not usual [12] (where our choice is referred to as *upward abortion propagation*), we intend that an abort signal means it is signaled to the whole workflow, because the workflow is executed on a single thread.

4 Monadic embedding to Scala

Our approach to implementing ContextWorkflow is to embed the language into another language. We use a free monad transformer for representing and building the abstract syntax trees and define a monadic interpreter that follows the semantics in Section 3.

There are two differences between the core calculus and the embedding, though they closely correspond with each other. First, the `sub`-block is represented by two marks in the embedding, to indicate the beginning and the end of a block. Second, the semantics of `check` is deterministic in the embedding while it is nondeterministic in the core calculus. Our interpreter checks the context when evaluating `check` and chooses one branch. We represent the context by a stream of `Context`, which is essentially the same as the signal of `Context` in Section 2.

The underlying monad of our free monad transformer is a combination of an exception monad and a reader monad. The exception monad represents aborts, partial aborts, and suspensions. The reader monad keeps the context that is checked when `check` is evaluated. In other words, we develop ContextWorkflow on top of a monadic language that supports exceptions and readable environments. The monadic interpreter translates ContextWorkflow programs to monadic programs.

The main contribution of this section is (1) a simple implementation, i.e., clear correspondence between the semantics and implementation, and (2) efficiency in eager languages. A naive approach would be to extend the compensation monad [47], but it is hard to make such an extension simple. See Section 5 for a detailed discussion.

We use Scala as the language for demonstration and explanation. Although our implementation in Scala heavily relies on scalaz [1], we here show language/library-independent definitions for comprehensibility and generality.

4.1 Free monad transformers

This section gives a brief introduction to the free monad transformers along with the basic definitions and notations for monadic programming in Scala. Readers who would like to learn about monads and monadic programming are referred to other papers [43, 58]. Most of the definitions are simplified; although scalaz uses implicit conversions to use objects as functors and monads, here we define functors and monads using simple inheritance.

A free monad transformer `FreeT[F,M,_]` is a monad that is freely constructed from the given functor `F` and underlying monad `M`. One can understand free monad transformers as abstract syntax trees and therefore the functor `F` defines the “commands” of the language. The difference from free monads is that the nodes are some computations of which semantics is given by the underlying monad.

Functors and monads are defined by the traits `Functor` and `Monad`, respectively. Free monad transformers are defined by the abstract class `FreeT`. `Functor` provides `map` (`fmap` in Haskell) and `Monad` provides `flatMap` (`>>=` in Haskell) and `point` (`return` in Haskell).

```
trait Functor[F[_]]{ def map[A,B](f: A => B): F[B] }
trait Monad[M[_]] extends Functor[M]{
 def flatMap[A,B](f: A => M[B]): M[B]
 def point[A](a: => A): M[A] }
```

Because `Monad` provides `flatMap`, we can use the `for`-comprehension in Scala, similarly to the `do`-notation in Haskell. For example, for values `m1` and `m2` of the type monad `M`, the code

```
for{ a <- m1 ; b <- m2 } yield a + b
```

is equivalent to the following code.

```
m1.flatMap(a => m2.map(b => a + b))
```

`FreeT` is defined using the auxiliary trait `FreeF` and provides the two functions `iterT` and `interpretS`.[3] Intuitively, `FreeT` is a list-like structure and `iterT` works as `foldr` over lists. `interpretS` replaces the "commands" of the language with other "commands."

```
class FreeT[F[_],M[_],A](run: M[FreeF[F,A,FreeT[F,M,A]]])
 extends Monad[FreeT[F,M,?]]{
 def iterT(interp: F[M[A]] => M[A]): M[A]
 def interpretS[G[_]](st: ~>[F,G]): FreeT[G,M,A]
}
```

`iterT` takes an interpretation of "commands" and translates a "program" of type `FreeT[F,M,A]` to that of `M[A]`. `interpretS` takes a natural transformation from the functor `F` to another functor `G` and translates a "program" of type `FreeT[F,M,A]` into that of type `FreeT[G,M,A]`. The question mark `?` in a type parameter means that a surrounding expression is a type-level anonymous function, e.g., `M[A,?]` takes one type parameter and `M[A,?,?]` takes two.[4]

The trait `FreeF` takes three types `F`, `A`, and `B` and has two constructions `Pure` and `Free`. `F` is the functor that defines "commands." `Pure` lifts a pure value of type `A` to the "program" represented by the free monad transformer. `Free` lifts a "command" followed by a computation of type `B` to the "program."

```
trait FreeF[F[_],A,B]
case class Pure[F[_],A,B](a:A) extends FreeF[F,A,B]
case class Free[F[_],A,B](fb: F[B]) extends FreeF[F,A,B]
```

4.2 ContextWorkflow Monad

The ContextWorkflow monad `CW` is a free monad transformer defined as:[5]

---

[3] Here we borrow `iterT` from the free package of Haskell. Although `iterT` can be defined in Scala, it is not good in practice. We will visit the problem in Sec 4.6.

[4] This feature is enabled by kind-projector (`https://github.com/non/kind-projector`).

[5] Again, the definition is simplified from the actual definition just for avoiding unnecessary complexity of implicit conversions.

```
case class CW[E,M[_],S,A](
 run: FreeT[CWT[M,S,?], EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]],?], A])
extends Monad[CW[E,M,S,?]] { /* map, point and flatMap */ }
```

The type parameter `E` is for the exception type; `M` is for the monad that represents effects in the atomic actions; `S` is for the suspended workflow type (explained later); and `A` is for the successful result value type. `Sig` is the type of the context, which is just an alias of `Stream[Context]`. A `Context` is either `Continue`, `Abort`, `PAbort` or `Suspend`, which are `objects` that extend `Context`. `EV` is the type of exceptional values that consists of the compensation actions to be executed and the suspended workflow. `InSubL` keeps track of the depth of the `sub`-block to skip compensation actions. We call `EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]], ?]` the underlying monad of `CW[E,M,S,A]` in the rest of the paper.

`CWT` represents the "commands" of ContextWorkflow. Concrete commands and `CWT` are defined as follows.

```
trait CWT[M[_],S,A] extends Functor[CWT[M,S,?]] { /* map */ }
case class Comp[M[_],S,A](comp:M[Unit], a:A) extends CWT[M,S,A]
case class SubB[M[_],S,A](a:A) extends CWT[M,S,A]
case class SubE[M[_],S,A](a:A) extends CWT[M,S,A]
case class Cp[M[_],S,A](a:A) extends CWT[M,S,A]
case class Cpn[M[_],S,A](s:S,a:A) extends CWT[M,S,A]
case class Check[M[_],S,A](a:A) extends CWT[M,S,A]
```

`M` is a monad for atomic actions; `S` is the type of a suspended workflow that corresponds to the evaluation contexts in the calculus. `Comp` is for specifying compensation action. `SubB` and `SubE` are the beginning and end marks of a `sub`-block, respectively. `Cp` and `Cpn` are checkpoints that correspond to `cp` and `cp#E` in the calculus, respectively. `Cpn` has a suspended workflow, which corresponds to the fact that `cp#E` has an evaluation context `E`. `Check` corresponds to `check` in the calculus.

One may wonder why we do not have a command for normal actions while we have one for compensation actions. This is because the normal actions of type `M[A]` are handled by the underlying monad `EitherT[ReaderT[M,...],...]` of the free monad transformer.

The exception type `EV` consists of three constructors as follows:

```
sealed trait EV[ME,S]
case class Aborting[ME,S](e:ME) extends EV[ME,S]
case class Suspending[ME,S](s:S) extends EV[ME,S]
case class PAborting[ME,S](s:Option[S],e:ME) extends EV[ME,S]
```

The type parameter `ME` is for the type of compensation actions. `Aborting` represents that the workflow is aborted. The field `e` keeps the compensations to be executed. `Suspending` represents that the workflow is suspended. The field `s` keeps the suspended workflow of type `S`. `PAborting` represents that the workflow is partially aborted. The suspended workflow `s` is optional because a workflow may not have a checkpoint and in that case, there is no suspended workflow.

`InSubL` represents whether the workflow execution is in the sub-block or not.

```
sealed trait InSubL[A]
case class InSub[A](n:InSubL[A]) extends InSubL[A]
case class NonSub[A](a:A) extends InSubL[A]
```

`InSub` and `NonSub` represent that the workflow execution is in a sub-workflow and not, respectively. Notice that only executions of compensation actions are changed by sub-workflows and programmable compensations. It is therefore sufficient to wrap only the exceptional values propagated backwards with `InSubL`.

Readers may wonder what `CW[A]` that appeared in Section 2 is. This abbreviates `CW[Unit,IO,Nothing,A]`; see Appendix A.1 for further details.

4.3 Auxiliary Definitions

This section gives the auxiliary functions and macros that correspond to the syntax for the users of ContextWorkflow. For readability and simplicity, we omit the type and implicit arguments of method invocations necessary to compile if they are clear from the context.

The functions `check` and `checkpoint` correspond to `check` and `cp` in the calculus, respectively.

```
def check[E,M[_],S]: CW[E,M,S,Unit] = CW(liftF(Check(())))
def checkpoint[E,M[_],S]: CW[E,M,S,Unit] = CW(liftF(Cp(())))
```

`liftF` lifts the objects of type `F[A]` for any functor `F` and type `A` to a free monad transformer `FreeT[F,M,A]` for any monad `M`.

The primitive workflow `A/C` in the calculus is written as `compL(A,C)` where `compL` is an auxiliary function defined as follows:

```
def compL[E,M[_],S,A](na:M[A])(ca:A => M[Unit]): CW[E,M,S,A] = CW{
 na.liftM.liftM.liftM.flatMap(x => liftF(Comp(ca(x),x))) }
```

`liftM` lifts the monadic values of type `G[A]` to another monadic value of type `H[G,A]` where `G` and `H` are a monad and a monad transformer, respectively. We also define another auxiliary function `/+` that corresponds to `check;A/C` [6].

```
def /+[E,M[_],S,A](na:M[A])(ca:A => M[Unit]): CW[E,M,S,A] =
 check.flatMap(_ => compL(na)(ca))
```

For the programmable compensations and sub-workflows, we define the two auxiliary functions `subC` and `sub`, respectively. `subC` takes a workflow and a compensation and `sub` takes only a workflow. `sub` concatenates the beginning mark of the block, the given workflow, and the end mark of the block. `subC` additionally concatenates the `sub`-workflow created from the given workflow and the given compensation action.

```
def sub[E,M[_],S,A](cw :CW[E,M,S,A]): CW[E,M,S,A] = CW{ for{
 _ <- liftF(SubB(()))
 r <- cw.run
 _ <- liftF(SubE(()))
 } yield r }
def subC[E,M[_],S,A](cw :CW[E,M,S,A])(ca :A => M[Unit]): CW[E,M,S,A] = CW{
 sub(cw).flatMap(r => liftF(Comp(ca(r),r))) }
```

We also define two macros `!!` and `workflow` using the Monadless [2] library. The macro `!!` takes a workflow and escapes it from the program transformation. The macro `workflow` works as a block that specifies the target area of the program transformation. Assignments

---

[6] Though omitted here, to regard `/+` as an infix operator, we have to define it using implicit conversions in Scala.

and sequential compositions in `workflow` are transformed into a chain of monadic binds. For example,

```
workflow { val x = !!(w1); val y = !!(w2); x + y }
```

is transformed into

```
w1.flatMap(x => w2.map(y => x + y))
```

4.4 Types of Suspended Workflows

Before showing the monadic interpreter for the `CW` monad, we need to fix the type of the suspended workflows. Clearly, it must be equal to the type of the workflow to be executed, i.e., `S` in `CW[E,M,S,A]` must be again `CW[E,M,S,A]`. This means that `S` is a fixpoint of the functor `CW[E,M,?,A]` [23, 45]. The data type `Fix` is parameterized over functors

```
case class Fix[F[_]](out: F[Fix[F]])
```

and the type of suspended workflows is represented as `Fix[CW[E,M,?,A]]`.

4.5 Monadic interpreter

Our monadic interpreter of the `CW` language is the function `runCWT` from, for any monad `M` and type `A`, `CW[Unit,M,Fix[CW[Unit,M,?,A]],A]`, which is equal to `Fix[CW[Unit,M,?,A]]`, to `MM[A]` where `MM` is the underlying monad defined as follows.

```
def runCWT[M[_],A](s: Fix[CW[Unit,M,?,A]])
: EitherT[ReaderT[M,Sig,?],InSubL[EV[M[Unit],Fix[CW[Unit,M,?,A]]]],A] = {
 type S = Fix[CW[Unit, M, ?, A]] // the type of suspended workflows
 type R = EV[M[Unit], S] // the type of exceptional results
 type F[X] = CWT[M, S, X] // the term functor
 type MM[X] = EitherT[ReaderT[M, Sig, ?], InSubL[R], X] // the underlying monad

 def runCWT0(cl: F[MM[A]]): MM[A] = cl match{
 case Comp(c, k) => ...
 ...
 }
 s.out.run.iterT(runCWT0)
}
```

The function `runCWT0` translates each command of the `CW` language defined by `CWT` to the program of the language given by the underlying monad `MM`. Because the translation proceeds from the last terms to the first terms by `iterT`, each command object has the subsequent translated program. In other words, the result of the rest of the workflow is always available.

The interpretation of `Check` follows CW-Check-*. It installs a context check to the resulting program. If the context is `Continue`, it returns the result of the subsequent program. It otherwise throws exceptions. Note that the exceptions are just the values of type `EitherT[...]`, that is the underlying monad, and we do not use the exception handling mechanism of Scala.

```
case Check(k) => { // k: EitherT[ReaderT[M, Sig, ?], InSubL[R], A]
 ask.liftM.flatMap{ sig =>
 sig.head match {
 case Abort => raiseException(InSubL.point(Aborting(M.point(()))))
```

```
 case PAbort => raiseException(InSubL.point(PAborting(None, M.point(()))))
 case Suspend => raiseException(InSubL.point(Suspending(
 Fix(CW(FreeT.roll(Check(k.liftM)))))) // creates the suspended workflow
 case Continue => local(_.tail)(k)
 }}}
```

`k` is the interpretation of the subsequent workflow. The method `ask` gets a value from the environment. In our case, they are the context that is represented by the streams of type `Stream[Context]`. The variable `sig` is bound to a stream. If the head, which represents the current context, is `Abort`, `Aborting` of `point` of the unit value is thrown. This is because there is no compensation to be executed at this point. If the current context is `PAbort`, `PAborting` of `None` and `point` of the unit value is thrown. If the current context is `Suspend`, we throw the translated program `k` as the suspended workflow. If the current context is `Continue`, we drop the head of the stream and continue interpreting the workflow.

The interpretation of `Comp` corresponds to CW-Seq-Int-*, CW-Program-*, CW-Comp-Action and CW-Comp-Seq-*. The parameters `comp` and `k` are the compensation action and the interpretation of the rest of the workflow, respectively.

```
case Comp(comp, k) => EitherT {
 k.run.map{ ev => ev match {
 case \/-(_) => ev // successful execution
 case -\/(err) =>
 extendSuspending(liftF(Comp(c, ())))(err) match { // at compensation
 case NonSub(p) => p match { // binding compensation
 case Aborting(cp) => \/.left(NonSub(Aborting(cp.flatMap(res => comp.flatMap(_ => M
     .point(res))))))
 case PAborting(None, cp) => \/.left(NonSub(PAborting(None, cp.flatMap(res => comp.
     flatMap(_ => M.point(res))))))
 case Suspending(sp) => \/.left(NonSub(Suspending(sp)))
 }
 case x => \/.left(x) // skipping compensation of a complete sub-workflow
 }}}}
```

If the result of the subsequent workflow is an exception, the interpreter adds the compensation command `Comp(c, ())` at the head of the suspended workflow in `err` by `extendSuspending`. Following the operational semantics, we skip the compensation actions that (1) are in sub-workflows and (2) are followed by a checkpoint that is not in any sub-workflow and the execution is partially aborted after executing the checkpoint. The first condition is represented by `InSubL`. The last condition is represented by `Option`.

Following CW-Checkpoint and CW-Comp-Ccp, the interpretation of `Cp` (1) puts the command represented by `Cpn` at the head of the suspended workflow and (2) puts a suspended workflow to the exception if it is of type `PAborting`. The suspended workflow that corresponds to `E` of `cp#E` and `ccp#E` is just the argument of `Cp`.

```
case Cp(k) => EitherT {
 k.run.map{r => r match {
 case \/-(_) => r
 case -\/(err) => {
 val s = Fix(CW(k.liftM))
 val kp = liftF(Cpn(s, ())) // creates Cpn that is substituted for the Cp
 \/.left(setPAbort(s)(extendSuspending(kp)(err))) // set pabort with suspension
 }}}}
```

`s` is the suspended workflow. The function `setPAbort` merely replaces the first parameter of `PAborting` with `s` if it is `None`. The interpretation of `Cpn` is similar.

The interpretations of `SubB` and `SubE` just remove and add `InSub` layers in the exceptional values, respectively.

4.6 Stack Safety

Implementations of free monad transformers in eager languages usually need some care to avoid stack overflow (so-called stack safety) and do not provide `iterT`. Instead, they provide a "`foldl` variant" of `iterT` [21], namely `runFreeT` in Purescript and `runM` in scalaz, which takes a function from `F[FreeT[F,M,A]]` to `M[FreeT[F,M,A]]` and returns a value of type `M[A]` for any functor `F`, monad `M` and type `A`.

It is necessary to know whether the subsequent workflow is interrupted or not to perform compensation actions. We use continuation monads to achieve this as the compensation monad [47]. We wrap the underlying monad of `CW` with a continuation monad transformer `ContT`.[7]

```
case class CW[E,M[_],S,R,A](
 run: FreeT[CWT[M,S,?],
 ContT[EitherT[ReaderT[M,Sig,?], InSubL[EV[M[E],S]],?], R, ?],
 A])
extends Monad[CW[E,M,S,R,?]] { /* map, point and flatMap */ }
```

The function `runCWT0` for `runM` takes a command followed by an uninterpreted workflow and returns a continuation monad transformer followed by the workflow left uninterpreted.

```
def runCWT[M[_],R,A](s: Fix[CW[Unit,M,?,R,A]]) = {
 type S = Fix[CW[Unit, M, ?, R, A]]
 type F[X] = CWT[M, S, X]
 type MM[X] = ContT[EitherT[ReaderT[M, Sig, ?], InSubL[EV[M[Unit], S]], X], R, X]
 def runCWT0[M[_],R,A](cl: F[FreeT[F, MM, A]]): MM[FreeT[F, MM, A]]
 = ...
}
```

The change on the definition of `runCWT0` is straightforward. All we need to do is just wrap the exception monad transformer with the continuation monad transformer. For example, the interpretation of the command `CompL` is defined as follows.

```
case Comp(comp, k) = ContT{knt =>
 EitherT{
 knt(k).run.map{ev => ev match {
 ... /* the same to the previous definition */
 }}}}
```

4.7 Atomicity

In this section, we extend `CWT` and the `CW` monad to support the `atomic` and `nonatomic` blocks.

---

[7] The continuation monad transformer must be stack safe. Unfortunately, neither scalaz nor cats (another library similar to scalaz) provides it. Our Scala implementation employs a workaround that relies on `Trampoline` [8] in the `IO` monad. In other words, we always use the `IO` monad as the underlying user monad of the `CW` monad.

We add a command `CheckA` for active context checking and `CheckI` for inactive context checking, whose definitions are similar to that of `Check`.

```
case class CheckA[M[_],S,A](a:A) extends CWT[M,S,A]
case class CheckI[M[_],S,A](a:A) extends CWT[M,S,A]
```

The interpretation of `CheckA` is similar to that of `Check` and that of `CheckI` is just continuing the evaluation of the subsequent workflow without checking the context.

The two blocks are implemented as two functions, similarly to how `sub` sub-workflows are implemented. The functions `atomic` and `nonatomic` replace `Check` with `CheckI` and `CheckA`, respectively, as follows.

```
def atomic[E, M[_], S, A](cw: CW[E, M, S, A]) : CW[E, M, S, A] = CW {
 cw.run.interpretS[CWT[M,S,?]](new (~>[CWT[M,S,?], CWT[M,S,?]]) {
 def apply[A](c: CWT[M,S,A]): CWT[M,S,A] = c match {
 case Check(a) => CheckI(a)
 case _ => c
 }})}
```

4.8 Abnormal Termination and Exceptions in Scala

We have already mentioned abnormal termination in Section 3. In our implementation in Scala, abnormal termination is realized by exceptions of the language. Basically, if an exception is thrown in an atomic action, the whole execution stops. However, we sometimes want to convert an exception in normal action to context, and it can be done using a new form of primitive workflow (*normal* `/~` *compensation*). This is mostly the same as `/+`, but absorbs some particular exceptions `AbortE` and `PAbortE` in the normal action, and raises the interruption `Abort` or `PAbort`. For example:

```
trait CWException extends Exception
class AbortE extends CWException
class PAbortE extends CWException
val cw0 = {if(...) "success" else throw e} /~ comp
```

When running `cw0`, if the exception `e` is `AbortE` or `PAbortE`, it will abort or partially abort; otherwise, the exception is raised as usual. In both cases, it does not do the corresponding compensation `comp`.

`/~` is defined as follows.

```
def /~[E,M[_],S,A](na:M[Try[A]])(comp:A => M[Unit]): CW[E,M,S,A] = for {
 tried <- compL(na)(_ match {
 case Success(a) => comp(a) // same as /+
 case Failure(e) => M.point(()) // skip the compensation comp
 })
 a <- tried match {
 case Failure(AbortE) => throwCWException(Abort) // raise abort
 case Failure(RestartE) => throwCWException(PAbort) // raise pabort
 case Success(a) => compL(M.point(a))(_ => M.point(())) // same as /+
 case Failure(e) => compL(M.point[A]{throw e})(_ => M.point(())) // rethrowing e
 }} yield a
```

The argument `na` is of the type `M[Try[A]]`. `Try[T]` is a Scala's class that represents a computation that may either result in an exception (`Failure[T]`) or return a successfully computed value (`Success[T]`). What `/~` does is first binding the result of `compL` to `tried` of the type `Try[A]` and then carry out one of the following: (1) raising `Abort` or `PAbort` inside ContextWorkflow, (2) successfully committing `na`, or (3) throwing the exception `e` of Scala.

Readers may wonder that the type of `/~` (and also `/+`) is different from that of actual use in examples so far. To omit the explicit type constructors of `M` and `Try`, we use implicit conversions. For further details, see Appendix A.1.

5 Related Work

This work is the direct descendant of our previous work [32]. The main differences between the two are the monadic interpreter, a formalization of semantics, the realization of suspension and checkpoint, and advanced implementation.

Context-Oriented Programming. The literature on context-oriented programming [30], which advocates the use of layers to modularize context-dependent behavior, includes several reports on behavioral change in response to asynchronous context changes [33, 57, 7]. Among them, the closest to the present work is Flute [7] in that 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 reflects the new context starts. The main difference from ContextWorkflow is that ContextWorkflow provides a wider variety of reactions to interruptions, using compensations, sub-workflows, and checkpoints, while Flute emphasizes changing program behavior according to context change.

Termination and Suspension. Rudys and Wallach [50] argue that in language run-time systems such as JVM that execute mobile code, it is important to be able to terminate such code for security reasons. For example, it can be critical to stop executing potentially buggy or untrusted mobile code. They propose a concept called *soft termination* to ensure that mobile code is properly terminated. For example, it makes a program with potentially infinite loops interruptible. Unlike our approach, theirs automatically transforms mobile code using code rewriting.

Several languages provide features to easily realize suspensions, such as first-class continuations [29, 15], which are supported in languages such as Scheme [55] and Scala [49], and *coroutines* [13]. Coroutines are a generalization of subroutines in the sense that they do not exit but call another coroutine as the caller coroutine suspends, and are supported in languages such as Lua [16]. We expect that these facilities are also useful for implementing ContextWorkflow.

Asynchronous Exception. Asynchronous exception, found in, e.g., Haskell [40], Ruby and OCaml [18], is also used to realize interruption. Java and Scala threads take a so-called semi-asynchronous approach [40], where asynchronous exceptions are 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. The design of ContextWorkflow is closer to the former languages in the sense that such a flag to denote interruption is completely implicit.

Workflow. Workflow is a broadly used notion [22, 12] and is provided in several languages such as Windows Workflow Foundation [42] in .NET and Windows PowerShell [41]. PowerShell also supports checkpointing for fault tolerance. There are many studies for the formalization of workflow [9, 10, 38]. Among them, our core ContextWorkflow is based on Bruni et al.'s formalization [9].

In a scientific workflow [39], which is an adaptation of the workflow to scientific computations, a series of heavy computations are executed. In a scientific workflow, checkpoints are also useful to avoid wasteful recomputation [14]. We suppose that ContextWorkflow can be used to develop these applications.

Software Transactional Memory. The software transactional memory (STM) [53], provided, e.g., by Scala [52] and Haskell [26], is a language-level approach to concurrency control, which is similar to a database transaction. STM provides the *atomic block* for atomic execution of all of the loads and stores of a critical section. If multiple atomic blocks are executed on multiple threads and inconsistency is found by interleaving execution, all the atomic blocks will be automatically rolled back. Checkpoints and continuations are also introduced in STM to realize partial aborts without using nested atomic block and gain efficiency [37]. STM is similar to our ContextWorkflow in the sense that they are automatically rolled back when some inconsistency occurs, although inconsistency is caused by rather different events (racy access to memory and context change).

Interruption in Functional Reactive Programming. The ideas of interruption and roll-back are also found in the context of FRP, such as P-FRP [34]. P-FRP is an FRP language for real-time systems, based on E-FRP [59]. In E-FRP, discrete events trigger executions of event handlers, which update reactive values. While E-FRP requires that each event handler execute atomically, P-FRP introduces priorities between events and allows event handlers to be interrupted when an urgent event occurs. To realize such an interruption, P-FRP adopts roll-back mechanisms like STM.

A difference from ours appears in what is rolled back and what kind of effect is removed. While P-FRP rolls back each event handlers and prevents reactive values from being updated incorrectly, ours rolls back the entire execution of a workflow and may remove any computational effects.

Compensation and Asynchronous Exception Monads. Ramalingam et al. showed that workflows with compensating actions can be represented by the compensation monad [47]. Besides the compensation monad, we also got the idea that computations with asynchronous exceptions can be represented by using the resumption monads [27, 28], which are structurally equal to the free monad [46].

Modular Exception Handling. Modularization of exception-handling code has been a significant concern in aspect-oriented programming [35, 11] because the separation of exception-handling code from normal code enhances the re-usability of each module. The compensation approach [60], which we adopt here, regards a pair of a normal code and a compensation as a unit of reuse instead, and also is modular.

Reversible Programming. Compensation actions can be seen as weak manual inversions of normal actions. In reversible programming languages [61], 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. Therefore, integrating reversible programming to ContextWorkflow will be interesting because it can release programmers from the burden of manually specifying compensation actions. Programming compensations is often cumbersome, but has an advantage that we may be able to avoid redundant compensation – such as visiting unnecessary nodes to go back to the start node as we saw in the maze search example in Section 2.

6 Conclusions

In this work, we have proposed ContextWorkflow for developing interruptible context-aware applications. ContextWorkflow basically combines the ideas of workflow and FRP and supports compensations, asynchronous interruption, checkpointing, nested-workflow and suspension. We also formalized the core idea of our language by developing a big-step operational semantics. Further, we proposed a method to embed our ContextWorkflow in existing languages such as Scala and Haskell, mainly using free monads; and the embedded DSL empowers host languages to treat the above features.

One important direction of future work is to support parallelism as many other workflow languages do, that is, atomic actions are executed in parallel on several threads. With parallelism, we expect the semantics of suspension, checkpoints, and sub-workflows to be changed drastically. A question is, for example, if only one sub-workflow of several concurrently running sub-workflows has a checkpoint, how does the whole workflow partially abort? In addition, in a parallel setting, an abort of a sub-workflow need not result in the abort of the parent workflow.

Another direction of future work is efficient implementation. Currently, since we use monad transformers naively, our implementation is not efficient; at least, we should unroll the monad transformer stack as is the standard practice in Haskell programming. It would also be valuable to develop ContextWorkflow with other implementation techniques such as first-class continuations and extensible effects [36], which are also introduced in Scala, and compare different implementations.

One tediousness in ContextWorkflow is that we have to write compensations manually, while we do not need to do so in database transaction and STM. Therefore, it would be interesting to develop a method to construct compensation actions from normal actions. Existing studies such as reversible computing would be helpful to achieve this.

In the current design, programmers can write as long atomic actions as they wish. Since we suppose that one application of ContextWorkflow is battery-aware software, it is interesting to automatically estimate how much execution time an atomic action will consume; then we can perform a kind of verification, e.g., by estimating that 10% of battery level would be enough to complete any compensations of the workflow. We expect that we can rely on existing studies about complexity estimation such as Gulwani et al. [25].

References

1 scalaz. URL: `https://github.com/scalaz/scalaz`.

2 Monadless. URL: `http://monadless.io/`.

3 Gregory Abowd, Anind Dey, Peter Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, volume 1707 of *Springer LNCS*, pages 304–307, 1999.

4 Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan. Context-aware workflow management. In *International Conference on Web Engineering*, volume 4607 of *Springer LNCS*, pages 47–52, 2007.

5 Steve Awodey. *Category Theory*. Oxford University Press, Inc., 2nd edition, 2010.

6 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

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

8 Rúnar Óli Bjarnarson. Stackless scala with free monads. *Scala Days*, 2012.

9 Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages Pages (POPL 2005)*, pages 209–220. ACM, 2005.

10 Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Springer LNCS*, pages 133–150. Springer, 2005.

11 Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented programming. In *Proc. of AOSD'08*, pages 72–83, New York, NY, USA, 2008. ACM.

12 Christian Colombo and Gordon J. Pace. Recovery within long-running transactions. *ACM Comput. Surv.*, 45(3):28:1–28:35, 2013.

13 Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

14 Daniel Crawl and Ilkay Altintas. A provenance-based fault tolerance mechanism for scientific workflows. In *Proc. of Provenance and Annotation of Data and Processes*, volume 5272 of *Springer LNCS*, pages 152–159, 2008.

15 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. of Lisp and Functional Programming*, pages 151–160, 1990.

16 Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

17 William R. Dieter and James E. Lumpp. A user-level checkpointing library for POSIX threads programs. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 224–227. IEEE, 1999.

18 Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming, TFP*, 2017.

19 Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 263–273. ACM, 1997.

20 Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 179–187. ACM, 1993.

21 Phil Freeman. Stack safety for free. URL: `http://functorial.com/stack-safety-for-free/index.pdf`.

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

23 Jeremy Gibbons. Datatype-generic programming. In *Datatype-Generic Programming*, pages 1–71. Springer, 2007.

24 Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.

25 Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proc. of ACM POPL*, pages 127–139, New York, NY, USA, 2009. ACM.

26 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

27 William L. Harrison. The essence of multitasking. In *International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *Springer LNCS*, pages 158–172. Springer, 2006.

28 William L. Harrison, Gerard Allwein, Andy Gill, and Adam Procter. Asynchronous exceptions as an effect. In *Proceedings of the 9th international conference on Mathematics of Program Construction*, volume 5133 of *Springer LNCS*, pages 153–176. Springer-Verlag, 2008.

29 Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298. ACM, 1984.

30 Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

31 Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

32 Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. A DSL for compensable and interruptible executions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2017, pages 8–14, New York, NY, USA, 2017. ACM.

33 Hiroaki Inoue and Atsushi Igarashi. 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*, pages 50–54, New York, NY, USA, 2016. ACM.

34 Roumen Kaiabachev, Walid Taha, and Angela Zhu. E-FRP with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 221–230. ACM, 2007.

35 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. of ECOOP*, volume 1241 of *Springer LNCS*, pages 220–242. Springer, 1997.

36 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM.

37 Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 160–168. ACM, 2008.

38 Jing Li, Huibiao Zhu, Geguang Pu, and Jifeng He. Looking into compensable transactions. In *Software Engineering Workshop, 2007. SEW 2007. 31st IEEE*, pages 154–166. IEEE, 2007.

39 Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

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

41 Microsoft. Powershell documentation. URL: `https://docs.microsoft.com/powershell/`.

42 Microsoft. Windows workflow foundation. URL: `https://docs.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/`.

43 Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23. IEEE, 1989.

44 N.C. Narendra and S Gundugola. Automated context-aware adaptation of web service executions. In *Proceedings of the IEEE International Conference on Computer Systems and Applications*, pages 179–187. IEEE Computer Society, 2006.

45 Bruno C. d. S. Oliveira and Jeremy Gibbons. Scala for generic programmers: comparing Haskell and Scala support for generic programming. *Journal of functional programming*, 20(3-4):303–352, 2010.

46 Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In *Mathematical Foundations of Programming Semantics Thirtieth Annual Conference*, page 273, 2014.

47 Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proc. of ACM POPL*, POPL '13, pages 249–262, New York, NY, USA, 2013. ACM.

48 Brian Randell, Peter Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

49 Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328. ACM, 2009.

50 Algis Rudys and Dan S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):138–168, 2002.

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

52 STM Scala. Expert group. scalastm. web, 2011. URL: `https://nbronson.github.io/scala-stm/`.

53 Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

54 Sucha Smanchat, Sea Ling, and Maria Indrawan. A survey on context-aware workflow adaptations. In *Proceedings of the 6th International Conference on Advances in Mobile Computing and Multimedia*, pages 414–417. ACM, 2008.

55 Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

56 Janis Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, pages 388–403. Springer-Verlag, 2008.

57 Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proc. of Intl. Conf. on Dynamic Languages*, pages 143–156, New York, NY, USA, 2007. ACM.

58 Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.

59 Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *International Symposium on Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

60 Westley Weimer. Exception-handling bugs in Java and a language extension to avoid them. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Springer LNCS*, pages 22–41, 2006.

61 Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proc. ACM PEPM*, pages 144–153, New York, NY, USA, 2007. ACM.

62 Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ML. *Journal of Functional Programming*, 20(2):137–173, 2010.

A Appendix

A.1 Hiding Type Parameters for Simplicity

The type `CW[A]` in Section 2 is in fact an abbreviation of `CW[Unit,IO,Nothing,A]`. The important point is to fix `M` to `IO` and `S` to `Nothing`. In Scala, `Nothing` is a subtype of every other type.

The monad `IO` is the standard way to treat effectful code in monadic programming, but explicit use of the `IO` monad constructor is redundant and not kind to many programmers. Therefore, we hide the explicit appearance of `IO` using implicit conversions of Scala. For example, the way `a /+ c` is converted to the corresponding monadic value is that (1) `a` of the type `A` is converted to a special object of the type `CWOps` that contains a field of the type `IO[A]` by implicit conversions, and then (2) the method `/+` of the special object is invoked. It takes an argument of the type `A => Unit` and returns a value of the type `CW[A]`. Here is the definition of the implicit conversion and the class `CWOps`:

```
implicit def toCWOps[A](proc: => A): CWOps[A] = new CWOps[A](IO(proc))
class CWOps[A](t: IO[A]) {
 def /+ (comp: => A => Unit): CW[A] = /+(t)(a => IO(comp(a)))
}
```

`toCWOps` is the definition for the implicit conversion. `IO(a)` is the `IO` monad constructor. We define the method `/+` in class `CWOps` using the function `/+` that appeared in Section 4.

The reason for using `Nothing` as the suspended workflow type is that, to treat `CW` as a monad, type parameters except for `A` must be fixed or parameterized. Although the latter approach appears good, it would become redundant in Scala. For example, let `CWS[S,A]` be `CW[Unit,IO,S,A]`, and let us combine two `CWS`:

```
def testU[S]: CWS[S,Unit] = ...; def testI[S]: CWS[S,Int] = ...
def testUI[S]: CWS[S,Int] = testU[S].flatMap(_ => testI[S])
```

We would have to use `def` and then type parameter `S` would appear everywhere, since Scala's `value` is not polymorphic. While such definitions can be treated well in Haskell, we would have to manually parameterize it one by one in Scala. Instead, we fix `S` to `Nothing` and cast `Nothing` to a proper suspended workflow type `Fix[CW[Unit,IO,?,A]]` at run time.

A.2 Derivation Example

Let $\mathtt{t}_k = \mathtt{A}_k/\mathtt{C}_k$ *for* $k = a, b, 1, 2, ...$

$$\boxed{\textbf{subgoal 1: } \langle \mathtt{sub\ \{sub\ \{t_1;cp;sub(t_2)/C_a;check\}/c_b;t_3\};t_4}, \mathtt{[]}, \bullet\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \mathtt{c_a;ccp_0;c_1;sub;sub}, \mathtt{[]}\rangle}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle \mathtt{A_1/C_1}, \mathtt{E_0[[];cp;sub(t_2)/C_a;check]}, \mathtt{sub;sub}\rangle \Downarrow^{\mathtt{A_1}} \langle \mathtt{C_1;sub;}\rangle}\,\mathrm{PW} \quad \dfrac{\dfrac{}{\langle \mathtt{cp}, \mathtt{E_1}, \mathtt{C_1;sub;sub}\rangle \Downarrow^{\epsilon} \langle \mathtt{ccp_0;C_1;sub;sub}\rangle}\,\mathrm{CP} \quad (\mathit{subgoal2})}{\langle \mathtt{cp;sub(t_2)/C_a;check}, \mathtt{E_0}, \mathtt{C_1;sub;sub}\rangle \Uparrow_P^{\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Seq\text{-}Int2}}{\langle \mathtt{t_1;cp;sub(t_2)/C_a;check}, \mathtt{E_0}, \mathtt{sub;sub}\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Seq\text{-}Int2}}{\langle \mathtt{sub(t_1;cp;sub(t_2)/C_a;check)/C_b}, \mathtt{sub([];t_3);t_4}, \mathtt{sub}\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Sub\text{-}Int}}{\langle \mathtt{sub(t_1;cp;sub(t_2)/C_a;check)/C_b;t_3}, \mathtt{sub([]);t_4}, \mathtt{sub}\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Seq\text{-}Int1}}{\langle \mathtt{sub(sub(t_1;cp;sub(t_2)/C_a;check)/C_b;t_3)}, \mathtt{[];t_4}, \bullet\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Sub\text{-}Int}$$

$$\dfrac{\langle \mathtt{sub(sub(t_1;cp;sub(t_2)/C_a;check)/C_b;t_3)}, \mathtt{[];t_4}, \bullet\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}{\langle \mathtt{sub(sub(t_1;cp;sub(t_2)/C_a;check)/C_b;t_3);t_4}, \mathtt{[]}, \bullet\rangle \Uparrow_P^{\mathtt{A_1},\mathtt{A_2}} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Seq\text{-}Int1}$$

$$\boxed{\textbf{subgoal 2: } \langle \mathtt{sub(t_2)/C_a;check}, \mathtt{E_0}, \mathtt{ccp_0;C_1;sub;sub}\rangle \Uparrow_P^{\mathtt{A_2}} \langle \mathtt{C_a;ccp_0;C_1;sub;sub}, \mathtt{[]}\rangle}$$

$$\dfrac{\dfrac{\dfrac{}{\langle \mathtt{A_2/C_2}, \mathtt{E_2}, \mathtt{sub;ccp_0;C_1;sub;sub}\rangle \Downarrow^{\mathtt{A_2}} \langle \mathtt{C_2;sub;ccp_0;C_1;sub;sub}\rangle}\,\mathrm{PW}}{\langle \mathtt{sub(t_2)/C_a}, \mathtt{E_0[[];check]}, \mathtt{ccp_0;C_1;sub;sub}\rangle \Downarrow^{\mathtt{A_2}} \langle \overline{\mathtt{c}}_0\rangle}\,\mathrm{Sub} \quad \dfrac{}{\langle \mathtt{check}, \mathtt{E_0}, \mathtt{C_a;ccp_0;C_1;sub;sub}\rangle \Uparrow_P^{\epsilon} \langle \overline{\mathtt{c}}_0, \mathtt{[]}\rangle}\,\mathrm{Check\text{-}PAbort}}{\langle \mathtt{sub(t_2)/C_a;check}, \mathtt{E_0}, \mathtt{ccp_0;C_1;sub;sub}\rangle \Uparrow_P^{\mathtt{A_2}} \langle \mathtt{C_a;ccp_0;C_1;sub;sub}, \mathtt{[]}\rangle}\,\mathrm{Seq\text{-}Int2}$$

$$\boxed{\textbf{goal: } \langle \mathtt{sub\{sub\{t_1;cp;sub(t_2)/C_a;check\}/C_b;t_3\};t_4}\rangle \Downarrow_P^{\mathtt{A_1},\mathtt{A_2},\mathtt{C_a}} \langle \mathtt{C_1;sub;sub}, \mathtt{sub(sub([];cp\#E_1;sub(t_2)/C_a;check)/C_b;t_3);t_4}\rangle}$$

$$\dfrac{(\mathit{subgoal1}) \qquad \dfrac{\dfrac{}{\langle \mathtt{C_a}\rangle \Downarrow^{\mathtt{C_a}} \langle \bullet, \mathtt{[]}\rangle}\,\mathrm{Comp\text{-}Action} \quad \dfrac{\dfrac{}{\langle \mathtt{ccp\#E_1\}[[];cp\#E_1\}]}\rangle \Downarrow^{\epsilon} \langle \bullet, \mathtt{E_1[[];cp\#E_1]}\rangle}\,\mathrm{Comp\text{-}Ccp}}{\langle \mathtt{ccp_0;C_1;sub;sub}\rangle \Downarrow^{\epsilon} \langle \mathtt{C_1;sub;sub}, \mathtt{E_1[[];cp\#E_1]}\rangle}\,\mathrm{Comp\text{-}Seq\text{-}PAbort1}}{\langle \overline{\mathtt{c}}_0\rangle \Downarrow^{C_a} \langle \mathtt{C_1;sub;sub}, \mathtt{E_1[[];cp\#E_1]]}\rangle}\,\mathrm{Comp\text{-}Seq\text{-}PAbort2}}{\langle \mathtt{sub\{sub\{t_1;cp;sub(t_2)/C_a;check\}/C_b;t_3\};t_4}\rangle \Downarrow_P^{\mathtt{A_1},\mathtt{A_2},\mathtt{C_a}} \langle \mathtt{C_1;sub;sub}, \mathtt{E}\rangle}\,\mathrm{Program\text{-}PAbort}$$

where

$$\begin{array}{lll}
\mathtt{E_0} & & = \mathtt{sub(sub([])/C_b;t_3);t_4} \\
\mathtt{E_1} & = \mathtt{E_0[[];sub(t_2)/C_a;check]} & = \mathtt{sub(sub([];sub(t_2)/C_a;check)/C_b;t_3);t_4} \\
\mathtt{E_2} & = \mathtt{E_0[(sub\ \ [])/C_a;check]} & = \mathtt{sub(sub((sub\ \ [])/C_a;check)/C_b;t_3);t_4} \\
\mathtt{ccp_0} & & = \mathtt{ccp\#E_1[[];cp\#E_1]} \\
\overline{\mathtt{c}}_0 & & = \mathtt{C_a;ccp_0;C_1;sub;sub} \\
\mathtt{E} & = \mathtt{E_1[[];cp\#E_1]} & = \mathtt{sub(sub([];cp\#E_1;sub(t_2)/C_a;check)/C_b;t_3);t_4}
\end{array}$$

Figure 6 A derivation of an execution of $\mathtt{sub\{sub\{t_1;cp;sub(t_2)/C_a;check\}/C_b;t_3\};t_4}$ with `pabort` at `check`.

A.3 Proofs of Properties

In the following theorems, let $\mathtt{p}_k = \mathtt{A}_k/\mathtt{C}_k$ for some k, and we define the functions as follows.

- $\flat(\mathtt{t})$ be a workflow that is obtained by removing `sub`, `check`, `cp` and `cp#E` from `t`.
- $\mathit{includes}(\mathtt{t}, m, n)$ iff $\flat(\mathtt{t}) = \mathtt{p}_m\mathtt{;}\cdots\mathtt{;}\mathtt{p}_n$ and $m \le n$; or `t` has no primitive workflows.
- $\mathit{includes}(\mathtt{E}, m, n)$ iff $\mathit{includes}(\mathtt{E[check]}, m, n)$.
- $\mathit{includes}(\overline{\mathtt{c}}, m, n)$ iff $\overline{\mathtt{c}} \setminus \{\mathtt{sub}, \mathtt{ccp\#E}\} = \mathtt{C}_m\mathtt{;}\ldots\mathtt{;}\mathtt{C}_n$ and $m \ge n$; or $\overline{\mathtt{c}}$ has no atomic actions $\mathtt{C}_*$.
- $\mathit{nosub}(\mathtt{t}, m, n)$ iff $\mathit{includes}(\mathtt{t}, m, n)$ and `t` has no `sub`-workflows.

▶ **Lemma 1** (Commit). *If* $\mathit{includes}(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}}\rangle \Downarrow^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}'\rangle$, *then* $\vec{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_n$ *(if* $m \le n$*) or* $\vec{\mathtt{A}} = \epsilon$ *(otherwise).*

Proof. By straightforward induction on the derivation. ◀

▶ **Lemma 2** (Abort). *If* $\mathit{nosub}(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}}\rangle \Uparrow_{A|P}^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{[]}\rangle$, *then* $\vec{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_i$ *and* $\mathit{includes}(\overline{\mathtt{c}}', i, m)$ *for some* i *such that* $m \le i \le n$ *(when* $m \le n$*), or* $\vec{\mathtt{A}} = \epsilon \wedge \mathit{includes}(\overline{\mathtt{c}}', 0, 1)$ *(otherwise).*

Proof. By straightforward induction on the derivation. ◀

▶ **Lemma 3** (Compensation). *If* $\overline{\mathtt{c}} = \mathtt{C}_m, \cdots, \mathtt{C}_n$ *and* $\langle \overline{\mathtt{c}} \rangle \Downarrow^{\vec{\mathtt{C}}} \langle \bullet, \mathtt{[]} \rangle$*, then* $\vec{\mathtt{C}} = \mathtt{C}_m, \cdots, \mathtt{C}_n$*.*

Proof. By straightforward induction on the derivation. ◀

▶ **Lemma 4** (Checkpoint). *Suppose* $\mathit{nosub}(\mathtt{t}, m, k)$ *and* $\mathtt{t}$ *has no* $\mathtt{cp\#E}_*$ *and* $\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}} \rangle \Downarrow^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{[]} \rangle$ *and* $\mathit{includes}(\mathtt{E}, k+1, n)$ *and* $\mathit{includes}(\overline{\mathtt{c}}, m-1, l)$ *and* $l \leq m$ *and* $\mathtt{ccp\#E}_s \notin \overline{\mathtt{c}}$ *and* $\mathtt{ccp\#E}_s \in \overline{\mathtt{c}}'$ *and* $\mathtt{ccp\#E}_s$ *comes just after* $\mathtt{C}_j$ *(or just before* $\mathtt{C}_{j+1}$*, so* $\overline{\mathtt{c}}'$ *usually becomes* $\mathtt{C}_k, \ldots, \mathtt{C}_{j+1}, \ldots, \mathtt{ccp\#E}_s, \ldots, \mathtt{C}_j \ldots, \mathtt{C}_m$*) and* $m-1 \leq j \leq k$*.*

1. *If* $m-1 \leq k \leq n \wedge m \leq n$*, then* $\mathit{includes}(\mathtt{E}_s, j+1, n)$*.*
2. *If* $n \leq k < m$*, then* $\mathit{includes}(\mathtt{E}_s, m, k)$*.*

Proof. Proof by induction on the derivation of $\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}} \rangle \Downarrow^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{[]} \rangle$. We show only main cases for the first item.

Case CW-Checkpoint: $\mathtt{E}_s = \mathtt{E[[];cp\#E]}$ $\quad j = m-1$

It is the case that $k = m-1$, and so $\mathit{includes}(\mathtt{E}, m, n)$. Clearly, $\mathit{includes}(\mathtt{E}_s, m, n)$, finishing the case.

Case CW-Seq: $\mathtt{t} = \mathtt{t}_1\mathtt{;t}_2$ $\quad \langle \mathtt{t}_1, \mathtt{E[[];t}_2\mathtt{]}, \overline{\mathtt{c}} \rangle \Downarrow^{\vec{\mathtt{A_1}}} \langle \overline{\mathtt{c}}'' \rangle$
$\langle \mathtt{t}_2, \mathtt{E}, \overline{\mathtt{c}}'' \rangle \Downarrow^{\vec{\mathtt{A_2}}} \langle \overline{\mathtt{c}}' \rangle$

We get $\mathit{includes}(\mathtt{t}_1, m, i)$ and $\mathit{includes}(\mathtt{t}_2, i+1, k)$ for some i s.t. $m-1 \leq i \leq k$. The induction hypothesis finishes the case. ◀

▶ **Lemma 5** (Partial Abort). *Suppose* $\mathit{nosub}(\mathtt{t}, m, n_0)$ *and* $\mathtt{t}$ *has no* $\mathtt{cp\#E}_*$ *and* $\langle \mathtt{t}, \mathtt{[]}, \bullet \rangle \Uparrow_P^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{[]} \rangle$ *and* $\vec{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_n$ *and* $\mathit{includes}(\overline{\mathtt{c}}', n, m)$ *and* $\langle \overline{\mathtt{c}}' \rangle \Downarrow^{\vec{\mathtt{C}}} \langle \overline{\mathtt{c}}'', \mathtt{E}_s \rangle$*.*

- *If* $m \leq n$*, then* $\vec{\mathtt{C}} = \epsilon$ *and* $\mathit{includes}(\mathtt{E}_s, m, n)$ *and* $\mathit{includes}(\overline{\mathtt{c}}'', n, m)$*, or* $\vec{\mathtt{C}} = \mathtt{C}_n, \cdots, \mathtt{C}_{k+1}$ *and* $\mathit{includes}(\mathtt{E}_s, k+1, n)$ *and* $\mathit{includes}(\overline{\mathtt{c}}'', k, m)$ *for some* k *s.t.* $m-1 \leq k < n$*.*
- *If* $m > n$*, then* $\vec{\mathtt{C}} = \epsilon$ *and* $\mathit{includes}(\mathtt{E}_s, m, n)$ *and* $\mathit{includes}(\overline{\mathtt{c}}'', n, m)$*.*

Proof. Proof by induction on the derivation of $\langle \overline{\mathtt{c}} \rangle \Downarrow^{\vec{\mathtt{C}}} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$, using Lemma 4. ◀

▶ **Lemma 6** (Suspend). *Suppose* $\mathit{includes}(\mathtt{t}, m, k)$ *and* $\langle \mathtt{t}, \mathtt{E}, \overline{\mathtt{c}} \rangle \Uparrow_S^{\vec{\mathtt{A}}} \langle \overline{\mathtt{c}}', \mathtt{E}_s \rangle$ *and* $\mathit{includes}(\mathtt{E}, k+1, n)$*.*

1. *If* $m-1 \leq k \leq n \wedge m \leq n$*, then* $\vec{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_i$ *for some* i *such that* $m \leq i \leq k$ *and* $\mathit{includes}(\mathtt{E}_s, i+1, n)$*, or* $\vec{\mathtt{A}} = \epsilon$ *and* $\mathit{includes}(\mathtt{E}_s, m, n)$*.*
2. *If* $n \leq k < m$*, then* $\mathit{includes}(\mathtt{E}_s, m, k)$*.*

Proof. Proof by induction on the derivation. We show only main cases for the first item.

Case CW-Check-Suspend:

It is the case that $k = m-1$, and so $\mathit{includes}(\mathtt{E}, m, n)$, finishing the case.

Case CW-Sub-Int: $\mathtt{t} = \mathtt{sub(t')/c}$

We can get $\mathit{includes}(\mathtt{t}', m, k)$ and $\mathit{includes}(\mathtt{E[(sub\ \ [])/c]}, k+1, n)$. Then, the induction hypothesis finishes the case.

Case CW-Seq-Int1: $\mathtt{t} = \mathtt{t}_1\mathtt{;t}_2$

We get $\mathit{includes}(\mathtt{t}_1, m, j)$ for some j s.t., $m-1 \leq j \leq k$. We also get $\mathit{includes}(\mathtt{E[[];t}_2\mathtt{]}, j+1, n)$. Then, the induction hypothesis finishes the case.

Case CW-Seq-Int2: $\mathtt{t} = \mathtt{t}_1\mathtt{;}\mathtt{t}_2 \quad \langle \mathtt{t}_1, \mathtt{E[[];t_2]}, \overline{\mathtt{c}}\rangle \Downarrow^{\overrightarrow{\mathtt{A}_1}} \langle \overline{\mathtt{c}}''\rangle$

$\langle \mathtt{t}_2, \mathtt{E}, \overline{\mathtt{c}}''\rangle \Uparrow_S^{\overrightarrow{\mathtt{A}_2}} \langle \overline{\mathtt{c}}', \mathtt{E}_s\rangle$

We get $includes(\mathtt{t}_1, m, j)$ for some j s.t., $m-1 \leq j \leq k$. By Lemma 1, $\overrightarrow{\mathtt{A}_1} = \mathtt{A}_m, \cdots, \mathtt{A}_{j-1}$ (when $m \leq j$), or $\overrightarrow{\mathtt{A}_1} = \epsilon$ (when $j = m-1$). We also get $includes(\mathtt{t}_2, j+1, k)$ from $includes(\mathtt{t}, m, k)$ and $includes(\mathtt{t}_1, m, j)$. We still have $includes(\mathtt{E}, k, n)$.

Then, by the induction hypothesis, $\overrightarrow{\mathtt{A}_2} = \mathtt{A}_j, \cdots, \mathtt{A}_i$ for some i such that $j \leq i \leq k$ and $includes(\mathtt{E}_s, i+1, n)$, or $\overrightarrow{\mathtt{A}_2} = \epsilon$ and $includes(\mathtt{E}_s, m, n)$.

Finally, we can finishes the case concatenating $\overrightarrow{\mathtt{A}_1}$ and $\overrightarrow{\mathtt{A}_2}$. ◀

▶ **Theorem 1** (Workflow commits). *If* $includes(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \overline{\mathtt{c}}\rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle\rangle$ *and* $m \leq n$*, then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots \mathtt{A}_n$.

Proof. By Lemma 1 and CW-Program-Commit. ◀

▶ **Theorem 2** (Workflow aborts (Successful Compensation)). *If* $nosub(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \overline{\mathtt{c}}\rangle \Downarrow_A^{\overrightarrow{\mathtt{A}}} \langle\rangle$ *and* $m \leq n$ *and* $\overline{\mathtt{c}} = \mathtt{C}_k\mathtt{,}\ldots\mathtt{,}\mathtt{C}_l$*, then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_m, \mathtt{C}_k, \cdots, \mathtt{C}_l$ *for some* i *s.t.* $m \leq i \leq n$.

Proof. By Lemmas 2 and 3 and CW-Program-Abort. ◀

▶ **Theorem 3** (Restarted suspended workflow commits). *If* $\langle \mathtt{t}, \bullet\rangle \Downarrow^{\overrightarrow{\mathtt{A}}} \langle\rangle$ *and* $\langle \mathtt{t}, \bullet\rangle \Downarrow_S^{\overrightarrow{\mathtt{C}}} \langle \overline{\mathtt{c}}, \mathtt{E}\rangle$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow^{\overrightarrow{\mathtt{C}'}} \langle\rangle$*, then* $\overrightarrow{\mathtt{A}} = \overrightarrow{\mathtt{C}}, \overrightarrow{\mathtt{C}'}$.

Proof. By Theorem 1, Lemma 6 and CW-Program-Suspend. ◀

▶ **Theorem 4** (Workflow partially aborts). *If* $nosub(\mathtt{t}, m, n)$ *and* $\langle \mathtt{t}, \bullet\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}}} \langle \overline{\mathtt{c}}, \mathtt{E}\rangle$ *and* $m \leq n$*, then either of the followings hold.*

- $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_i, \mathtt{C}_i, \mathtt{C}_{i-1}, \cdots, \mathtt{C}_j$ *and* $includes(\mathtt{E}, j, n)$ *and* $includes(\overline{\mathtt{c}}, j-1, m)$ *for some* i *and* j *(*$m \leq j \leq i \leq n$*).*
- $\overrightarrow{\mathtt{A}} = \mathtt{A}_m, \cdots, \mathtt{A}_n$ *and* $includes(\mathtt{E}, 1, 0)$ *and* $includes(\overline{\mathtt{c}}, n, m)$*.*

Moreover, the followings hold.

1. *(Suspended workflow commits) If* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow^{\overrightarrow{\mathtt{A}'}} \langle\rangle$*, then* $\overrightarrow{\mathtt{A}'} = \mathtt{A}_j, \cdots, \mathtt{A}_n$*, or* $\overrightarrow{\mathtt{A}'} = \epsilon$ *(if* $includes(\mathtt{E}, 1, 0)$*).*
2. *(Suspended workflow aborts) If* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow_A^{\overrightarrow{\mathtt{A}'}} \langle\rangle$*, then* $\overrightarrow{\mathtt{A}'} = \epsilon$ *(if* $j = m$*), or* $\overrightarrow{\mathtt{A}'} = \mathtt{C}_{j-1}, \cdots, \mathtt{C}_m$.

Proof. By Lemma 2, Lemma 5 and CW-Program-PAbort.

1. By Theorem 1.
2. By Theorem 2. ◀

▶ **Theorem 5** (Partial abort, checkpoint and nested workflow). *Suppose that includes(*$\mathtt{t}$*,1,n) and* $\mathtt{t\backslash check} =$
$\mathtt{p_1;\ldots;cp;p_k;\ldots;p_m;sub(p_{m+1};\ldots;cp;p_j;\ldots;p_l)/C_a;p_{l+1};\ldots;p_n}$ *and* $\langle \mathtt{t}, \bullet\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}}} \langle \overline{\mathtt{c}}, \mathtt{E}\rangle$.

1. *(Partial abort skips compensations of complete sub-workflow) If* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}}\}$*, then* $\overrightarrow{\mathtt{A}} = \mathtt{A}_1, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_{l+1}, \mathtt{C}_a, \mathtt{C}_m, \cdots, \mathtt{C}_k$ *for some* $i > l$*.*
2. *(A suspended workflow remembers checkpoints in a sub-workflow) If* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}}\}$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}'\rangle$ *and* $\mathtt{A}_j \in \{\overrightarrow{\mathtt{A}'}\} \wedge \mathtt{A}_l \not\in \{\overrightarrow{\mathtt{A}'}\}$*, then* $\overrightarrow{\mathtt{A}'} = \mathtt{A}_k, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_j$ *for some* i *s.t.* $j \leq i \leq l$*.*
3. *(A suspended workflow remembers checkpoints before a sub-workflow) If* $\mathtt{C}_j \in \{\overrightarrow{\mathtt{A}}\}$ *and* $\langle \mathtt{E[check]}, \overline{\mathtt{c}}\rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}'\rangle$ *and* $\mathtt{A}_{l+1} \in \{\overrightarrow{\mathtt{A}'}\}$*, then* $\overrightarrow{\mathtt{A}'} = \mathtt{A}_j, \cdots, \mathtt{A}_i, \mathtt{C}_i, \cdots, \mathtt{C}_{l+1}, \mathtt{C}_a, \mathtt{C}_m, \cdots, \mathtt{C}_k$ *for some* $i > l$*.*

Proof. Let $\mathtt{E}_0 = \mathtt{[];cp\#E}_0\mathtt{;p}_k\mathtt{;...;sub(...;cp;...)/C}_a\mathtt{;...;p}_n$ and $\mathtt{E}_1 = \mathtt{cp\#E}_0\mathtt{;sub([];cp\#E}_1\mathtt{;...)/C}_a\mathtt{;...;p}_n$.

1. Straightforwardly from the derivation, using Lemma 1 and Lemma 2. Notice that the CW-Sub deletes the `cp` inside the `sub` and installs the other compensation $\mathtt{C}_a$.
2. We can get $\mathtt{E} = \mathtt{E}_0$ from the derivation tree. Then, the conclusion follows straightforwardly from the derivation of $\langle \mathtt{E[check]}, \overline{\mathtt{c}} \rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}' \rangle$ using Lemma 1 and Lemma 2.
3. We can get $\mathtt{E} = \mathtt{E}_1$ from the derivation tree. Then, the conclusion follows straightforwardly from the derivation of $\langle \mathtt{E[check]}, \overline{\mathtt{c}} \rangle \Downarrow_P^{\overrightarrow{\mathtt{A}'}} \langle \overline{\mathtt{c}}'', \mathtt{E}' \rangle$ using Lemma 1 and Lemma 2. ◀