

KYOTO UNIVERSITY

DOCTORAL THESIS

**A Language-Based Approach to
Robust Context-Aware Software**

Author:
Hiroaki Inoue

Supervisor:
Professor Atsushi Igarashi

Department of Communications and Computer Engineering
Graduate School of Informatics

February 27, 2018

KYOTO UNIVERSITY

Abstract

Graduate School of Informatics
Department of Communications and Computer Engineering

Doctor of Informatics

A Language-Based Approach to Robust Context-Aware Software

by Hiroaki Inoue

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. Context-aware software has some context-dependent behaviors to which software adapts at run-time. However, such a concept brings a new and important problem of *achieving safety from harmful context changes* to software development: run-time adaptation of context-dependent behavior sometimes leads to a failure. For example, a facility of software corresponding to some contexts may be unavailable at some time since it is not implemented yet. Another example is that the program lacks reactivity to context changes, causing delay of adaptation. Therefore, it is important to make robust software free of failures concerning context-awareness.

Our goal is to create a programming language for developing robust context-aware software. In this thesis, we focus on two kinds of studies: (1) static fault detection via a type system; (2) run-time error recovery via workflow.

Type systems are a well-known technique for light-weight static verification of a program, which is useful to detect erroneous code before running applications. We develop such a type system with the calculus $\text{ContextFJ}_{<}$, which models JCop language, which embodies Context-Oriented Programming, a new paradigm for developing context-aware applications. Using the type system, a type checker can detect invalid behavioral adaptation statically, in the sense that no *method not found error* arises. The novelty of $\text{ContextFJ}_{<}$ is supporting inheritance and subtyping of contexts, which are similar to those of objects in Object-Oriented Programming languages and helpful to model and modularize applications. Our contributions are (1) language design of a type-safe version of JCop including layer swapping, (2) the semantics of layer inheritance, which adds another “dimension” to the space of method lookup, (3) sound subtyping for first-class layers, which led us to two kinds of subtyping relation.

Workflow is a broadly used notion to coordinate long-running transactional activities and realizes robustness via compensations, which roughly correspond to undo operations for already executed activities. We develop a context-aware version of workflow, named *ContextWorkflow*, which mainly treats the sudden interruption of a program that would arise from context changes: responding to sudden interruption, a program should abort with doing recovery, or suspend. The novelty of *ContextWorkflow* is to support asynchronous interruption, suspension and checkpointing, which saves a snapshot of the program states; and being able to be implemented as an embedded domain-specific language (E-DSL) in some major host languages such as Scala and Haskell, which programmers can use easily as a part of a host language. Our contributions are (1) the design of *ContextWorkflow*; (2) a formal semantics of the core

language; (3) a monadic interpreter corresponding to the semantics; and (4) its concrete implementation as an E-DSL in Scala.

Acknowledgements

First of all, I am extremely grateful to my supervisor, Prof. Atsushi Igarashi, for his endless enthusiasm, patience, wisdom, and support. He gave me a chance to study Context-Oriented Programming, which is the base of my work. He is quite proficient to make abstract ideas and complex notations elegant and concrete, and that has been quite fruitful for my research. He has raised my ability to make up papers, slides, and documents and do presentations, from the viewpoints of not only contents and structures but also English.

I thank my Ph.D. examiners, Prof. Toru Ishida and Prof. Akihiro Yamamoto for their comments to improve the quality of this thesis significantly. Besides, Prof. Toru Ishida passionately involved me in Design School and has given me valuable comments on my research and future.

I express my gratitude to my senior collaborators. Dr. Tomoyuki Aotani has lent me a helping hand to make up ContextWorkflow and introduced me to the mysterious monadic world. He also has been kind to me since I was in bachelor and greatly influenced my understanding of Context-Oriented Programming and Functional Reactive Programming. Prof. Robert Hirschfeld kindly invited me to Berlin and assisted me in creating the idea of type-safe JCop. He has occasionally encouraged me over the years and taught me the importance of sightseeing. Dr. Malte Appeltauer took some of his time to invite me to SAP innovation center and teach me JCop.

I thank all members of Software Foundation Group. Dr. Kohei Suenaga, Dr. Seiji Umatani, and Dr. Koji Nakazawa have given me many kinds of valuable advice and taught me computer science and mathematics. Dr. Taro Sekiyama is a great pioneer of Ph.D. course in our laboratory and quite helpful for my graduation. Ms. Kaori Takei, a secretary of our laboratory, has assisted me a lot particularly in following procedures of Design School. All students have made my labo-life fun and stimulative.

I am grateful to Kyoto University Design School and its members. I am especially thankful to Prof. Tetsuo Sawaragi, who has involved my work for about five years as an advisor. He also suggested to me participating the internship in Mitsubishi Electric Corporation. Prof. Kumiyo Nakakoji, Prof. Hiroyuki Matsui, Dr. Takushi Sogo, Dr. Yusuke Kita, and Ms. Eri Tsunokawa guided my first overseas tour in Europe and have given me valuable advice. Prof. Fumitoshi Matsuno delivered me a special lecture on robotics (FBL/PBL) and also helpfully commented on my Ph.D. work. All teachers and staffs have supported me. I have gratitude to all my friends in the same year too, with who collaborate in a wide variety of classes such as about robotics, education, English/Japanese speaking/writing, learning commons, etc., and go to Europe, Okinawa, etc. They have made my Ph.D. program pleasant and motive. Additionally, I have been supported by this program financially.

I say thank you to all those whom I have visited, or had conversations with over the years. Dr. Tetsuo Kamina and Prof. Hidehiko Masuhara, who are in the similar research area, have given me valuable comments. Staffs of Graduate School of Informatics and Dr. Kensuke Kojima lent me a helping hand in “successfully entering” the graduate school. Mitsubishi Electric Corporation took in me as an internship. The internship helped me to concretize my work and refresh myself. I also thank Prof. Shunichi Fukuhara, Ms. Shizue Nagai, Dr. Yoshihiro Onishi, Mr. Ryo Aoto, and all staffs of iHope International and Department of Healthcare Epidemiology, who gave me an opportunity to work as a fledgling programmer. L’s sports Kyoto, a health club, supported me to maintain my health. Daisuke Baba, Dr. Shinichiro Ogawa, and Takahiko Uchiyama have assisted me in completing a hand called ‘chanta’.

Finally, I thank my family, who supported me from all of financial, physical and mental sides since childhood.

List of Publications

Chapter 2 consists of:

Hiroaki Inoue, Atsushi Igarashi, Malte Appeltauer, and Robert Hirschfeld. Towards type-safe JCop: A type system for layer inheritance and first-class layers. In *Proceedings of the International Workshop on Context-Oriented Programming (COP'14)*, Uppsala, Sweden, July 2014. <https://doi.org/10.1145/2637066.2637073>

Hiroaki Inoue and Atsushi Igarashi. A Sound Type System for Layer Subtyping and Dynamically Activated First-Class Layers. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS 2015)*, volume 9458 of Lecture Notes in Computer Science, pages 445–464, Pohang, Korea, November 2015. http://dx.doi.org/10.1007/978-3-319-26529-2_24

Hiroaki Inoue and Atsushi Igarashi. A type system for first-class layers with inheritance, subtyping, and swapping, November 2017. Submitted for publication.

Chapter 3 consists of:

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'17)*, Vancouver, Canada, October 2017. <http://dx.doi.org/10.1145/3141858.3141860>

Hiroaki Inoue, Tomoyuki Aotani and Atsushi Igarashi. ContextWorkflow: A Monadic DSL for compensable and interruptible executions, January 2018. Submitted for publication.

Chapter 4 consists of:

Hiroaki Inoue and Atsushi Igarashi. A Library-Based Approach to Context-Dependent Computation with Reactive Values. In *Proceedings of the Constrained and Reactive Objects Workshop (CROW 2016)*, Málaga, Spain, March 2016. <https://doi.org/10.1145/2892664.2892669>

Contents

Contents	vii
List of Abbreviations	xi
List of Figures	xiii
1 Introduction	1
1.1 Context-Aware Applications	1
1.2 Language-Based Approaches to Developing Context-Aware Applications	1
1.2.1 Context-Oriented Programming	2
JCop	3
1.2.2 Languages for Asynchronously Interruptible Program	4
1.3 Problem Statement	4
1.4 Our Approach	5
1.4.1 Static Verification via Type System	5
1.4.2 Error Recovery via Workflow	6
1.5 The Reason for a Language-Based Approach	7
1.6 This Thesis	7
1.6.1 A Type System for JCop	7
1.6.2 ContextWorkflow	8
1.6.3 Modularizing Context-Dependent Behaviors in ContextWorkflow	9
1.7 Organization of the Thesis	9
2 A Type System for JCop	11
2.1 Language Constructs of Safe JCop	11
2.1.1 Layers and Partial Methods	11
2.1.2 Layer Activation and First-Class Layers	12
2.1.3 Dependencies between Layers	13
2.1.4 Layer Inheritance and Subtyping	13
2.1.5 Layer Swapping and Deactivation	15
2.1.6 Method Lookup	16
2.2 ContextFJ _{<}	18
2.2.1 Syntax	18
2.2.2 Operational Semantics	20
2.2.3 Type System	25
2.3 Type Soundness	32
2.3.1 Typing Rules for Run-time Expressions	32
2.3.2 Subject Reduction	34
2.3.3 Progress	36
2.4 Summary	36

3	ContextWorkflow	37
3.1	ContextWorkflow Constructs	37
3.1.1	Example: Explorer Robot	37
3.1.2	Interruptible Workflow	38
3.1.3	Interruption and Context	39
3.1.4	Nested Workflow and Programmable Compensations	40
3.1.5	Checkpoint	41
3.1.6	Blocking Context Checking	42
3.2	Operational Semantics of Core ContextWorkflow	42
3.2.1	Syntax	43
3.2.2	Big-Step Semantics	43
3.2.3	Properties	46
3.2.4	Discussion	47
3.3	Monadic Embedding to Scala	48
3.3.1	Free Monad Transformers	49
3.3.2	ContextWorkflow Monad	50
3.3.3	Auxiliary Definitions	51
3.3.4	Types of Suspended Workflows	52
3.3.5	Monadic Interpreter	52
3.3.6	Stack Safety	54
3.3.7	Atomicity	55
3.3.8	Abnormal Termination and Exceptions in Scala	55
3.3.9	Hiding Type Parameters for Simplicity	56
3.4	Summary	57
4	Modularizing Context-Dependent Behaviors in ContextWorkflow	59
4.1	Overview	59
4.2	Contexts as Reactive Values	60
4.2.1	Functional Reactive Programming	60
4.2.2	Context-Oriented Programming	60
4.2.3	Characteristics of Contexts	61
4.2.4	Layer Activation as Reactive Values	61
4.2.5	Problem of Asynchronous Context Change	62
4.3	Context-Dependent Computation with Reactive Values	62
4.3.1	Core Classes of GEAR	62
4.3.2	Semantics	63
4.3.3	Dynamic Binding of Context Sources	64
4.4	Implementation	64
4.5	Summary	66
5	Related Work	67
5.1	Foundation of Context-Oriented Programming	67
5.2	Advanced Modularization Mechanisms	68
5.3	Ancestors and Cousins of ContextWorkflow	69
5.4	On Modularization of Context-Dependent Behaviors	70
6	Conclusion	73
6.1	This Thesis	73
6.2	Future Work	73

A	Proof of Type Soundness of ContextFJ_{<}:	85
A.1	Proofs	85
B	Proof of Properties of Core ContextWorkflow	101
B.1	Proofs	101

List of Abbreviations

CAA	Context-Aware Application
CDB	Context-Dependent Behavior
CDF	Context-Dependent Function
COP	Context-Oriented Programming
DOP	Delta-Oriented Programming
E-DSL	Embedded domain specific language
FRP	Functional-Reactive Programming
LF	Layered Function
OOP	Object-Oriented Programming
SPL	Software product line

List of Figures

2.1	An example of layer subtyping hierarchy.	15
2.2	Method Lookup Example.	18
2.3	ContextFJ _{<} : Syntax.	19
2.4	ContextFJ _{<} : Lookup Functions.	21
2.5	ContextFJ _{<} : Reduction Rules 1.	22
2.6	ContextFJ _{<} : Reduction Rules 2.	23
2.7	ContextFJ _{<} : Subtyping Relations.	26
2.8	ContextFJ _{<} : Method Type Lookup Functions.	27
2.9	ContextFJ _{<} : Expression Typing.	28
2.10	ContextFJ _{<} : Method/Class/Layer Typing.	29
2.11	ContextFJ _{<} : Program Typing.	30
2.12	ContextFJ _{<} : Layer set well-formedness.	32
2.13	ContextFJ _{<} : Runtime Expression Typing.	33
3.1	Maze search simulation: Abort (left) and Suspend (right).	40
3.2	Maze search simulation: Abort (refined)	41
3.3	Maze search simulation: Partial abort (left) and its restart (right).	42
3.4	Syntax of core ContextWorkflow.	43
3.5	Big step semantics of core ContextWorkflow.	45
3.6	ContextWorkflow: A derivation example.	46

Chapter 1

Introduction

1.1 Context-Aware Applications

Software is much more interactive than it used to be: it interacts with not only users but also external resources such as network and sensors and changes its behavior according to inputs from these resources. Such external information that affects the behavior of software is often referred to as *contexts*, and software that is aware of contexts as *context-aware applications* (CAA) [SAW94, ADB⁺99].

One typical instance of CAA is robot software. Suppose, for example, a robot has limited energy and moves around to do some task. In case the battery is running out before accomplishing the given job, he should stop what he is doing, return to a refueling point and resume the task. In fact, such a robot has already been developed and sold, such as Roomba[®] ¹.

Besides robot applications, we can consider a lot of computer software as CAA. One example is an e-mail reader which switches to a text-based mode when network throughput is low. Another example is a package manager application that installs or updates packages in an operating system or a software development environment, whose execution largely depends on network availability.

1.2 Language-Based Approaches to Developing Context-Aware Applications

From the viewpoint of software development, researchers have shown that CAA has following two features.

1. CAA has several *context-dependent behaviors* (CDB), each of which depends on some contexts. One characteristic feature of CDB is that it is activated and deactivated at run-time according to contexts changes; that is, *behavioral changes* occur dynamically. An example is an e-mail reader that has the rich-text mode and plain-text mode, which are CDBs and depend on different contexts respectively, high or low of network throughput. In an example of the above robot, doing given task and returning to a refueling point are also CDBs, and each of them relates to his battery level.
2. CAA usually reacts to context changes promptly since contexts are time-varying in many situations; so they are *interruptible* [BVDR⁺12] or support *asynchronous interruption*. An example is a working robot that should immediately stop his task when the surroundings become dark; if he lacks rapid reaction, he may work haphazardly in the dark.

¹<http://www.irobot-jp.com/roomba/>

Researchers have also found a problem that, by using current main-stream languages, CAA is hard to develop and maintain. The main reason is that existing main-stream languages lack constructs to deal with the above features of CAA. We look at two problems related to the above features and corresponding current solutions in detail.

1.2.1 Context-Oriented Programming

The first issue is about modularization of CAA program. Generally, it is desirable for a software program to be modularized into some meaningful pieces of code so that programmers can reuse each module. In this spirit, many modularization mechanisms are proposed such as Object-Oriented Programming (OOP) [Sny86], Aspect-Oriented Programming [KLM⁺97], and Feature-Oriented Programming [Pre97, ABKS13].

Similarly, a program constituting a CAA should be appropriately modularized so that CDBs are separated from each other [HCN08]. That is, it is desirable to package one CDB into one module in order to let programmers not to change many modules at once. Since a CDB will be added newly or removed from software through the life cycle of the software, such modularization is very significant, e.g., introducing a new mode related to about half amount of the battery level in a robot example.

However, existing module structures, such as *classes* of OOP languages, are not suitable to arrange CDBs. This is because that the description of CDBs often crosscuts with the dominating module structure. Besides module structures, it is difficult to treat dynamic behavioral changes, i.e., the context-aware software has to be adapted for different CDBs dynamically.

In order to address this problem from a programming-language perspective, Context-Oriented Programming (COP) has been proposed by Hirschfeld et al. [HCN08]. COP usually extends OOP. The main idea of COP is to provide language constructs *layers*, which are modules to specify context-dependent behavior, and their *dynamic layer activation*. A layer is basically a collection of what are called *partial methods*, which intuitively correspond to CDBs, and add new behavior to existing objects or override existing methods. When a layer is activated at run-time by a designated construct, the partial methods defined in it become effective, changing the behavior of objects until the activation ends. Roughly speaking, a layer abstracts a context and corresponding CDBs, and dynamic layer activation abstracts change of contexts.

COP is introduced in many languages such as Java [AHHM11], Python [VLDN07], Common Lisp [CH05], and JavaScript [GMCC13]. Here is a simple example of a battery-aware robot using COP in Java.

```
public layer LowBattery {
    public void Robot.doTask(){ // partial method
        ... /* go back to a refueling point */
    }
}
public layer HighBattery {
    public void Robot.doTask(){ // partial method
        ... /* do given task */
        ... proceed(); ... // invoking an overridden doTask()
    }
}
```

We define two layers `LowBattery` and `HighBattery`, each of which expresses modules containing CDBs relating to low and high battery level respectively. `Robot.doTask` is

a partial method that is declared into existing class `Robot` and represents a CDB. Since a partial method can override original definitions in a class or other layers, it is able to have a `proceed` call, which invokes an overridden method and similar to the `super` call of Java.

While layers are a way to separate CDBs, layer activations are for dynamic behavior changes. A layer can be activated by using a `with`-block, in which the selected layers are activated and method invocations are affected by activated layers.

```
Robot robot = new Robot();
with(HighBattery){
    ...
    robot.doTask(); // it will be affected by HighBattery
    ...
}
```

Then, we can clearly separate CDBs into layers relating to some contexts and dynamically adopt CDBs by activating layers.

JCop

JCop language [AH12] is a leading COP language that is an extension of Java. Not only does it support basic COP constructs described above, but also it introduces many advanced features such as inheritance of layer implementations and first-class layers; they enhance the expressibility and are helpful to model and modularize CAAs.

For example, it would be smart to let `LowBattery` and `HighBattery` extend the same layer `BatteryLevel` using layer inheritance if they have similar definitions. Suppose we add a new method `printBatteryLevel` in the class `Robot` by these layers, which returns the current battery level as a string, and write a common string in the partial method of `BatteryLevel`.

```
public layer BatteryLevel{
    public String Robot.currentBatteryLevel(){
        return "BatteryLevel:";
    }
}
public layer LowBattery extends BatteryLevel {
    public String Robot.currentBatteryLevel(){
        return superproceed() + "Low";
    }
}
public layer HighBattery extends BatteryLevel {
    public String Robot.currentBatteryLevel(){
        return superproceed() + "High";
    }
}
```

We can use a `superproceed` call, which invokes a method overridden by layer inheritance, e.g., here the `superproceed` calls will invoke the `Robot.currentBatteryLevel()` method defined in the layer `BatteryLevel`.

This mechanism is not special one: classes of OOP use inheritance commonly. Similar to objects in Java, a layer is also a first-class inhabitant in JCop, that is, it can be a value and passed to as an argument of methods.

```

BatteryLevel bat = batteryLevel() < 50 ? new LowBattery() : new
    HighBattery();
with(bat){
    ...
}

```

Here, `bat` represents a layer value whose content is decided by conditional expression. It can also be activated by `with`.

Therefore, layer inheritance and first-class layers enhance expressibility of COP.

1.2.2 Languages for Asynchronously Interruptible Program

The second problem is about reactivity of CAA. Reactiveness has emerged in many applications such as GUI, which interacts with some user inputs; and embedded systems, which respond to many sensory data. Generally, it is difficult to develop reactive software because of asynchronous nature, i.e., programmers cannot predict when a stimulus occurs. Therefore, many technologies have been proposed to deal with reactivity well, such as the observer pattern [GHJV94] of OOP and functional reactive programming (FRP) [EH97, BLCVC⁺13].

We explain more about FRP. One unique construct of FRP is first-class time-varying values, called *signals*. A signal is intuitively a function from time to some value. It can depend on other signals, and a dependency network of signals consists a directed acyclic graph. Moreover, a value change of a signal automatically propagates to other signals which are dependent on the signal. Then, programmers need not manage time-varying values by themselves, i.e., they are free from manual assignments of updated values.

Here is an example of signals in Scala using an FRP library REScala [SHM14].

```

val battery: Signal[Int] = /* a signal representing the battery level */
val lowbattery: Signal[Boolean] = Signal{ if(battery() < 20) false else true }

```

Both `battery` and `lowbattery` are signals. We suppose `battery` is updated periodically by a built-in timer. `lowbattery` depends on `battery` and is automatically updated if the value of the dependent signal changes.

Researchers have already studied to use FRP for developing CAAs [MCDM08, PHE99, HCNP02]. Among them, Bainomugisha et al. proposed a new programming language *Flute* [BVDR⁺12] which supports asynchronous interruption with suspensions. One novel approach of *Flute* is that it adopts FRP to express what manages interruption. In *Flute*, a function relates to signals of FRP and a predicate about the signals, and its execution will abort or suspend when values of the dependent signals do not satisfy the predicate. In other words, we can execute the program under some contexts. We think that this approach is a natural one since it can be seen as a generalization of *time-out*, where program execution depends on time; it is needless to say time is time-varying.

1.3 Problem Statement

Developing CAA, though novel concepts described above have been introduced, however, a new and crucial problem *achieving safety from harmful context changes* has emerged: a run-time adaptation of context-dependent behavior sometimes leads to failure. Though the meaning of this problem is wide, here we focus on two concretizations.

1. The first sub-problem is about the unavailability of CDBs. A facility of software corresponding to some contexts may be unavailable at run-time for several reasons, such as it is not implemented yet. This problem is crucial since it will cause an unexpected termination of software.
2. The second one is about robustness against context changes. Since many CAAs should respond to context changes promptly, some tasks may be aborted halfway. The issue here is that just aborting such an incompleted task is problematic since it lets computational effects as they are, e.g., a robot may stop without going back to the refueling point. We need some appropriate error-recovery, but it is usually not clear how to do it.

1.4 Our Approach

Our research goal is to develop a programming language to address this problem. In this thesis, we focus on the above two sub-problems and take different approaches respectively; static fault detection via a type system for the first one; run-time error recovery via workflow for the second.

1.4.1 Static Verification via Type System

From the viewpoint of COP, we can state the first problem in a different way, i.e., invoking a (partial) method is failed by “method not found” error. Such a problem is broadly recognized in programming languages; for example, many dynamically typed languages such as Ruby and Python can easily cause this error.

To prevent “method not found” error, *type systems* are a well-known technique, which is useful to detect erroneous code before running applications. Many languages such as Java, ML, Scala, and Haskell, have its own type system. Although we can say dynamically typed languages such as Ruby and Python also have type systems, only statically typed languages are called *type-safe*, which means that if a program is well-typed, it does not cause a certain kind of run-time errors. For example, in Java, a well-typed program does not cause “method not found” errors.

Developing a type system and providing type-safety for a language, we usually take an approach to constructing a formal calculus which models the language and includes a type system. In Java, for example, *Featherweight Java* (FJ) [IPW01] is used as the defacto standard of the core calculus, and many derivatives are born in many programming paradigms such as aspect-oriented programming [CIZ10] and feature-oriented programming [AKL08]. A formal calculus generally includes formal syntax and semantics and provides a significant property *type soundness* [WF94] with its proofs, which ensures type-safety of the language.

Typechecking a COP language is also an interesting study and is challenging because of its dynamic nature. In Java-based COP languages, typechecking programs is more difficult than in Java because layer (de)activation can dynamically change the interfaces of objects (if a layer adds a new method to base classes). Igarashi, Hirschfeld, and Masuhara [IHM12] have studied this problem and developed a formal calculus ContextFJ, which models a core of COP, with a provably sound type system.

However, there were no studies of typechecking advanced COP mechanisms introduced in JCop. In other words, ContextFJ supports only the fundamental COP mechanisms, namely, global and second-class layers and block-style layer activation and it is

not obvious how their type system can extend to other features such as first-class layers and layer inheritance of JCop.

Therefore, we address this problem by developing a core calculus based on JCop.

1.4.2 Error Recovery via Workflow

To address the second issue, a desirable programming language should have an error-handling mechanism that is well suited to reactivity of CAA. To find out what kind of mechanism is well suited and understand why this is an exciting problem to solve, we look some examples.

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 considerable 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 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 promptly (1) abort with reverting the “effects” that comes from incompleting tasks (file replacement in the first example and robot moves in the second example) or (2) suspend until the time contexts get back.

It is, however, difficult to develop context-aware applications in existing mainstream languages because of the following two problems. First, as Bainomugisha et al. indicated [BVDR⁺12], 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. Using *asynchronous exceptions* [MJMR01] 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 this.

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 the reversion and suspension; especially, the reversion is similar to resource handling with exceptions, which is hard with the constructs [Wei06]. A more complicated and difficult reaction is *partial abort* [GMC09], which is a combination of the reversion and suspension and is realized by using *checkpoints* [RLT78, ZJ10, DL99]. Checkpoints are useful in order to make applications robust [pow] and avoid wasteful recomputation [CA08].

Our solution to the problems is based on the ideas of Flute [BVDR⁺12] and *workflow* [GMS87, CP13]. As described above, Flute is a programming language originally

proposed to solve the first problem. To represent the context depending on multiple time-varying data, Flute uses FRP. Flute also supports suspending the program execution.

Workflow [GMS87, CP13] represents a long-running interruptible transaction that consists of several atomic transactions. The typical applications are web applications and business process management, and recently it is adapted to context-aware applications [NG06, AFG⁺07, SLI08]. One import idea of workflow for us is *compensation* [Wei06], 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.

1.5 The Reason for a Language-Based Approach

Here, we explain why we take a language-based approach. We have several reasons. One is for the safety of a language. Setting limitation on elements of a language (such as syntax) can increase the safety of the language, but on the other hand, it reduces expressibility of the language. Static typing and dynamic typing are a typical example. Therefore, taking a language-based approach, we can discuss tradeoffs between safety and expressibility of the language.

Another reason is for simplicity and genericity. Libraries and frameworks are useful in many cases, but they are sometimes too specific. For example, existing workflow engines such as Windows Workflow Foundation [wwf] are helpful for defining and executing long-running tasks and business processes, but they are exaggerating for generic error-handling of a program. For error recovery of our setting, what we want is a simple and lightweight technique like asynchronous exceptions and try-catch-finally, which are embedded into languages.

1.6 This Thesis

In this thesis, we propose solutions to the above problems. First, we develop a small COP language called *ContextFJ_<* as a method of static verification, which extends ContextFJ and includes a provably sound type system. Second, we provide *Context-Workflow* for developing an interruptible program. Finally, we try to combine Context-Workflow and COP notion. Their details are as follows.

1.6.1 A Type System for JCop

In this work, first, we design a type-safe version of JCop, called Safe JCop, based on the notion of ContextFJ. Then, we formalize Safe JCop by developing a small COP language called *ContextFJ_<*. The language extends ContextFJ to layer inheritance, subtyping of layer types, first-class layers, and a type-safe layer deactivation mechanism called *layer swapping*; and we prove a type soundness theorem for *ContextFJ_<*.

Main issues we have to deal with are (1) language design of Safe JCop, including layer swapping (2) the semantics of layer inheritance, which adds another “dimension” to the space of method lookup, (3) sound subtyping for first-class layers, which led us to two kinds of subtyping relation.

Designing Safe JCop First, we design a type-safe version of JCop. Safe JCop consists of a part of JCop features, such as layers, with-block layer activation, layer inheritance,

and first-class layers. It also includes explicit declarations of dependencies between layers which are introduced in ContextFJ.

Moreover, we newly introduce a type-safe layer deactivation mechanism called layer swapping. It is known to be difficult to introduce layer deactivation in a type-safe manner [IHM12]. Therefore, we introduced a way to safely realize layer deactivation, layer swapping, which deactivates one layer and activates another one simultaneously.

Semantics Second, we extend the semantics of ContextFJ to support layer inheritance. Besides `proceed` call of COP, JCop provides `superproceed` call which is almost similar to `super` call but follows the layer inheritance, not class inheritance. Then, JCop’s method lookup mechanism adds another dimension to the one of original COP. Therefore, we formalize the new method lookup mechanism.

Type System Third, we extend a type system of ContextFJ by Safe JCop constructs. Especially, we study about layer subtyping considering first-class layers; we introduce two kinds of subtyping relation, normal and weak subtyping. We also prove the type soundness of the calculus.

1.6.2 ContextWorkflow

We propose a language *ContextWorkflow* as a solution to the above second problem. ContextWorkflow is a workflow-based language that supports compensation, asynchronous interruption, suspension, and checkpoints. It also provides *sub-workflows* and *programmable compensations* [BMM05, CP13] that ignore and replace the compensations of completed portions of workflow, respectively.

Our approach to implementing ContextWorkflow is embedding to other “host” languages [Hud96], that is, we develop an *embedded domain-specific language* (E-DSL). 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) design of the workflow-based programming language with asynchronous interruption, (2) formalization including the big-step operational semantics, (3) monadic interpreter corresponding to the semantics, and (4) implementation of ContextWorkflow by embedding to Scala. The details are as follows.

Asynchronous Interruption in Workflow Our approach to asynchronous interruption uses signals of FRP and polling [Fee93], and our novel finding is that the idea of transactions fits 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 the transactions in database [Gra81] and software transactional memory [ST97]. The difference between our workflow and the transactions is when a check runs. In the transactions, a check runs at the end. We also introduce constructs for blocking interruptions as in Concurrent Haskell [MJMR01] 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 the Bruni et al.'s formalization [BMM05] of Sagas [GMS87], which is a foundation of workflow. We provide basic properties of the calculus and describe small extensions. We also discuss whether the polling code should be inserted before or after an atomic transaction using the core calculus.

Monadic Interpreter We develop a monadic interpreter closely corresponding with the operational semantics in lazy and eager languages. We define the CW monad using the reader, either and free [Swi08] monads and monad transformers to represent abstract syntax trees. The monadic interpreter translates ContextWorkflow monadic values to values of the underlying monad using fold-like functions. Because free monad transformers and the fold functions over them are usually different between lazy and eager languages, two implementations are necessary.

Implementation as E-DSL 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 look the ContextWorkflow programs naturally.

1.6.3 Modularizing Context-Dependent Behaviors in ContextWorkflow

Finally, we provide a mechanism to modularize CDBs in ContextWorkflow. The motivation is that though we use contexts as an execution status in ContextWorkflow, we want to use mechanisms of behavioral changes of COP also in ContextWorkflow. For example, we would like to program CAAs such as energy-aware computing [SDF⁺11, ZLL15], where several modes relate to energy consumption and are changed dynamically. In this work, we develop a stand-alone library to modular CDBs in Scala, called GEAR.

In our library, we mainly introduce two constructs, context-dependent functions (CDFs) and layered-functions (LFs). A CDF is a partial function that modularizes one context-dependent behavior and relates to signals representing contextual data with a predicate about signals. A LF is a list of CDFs and an executable unit of the library: when it is executed, containing CDFs which satisfy their own predicates are going to be executed in order. We suppose LF is used to write atomic actions of ContextWorkflow.

The contributions of this work are (1) developing a prototypical library that realizes modular behavioral changes and (2) implementation of proceed call of COP using delimited continuations [DF90], which is also provided in Scala [RMO09].

1.7 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we describe ContextFJ_{<.. Chapter 3 is for ContextWorkflow. Chapter 4 explains about the third work. Chapter 5 describes related work. Chapter 6 concludes this thesis with future work. We also give the proof of type soundness of ContextFJ_{<.. in Appendix A and the proofs of properties of core ContextWorkflow in Appendix B.}}

Chapter 2

A Type System for JCop

In this chapter, we develop a type-safe version of JCop, called Safe JCop. Then, we formalize Safe JCop by developing a small COP language called $\text{ContextFJ}_{<}$, which extends ContextFJ [IHM12] to layer inheritance, subtyping of layer types, first-class layers, and a type-safe layer deactivation mechanism called layer swapping. We also provide a provable type soundness property with several theorems and lemmas.

The rest of the chapter is organized as follows. After informally reviewing features of Safe JCop in Section 2.1, we develop $\text{ContextFJ}_{<}$ with its syntax, operational semantics, and type system in Section 2.2; and we prove type soundness in Section 2.3. The rest summarizes this chapter.

2.1 Language Constructs of Safe JCop

In this section, we review language constructs of Safe JCop including first-class layers, layer inheritance/subtyping, and layer swapping along informal discussions about the type system.

As a running example, we consider programming a graphical computer game called *RetroAdventure* [AHL13]. In this game, a player has a character “hero” that wanders around the game world. Here, we introduce class `Hero` that represents the hero, which has method `move` to walk around, and class `World` that represents the game world.

```
public class Hero {
    Position pos;
    public void move(Direction dir){
        pos = /* changes pos according to dir */;
    }
}
public class World { ... }
```

2.1.1 Layers and Partial Methods

As mentioned already, a first distinctive feature of COP is *layers*—collections of *partial methods* to modify the behavior of existing objects. A partial method is syntactically similar to an ordinary method declared in a class, except that the name is given in a qualified form `Hero.move()`; this means the partial method is going to override method `move` defined in `Hero` or (if it does not exist) add to `Hero`. A layer can contain partial methods for different classes, so, when it is activated, it can affect objects from various classes at once. Similarly to super calls in Java, the body of a partial method can contain proceed calls to invoke the original method overridden by this partial method.

Here, suppose that the hero’s behavior is influenced by weather conditions in the game world. For example, in a foggy weather, the hero gets slow and, in a stormy

weather, the hero cannot move as he likes. Here are layers that denote weathers of the game world.

```
public layer Foggy {
    /* partial method */
    public void Hero.move(Direction dir){
        pos = /* the distance of move is shorter */;
    }
}

public layer Stormy {
    /* partial method */
    public void Hero.move(Direction dir){
        proceed(randomDirection(dir));
    }
    /* baseless partial method */
    public Direction Hero.randomDirection(Direction dir){
        return /* add randomness to dir */;
    }
}

public layer Sunny { ... }
```

Foggy and Stormy have the definitions of `Hero.move`, which change the behavior of the original definition in different ways. In particular, `Hero.move` in Stormy uses `proceed`, replacing the arguments to calls to `move`. It also has `Hero.randomDirection`, used to determine a new randomized direction to which the hero is going to move.

Methods defined in classes are often referred to as *base methods* and partial methods without corresponding base methods as *baseless partial methods*. Notice that activating a layer with baseless partial methods extends object interfaces and `proceed` in a baseless partial method is unsafe unless another layer activation provides a baseless partial method of the same signature.

2.1.2 Layer Activation and First-Class Layers

In Safe JCop, a layer can be activated by using a layer instance (created by a new expression, just as an ordinary Java object, from a layer definition) in a `with` statement. The following code snippet shows how Foggy can be activated.

```
with(new Foggy()){
    hero.move(); /* The hero will get slow by Rainy weather. */
}
```

Inside the body of `with`, dynamic method dispatch is affected by the activated layers so that partial methods are looked up first. So, movement of the hero will be slow.

Layer activation has a dynamic extent in the sense that the behavior of objects changes even in methods called from inside `with`. If more than one layer is activated, a more recent activation has precedence and a `proceed` call in a more recently activated layer may call another partial method (of the same name) in another layer.

In Safe JCop, a layer instance is a first-class citizen and can be stored in a variable, passed to, or returned from a method. A layer name can be used as a type. Combining with layer subtyping discussed later, we can switch layers to activate by a run-time condition. For example, suppose that the game has *difficulty* levels, determined at run time according to some parameters, and each level is represented by an instance of a sublayer of `Difficulty`. Then, we can set the initial difficulty level by code like this:

```
Difficulty diff = /* an expression to compute difficulty */ ;
with(diff){ ... }
```

Moreover, a layer can declare own fields and methods (although we do not model them in layers in this article). So, first-class layers significantly enhance expressiveness of the language.

2.1.3 Dependencies between Layers

Baseless partial methods and layer activation that has dynamic extent pose a challenge on typechecking because activation of a layer including baseless partial methods can change object interfaces. So, a method invocation, including a `proceed` call, may or may not be safe depending on what layers are activated at the program point. Safe JCop adopts `requires` clauses [IHM12] for layer definitions to express which layers should have been activated before activating each layer (instance). The type system checks whether each activation satisfies the `requires` clause associated to the activated layer and also uses `requires` clauses to estimate interfaces of objects at every program point.

For example, consider another layer `ThunderInStorm`, which expresses an event in a game. It affects the way how the hero's direction is randomized during a storm and includes a baseless partial method with a `proceed` call. To prevent `ThunderInStorm` from being activated in a weather other than a storm, the layer `requires Stormy` as follows:

```
public layer ThunderInStorm requires Stormy {
  public Direction Hero.randomDirection(Direction dir){
    Direction tmpd = proceed(dir);
    ... /* change tmpd to speed up */
    return tmpd;
  }
}
```

An attempt at activating `ThunderInStorm` without activating `Stormy` will be rejected by the type system (unless the activation appears in a layer requiring `Stormy`). Thanks to the `requires` clause, the type system knows that the `proceed` call will not fail. (It will call the partial method of the same name in `Stormy` or some other depending on what layers are activated at run time.)

2.1.4 Layer Inheritance and Subtyping

In Safe JCop, a layer can inherit definitions from another layer by using the keyword `extends` and the `extends` relation between layers yields subtyping, just like Java classes. If weather layers have many definitions in common, it is a good idea to define a super-layer `Weather` and concrete weather layers as its sublayers.

```
public layer Weather {
  public Text People.sayWeather(){ return new Text(""); }
  ...
}
public layer Stormy extends Weather {
  public Text People.sayWeather(){
    Text buf = superproceed();
    buf.setText("It's stormy today.");
    return buf;
  } ...
}
public layer Foggy extends Weather {
```

```

    public Text People.sayWeather(){ ... }
    ...
}

```

Here, Weather provides (baseless) partial method `sayWeather` to the class `People`, which returns `Text` data that people say about weather condition. The implementation of `People.sayWeather` just returns an empty `Text` and sublayers of the Weather override it. Safe JCop provides `superproceed`, which calls a partial method overridden because of layer inheritance. The partial method of `Stormy` sets the contents of the text using `superproceed`.

Since class subtyping equals to the reflexive and transitive closure of the `extends` relation, we expect layer subtyping to be the same; an instance of a sublayer can be substituted for that of its superlayer. However, substitutability is more subtle than one might expect and we are led to distinguishing two kinds of substitutability and introducing two kinds of subtyping relation, called weak and normal subtyping. The difference arises from `requires` clauses. To explain the issue, we define layer `Thunder`, which is the superlayer of `ThunderInStorm` and `ThunderInFog` and a sublayer of a marker layer `Event`.

```

public layer Event { ... }
public layer Thunder extends Event requires Weather {
    public void change_font(Text label){ label.setFont("Italic"); }
    public Text People.sayWeather(){ change_font(proceed()); }
}
public layer ThunderInStorm extends Thunder requires Stormy {
    public Text People.sayWeather(){
        Text buf = superproceed();
        buf.setText("Escape from here right now!!");
        return buf;
    }
    ...
}
public layer ThunderInFog extends Thunder requires Foggy {
    public Text People.sayWeather(){ ... }
}

```

`Thunder` changes the font of the text of what `People` say. It seems natural to set the `requires` clause of `Thunder` to be `Weather`, since its two sublayers require `Stormy` and `Foggy` respectively.

Weak subtyping An instance of a sublayer can be used where one of its superlayers is required, since a sublayer defines more partial methods than its superlayer. For example, to activate the following layer called `Thunder`, which requires `Weather`, it suffices to activate `Foggy`, a sublayer of `Weather`, beforehand.

```

with(new Foggy()){
    // Thunder requires Weather and Foggy extends Weather
    with(new Thunder()){ ... }
}

```

We will formalize substitutability about `requires` as *weak subtyping*, which is the reflexive transitive closure of the `extends` relation between layer types. For the weak subtyping to work, we require that a sublayer declare, at least, what its superlayer requires because partial methods inherited from the superlayer may depend on them.

We could relax this condition if a sublayer overrides all the partial methods but such a case is expected to be rare and so not taken into account.¹

Normal subtyping The above notion of subtyping is called weak because it does *not* guarantee safe substitutability for *first-class* layers. Consider layer `Difficulty` again and assume that it requires no other layers and has sublayers `Easy` and `Hard`. In the following code snippet, the activation of `diff` appears safe because its static type `Difficulty` does not require any layers to have been activated.

```
Difficulty diff = someCondition() ? new Easy() : new Hard();
with(diff){ ... }
```

However, the case where `Easy` or `Hard` requires some layers breaks the expected invariant that the dependency expressed by the `requires` clauses is satisfied at run time. So, for assignments and parameter passing, we need one more condition for subtyping, namely, `requires` of a sublayer must be the same as that of its superlayer. We call this strong notion of subtyping *normal subtyping*.

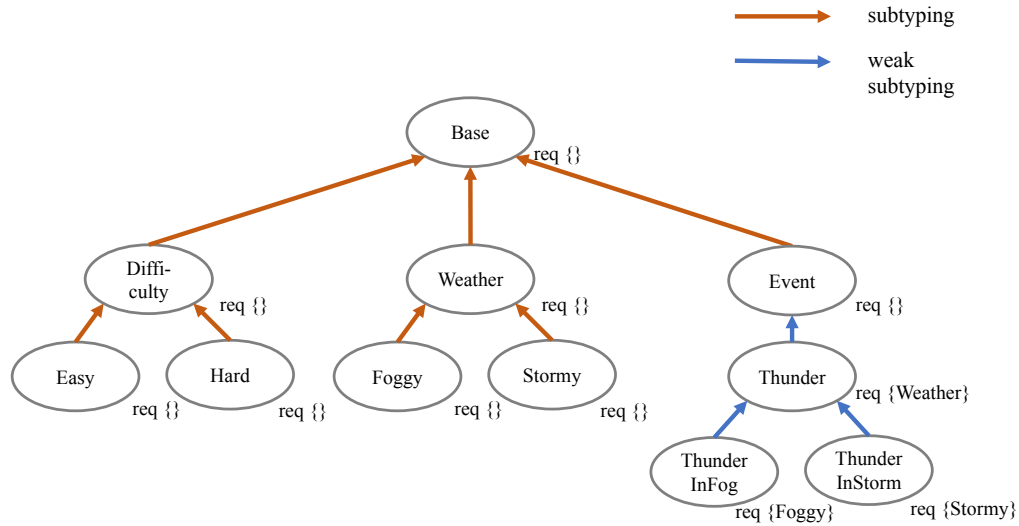


FIGURE 2.1: An example of layer subtyping hierarchy.

In Fig. 2.1, we show the layer subtyping hierarchy of the examples so far. An oval means a layer and the notation `req {X}` beside an oval means its requiring layers. Just like `Object` in Java, there is `Base`, which is a superlayer of all layers, in Safe JCop. If a layer omits the `extends` clause, it is implicitly assumed that the layer extends `Base`.

2.1.5 Layer Swapping and Deactivation

The original JCop provides constructs to *deactivate* layers, called `without`. However, only with `requires`, it is not easy to guarantee that layer deactivation does not lead to an error. For safe deactivation, it has to be checked that there is no layer that `requires` the deactivated layer, but the type system is not designed to keep track of the *absence* of certain layers. Instead of general-purpose layer deactivation mechanisms, Safe JCop

¹Re-typechecking inherited methods under the new `requires` clause would be another way to relax this condition but this is against modular checking.

introduces a special construct to express one important idiom that uses deactivation, namely *layer swapping* to deactivate some layers and activate a layer at once.

In Safe JCop, we can define a layer as *swappable*, which means that all its sublayers can be swapped with each other, by adding the modifier *swappable*. The swap statement for layer swapping is of the following form:

```
swap(activation_layer, deactivation_layer_type){ ... }
```

The *activation_layer* is an expression whose static type must be a sublayer of *deactivation_layer_type*, which in turn has to be swappable. It deactivates *all* instances of *deactivation_layer_type* (and its sublayers), and activates the *activation_layer*.

Let's consider *Difficulty* once again. We could define *Difficulty* as a swappable layer and use *swap* to switch to another mode temporarily.

```
swappable layer Difficulty { ... }
...
Difficulty diff = someCondition() ? new Easy() : new Hard();
with(diff){
    ...
    swap(new Hard(), Difficulty){
        ... // Enforce hard mode
    }
}
```

Unfortunately, for type safety, the necessary restriction for layer swapping is not weak: No sublayers of a swappable layer can be required by other layers or can change their interfaces or requires clauses from the swappable layer.

2.1.6 Method Lookup

We informally explain how Safe JCop's method lookup mechanism works, before proceeding to the formal calculus.

When method *m* is invoked on an instance of class *C* with layers $L_1; \dots; L_n$ activated, the corresponding method definition is sought as follows: first, the activated layers L_n, L_{n-1} , down to L_1 are searched (in this order) for a partial method named *C.m*; if *C.m* is not found, the base class *C* is searched for the base definition; if *m* is not found, similar search continues on the *C*'s superclass *D*—namely, the activated layers are searched again for a partial method named *D.m* and the base class *D* is searched for the base definition, and so on. In addition to the usual inheritance chain in class-based object-oriented languages, COP adds another dimension to the space of method lookup. Actually, there is yet another dimension in (Safe) JCop because of layer inheritance: When L_i is searched for a partial method, its superlayers are searched, too, before going to L_{i-1} . For example, under the following class and layer definitions

```
class C extends D    { }
class D extends E    { void m(){ ... } }
class E              { void m(){ ... } }
layer L1             { void D.m(){ ... } }
layer L2 extends L3 { void E.m(){ ... } }
layer L3             { void C.m(){ ... } }
```

the following statement

```
with(new L1()) {
    with(new L2()) {
```

```

    new C().m();
  }
}

```

will execute partial method $C.m$ defined in $L3$, whereas the statement

```
with(new L1()) { new C().m(); }
```

will execute $D.m$ in $L1$.

Now, we turn our attention to the semantics of `super`, `proceed`, and `superproceed`. When a `super`, `proceed` or `superproceed` call is encountered during execution of a (partial) method, it continues to look for a method definition of the same name as follows.

Suppose that $C.m$ is found in layer L_i with layers $\bar{L} = L_1; \dots; L_n$ activated ($0 < i \leq n$) and that D is a superclass of C .

- A call `super.m()` starts looking for a partial method $D.m$ from L_n and so on.
- A `proceed` call starts looking for a partial method $C.m$ from L_{i-1} or the base method of class C (when $i = 1$), and so on.
- A `superproceed` call starts looking for $C.m$ in L_i' (where L_i' is the superlayer of L_i), L_i'' (where L_i'' is the superlayer of L_i'), and so on. If $C.m$ is not found in the superlayers, it is a run-time error (which the type system will prevent).

For example, consider the following class and layer definitions and suppose $L1$, $L2$ and $L3$ are activated in this order. (In what follows, the notation $L.C.m$ means the partial method $C.m$ defined in layer L .)

```

class C extends D { }
class D extends E { void m(){ return super.m(); } }
class E           { void m(){ return; } }
layer L1 {
  void C.m(){ return e1; }
  void D.m(){ return e2; }
}
layer L2 {
  void C.m(){ return e3; }
}
layer L4 {
  void C.m(){ return e5; }
  void E.m(){ return e6; }
}
layer L3 extends L4 {
  void C.m(){ return e4; }
}

```

- `super.m` calls from $L4.C.m$ and $L1.C.m$ will invoke $L1.D.m$; and ones from $L1.D.m$ and $D.m$ will invoke $L4.E.m$, since $L3$ inherits $E.m$ from $L4$.
- `proceed` calls from $L4.C.m$ will invoke $L2.C.m$ and ones from $L1.C.m$ will invoke $L1.D.m$.
- `superproceed` calls from $L3.C.m$ will invoke $L4.C.m$.

Fig. 2.2 summarizes how `super`, `proceed`, and `superproceed` calls are resolved. Each ball represents a (partial) method definition and its location where it is put. The

three axes stands for class inheritance (C extends D and D extends E), activated layers (L1, L2, and L3 are activated in this order), and layer inheritance (L3 extends L4). Orange arrows represent how proceed calls at each method definition are resolved. For example, the top-most long orange arrow means that proceed from L4.E.m will invoke E.m. Green arrows represent super and blue superproceed.

Finally, we should note that, for super, proceed, and superproceed calls, the activated layers are the same as those when the current method is found. So, with or swap around super, proceed, and superproceed does not affect which definition is invoked.

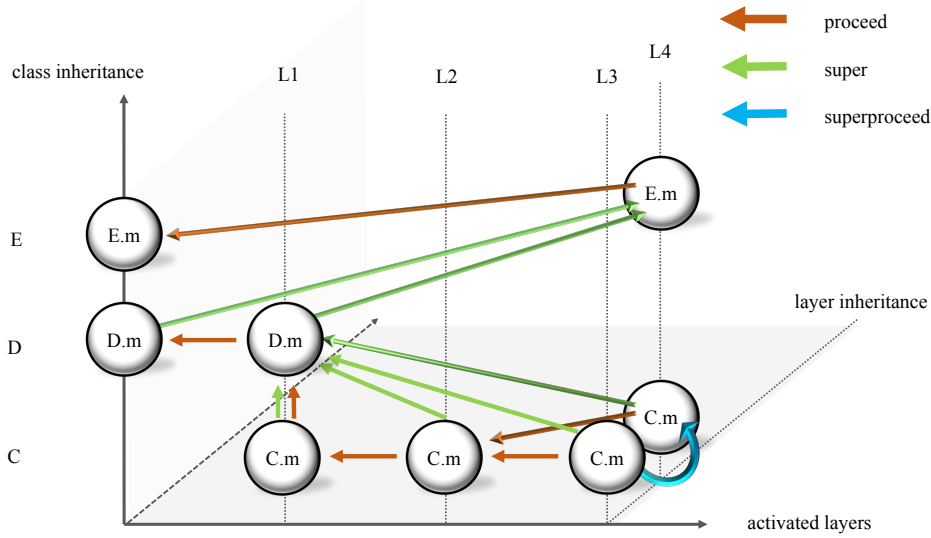


FIGURE 2.2: Method Lookup Example.

2.2 ContextFJ_<:

In this section, we formalize a core functional subset of Safe JCop as ContextFJ_<; with its syntax, operational semantics and type system. ContextFJ_<, a descendant of Featherweight Java (FJ) [IPW01], extends ContextFJ [HIM11, IHM12] with layer inheritance, superproceed, layer subtyping, first-class layers, and swappable layers. JCop features that ContextFJ_< does *not* model for simplicity include: fields and (ordinary) methods in layers, special variable `thislayer` to refer to the current layer instance, `superlayer` to invoke an ordinary method in a superlayer, and declarative layer composition.

2.2.1 Syntax

Let metavariables C, D and E range over class names; L over layer names; f and g over field names; m over method names; x and y over variables, which contains special variable `this`. The abstract syntax of ContextFJ_< is given in Fig. 2.3.

Following FJ, we use overlines to denote sequences: So, $\overline{f}$ stands for a possibly empty sequence $f_1, \dots, f_n$ and similarly for $\overline{T}$, $\overline{x}$, $\overline{e}$, and so on. The empty sequence is denoted by $\bullet$. Concatenation of sequences is often denoted by a comma except for layer names, for which we use a semicolon. We also abbreviate pairs of sequences, writing " $\overline{T} \ \overline{f}$ " for " $T_1 \ f_1, \dots, T_n \ f_n$ ", where n is the length of $\overline{T}$ and $\overline{f}$, and similarly " $\overline{T} \ \overline{f};$ " as shorthand for the sequence of declarations " $T_1 \ f_1; \dots T_n \ f_n;$ " and "`this. $\overline{f} = \overline{f};$` " for

T	$::= C \mid L$	(types)
CL	$::= \text{class } C \triangleleft C \{ \bar{T} \bar{f}; K \bar{M} \}$	(classes)
LA	$::= [\text{swappable}] \text{ layer } L \triangleleft L \text{ req } \bar{L} \{ \bar{PM} \}$	(layers)
K	$::= C(\bar{T} \bar{f}) \{ \text{super}(\bar{f}); \text{this}.\bar{f} = \bar{f}; \}$	(constructors)
M	$::= T \ m(\bar{T} \bar{x}) \{ \text{return } e; \}$	(methods)
PM	$::= T \ C.m(\bar{T} \bar{x}) \{ \text{return } e; \}$	(partial methods)
e, d	$::= x \mid e.f \mid e.m(\bar{e}) \mid \text{new } T(\bar{e}) \mid \text{with } e \ e \mid \text{swap } (e, L) \ e$ $\mid \text{proceed}(\bar{e}) \mid \text{super}.m(\bar{e}) \mid \text{superproceed}(\bar{e})$ $\mid \text{new } C(\bar{v}) \langle C, \bar{L}, \bar{L}' \rangle .m(\bar{e}) \mid \text{new } C(\bar{v}) \langle C, L, \bar{L}, \bar{L}' \rangle .m(\bar{e})$	(expressions)
v, w	$::= \text{new } C(\bar{v}) \mid \text{new } L()$	(values)

FIGURE 2.3: ContextFJ_<: Syntax.

“ $\text{this}.f_1=f_1; \dots; \text{this}.f_n=f_n;$ ”. Given layer sequence $\bar{L}$, We write $\{\bar{L}\}$ for the set of layers (obtained by ignoring the order). Sequences of field declarations, parameter names, layer names, and method declarations are assumed to contain no duplicate names.

We briefly explain the syntax, focusing on COP-related constructs. A layer definition LA consists of optional modifier *swappable*, its name, its superlayer name, layers that it requires, and partial methods. A partial method (defined as PM) is similar to a method but specifies which m to modify by qualifying the simple method name with a class name C .

Instantiation can be a layer instance $\text{new } L()$, as well as a class instance $\text{new } C(\bar{e})$. Note that arguments to $\text{new } L$ are always empty because ContextFJ_< does not model fields of layer instances. In the expression $\text{with } e_1 \ e_2$, e_1 stands for the layer to be activated and e_2 the body of *with*. In the expression $\text{swap } (e_1, L) \ e_2$, e_1 means the layer to be activated, L the swappable layer, e_2 the body of *swap*. By this expression, during the evaluation of e_2 , all instances of the swappable layer L and its sublayers are deactivated, and e_1 is activated. $\text{super}.m(\bar{e})$, $\text{proceed}(\bar{e})$ and $\text{superproceed}(\bar{e})$ are keywords to invoke methods of the superclass, a previously activated layer, and the superlayer, respectively.

Expressions $\text{new } C(\bar{v}) \langle D, \bar{L}', \bar{L} \rangle .m(\bar{e})$ and $\text{new } C(\bar{v}) \langle D, L, \bar{L}', \bar{L} \rangle .m(\bar{e})$ are special run-time expressions that are related to method invocation mechanism of COP, and not supposed to appear in classes and layers. They basically mean that m is going to be invoked on $\text{new } C(\bar{v})$. The annotation $\langle D, \bar{L}', \bar{L} \rangle$ is used to model *super* and *proceed* whereas $\langle D, L, \bar{L}', \bar{L} \rangle$ is used for *superproceed*. $\bar{L}$ stands for a sequence of activated layers and D, L and $\bar{L}'$ (which is assumed to be a prefix of $\bar{L}$) play a role of a “cursor” where the method lookup starts from. We explain how they work in detail in Section 2.2.2.

Program A ContextFJ_< program (CT, LT, e) consists of a class table CT , a layer table LT and an expression e , which stands for the body of the main method. CT maps a class name to a class definition and LT a layer name to a layer definition. A layer definition can be regarded as a function that maps a partial method name $C.m$ to a partial method definition. So, we can view LT as a Curried function, and we often write $LT(L)(C.m)$ for the partial method $C.m$ in L in a program. We assume that the domains of CT and LT are finite. Precisely speaking, the semantics and type system are parameterized over CT and LT but, to lighten the notation, we assume them to be fixed and omit from judgments.

Given CT and LT , **extends** and **requires** clauses are considered relations, written $\triangleleft$ and **req**, respectively, over class/layer names. Namely, we write $L \text{ req } L_i$ if $LT(L) = \text{layer } L \text{ req } \bar{L}$ and $L_i \in \bar{L}$. We also write $L \text{ req } \{\bar{L}\}$ if $LT(L) = \text{layer } L \text{ req } \bar{L}$.² As usual, we write $\mathcal{R}^+$ for the transitive closure of relation $\mathcal{R}$; similarly for $\mathcal{R}^*$ for the reflexive transitive closure of $\mathcal{R}$. We write $L \text{ swappable}$ if $LT(L)$ is defined with the swappable modifier.

We assume the following sanity conditions are satisfied by a given program:

1. $CT(C) = \text{class } C \dots$ for any $C \in \text{dom}(CT)$.
2. $\text{Object} \notin \text{dom}(CT)$.
3. For every class name C (except **Object**) appearing anywhere in CT , $C \in \text{dom}(CT)$.
4. $LT(L) = \dots \text{layer } L \dots$ for any $L \in \text{dom}(LT)$.
5. $\text{Base} \notin \text{dom}(LT)$.
6. For every layer name L (except **Base**) appearing anywhere in LT , $L \in \text{dom}(LT)$.
7. Both for classes and layers, there are no cycles in the transitive closure of the **extends** clauses.
8. $LT(L)(C.m) = \dots C.m(\dots)\{\dots\}$ for any $L \in \text{dom}(LT)$ and $C \neq \text{Object}$ and $(C.m) \in \text{dom}(LT(L))$.

These sanity conditions are an extension of those of FJ: conditions for layers (4–7) are similar to those for classes (1–3, 7). In Condition 6, like **Object** of classes, layer **Base** is defined as the root of the layer inheritance/subtyping hierarchy. In the condition (8), $C \neq \text{Object}$ means that a layer cannot introduce a method to **Object**, which has no base methods. We could allow a layer to add methods to **Object** but doing so would just clutter presentation—there are more rules to deal with the fact that **super** calls cannot be made in partial methods for **Object**.

2.2.2 Operational Semantics

Lookup Functions We need a few auxiliary lookup functions to define operational semantics and they are defined in Fig. 2.4. The function $fields(C)$ returns a sequence $\bar{T} \bar{f}$ of pairs of a field name and its type by collecting all field declarations from C and its superclasses.

The function $pmbody(m, C, L)$ returns the parameters and body $\bar{x}.e$ of the partial method $C.m$ defined in layer L . It also returns the layer name L_0 at which $C.m$ is found, which will be used in reduction rules to deal with **superproceed**. If partial method $C.m$ is not found in L , its superlayer L' is searched and so on. The function $mbody(m, C, \bar{L}_1, \bar{L}_2)$ returns the parameters and body $\bar{x}.e$ of method m in class C when the search starts from $\bar{L}_1$; the other sequence $\bar{L}_2$ keeps track of the layers that are activated when the search initially started. It also returns D and $\bar{L}''$ (which will be a prefix of $\bar{L}_2$), information on where the method has been found. For example, in the rule **MB-LAYER**, which means that the method is found in class C and layer L_0 (or its superlayers), $mbody$ returns C and $(\bar{L}'; L_0)$. Such information will be used in reduction rules to deal with **proceed** and

²Note that $L_1 \text{ req } L_2$ and $L_1 \text{ req } \{L_2\}$ have slightly different meanings; the former means L_2 is one of the layers required by L_1 , whereas the latter means L_2 is the only layer required by L_1 .

$$\boxed{fields(C) = \bar{T} \ \bar{f}}$$

$$\begin{array}{c}
fields(Object) = \bullet \\
\text{class } C \triangleleft D \{ \bar{T} \ \bar{f}; \dots \} \quad fields(D) = \bar{S} \ \bar{g} \\
\hline
fields(C) = \bar{S} \ \bar{g}, \bar{T} \ \bar{f}
\end{array}
\quad \begin{array}{l} \text{(F-OBJECT)} \\ \text{(F-CLASS)} \end{array}$$

$$\boxed{pmbody(m, C, L) = \bar{x}.e \text{ in } L_0}$$

$$\begin{array}{c}
LT(L)(C.m) = T_0 \ C.m(\bar{T} \ \bar{x}) \{ \text{return } e; \} \\
\hline
pmbody(m, C, L) = \bar{x}.e \text{ in } L
\end{array}
\quad \text{(PMB-LAYER)}$$

$$\begin{array}{c}
LT(L)(C.m) \text{ undefined} \quad L \triangleleft L' \quad pmbody(m, C, L') = \bar{x}.e \text{ in } L_0 \\
\hline
pmbody(m, C, L) = \bar{x}.e \text{ in } L_0
\end{array}
\quad \text{(PMB-SUPER)}$$

$$\boxed{mbody(m, C, \bar{L}', \bar{L}) = \bar{x}.e \text{ in } D, \bar{L}''}$$

$$\begin{array}{c}
\text{class } C \triangleleft D \{ \dots \ T_0 \ m(\bar{T} \ \bar{x}) \{ \text{return } e; \} \ \dots \} \\
\hline
mbody(m, C, \bullet, \bar{L}) = \bar{x}.e \text{ in } C, \bullet
\end{array}
\quad \text{(MB-CLASS)}$$

$$\begin{array}{c}
pmbody(m, C, L_0) = \bar{x}.e \text{ in } L_1 \\
\hline
mbody(m, C, (\bar{L}'; L_0), \bar{L}) = \bar{x}.e \text{ in } C, (\bar{L}'; L_0)
\end{array}
\quad \text{(MB-LAYER)}$$

$$\begin{array}{c}
\text{class } C \triangleleft D \{ \dots \ \bar{M} \} \quad m \notin \bar{M} \quad mbody(m, D, \bar{L}, \bar{L}) = \bar{x}.e \text{ in } E, \bar{L}' \\
\hline
mbody(m, C, \bullet, \bar{L}) = \bar{x}.e \text{ in } E, \bar{L}'
\end{array}
\quad \text{(MB-SUPER)}$$

$$\begin{array}{c}
pmbody(m, C, L_0) \text{ undefined} \quad mbody(m, C, \bar{L}', \bar{L}) = \bar{x}.e \text{ in } D, \bar{L}'' \\
\hline
mbody(m, C, (\bar{L}'; L_0), \bar{L}) = \bar{x}.e \text{ in } D, \bar{L}''
\end{array}
\quad \text{(MB-NEXTLAYER)}$$

FIGURE 2.4: ContextFJ_<: Lookup Functions.

super. Readers familiar with ContextFJ will notice that the rules for *mbody* are mostly the same as those in ContextFJ, except that *pmbody*(*m*, *C*, *L*) is substituted for *PT*(*m*, *C*, *L*) to take layer inheritance into account. By reading the four rules defining the two functions in a bottom-up manner, it is not hard to see the correspondence with the method lookup procedure, informally described in Section 2.1.6.

Reduction The operational semantics of ContextFJ_< is given by a reduction relation of the form $\bar{L} \vdash e \longrightarrow e'$, read “expression *e* reduces to *e'* under the activated layers $\bar{L}$.” The sequence $\bar{L}$ of layer names stands for nesting of *with* and the rightmost name stands for the most recently activated layer. As for other sequences, $\bar{L}$ do not contain duplicate names. Note that we put a sequence of layer names $\bar{L}$ rather than layer instances because layer instances have no fields and *new* *L*() and *L* can be identified. If we modelled fields in layer instances, we would have to put instances for layer names.

Reduction rules are found in Fig. 2.5 and Fig. 2.6. R-FIELD is for field access and four rules R-INVKXX are for method invocation: R-INVK initializes the cursor according to the currently activated layers $\bar{L}$; the rules R-INVKB and R-INVKP represent invocation of a base and partial method, respectively, depending on which kind is found by *mbody*; the rule R-INVKSP deals with the case where the cursor in the receiver object is a quadruple, which occurs when the entire expression was a *super* proceed call. In

$$\begin{array}{c}
\frac{fields(C) = \bar{C} \ \bar{f}}{\bar{L} \vdash \text{new } C(\bar{v}) . f_i \longrightarrow v_i} \quad (\text{R-FIELD}) \\
\\
\frac{\bar{L} \vdash \text{new } C(\bar{v}) \langle C, \bar{L}, \bar{L} \rangle . m(\bar{w}) \longrightarrow e'}{\bar{L} \vdash \text{new } C(\bar{v}) . m(\bar{w}) \longrightarrow e'} \quad (\text{R-INVK}) \\
\\
\frac{mbody(m, C', \bar{L}'', \bar{L}') = \bar{x} . e_0 \text{ in } C'', \bullet \quad \text{class } C'' \triangleleft D\{\dots\}}{\bar{L} \vdash \text{new } C(\bar{v}) \langle C', \bar{L}'', \bar{L}' \rangle . m(\bar{w}) \longrightarrow} \quad (\text{R-INVKB}) \\
\left[\begin{array}{ll} \text{new } C(\bar{v}) & / \text{this,} \\ \bar{w} & / \bar{x}, \\ \text{new } C(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super} \end{array} \right] e_0 \\
\\
\frac{mbody(m, C', \bar{L}'', \bar{L}') = \bar{x} . e_0 \text{ in } C'', (\bar{L}'''; L_0) \quad \text{class } C'' \triangleleft D\{\dots\} \quad \text{layer } L_0 \triangleleft L_1}{\bar{L} \vdash \text{new } C(\bar{v}) \langle C', \bar{L}'', \bar{L}' \rangle . m(\bar{w}) \longrightarrow} \\
\left[\begin{array}{ll} \text{new } C(\bar{v}) & / \text{this,} \\ \bar{w} & / \bar{x}, \\ \text{new } C(\bar{v}) \langle C'', \bar{L}''', \bar{L}' \rangle . m & / \text{proceed,} \\ \text{new } C(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super,} \\ \text{new } C(\bar{v}) \langle C'', L_1, (\bar{L}'''; L_0), \bar{L}' \rangle . m / \text{superproceed} & \end{array} \right] e_0 \quad (\text{R-INVKP}) \\
\\
\frac{pmbody(m, C', L_1) = \bar{x} . e_0 \text{ in } L_2 \quad \text{class } C' \triangleleft D\{\dots\} \quad \text{layer } L_2 \triangleleft L_3}{\bar{L} \vdash \text{new } C(\bar{v}) \langle C', L_1, (\bar{L}''; L_0), \bar{L}' \rangle . m(\bar{w}) \longrightarrow} \quad (\text{R-INVKSP}) \\
\left[\begin{array}{ll} \text{new } C(\bar{v}) & / \text{this,} \\ \bar{w} & / \bar{x}, \\ \text{new } C(\bar{v}) \langle C', \bar{L}'', \bar{L}' \rangle . m & / \text{proceed,} \\ \text{new } C(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super,} \\ \text{new } C(\bar{v}) \langle C', L_3, (\bar{L}''; L_0), \bar{L}' \rangle . m / \text{superproceed} & \end{array} \right] e_0
\end{array}$$

FIGURE 2.5: ContextFJ_<: Reduction Rules 1.

$$\begin{array}{c}
\frac{with(L, \bar{L}) = \bar{L}' \quad \bar{L}' \vdash e \longrightarrow e'}{\bar{L} \vdash \text{with new } L() \ e \longrightarrow \text{with new } L() \ e'} \quad (\text{RC-WITH}) \\
\\
\frac{swap(L, L_{sw}, \bar{L}) = \bar{L}' \quad \bar{L}' \vdash e \longrightarrow e'}{\bar{L} \vdash \text{swap}(\text{new } L(), L_{sw}) \ e \longrightarrow \text{swap}(\text{new } L(), L_{sw}) \ e'} \quad (\text{RC-SWAP}) \\
\\
\frac{\bar{L} \vdash e_l \longrightarrow e'_l}{\bar{L} \vdash \text{with } e_l \ e \longrightarrow \text{with } e'_l \ e} \quad (\text{RC-WITHARG}) \\
\\
\frac{\bar{L} \vdash e_l \longrightarrow e'_l}{\bar{L} \vdash \text{swap}(e_l, L_{sw}) \ e \longrightarrow \text{swap}(e'_l, L_{sw}) \ e} \quad (\text{RC-SWAPARG}) \\
\\
\frac{}{\bar{L} \vdash \text{with new } L() \ v \longrightarrow v} \quad (\text{R-WITHVAL}) \\
\\
\frac{}{\bar{L} \vdash \text{swap}(\text{new } L(), L_{sw}) \ v \longrightarrow v} \quad (\text{R-SWAPVAL}) \\
\\
\frac{\bar{L} \vdash e_0 \longrightarrow e'_0}{\bar{L} \vdash e_0.f \longrightarrow e'_0.f} \quad (\text{RC-FIELD}) \\
\\
\frac{\bar{L} \vdash e_i \longrightarrow e'_i}{\bar{L} \vdash e_0.m(\dots, e_i, \dots) \longrightarrow e_0.m(\dots, e'_i, \dots)} \quad (\text{RC-INVKARG}) \\
\\
\frac{\bar{L} \vdash e_0 \longrightarrow e'_0}{\bar{L} \vdash e_0.m(\bar{e}) \longrightarrow e'_0.m(\bar{e})} \quad (\text{RC-INVKRECV}) \\
\\
\frac{\bar{L} \vdash e_i \longrightarrow e'_i}{\bar{L} \vdash \text{new } C(\dots, e_i, \dots) \longrightarrow \text{new } C(\dots, e'_i, \dots)} \quad (\text{RC-NEW}) \\
\\
\frac{\bar{L} \vdash e_i \longrightarrow e'_i}{\bar{L} \vdash \text{new } C(\bar{v})\langle C', \bar{L}'', \bar{L}' \rangle.m(\dots, e_i, \dots) \longrightarrow \text{new } C(\bar{v})\langle C', \bar{L}', \bar{L}' \rangle.m(\dots, e'_i, \dots)} \quad (\text{RC-INVKAARG1}) \\
\\
\frac{\bar{L} \vdash e_i \longrightarrow e'_i}{\bar{L} \vdash \text{new } C(\bar{v})\langle C', L, \bar{L}'', \bar{L}' \rangle.m(\dots, e_i, \dots) \longrightarrow \text{new } C(\bar{v})\langle C', L, \bar{L}', \bar{L}' \rangle.m(\dots, e'_i, \dots)} \quad (\text{RC-INVKAARG2})
\end{array}$$

FIGURE 2.6: ContextFJ_<: Reduction Rules 2.

the last case, *pmbody* is used to find a method body because *superproceed* denotes a partial method in one of the superlayers.

Note how *this*, *proceed*, *super* and *superproceed* are replaced with the receiver with different cursor locations. For *proceed*, the cursor of triple moves one layer to the left and, for *super*, the cursor moves one level up in the direction of class inheritance, resetting the layers. Thanks to Sanity Condition (8), the superclass *D* is always found. If we allowed a layer to add baseless partial methods to *Object*, we would have to have special rules, in which there is no substitution for *super* (and typing rules to disallow the use of *super* in such partial methods). Igarashi et al. [IHM12] have overlooked this subtlety. For *superproceed*, the cursor moves one level up in the direction of layer inheritance (generating a quadruple from a triple in R-INVKP). For example, we show how cursors of a triple and a quadruple work using example in Section 2.1.6. Let *e* be *new C().m()*. Then, the derivation of $L1;L2;L3 \vdash e \longrightarrow e'$ will take the form:

$$\frac{\frac{mbody(m, C, (L1;L2;L3), (L1;L2;L3)) = \bullet.e4 \text{ in } C, (L1;L2;L3)}{L1;L2;L3 \vdash \text{new } C < C, (L1;L2;L3), (L1;L2;L3) > ().m() \longrightarrow e'} \text{ R-INVKP}}{L1;L2;L3 \vdash \text{new } C() .m() \longrightarrow e'} \text{ R-INVK}$$

where *e'* is

$$\left[\begin{array}{ll} \text{new } C < C, (L1;L2;L3), (L1;L2;L3) > () & / \text{this} \\ \text{new } C < D, (L1;L2;L3), (L1;L2;L3) > () & / \text{super} \\ \text{new } C < C, (L2;L3), (L1;L2;L3) > ().m & / \text{proceed} \\ \text{new } C < C, L4, (L1;L2;L3), (L1;L2;L3) > ().m / \text{superproceed} \end{array} \right] e4.$$

Now, we go back to Fig. 2.6. The rules RC-WITH and RC-SWAP express layer activation and swapping, respectively. The auxiliary functions *with*(*L*, $\bar{L}$) and *swap*(*L*, *L_{sw}*, $\bar{L}$) for context manipulation are defined by:

$$with(L, \bar{L}) = (\bar{L} \setminus \{L\}); L \quad swap(L, L_{sw}, \bar{L}) = (\bar{L} \setminus \{L' \mid L' \triangleleft^* L_{sw}\}); L$$

The function *with* removes *L* (if exists) from layer sequence $\bar{L}$ and adds *L* to the end of $\bar{L}$ and *swap* removes all sublayers of *L_{sw}* from $\bar{L}$, and adds *L* to the end of $\bar{L}$.³ The type system checks that *L_{sw}* is a swappable layer. Based on the above, the rule RC-WITH means that *with* (*new L()*) *e* executes *e* with *L* activated (as the first layer). The rule RC-SWAP is similar; it means that *swap* (*new L()*, *L_{sw}*) *e* executes by deactivating all sublayers of *L_{sw}* and activating layer *L*. For example, we can derive:

$$\frac{\frac{\vdots}{L1;L2;L3 \vdash \text{new } C() .m() \longrightarrow e'} \text{ R-INVK}}{L1;L2 \vdash \text{with new } L3() \text{ } e \longrightarrow e'} \text{ R-WITH}}{L1 \vdash \text{with new } L2() \text{ (with new } L3() \text{ } e) \longrightarrow e'} \text{ R-WITH}}{\bullet \vdash \text{with new } L1() \text{ (with new } L2() \text{ (with new } L3() \text{ } e)) \longrightarrow e'} \text{ R-WITH}$$

The rules RC-WITHARG and RC-SWAPARG are for reduction of expression *e_i* that is expected to become a layer instance. Rules RC-WITHVAL and RC-SWAPVAL are for final reduction steps of *with* and *swap* blocks, respectively, that pass the value *v* as it

³The symbol $\setminus$ is usually used to remove entities from a set, but we informally use it for a sequence here.

is. Other rules for congruence are same as those of ContextFJ: ContextFJ_<; reduction is call by value but the order of reduction of subexpressions is unspecified.

2.2.3 Type System

As usual, the role of a type system is to ensure the absence of a certain class of run-time errors. Here, they are “field-not-found” and “method-not-found” errors, including the failure of `proceed`, `superproceed` or `super` calls.

As discussed in the last section, the type system takes information on activated layers at every program point into account. We approximate such information by a set Λ of layer names, which mean that, for any layer in Λ , an instance of one of its sublayers has to be activated at run time. This set gives underapproximation of activated layers; other layers might be activated. Activated layers are approximated by sets rather than sequences because the type system is mainly concerned about access to fields and methods and the order of activated layers does not influence which fields and methods are accessible.

In our type system, a type judgment for an expression is of the form $\mathcal{L}; \Lambda; \Gamma \vdash e : T$, where Γ is a type environment, which records types of variables, and $\mathcal{L}$ stands for where e appears, namely, a method in a class (denoted by $C.m$) or a partial method in a layer (denoted by $L.C.m$). For example, the `proceed` call in the body of the partial method `People.sayWeather()` of layer `Thunder` is typed as follows:

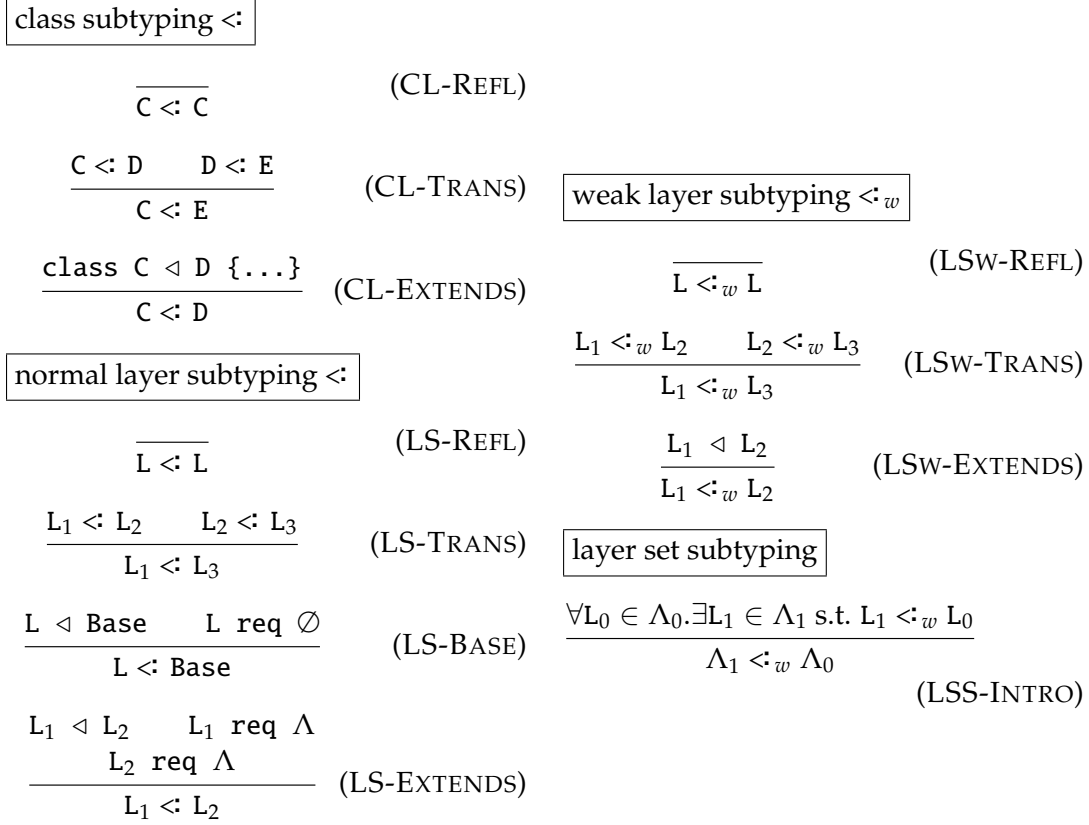
`Thunder.People.sayWeather; {Weather, Thunder}; this : People ⊢ proceed() : Text`

The layer name set `{Weather, Thunder}` comes from the fact that `Thunder` requires `Weather`. `Thunder` is also included because `Thunder` (or one of its sublayers) is obviously activated when a partial method defined in this very layer is executed.

We start with the definitions of two kinds of layer subtyping discussed in the last section and proceed to functions to look up method types and typing rules.

Subtyping We define subtyping $C \prec D$ for class types, weak subtyping $L_1 \prec_w L_2$ and normal subtyping $L_1 \prec L_2$ for layer types by the rules in Fig. 2.7. Class subtyping $C \prec D$ is defined as the reflexive and transitive closure of $\prec$, just as FJ. Weak layer subtyping is also the reflexive and transitive closure of $\prec$. We extend it to the relation $\Lambda_1 \prec_w \Lambda_2$ between layer name sets by LSS-INTRO: $\Lambda_1 \prec_w \Lambda_2$ if and only if for every element in Λ_2 , there must exist a sublayer of it in Λ_1 . It is used to check activated layers Λ_1 satisfy the requirement Λ_2 given by a `requires` clause in typechecking a layer activation. Normal subtyping is almost the reflexive and transitive closure of $\prec$ but there is one additional condition: for L_1 to be a normal subtype of L_2 , the layers they require must be the same (LS-EXTENDS). Obviously, if $L_1 \prec L_2$, then $L_1 \prec_w L_2$ (but not vice versa).

Method type lookup Similarly to *pmbody* and *mbody*, we define two auxiliary functions *pmtype* and *mtype* to look up the signature $\bar{T} \rightarrow T_0$ (consisting of argument type $\bar{T}$ and a return type T_0) of a (partial) method. *pmtype*(m, C, L) returns the signature of $C.m$ in L (or one of its superlayers). *mtype*($m, C, \Lambda_1, \Lambda_2$) returns the type of m in C under the assumption that Λ_1 is activated. The other layer set Λ_2 ($\supseteq \Lambda_1$) is used when the lookup goes on to a superclass. If Λ_1 and Λ_2 are the same, which is mostly the case, we write *mtype*(m, C, Λ_1).

FIGURE 2.7: ContextFJ $_{<:}$: Subtyping Relations.

These rules by themselves do not define *mtype* as a function, because different layers may contain partial methods of the same name with different signatures. So, precisely speaking, it should rather be understood as a relation; in a well-typed program, it will behave as a function, though.

Expression Typing As mentioned already, the type judgment for expressions is of the form $\mathcal{L}; \Lambda; \Gamma \vdash e : T$, read “ e is given type T under context Γ , location $\mathcal{L}$ and layer set Λ ”. In addition to $C.m$ and $L.C.m$, $\mathcal{L}$ can be $\bullet$, which means the top-level (i.e., under execution). Typing rules are given in Fig. 2.9. We defer typing rules for run-time expressions $\text{new } C(\bar{v}) \triangleleft D, \bar{L}', \bar{L} \rangle .m(\bar{e})$ and $\text{new } C(\bar{v}) \triangleleft D, L, \bar{L}', \bar{L} \rangle .m(\bar{e})$ to the next section and focus on expressions that appear class and layer definitions.

Rules T-VAR, T-FIELD are easy. T-NEW and T-NEWL are for instance of classes and instance of layers, respectively. The rule T-INVK is straightforward: the method signature $\bar{T} \rightarrow T_0$ is retrieved from the receiver type C_0 and Λ ; the types of the actual arguments must be subtypes of $\bar{T}$; and the whole expression is given the method return type T_0 . The rule T-WITH checks, by $\Lambda <:_w \Lambda'$, that the layers required by L —the type of the layer to be activated—are already activated and that the body e_0 is well typed under the assumption that L is additionally activated. T-SWAP is similar; the set Λ_{rm} stands for the set of layers after deactivation and must be a weak subtype of the required set Λ' . The last four rules are for *super*, *proceed*, and *superproceed* calls and so they are similar to T-INVK. Differences are in how the method signature is

$$\boxed{pmtype(m, C, L) = \bar{T} \rightarrow T_0}$$

$$\frac{LT(L)(C.m) = T_0 \quad C.m(\bar{T} \ \bar{x})\{ \text{return } e; \} }{pmtype(m, C, L) = \bar{T} \rightarrow T_0} \quad (\text{PMT-LAYER})$$

$$\frac{LT(L)(C.m) \text{ undefined} \quad L \triangleleft L' \quad pmtype(m, C, L') = \bar{T} \rightarrow T_0}{pmtype(m, C, L) = \bar{T} \rightarrow T_0} \quad (\text{PMT-SUPER})$$

$$\boxed{mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0}$$

$$\frac{\text{class } C \triangleleft D \{ \dots \quad T_0 \ m(\bar{T} \ \bar{x})\{ \text{return } e; \} \quad \dots \} }{mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0} \quad (\text{MT-CLASS})$$

$$\frac{\exists L \in \Lambda_1. pmtype(m, C, L) = \bar{T} \rightarrow T_0}{mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0} \quad (\text{MT-PMETHOD})$$

$$\frac{\text{class } C \triangleleft D \{ \dots \quad \bar{M} \} \quad m \notin \bar{M} \quad \forall L \in \Lambda_1. pmtype(m, C, L) \text{ undefined} \quad mtype(m, D, \Lambda_2, \Lambda_2) = \bar{T} \rightarrow T_0}{mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0} \quad (\text{MT-SUPER})$$

FIGURE 2.8: ContextFJ_<: Method Type Lookup Functions.

obtained. In the rules T-SUPERB and T-SUPERP for a *super* call in a method defined in a class and in a partial method, respectively, the superclass E is given to $mtype$. Layer names are taken from the *requires* clause instead of Λ —corresponding to the fact that the method to be invoked is not affected by *with* or *swap* surrounding *super* (a class cannot require any layer, hence the empty set). In the rule T-PROCEED for a *proceed* call, the current class name C is used. Similarly to T-SUPERP, layer names are taken from the *require* clause. The last argument to $mtype$ is $\Lambda \cup \{L\}$ because a *proceed* call can proceed to a partial method $D.m$ (where D is a superclass of C) defined in the same layer L . In the rule T-SUPERPROCEED, $pmtype$ is used instead of $mtype$.

In Igarashi et al. [IHM12], in which a type system for ContextFJ is developed, another layer activation construct called *ensure* is adopted. The difference from *with* is that, if an already activated layer is to be activated, *ensure* does not change the activated layer sequence, whereas *with* will pull that layer to the head of the sequence so that partial methods in it are invoked first. For example, activating layers $L1, L2, L1$ in this order results in $L1; L2$ with *ensure* but in $L2; L1$ with the *with* statement. Igarashi et al. argue that the rearrangement of layers by *with* destroys the layer ordering in which interlayer dependency is respected. For example, if $L2$ requires $L1$, then $L2; L1$ violates the *require* clause in the sense that the layers that $L2$ requires do not precede $L2$ in the sequence. So, for simplicity, Igarashi et al. considered only *ensure*, which does not have this problem.

Our discovery is that, in fact, this anomaly caused by *with* is not really a problem for type soundness and essentially the same typing rule works—Our typing rule T-WITH for *with* is indeed very similar to that for *ensure* in ContextFJ; the only difference is the use of $\subseteq$ in the place of weak subtyping $<_w$ (ContextFJ does not have layer subtyping). The reason why a layer sequence like $L2; L1$ is not problematic can be explained as follows. Actually, problematic would be a partial method defined in $L2$ calling another (partial) method, say $C.m$, that exists only in $L1$ —that is, one that is undefined in a base

$\mathcal{L}; \Lambda; \Gamma \vdash e : T$

$$\begin{array}{c}
\frac{(\Gamma = \bar{x} : \bar{T})}{\mathcal{L}; \Lambda; \Gamma \vdash x_i : T_i} \quad \text{(T-VAR)} \\
\\
\frac{\mathcal{L}; \Lambda; \Gamma \vdash e_0 : C_0 \quad \text{fields}(C_0) = \bar{T} \ \bar{f}}{\mathcal{L}; \Lambda; \Gamma \vdash e_0.f_i : T_i} \quad \text{(T-FIELD)} \\
\\
\frac{\mathcal{L}; \Lambda; \Gamma \vdash e_0 : C_0 \quad \text{mtype}(\mathbf{m}, C_0, \Lambda) = \bar{T} \rightarrow T_0 \quad \mathcal{L}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{\mathcal{L}; \Lambda; \Gamma \vdash e_0.\mathbf{m}(\bar{e}) : T_0} \quad \text{(T-INVK)} \\
\\
\frac{\text{fields}(C_0) = \bar{T} \ \bar{f} \quad \mathcal{L}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{\mathcal{L}; \Lambda; \Gamma \vdash \text{new } C_0(\bar{e}) : C_0} \quad \text{(T-NEW)} \\
\\
\frac{}{\mathcal{L}; \Lambda; \Gamma \vdash \text{new } L_0() : L_0} \quad \text{(T-NEWL)} \\
\\
\frac{\mathcal{L}; \Lambda; \Gamma \vdash e_l : L \quad L \text{ req } \Lambda' \quad \Lambda <_w \Lambda' \quad \mathcal{L}; \Lambda \cup \{L\}; \Gamma \vdash e_0 : T_0}{\mathcal{L}; \Lambda; \Gamma \vdash \text{with } e_l \ e_0 : T_0} \quad \text{(T-WITH)} \\
\\
\frac{\mathcal{L}; \Lambda; \Gamma \vdash e_l : L \quad L <_w L_{sw} \quad L_{sw} \text{ swappable} \quad L \text{ req } \Lambda' \quad \Lambda_{rm} = \Lambda \setminus \{L' \mid L' <_w L_{sw}\} \quad \Lambda_{rm} <_w \Lambda' \quad \mathcal{L}; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T_0}{\mathcal{L}; \Lambda; \Gamma \vdash \text{swap } (e_l, L_{sw}) \ e_0 : T_0} \quad \text{(T-SWAP)} \\
\\
\frac{\text{class } C \triangleleft E \ \{\dots\} \quad \text{mtype}(\mathbf{m}', E, \emptyset) = \bar{T} \rightarrow T_0 \quad C.\mathbf{m}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{C.\mathbf{m}; \Lambda; \Gamma \vdash \text{super}.\mathbf{m}'(\bar{e}) : T_0} \quad \text{(T-SUPERB)} \\
\\
\frac{\text{class } C \triangleleft E \ \{\dots\} \quad L \text{ req } \Lambda' \quad \text{mtype}(\mathbf{m}', E, \Lambda' \cup \{L\}) = \bar{T} \rightarrow T_0 \quad L.C.\mathbf{m}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{L.C.\mathbf{m}; \Lambda; \Gamma \vdash \text{super}.\mathbf{m}'(\bar{e}) : T_0} \quad \text{(T-SUPERP)} \\
\\
\frac{L \text{ req } \Lambda' \quad \text{mtype}(\mathbf{m}, C, \Lambda', \Lambda' \cup \{L\}) = \bar{T} \rightarrow T_0 \quad L.C.\mathbf{m}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{L.C.\mathbf{m}; \Lambda; \Gamma \vdash \text{proceed}(\bar{e}) : T_0} \quad \text{(T-PROCEED)} \\
\\
\frac{L \triangleleft L' \quad \text{pmtype}(\mathbf{m}, C, L') = \bar{T} \rightarrow T_0 \quad L.C.\mathbf{m}; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}}{L.C.\mathbf{m}; \Lambda; \Gamma \vdash \text{superproceed}(\bar{e}) : T_0} \quad \text{(T-SUPERPROCEED)}
\end{array}$$

FIGURE 2.9: ContextFJ_{<:}: Expression Typing.

M ok in C

$$\frac{C.m; \emptyset; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0 \quad S_0 <: T_0}{T_0 \text{ m}(\bar{T} \ \bar{x}) \ \{ \text{return } e_0; \} \text{ ok in } C} \quad (\text{T-METHOD})$$

PM ok in L

$$\frac{L \text{ req } \Lambda \quad L.C.m; \Lambda \cup \{L\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0 \quad S_0 <: T_0}{T_0 \text{ C.m}(\bar{T} \ \bar{x}) \ \{ \text{return } e_0; \} \text{ ok in } L} \quad (\text{T-PMETHOD})$$

CL ok

$$\frac{K = C(\bar{S} \ \bar{g}, \ \bar{T} \ \bar{f}) \{ \text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}; \} \quad \text{fields}(D) = \bar{S} \ \bar{g} \quad \bar{M} \text{ ok in } C}{\text{class } C < D \ \{ \bar{T} \ \bar{f}; \ K \ \bar{M} \} \text{ ok}} \quad (\text{T-CLASS})$$

LA ok

$$\frac{\begin{array}{l} L \text{ is not sublayer of any swappable layer} \\ L' \text{ req } \Lambda' \quad \{\bar{L}\} <_w \Lambda' \quad \bar{P}\bar{M} \text{ ok in } L \\ \text{[swappable]} \text{ layer } L \text{ req } \bar{L} < L' \ \{ \bar{P}\bar{M} \} \text{ ok} \end{array}}{\begin{array}{l} L <^+ L_{sw} \quad L_{sw} \text{ swappable} \\ L' \text{ req } \Lambda' \quad \{\bar{L}\} = \Lambda' \\ \bar{P}\bar{M} \text{ ok in } L \quad \forall C.m \in \{\bar{P}\bar{M}\}. \text{pmtyp}(m, C, L_{sw}) \text{ defined} \\ \neg \exists L_2 \in \text{dom}(LT). L_2 \text{ req } L \end{array}}{\text{layer } L \text{ req } \bar{L} < L' \ \{ \bar{P}\bar{M} \} \text{ ok}} \quad (\text{T-LAYER})$$

$$\frac{\begin{array}{l} L <^+ L_{sw} \quad L_{sw} \text{ swappable} \\ L' \text{ req } \Lambda' \quad \{\bar{L}\} = \Lambda' \\ \bar{P}\bar{M} \text{ ok in } L \quad \forall C.m \in \{\bar{P}\bar{M}\}. \text{pmtyp}(m, C, L_{sw}) \text{ defined} \\ \neg \exists L_2 \in \text{dom}(LT). L_2 \text{ req } L \end{array}}{\text{layer } L \text{ req } \bar{L} < L' \ \{ \bar{P}\bar{M} \} \text{ ok}} \quad (\text{T-LAYERSW})$$

FIGURE 2.10: ContextFJ_<: Method/Class/Layer Typing.

class—via `proceed`.⁴ Such a dangling partial method cannot be executed, however: $C.m$ in L_1 cannot contain `proceed`, which leads to execution of the dangling partial method, because L_1 is activated first, meaning that L_1 does not require any other layer, but it is assumed here that m is not defined in base class C .

Typing for Methods, Partial Methods, Classes, Layers, and Programs Typing rules for (partial) methods, layers, and classes and are given in Fig. 2.10. The rule T-METHOD is standard. Readers familiar with FJ may notice that a condition for valid overriding is missing; it is put in elsewhere—see below. The rule T-PMETHOD for a partial method means that the method body e_0 is typed under the layer set required by this layer. The rule T-LAYER is for layers that are not sublayers of any swappable layer and demands that the `requires` clause of the layer be *covariant* and all partial methods are well formed. The rule T-LAYERSW is for sublayers of swappable layers. It demands, in addition to the conditions described in T-LAYER, that the `requires` clause of this layer be the same as those of its parent swappable layer, that no partial method be newly introduced, and that this layer be not required by other layers. The last condition requires a global program analysis.

⁴Invoking m via `this` or `super` will find m in L_1 .

Valid overriding $noconflict(L_1, L_2), override^h(L, C), override^v(C)$

$$\frac{\forall m, C, \bar{T}, T_0, \bar{S}, S_0. \text{ if } LT(L_1)(C.m) = T_0 \ m(\bar{T} \ \bar{x})\{\dots\} \\ \text{ and } LT(L_2)(C.m) = S_0 \ m(\bar{S} \ \bar{y})\{\dots\}, \text{ then } \bar{T}, T_0 = \bar{S}, S_0}{noconflict(L_1, L_2)}$$

$$\frac{\forall m, \bar{T}, T_0, \bar{S}, S_0, \bar{x}. \text{ if } LT(L)(C.m) = S_0 \ m(\bar{S} \ \bar{x})\{\dots\} \\ \text{ and } mtype(m, C, \emptyset, dom(LT)) = \bar{T} \rightarrow T_0, \text{ then } \bar{T}, T_0 = \bar{S}, S_0}{override^h(L, C)}$$

$$\frac{\forall m, D, \bar{T}, T_0, \bar{S}, S_0. \text{ if class } C \triangleleft D \ \{\dots \ S_0 \ m(\bar{S} \ x)\{\dots\} \dots\} \\ \text{ and } mtype(m, D, dom(LT), dom(LT)) = \bar{T} \rightarrow T_0, \\ \text{ then } \bar{T} = \bar{S} \text{ and } S_0 <: T_0}{override^v(C)}$$

 $\vdash (CT, LT) \text{ ok} \quad \vdash (CT, LT, e) : T$

$$\frac{\forall C \in dom(CT). CT(C) \text{ ok} \quad \forall L \in dom(LT). LT(L) \text{ ok} \\ \forall L_1, L_2 \in dom(LT). noconflict(L_1, L_2) \\ \forall C \in dom(CT). L \in dom(LT). override^h(L, C) \quad \forall C \in dom(CT). override^v(C)}{\vdash (CT, LT) \text{ ok}} \quad (\text{T-TABLE})$$

$$\frac{\vdash (CT, LT) \text{ ok} \quad \bullet; \emptyset; \bullet \vdash e : T}{\vdash (CT, LT, e) : T} \quad (\text{T-PROG})$$

FIGURE 2.11: ContextFJ_<: Program Typing.

It is worth elaborating the rule T-LAYERSW in more detail. First, if the condition $\{\bar{L}\} = \Lambda'$ were $\{\bar{L}\} <_w \Lambda'$ (as in T-LAYER), the type system would be unsound. A counterexample is below:

```
class C {}
swappable layer L0 { int C.m() { return 0; } }
layer L1 extends L0 {}
layer L2 extends L0 requires L { int C.m() { return proceed(); } }
layer L requires L0 { int C.m() { return proceed(); } }
```

Layer L2 additionally requires L, which requires L0, a swappable superclass of L2. The condition $\{\bar{L}\} <_w \Lambda'$ would be trivially satisfied for L2 because the *requires* clause of L0 is empty. The partial methods in L2 and L are well formed because L and L0, respectively, provide definitions to *proceed*. Under these classes and layers, the following expression

```
with (new L1())
  with (new L())           // fulfills "requires L0"
    swap(L0, new L2())     // fulfills "requires L"
      new C().m()
```

is well typed, because L1, which is a subclass of L0, is activated before activating L, and L is activated before activating L2. However, the *swap* expression executed under L1; L

would get stuck as follows:

$$\begin{aligned}
 L1;L \vdash \text{swap}(L0, \text{new } L2()) \text{ new } C().m() \\
 \longrightarrow \text{swap}(L0, \text{new } L2()) \text{ new } C<C,L,(L;L2)>().m() \\
 \longrightarrow \text{swap}(L0, \text{new } L2()) \text{ new } C<C,\bullet,(L;L2)>().m() \\
 \not\rightarrow
 \end{aligned}$$

The method invocation would take place under $L;L2$, both of which have $C.m$ but the second proceed call goes nowhere.

Second, if a subclass of a swappable layer were allowed to define a new method (which is not defined in the swappable), then the type system would be unsound, too. Consider the following classes and layers.

```

class C {}
class D extends C {}

swappable layer L0 {}
layer L1 extends L0 {}
layer L2 extends L0 {
  int C.m() { return this.m(); }
  int D.m() { return swap(L, new L2()) super.m(); }
}

```

Layer $L2$ defines new partial methods $C.m$ and $D.m$. They are well formed: in particular, $\text{super.m}()$ is well typed because $L2$ itself provides $C.m$. The following expression

```
with (new L2()) new D().m()
```

is well typed, since $D.m$ invoked with $L2$ activated. However, reduction of $\text{new } D().m()$ under $L2$ would get stuck:

$$\begin{aligned}
 L2 \vdash \text{new } D().m() \\
 \longrightarrow \text{swap}(L, \text{new } L1()) \text{ new } D<C,L2,L2>().m() \\
 \longrightarrow \text{swap}(L, \text{new } L1()) \text{ new } D().m() \\
 \not\rightarrow
 \end{aligned}$$

Since super calls are not affected by swap , $\text{super.m}()$ in $D.m$ succeeds but, by the time $\text{this.m}()$ is executed, $L2$ will be swapped out.

Fig. 2.11 is for program typing; a program is well typed if all classes and layers in CT and LT , respectively, are well formed and the main expression e is typed (at the top-level $\bullet$).

The most involved is the rule to check valid method overriding used in T-TABLE. The predicate *noconflict* means that for two partial methods of the same (qualified) name must have the same signature. The predicate *override^h* means that, for any partial method, the overridden method (base method in C or partial methods for C 's superclass) must have the same signature. The predicate *override^v* means that a base method can override a (partial) method in its superclass (or layers modifying it) with a covariant return type. Note that, unlike Java, checking valid method overriding requires a whole program because a layer may add a new method to a base class, one of whose subclass may accidentally define a method of the same name without knowing of that layer.

$$\begin{array}{c}
\overline{\emptyset wf} \quad (WF-EMPTY) \\
\\
\frac{\Lambda wf \quad L_a \text{ req } \Lambda' \quad \Lambda <_w \Lambda'}{\Lambda \cup \{L_a\} wf} \quad (WF-WITH) \\
\\
\frac{\Lambda wf \quad L_{sw} \text{ swappable} \quad L <_w L_{sw} \quad L \text{ req } \Lambda' \quad \Lambda_{rm} = \Lambda \setminus \{L' \mid L' <_w L_{sw}\} \quad \Lambda_{rm} <_w \Lambda'}{\Lambda_{rm} \cup \{L\} wf} \quad (WF-SWAP)
\end{array}$$

FIGURE 2.12: ContextFJ_<: Layer set well-formedness.

2.3 Type Soundness

In this section, we prove type soundness of ContextFJ_< via subject reduction and progress [WF94]. Strictly speaking, we should present typing rules for run-time expressions first before stating these properties but, for ease of understanding, we will reverse the order and start with the statements of the properties.

Since we model the execution of a main method starting with no layers activated, we are mainly interested in the case where $\mathcal{L}$ is $\bullet$ and the layer sequence is empty. However, we have to strengthen the statements of these properties so that the layer sequence can be nonempty. We introduce the notion of *well-formed layer sets* for this purpose.

We define the relation $\{\bar{L}\} wf$, read “layer set $\{\bar{L}\}$ is well formed,” by the rules in Fig. 2.12. Intuitively, a set of layers is well-formed if one can obtain the layers by activating them one by one so that `requires` clauses are satisfied. We ignore the order of activation because the `with` statement can change the order of activated layers by activating an already activated layer again.

Aside from layer well-formedness, the statements of subject reduction, progress, and type soundness are standard:

2.3.1 Theorem [Subject Reduction]: Suppose $\vdash (CT, LT)$ ok. If $\bullet; \{\bar{L}\}; \Gamma \vdash e : T$ and $\{\bar{L}\} wf$ and $\bar{L} \vdash e \longrightarrow e'$, then $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S$ for some S such that $S < T$.

2.3.2 Theorem [Progress]: Suppose $\vdash (CT, LT)$ ok. If $\bullet; \{\bar{L}\}; \bullet \vdash e : T$ and $\{\bar{L}\} wf$, then e is a value or $\bar{L} \vdash e \longrightarrow e'$ for some e' .

2.3.3 Theorem [Type Soundness]: If $\vdash (CT, LT, e) : T$ and e reduces to a normal form under the empty set of layers, then the normal form is `new S( $\bar{v}$ )` for some $\bar{v}$ and S such that $S < T$.

2.3.1 Typing Rules for Run-time Expressions

To prove the theorems above, we have to give typing rules for run-time expressions of the forms `new C( $\bar{v}$ )<D,  $\bar{L}'$ ,  $\bar{L}$ >.m( $\bar{e}$ )` and `new C( $\bar{v}$ )<D, L,  $\bar{L}'$ ,  $\bar{L}$ >.m( $\bar{e}$ )`, which are not supposed to appear in a class/layer table. The typing rules with the rules for a few auxiliary judgments are given in Fig. 2.13:

In the rule T-INVKA for `new C0( $\bar{v}$ )<D0,  $\bar{L}'$ ,  $\bar{L}$ >.m( $\bar{e}$ )`, the premises except for $C_0.m \vdash \langle D_0, \bar{L}', \bar{L} \rangle \text{ ok}$ and $\Lambda <_{sw} \{\bar{L}\}$ —they are explained in detail below—are similar to T-INVK. The method signature is obtained by using the current cursor $\langle D_0, \bar{L}', \bar{L} \rangle$. The

$$\begin{array}{c}
\frac{\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0 \quad C_0.m \vdash \langle D_0, \bar{L}', \bar{L} \rangle \text{ ok} \quad \Lambda \prec_{sw} \{\bar{L}\} \quad mtype(m, D_0, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T}' \rightarrow T_0 \quad \bullet; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} \prec \bar{T}'}{\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) \langle D_0, \bar{L}', \bar{L} \rangle . m(\bar{e}) : T_0} \quad (\text{T-INVKA}) \\
\\
\frac{\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0 \quad C_0.m \vdash \langle D_0, (\bar{L}''; L_0), \bar{L} \rangle \text{ ok} \quad \Lambda \prec_{sw} \{\bar{L}\} \quad L_0 \prec_w L_1 \quad pmttype(m, D_0, L_1) = \bar{T}' \rightarrow T_0 \quad \bullet; \Lambda; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} \prec \bar{T}'}{\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) \langle D_0, L_1, (\bar{L}''; L_0), \bar{L} \rangle . m(\bar{e}) : T_0} \quad (\text{T-INVKAL}) \\
\\
\frac{\forall L_0 \in \Lambda_0. \exists L_1 \in \Lambda_1. (L_1 \prec_w L_0 \text{ or } \exists L_2 \in dom(LT). L_2 \text{ swappable and } L_0 \prec_w L_2 \text{ and } L_1 \prec_w L_2)}{\Lambda_1 \prec_{sw} \Lambda_0} \quad (\text{LSSW-INTRO}) \\
\\
\frac{C \prec D \quad \{\bar{L}_2\} wf \quad ndp(m, D, \bar{L}_1, \bar{L}_2)}{C.m \vdash \langle D, \bar{L}_1, \bar{L}_2 \rangle \text{ ok}} \quad (\text{WF-CURSOR}) \\
\\
\frac{\text{class } C \{ \dots C_0 \ m(\dots) \{ \dots \} \dots \}}{ndp(m, C, \bar{L}_1, (\bar{L}_1; \bar{L}_2))} \quad (\text{NDP-CLASS}) \\
\\
\frac{\exists L_0 \in \bar{L}_1. \text{proceed} \notin pmbody(m, C, L_0)}{ndp(m, C, \bar{L}_1, (\bar{L}_1; \bar{L}_2))} \quad (\text{NDP-LAYER}) \\
\\
\frac{C \triangleleft D \quad ndp(m, D, (\bar{L}_1; \bar{L}_2), (\bar{L}_1; \bar{L}_2))}{ndp(m, C, \bar{L}_1, (\bar{L}_1; \bar{L}_2))} \quad (\text{NDP-SUPER})
\end{array}$$

FIGURE 2.13: ContextFJ_<: Runtime Expression Typing.

rule T-INVKAL for a method invoked by superproceed is similar. One difference is that the method signature is obtained by using $pmtree$; the receiver is derived from a superproceed call that originated from a superlayer of L_0 , hence $L_0 <_{sw} L_1$.

The condition $\Lambda <_{sw} \{\bar{L}\}$ relates the layer sequence $\bar{L}$ in the cursor and Λ , which intuitively represents the set of layers activated at this program point. In many cases, $\Lambda = \{\bar{L}\}$ holds but if *super* and *proceed* calls are surrounded by *with* or *swap*, they can be different. The relation $<_{sw}$ is similar to $<_w$ but the additional clauses $\exists L_2 \in \text{dom}(LT). L_2 \text{ swappable}$ and $L_0 <_w L_2$ and $L_1 <_w L_2$ take into account the possibility that a layer in Λ may be activated by swapping layers in $\{\bar{L}\}$ out.

The judgment $C.m \vdash \langle D, \bar{L}_1, \bar{L}_2 \rangle \text{ ok}$, which means that the cursor is well formed with respect to method m in class C , is defined by WF-CURSOR. It requires that D to be a superclass of C and $\bar{L}_2$ to be well formed. The last condition $ndp(m, D_0, \bar{L}', \bar{L})$ (standing for “non-dangling proceed”) intuitively means “a chain of proceed calls from the given cursor location $\langle D_0, \bar{L}', \bar{L} \rangle$ eventually reaches a (partial) method that does *not* call *proceed*” and is defined by the rules NDP-CLASS, NDP-LAYER and NDP-SUPER, which are straightforward. (Here, “*proceed* $\notin pmbody(m, C, L_0)$ ” means that there is no *proceed* calls in the method body obtained by $pmbody(m, C, L_0)$.) This predicate represents an invariant condition throughout a chain of *proceed* calls and ensures there will not be a dangling *proceed* call.

2.3.2 Subject Reduction

The proof of subject reduction is done by induction on $\bar{L} \vdash e \longrightarrow e'$. Similarly to FJ, one main lemma is the Substitution Lemma, which is used in the case where e is a method invocation and states substitution of values of types $\bar{T}$ for variables of types $\bar{S}$, where $\bar{S}$ are subtypes of $\bar{T}$, in a well typed term preserves typing. Another important lemma here is Lemma 2.3.8, which states substitution for *proceed*, *super*, and *superproceed* preserves typing.

We state several main lemmas to prove the theorems above; their proofs as well as other lemmas and proofs are found in Appendix. We fix CT and LT and assume $(CT, LT) \text{ ok}$ in the rest of this section.

As usual, adding an unused variable to the type environment preserves typing (Weakening). Narrowing usually refers to the property that replacing the type of a variable in the type environment with its subtype preserves typing; here, we need narrowing with respect to (extended) layer set subtyping $<_{sw}$. The next lemma states that a well typed value remains well typed regardless of its typing context $(\mathcal{L}; \Lambda; \Gamma)$.

2.3.4 Lemma [Weakening]: If $\mathcal{L}; \Lambda; \Gamma \vdash e : T$, then $\mathcal{L}; \Lambda; \Gamma, x : S \vdash e : T$.

2.3.5 Lemma [Layer Set Narrowing]: If $\mathcal{L}; \Lambda; \Gamma \vdash e : T$ and $\Lambda' <_{sw} \Lambda$, then $\mathcal{L}; \Lambda'; \Gamma \vdash e : T$.

2.3.6 Lemma [Strengthening for values]: If $\mathcal{L}; \Lambda; \Gamma \vdash v : T$ then, $\mathcal{L}'; \Lambda'; \Gamma' \vdash v : T$.

The statement of the Substitution Lemma is straightforward.

2.3.7 Lemma [Substitution]: If $\mathcal{L}; \Lambda; \Gamma, \bar{x} : \bar{T} \vdash e : T$ and $\mathcal{L}; \Lambda; \Gamma \vdash \bar{v} : \bar{S}$ and $\bar{S} < \bar{T}$, then $\mathcal{L}; \Lambda; \Gamma \vdash [\bar{v}/\bar{x}]e : S$ and $S < T$ for some S .

The next lemma states that substitution for *proceed*, *super*, and *superproceed* preserves typing. The first item is for an invocation of a partial method, which may contain

proceed and superproceed calls as well as super calls; the second is for a base method, which may contain only super calls. The conditions, which look rather complicated, correspond to the premises of T-INVKA and T-INVKA_L.

2.3.8 Lemma [Substitution for super, proceed and superproceed]:

1. If $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ and $L.C.m; \Lambda; \Gamma \vdash e : T$ and $C_0.m \vdash \langle C, (\bar{L}'; L''), \bar{L} \rangle \text{ ok}$ and $C \triangleleft D$ and $L'' \prec_w L \triangleleft L'$ and $\Lambda \prec_{sw} \{\bar{L}\}$ and $\text{proceed} \in e \implies \text{ndp}(m, C, \bar{L}', \bar{L})$, then $\bullet; \Lambda; \Gamma \vdash Se : T$ where

$$S = \left[\begin{array}{ll} \text{new } C_0(\bar{v}) \langle C, \bar{L}', \bar{L} \rangle . m & / \text{proceed,} \\ \text{new } C_0(\bar{v}) \langle D, \bar{L}, \bar{L} \rangle & / \text{super,} \\ \text{new } C_0(\bar{v}) \langle C, L', (\bar{L}'; L''), \bar{L} \rangle . m & / \text{superproceed} \end{array} \right].$$

2. If $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ and $C.m; \Lambda; \Gamma \vdash e : T$ and $C_0.m \vdash \langle C, \bar{L}', \bar{L} \rangle \text{ ok}$ and $C \triangleleft D$ and $\Lambda \prec_{sw} \{\bar{L}\}$, then $\bullet; \Lambda; \Gamma \vdash [\text{new } C_0(\bar{v}) \langle D, \bar{L}, \bar{L} \rangle / \text{super}]e : T$.

The next two lemmas state method bodies obtained by *pmbody* and *mbody* are well typed according to the type information obtained by *pmtyp* and *mtyp*, respectively.

2.3.9 Lemma [Inversion for partial method body]: If $\text{pmbody}(m, C, L) = \bar{x}.e_0$ in L' and $L \text{ req } \Lambda$ and $\text{pmtyp}(m, C, L) = \bar{T} \rightarrow T_0$, then $L.C.m; \Lambda \cup \{L\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0$ for some $S_0 \prec_w T_0$.

2.3.10 Lemma [Inversion for method body]: Suppose $\{\bar{L}\} \text{ wf}$ and $\text{mbody}(m, C, \bar{L}', \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$ and $\text{mtyp}(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ and $\text{ndp}(m, C, \bar{L}', \bar{L})$.

1. If $\bar{L}'' = \bar{L}'''; L_0$, then $L_0 \text{ req } \Lambda$ and $L_0.C'.m; \Lambda \cup \{L_0\}; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : U_0$ and $C \prec C'$ and $U_0 \prec T_0$ and $\text{ndp}(m, C', \bar{L}'', \bar{L})$ for some Λ and U_0 .
2. If $\bar{L}'' = \bullet$, then $C'.m; \emptyset; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : U_0$ and $C \prec C'$ and $U_0 \prec T_0$ and $\text{ndp}(m, C', \bullet, \bar{L})$ for some U_0 .

We also need additional lemmas derived from runtime conditions. Layer-set well-formedness $\Lambda \text{ wf}$ provides two important properties. The first states that a well formed layer set is closed under the requires clause and the second that, if method m is found in C (under the assumption that Λ activated) but not in its direct superclass D , then at least one of those methods does not call proceed. This lemma is used to prove the next lemma (Lemma 2.3.13), which derives *ndp* for an initial cursor of the form $\langle C, \bar{L}, \bar{L} \rangle$.

2.3.11 Lemma: If $\Lambda \text{ wf}$, then $\forall L \in \Lambda, \forall L' \text{ s.t. } L \text{ req } L', \exists L'' \in \Lambda. L'' \prec_w L'$.

2.3.12 Lemma: If $\Lambda \text{ wf}$ and $\text{mtyp}(m, C, \Lambda)$ defined and $\text{mtyp}(m, D, \Lambda)$ undefined and $C \triangleleft D$, then $(\exists L' \in \Lambda. \text{proceed} \notin \text{pmbody}(m, C, L'))$ or $\text{mtyp}(m, C, \emptyset, \Lambda)$ defined.

2.3.13 Lemma: If $\{\bar{L}\} \text{ wf}$ and $\text{mtyp}(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$, then $\text{ndp}(m, C, \bar{L}, \bar{L})$.

As stated below, the predicate *ndp* ensures the existence of a method:

2.3.14 Lemma: If $\text{ndp}(m, C, \bar{L}', \bar{L})$ for some $m, C, \bar{L}'$ and $\bar{L}$, then $\text{mtyp}(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ for some $\bar{T}$ and T_0 .

2.3.3 Progress

To prove the Progress Theorem, we need the following two lemmas, which show the existence of a method body from well definedness of $pmttype$ and $mtype$.

2.3.15 Lemma: If $pmttype(m, C, L) = \bar{T} \rightarrow T_0$, then there exist $\bar{x}$ and e_0 and L' ($\neq \text{Base}$) such that $pmbody(m, C, L) = \bar{x}.e_0$ in L' and the lengths of $\bar{x}$ and $\bar{T}$ are equal and $L <_w L'$.

2.3.16 Lemma: If $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ and $\bar{L}'$ is a prefix of $\bar{L}$ and $\{\bar{L}\} wf$, then there exist $\bar{x}$ and e_0 and $\bar{L}''$ and C' ($\neq \text{Object}$) such that $mbody(m, C, \bar{L}, \bar{L}') = \bar{x}.e_0$ in $C', \bar{L}''$ and the lengths of $\bar{x}$ and $\bar{T}$ are equal and, if $\bar{L}''$ is not empty, the last layer name of $\bar{L}''$ is not Base.

2.4 Summary

We have developed a formal type system for a small COP language with layer inheritance, layer subtyping, layer swapping, and first-class layers, and shown that the type system is sound with respect to the operational semantics. As in previous work of Igarashi et al., `requires` declarations are important to guarantee safety in the presence of baseless partial methods. Subtyping for first-class layers is subtle because there are two kinds of substitutability. We have introduced weak subtyping for checking whether a `requires` clause is satisfied and normal subtyping for usual substitutability.

Chapter 3

ContextWorkflow

In this chapter, we propose a language ContextWorkflow, which is a workflow-based language that supports compensation, asynchronous interruption, suspension, checkpoints, nested-workflow, and programmable compensations. Further, we formalize the core of ContextWorkflow and provide some provable properties. We also develop the monadic interpreter and implement ContextWorkflow as an E-DSL in Scala.

The rest of this chapter is organized as follows. In Section 3.1, we informally introduce ContextWorkflow with a running example of maze search robot. Section 3.2 provides a formal calculus of the core ContextWorkflow. In Section 3.3, we construct a monadic interpreter and show further implementation techniques in Scala. The rest summarizes this chapter.

3.1 ContextWorkflow Constructs

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

A program in ContextWorkflow is a workflow, which 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, is suspended (and the rest of the computation is returned).

3.1.1 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 his battery. (We call such a special point simply a charger.)
2. When it starts to rain, the robot should suspend.

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; and 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 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 neighboring
nodes of the node 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 it has been visited or not. The function neighbors returns neighboring nodes of the given node n. The function 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 he is currently at its neighbor or the node itself. The function visit is our basic main function to 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 the 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 him 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 him at a charger while he is getting back.

3.1.2 Interruptible Workflow

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

Each function call to visited, move and visit should be lifted to a primitive workflow because it changes “states”, 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.¹ 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 is of the type Unit, and the compensation action 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,

¹The 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.

and the `!!` operator is used to sequence workflows in the `workflow` block. These notations are actually syntax sugar, where `!!` is monadic bind; it is possible to use for-comprehension instead of them in Scala. We can also use ordinary `if` for branching and `fold` (called `foldCW`) for iteration in `ContextWorkflow`.

```
1 def workflow[T](body: T): CW[T]
2 def !![T](m: CW[T]): T
3 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:

```
1 def visit(n: Node, maze: Set[Node]): CW[Unit] = workflow {
2   !! (visited(n) /+ (_ => unknown(n))) // reversible visited
3   !! { foldCW(neighbors(n, maze)) (()) { (_, neighbor) =>
4     if (!neighbor.visited) workflow {
5       !! (moveFromTo(n, neighbor)) // the robot moves to the neighbor
6       !! (visit(neighbor, maze))
7       !! (moveFromTo(neighbor, n)) // the robot gets back to the original node
8     }
9     else () /+ () } } }
```

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

In order to execute a workflow, we invoke the method `exec` of `CW` class:

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

where `\/` introduces disjunctions of two types of which constructors are `-\/(l)` (meaning the left value) and `-\/(r)` (meaning the right value); `Option` is the type of optional values consists 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.

3.1.3 Interruption and Context

We need contexts to interrupt execution of the workflow in `ContextWorkflow`. A context signals how the execution of 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`, `PAabort`, 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.
- `PAabort` 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.

Before the execution of the normal action of each primitive workflow, the current context is checked (called *context checking* and same as polling); if the current context is not equal to `Continue`, it is interrupted immediately.

To create a stream of `Contexts`, we use a signal in the functional reactive programming library `REScala` [SHM14]. For example, we can represent an interruption due to

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 }
```

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, thanks to expressiveness of REScala. For example, in order to suspend the robot when it starts raining, we need another sensory data that reflects the weather condition. It is achieved by creating another signal that relates to both of 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 the precedence between the two contexts here: Abort from lowbattery precedes Suspend from weather.

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

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

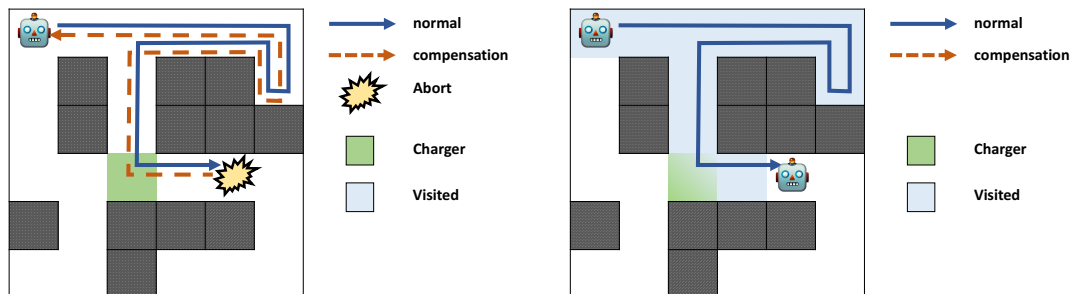


FIGURE 3.1: Maze search simulation: Abort (left) and Suspend (right).

Fig. 3.1 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(...)
...
r match { case -\/(Some(s)) => s.exec(...) } // restart if suspended
```

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

3.1.4 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 how

he came to the aborted point and does not come back straight to the start. A better compensation should take a shortcut to the start node as shown in Fig. 3.2.

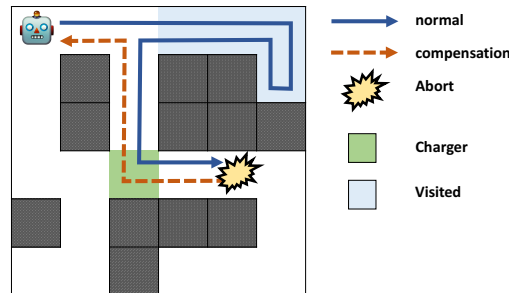


FIGURE 3.2: Maze search simulation: Abort (refined)

This can be achieved by delimiting a part of a workflow and ignoring the compensation actions of the delimited part if the part has completed successfully. We call such a part *sub-workflow* and provide a construct `sub` that makes a part of workflow a sub-workflow:

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

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

```
1 def visit(n: Node, maze: Set[Node]):CW[Unit] = workflow {
2   ...
3   if(!neighbor.visited)
4     sub{ workflow{
5       !! (moveFromTo(n, neighbor))
6       !! (visit(neighbor, maze))
7       !! (moveFromTo(neighbor, n))
8     } } ... }
```

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

It is possible to perform other compensation action instead of just skipping the compensation actions within sub-workflows by writing like `sub(...)/+ comp`, which is so-called programmable compensations [BMM05, CP13]. For example, we can add a log:

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

3.1.5 Checkpoint

Using 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 his 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` which represents whether the node has a charger or not. We just add a checkpoint 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(...)(...){...} }
}

```

The left side of Fig. 3.3 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. 3.3).

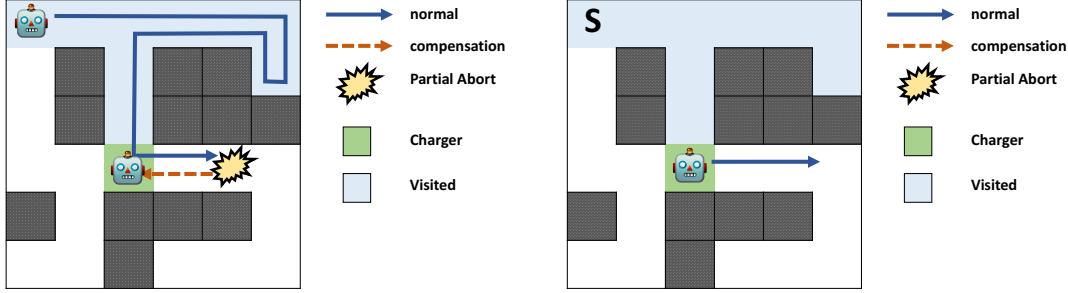


FIGURE 3.3: Maze search simulation: Partial abort (left) and its restart (right).

3.1.6 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, which correspond to blocking constructs of asynchronous exceptions `block` and `unblock` in Concurrent Haskell [MJMR01].

```

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` enforces context checking inside it. If they are nested 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.

3.2 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.

3.2.1 Syntax

We show the syntax of our calculus in Fig. 3.4. 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.

t	::=	$A/C \mid \text{check} \mid \text{cp} \mid \text{cp}\#E \mid \text{sub}(t)/C \mid t;t$	(workflows)
A, C	::=	$\epsilon \mid \dots$	(atomic actions)
c	::=	$C \mid \text{sub} \mid \text{ccp}\#E$	(compensations)
E	::=	$[] \mid E[[];t]$	(evaluation context)

FIGURE 3.4: Syntax of core ContextWorkflow.

A/C is a primitive workflow consisting of a pair of a normal action A and a compensation action C . $\text{sub}(t)/C$ is sub-workflow with a programmable compensation; if $/C$ is omitted, a no-op will be recovered. 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 3.2.4 for discussions. cp is the checkpoint declaration and $\text{cp}\#E$, which does not appear in the source program, is an automatically created checkpoint declaration that remembers the original evaluation context E and will substitute for normal cp at run-time.

In compensations, sub is the marker that indicates the start point of a sub-workflow; $\text{ccp}\#E$ 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.

3.2.2 Big-Step Semantics

In this section, we give a big-step semantics. We use overlines to denote sequences. For example, $\bar{c}$ stands for a possibly empty sequence $c_1; \dots; c_n$. We also use $\bar{c} \setminus c$ to represent the sequence obtained by removing c from $\bar{c}$, and similarly for other metavariables. Moreover, we use $\vec{A}$ for a sequence of atomic actions excluding ϵ , e.g., $A_1, \dots, A_n, C_n, \dots, C_1$.

The following relations give our semantics of core ContextWorkflow:

$\langle t, E, \bar{c} \rangle \Downarrow_{\vec{A}} \langle \bar{c}' \rangle$	workflow success
$\langle t, E, \bar{c} \rangle \Uparrow_{A P S}^{\vec{A}} \langle \bar{c}', E_s \rangle$	workflow interruption
$\langle \bar{c} \rangle \Downarrow_{\vec{A}} \langle \bar{c}', E_s \rangle$	compensations success
$\langle t, \bar{c} \rangle \Downarrow_{\vec{A}} \langle \rangle$	program commit
$\langle t, \bar{c} \rangle \Downarrow_{\vec{A}} \langle \rangle$	program abort
$\langle t, \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}', E_s \rangle$	program partial abort
$\langle t, \bar{c} \rangle \Downarrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle$	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 $\Downarrow_{\vec{A}}$ or $\Uparrow_{\vec{A}}^{\vec{A}}$ is executed, it terminates after executing $\vec{A}$ and returns the right side, which is a sequence of compensation actions $\bar{c}'$ possibly with a suspended computation E_s . The first two relations are for execution of

t under evaluation context E with compensation actions $\bar{c}$ recorded by past commands; the first relation is for successful execution and the second relation is for interrupted execution, where E_s is empty ($[]$) in the case of abort A or partial abort P . The third relation is for 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 t and compensation actions $\bar{c}$, which is 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 $\bar{c}'$ and the evaluation context 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 $\langle \bar{c}, E_s \rangle$ is returned by suspension, a restart of the suspended computation can be expressed by running a program $\langle E_s[\text{check}], \bar{c} \rangle$ —check means that the restart should check the context first to check if the context allows the restart.

We also define an auxiliary function *rmsub1* as follows in order to forget compensations in the nearest sub-workflow.

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

The semantics is defined by the rules in Fig. 3.5. The rule CW-PW is for the primitive workflow that does normal action A and adds the compensation C . The rule CW-CHECK-* are for check and one of them is chosen in nondeterministic. 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 $[]$; $\text{cp}\#E$, 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 $(\text{sub } [])/C$ onto the stack of frames (that is, the evaluation context) before executing t . The rules CW-SEQ* are for sequences, which push t_2 on the stack of frames. The rules CW-PROGRAM-* is for program execution, where CW-PROGRAM-ABORT is to run compensations except *ccp* (represented by $\bar{c}' \setminus \text{ccp}$), meaning that checkpoints are simply ignored. CW-PROGRAM-PABORT do compensations—if they include *ccp*, compensation will stop at the first *ccp* and returns the evaluation context recorded there (see CW-COMP-CCP). Rules CW-COMP* are for execution of compensations.

An example of workflow execution is as follows and the derivation tree is in Fig. 3.6.

$$\begin{aligned} &\langle \text{sub}\{\text{sub}\{t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}\}/C_b; t_3\}; t_4, \bullet \rangle \Downarrow_P^{A_1, A_2, C_a} \\ &\quad \langle C_1; \text{sub}; \text{sub}, \text{sub}(\text{sub}([], \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \rangle \\ &\text{where, } t_k = A_k/C_k \text{ for } k = 1, 2, \dots \text{ and} \\ &E_1 = \text{sub}(\text{sub}([], \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \end{aligned}$$

This is an example of partial abort at the check, so 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 A_2, A_3 , and A_4 and terminates: In other words, the relation below can be derived.

$$\langle \text{sub}(\text{sub}(\text{check}; \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4, C_1; \text{sub}; \text{sub} \rangle \Downarrow^{A_2, A_3, A_4} \langle \rangle$$

$\frac{}{\langle A/C, E, \bar{c} \rangle \Downarrow^A \langle C; \bar{c} \rangle}$	(CW-PW)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Downarrow^\epsilon \langle \bar{c} \rangle}$	(CW-CHECK-CONT)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_S^\epsilon \langle \bar{c}, E \rangle}$	(CW-CHECK-SUSPEND)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_A^\epsilon \langle \bar{c}, [] \rangle}$	(CW-CHECK-ABORT)
$\frac{}{\langle \text{check}, E, \bar{c} \rangle \Uparrow_P^\epsilon \langle \bar{c}, [] \rangle}$	(CW-CHECK-PABORT)
$\frac{}{\langle \text{cp}, E, \bar{c} \rangle \Downarrow^\epsilon \langle \text{ccp}\#E[[]; \text{cp}\#E]; \bar{c} \rangle}$	(CW-CHECKPOINT)
$\frac{}{\langle \text{cp}\#E_0, E, \bar{c} \rangle \Downarrow^\epsilon \langle \text{ccp}\#E_0[[]; \text{cp}\#E_0]; \bar{c} \rangle}$	(CW-CHECKPOINT-REVISIT)
$\frac{}{\langle t, E[(\text{sub } [])/C], \text{sub}; \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}' \rangle}$	(CW-SUB)
$\frac{}{\langle \text{sub}(t)/C, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle C; \text{rmsub1}(\bar{c}') \rangle}$	
$\frac{}{\langle t, E[(\text{sub } [])/C], \text{sub}; \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-SUB-INT)
$\frac{}{\langle \text{sub}(t)/C, E, \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-SEQ)
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}' \rangle \quad \langle t_2, E, \bar{c}' \rangle \Downarrow^{\vec{A}_2} \langle \bar{c}'' \rangle}$	
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Downarrow^{\vec{A}_1; \vec{A}_2} \langle \bar{c}'' \rangle}$	(CW-SEQ-INT1)
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Uparrow_*^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-SEQ-INT2)
$\frac{}{\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}' \rangle \quad \langle t_2, E, \bar{c}' \rangle \Uparrow_*^{\vec{A}_2} \langle \bar{c}'', E_s \rangle}$	
$\frac{}{\langle t_1; t_2, E, \bar{c} \rangle \Uparrow_*^{\vec{A}_1; \vec{A}_2} \langle \bar{c}'', E_s \rangle}$	
program	
$\frac{}{\langle t, [], \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bullet \rangle}$	(CW-PROGRAM-COMMIT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_A^{\vec{A}} \langle \bar{c}', [] \rangle \quad \langle \bar{c}' \setminus \text{ccp} \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle}$	(CW-PROGRAM-ABORT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_A^{\vec{A}; \vec{c}} \langle \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_P^{\vec{A}} \langle \bar{c}', [] \rangle \quad \langle \bar{c}' \rangle \Downarrow^{\vec{c}} \langle \bar{c}'', E_s \rangle}$	(CW-PROGRAM-PABORT)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_A^{\vec{A}; \vec{c}} \langle \bar{c}'', E_s \rangle}$	
$\frac{}{\langle t, [], \bar{c} \rangle \Uparrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle}$	(CW-PROGRAM-SUSPEND)
$\frac{}{\langle t, \bar{c} \rangle \Downarrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle}$	
compensation	
$\frac{}{\langle C \rangle \Downarrow^C \langle \bullet, [] \rangle}$	(CW-COMP-ACTION)
$\frac{}{\langle \text{sub} \rangle \Downarrow^\epsilon \langle \bullet, [] \rangle}$	(CW-COMP-SUB)
$\frac{}{\langle c \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle \quad \langle \bar{c} \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle}$	(CW-COMP-SEQ)
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}; \vec{c}} \langle \bullet, [] \rangle}$	
$\frac{}{\langle \text{ccp}\#E \rangle \Downarrow^\epsilon \langle \bullet, E \rangle}$	(CW-COMP-CCP)
$\frac{}{E_s \neq [] \quad \langle c \rangle \Downarrow^{\vec{c}} \langle \bullet, E_s \rangle}$	
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}} \langle \bar{c}, E_s \rangle}$	(CW-COMP-SEQ-PABORT1)
$\frac{}{E_s \neq [] \quad \langle c \rangle \Downarrow^{\vec{c}_1} \langle \bullet, [] \rangle \quad \langle \bar{c} \rangle \Downarrow^{\vec{c}_2} \langle \bar{c}', E_s \rangle}$	(CW-COMP-SEQ-PABORT2)
$\frac{}{\langle c; \bar{c} \rangle \Downarrow^{\vec{c}_1; \vec{c}_2} \langle \bar{c}', E_s \rangle}$	

FIGURE 3.5: Big step semantics of core ContextWorkflow.

let $t_k = A_k/C_k$ for $k = a, b, 1, 2, \dots$

$$\begin{array}{c}
\boxed{\text{subgoal 1: } \langle \text{sub} \{ \text{sub} \{ t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \}/C_b; t_3 \}; t_4, [] \bullet \rangle \uparrow_P^{A_1, A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, [] \rangle} \\
\hline
\frac{\langle A_1/C_1, E_0[[]]; \text{cp}; \text{sub}(t_2)/C_a; \text{check} \rangle, \text{sub}; \text{sub} \rangle \downarrow_P^{A_1} \langle C_1; \text{sub}; \rangle \quad \frac{\langle \text{cp}; E_1, C_1; \text{sub}; \text{sub} \rangle \downarrow_P^e \langle \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \quad \text{CP} \quad \langle \text{subgoal 2} \rangle}{\langle \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle \bar{C}_0, [] \rangle} \text{Seq-Int2} \\
\hline
\frac{\langle t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}, E_0, \text{sub}; \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{C}_0, [] \rangle \quad \text{Sub-Int} \\
\frac{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check})/C_b; \text{sub}([[]]; t_3); t_4, \text{sub} \rangle \uparrow_P^{A_1, A_2} \langle \bar{C}_0, [] \rangle \quad \text{Seq-Int1} \\
\frac{\langle \text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3, \text{sub}([[]]; t_4, \text{sub}) \rangle \uparrow_P^{A_1, A_2} \langle \bar{C}_0, [] \rangle \quad \text{Sub-Int} \\
\frac{\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3), []; t_4, \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{C}_0, [] \rangle \quad \text{Seq-Int1} \\
\langle \text{sub}(\text{sub}(t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4, [] \bullet \rangle \uparrow_P^{A_1, A_2} \langle \bar{C}_0, [] \rangle} \\
\boxed{\text{subgoal 2: } \langle \text{sub}(t_2)/C_a; \text{check}, E_0, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, [] \rangle} \\
\hline
\frac{\langle A_2/C_2, E_2, \text{sub}; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^{A_2} \langle C_2; \text{sub}; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \quad \text{PW} \\
\frac{\langle \text{sub}(t_2)/C_a, E_0[[]]; \text{check} \rangle, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^{A_2} \langle \bar{C}_0 \rangle \quad \text{Sub} \quad \langle \text{check}, E_0, C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^e \langle \bar{C}_0, [] \rangle \quad \text{Check-PAbsort} \\
\langle \text{sub}(t_2)/C_a; \text{check}, E_0, \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \uparrow_P^{A_2} \langle C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub}, [] \rangle \quad \text{Seq-Int2} \\
\hline
\boxed{\text{goal: } \langle \text{sub}\{\text{sub}\{t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}\}/C_b; t_3\}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, \text{sub}(\text{sub}([[]]; \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \rangle} \\
\hline
\frac{\langle C_a \rangle \downarrow_P^{C_a} \langle \bullet, [] \rangle \quad \text{Comp-Action} \quad \frac{\langle \text{ccp}\#E_1 \rangle[[]]; \text{cp}\#E_1 \rangle \downarrow_P^e \langle \bullet, E_1[[]]; \text{cp}\#E_1 \rangle \quad \text{Comp-Ccp} \\
\langle \text{ccp}_0; C_1; \text{sub}; \text{sub} \rangle \downarrow_P^e \langle C_1; \text{sub}; \text{sub}, E_1[[]]; \text{cp}\#E_1 \rangle \quad \text{Comp-Seq-PAbsort1} \\
\langle \bar{C}_0 \rangle \downarrow_P^{C_a} \langle C_1; \text{sub}; \text{sub}, E_1[[]]; \text{cp}\#E_1 \rangle \quad \text{Comp-Seq-PAbsort2} \\
\langle \text{sub}\{\text{sub}\{t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}\}/C_b; t_3\}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, E \rangle \quad \text{Program-PAbsort}} \\
\langle \text{sub}\{\text{sub}\{t_1; \text{cp}; \text{sub}(t_2)/C_a; \text{check}\}/C_b; t_3\}; t_4 \rangle \downarrow_P^{A_1, A_2, C_a} \langle C_1; \text{sub}; \text{sub}, E \rangle}
\end{array}$$

where,

$$\begin{array}{ll}
E_0 & = \text{sub}(\text{sub}([[]])/C_b; t_3); t_4 \\
E_1 & = E_0[[]; \text{sub}(t_2)/C_a; \text{check}] = \text{sub}(\text{sub}([[]]; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4 \\
E_2 & = E_0[[]; \text{sub}([[]])/C_a; \text{check}] = \text{sub}(\text{sub}(\text{sub}([[]])/C_a; \text{check})/C_b; t_3); t_4 \\
\text{ccp}_0 & = \text{ccp}\#E_1[[]]; \text{cp}\#E_1 \\
\bar{C}_0 & = C_a; \text{ccp}_0; C_1; \text{sub}; \text{sub} \\
E & = E_1[[]; \text{cp}\#E_1] = \text{sub}(\text{sub}([[]]; \text{cp}\#E_1; \text{sub}(t_2)/C_a; \text{check})/C_b; t_3); t_4
\end{array}$$

FIGURE 3.6: ContextWorkflow: A derivation example.

3.2.3 Properties

Here, we describe some properties that hold of the semantics. In the following theorems, let $p_k = A_k/C_k$ for some k , and we define a function $b(t)$ and predicates *includes* and *nosub* as follows.

- $b(t)$ be a workflow obtained from t by removing *sub*, *check*, *cp* and *cp#E* from t .
- *includes*(t, m, n) iff $b(t) = p_m; \dots; p_n$ and $m \leq n$; or t has no primitive workflows and $m \not\leq n$.
- *includes*(E, m, n) = *includes*($E[\text{check}], m, n$).
- *includes*($\bar{C}, n, m$) iff $\bar{C} \setminus \{\text{sub}, \text{ccp}\#E\} = C_n; \dots; C_m$ and $m \leq n$; or $\bar{C}$ has no atomic actions C_* and $m \not\leq n$.
- *nosub*(t, m, n) iff *includes*(t, m, n) and t has no sub-workflow.

Theorems 3.2.1 and 3.2.2 state about behaviors under contexts *Continue* and *Abort*.

3.2.1 Theorem [Workflow commits]: If *includes*(t, m, n) and $\langle t, \bar{C} \rangle \downarrow_{\vec{A}} \langle \rangle$ and $m \leq n$, then $\vec{A} = A_m, \dots, A_n$.

3.2.2 Theorem [Workflow aborts (Successful Compensation)]: If *nosub*(t, m, n) and $\langle t, \bar{C} \rangle \downarrow_{\vec{A}} \langle \rangle$ and $m \leq n$ and $\bar{C} = C_k, \dots, C_l$, then $\vec{A} = A_m, \dots, A_i, C_i, \dots, C_m, C_k, \dots, C_l$ for some i ($m \leq i \leq n$), or $\vec{A} = C_k, \dots, C_l$.

Theorems 3.2.3 and 3.2.4 are about workflows that may suspend by **PA**abort or **S**uspend, and describe how a suspended workflow behaves when it restarts.

3.2.3 Theorem [Restarted suspended workflow commits]: If $\langle t, \bullet \rangle \Downarrow^{\vec{A}} \langle \rangle$ and $\langle t, \bullet \rangle \Downarrow_S^{\vec{C}} \langle \bar{c}, E \rangle$ and $\langle E[\text{check}], \bar{c} \rangle \Downarrow^{\vec{C}'} \langle \rangle$, then $\vec{A} = \vec{C}, \vec{C}'$.

3.2.4 Theorem [Workflow partially aborts]: If $\text{nosub}(t, m, n)$ and $\langle t, \bullet \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}, E \rangle$ and $m \leq n$, then either of the followings hold.

- $\vec{A} = A_m, \dots, A_i, C_i, C_{i-1}, \dots, C_j$ and $\text{includes}(E, j, n)$ and $\text{includes}(\bar{c}, j-1, m)$ for some i and j ($m \leq j \leq i \leq n$).
- $\vec{A} = A_m, \dots, A_n$ and $\text{includes}(E, 1, 0)$ and $\text{includes}(\bar{c}, n, m)$.

Moreover, the following conditions hold.

1. (Suspended workflow commits) If $\langle E[\text{check}], \bar{c} \rangle \Downarrow^{\vec{A}'} \langle \rangle$, then $\vec{A}' = A_j, \dots, A_n$, or $\vec{A}' = \epsilon$ (if $\text{includes}(E, 1, 0)$).
2. (Suspended workflow aborts) If $\langle E[\text{check}], \bar{c} \rangle \Downarrow_A^{\vec{A}'} \langle \rangle$, then $\vec{A}' = \epsilon$ (if $j = m$), or $\vec{A}' = C_{j-1}, \dots, C_m$.

Theorem 3.2.5 is about a workflow with a sub-workflow, checkpoints and **PA**abort; 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.

3.2.5 Theorem [Partial abort, checkpoint and nested workflow]: Suppose that $\text{includes}(t, 1, n)$ and t without check is

$p_1; \dots; cp; p_k; \dots; p_m; \text{sub}(p_{m+1}; \dots; cp; p_j; \dots; p_l) / C_a; p_{l+1}; \dots; p_n$
and $\langle t, \bullet \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}, E \rangle$.

1. (Partial abort skips compensations of complete sub-workflow) If $A_{l+1} \in \{\vec{A}\}$, then $\vec{A} = A_1, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k, \dots, C_1$ for some $i > l$.
2. (A suspended workflow remembers checkpoints in a sub-workflow) If $A_{l+1} \in \{\vec{A}\}$ and $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}'} \langle \bar{c}', E' \rangle$ and $A_j \in \{\vec{A}'\} \wedge A_l \notin \{\vec{A}'\}$, then $\vec{A}' = A_k, \dots, A_i, C_i, \dots, C_j$ for some i such that ($j \leq i \leq l$).
3. (A suspended workflow remembers checkpoints before a sub-workflow) If $C_j \in \{\vec{A}\}$ and $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}'} \langle \bar{c}', E' \rangle$ and $A_{l+1} \in \{\vec{A}'\}$, then $\vec{A}' = A_j, \dots, A_i, C_i, \dots, C_{l+1}, C_a, C_m, \dots, C_k$ for some $i > l$.

3.2.4 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 them becomes clear when executing a sub-workflow. Let $t_k = A_k / + C_k$ for $k = 1, 2$. Then, when we execute $\text{sub}(t_1; t_2)$, is it possible that the resulting action sequence A_1, A_2, C_2, 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 $\bullet$, “ A_1, C_1 ”, or “ A_1, 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 the context checking actually occurs during execution of an atomic action. Such behavior is critical in cases where an atomic action must be done 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 so 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 of `check`.

Abnormal termination Although not modeled here, we can consider *abnormal termination* [BMM05], 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 from the calculus [BMM05] Here, we describe differences from the existing calculus [BMM05], by which ours are inspired.

- 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 [CP13] (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.

3.3 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 3.2.

There are two differences between the core calculus and the embedding. 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 to the signal of `Context` in Section 3.1.

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 programs of ContextWorkflow to programs of that language.

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

3.3.1 Free Monad Transformers

This sub-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 [Mog89, Wad95]. 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 as free monads 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 as the `do`-notation in Haskell. For example, for values `m1` and `m2` of the type monad `M`, the following 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`.² Intuitively, it defines 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]
}
```

²We here borrow `iterT` from the free package of Haskell. Although it is possible to define `iterT` in Scala, it is no good in practice. We will visit the problem in Sec 3.3.6.

`iterT` takes an interpretation of “commands” and translates a “program” of type `FreeT[F,M,A]` to that of type `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 and `M[A,?,?]` takes two types.³

The trait `FreeF` takes three types namely `F`, `A` and `B`, and has two constructions namely `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]
```

3.3.2 ContextWorkflow Monad

The ContextWorkflow monad, namely `CW`, is a free monad transformer defined as:⁴

```
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, which 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, which is 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

³This feature is enabled by kind-projector (<https://github.com/non/kind-projector>)

⁴Again, the definition is simplified from the actual definition just for avoiding unnecessary complexity of implicit conversions.

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 enough to wrap only the exceptional values propagated backwards with `InSubL`.

Readers may wonder what is `CW[A]` appeared in Section 3.1. This is the abbreviation of `CW[Unit, IO, Nothing, A]`, and see Section 3.3.9 for further detail.

3.3.3 Auxiliary Definitions

This section gives auxiliary functions and macros that correspond to the syntax for the users of `ContextWorkflow`. For readability and simplicity, we omit 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 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 monadic values of type `G[A]` to another monadic value of type `H[G,A]` where `G` and `H` is a monad and monad transformer, respectively. We also define another auxiliary function `/+` that corresponds to `check; A/C`⁵.

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

⁵Though omitted here, in order to regard `/+` as an infix operator, we have to define it using implicit conversions in Scala.

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 [mon] 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 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))
```

3.3.4 Types of Suspended Workflows

Before showing the monadic interpreter for the CW monad, we need to fix the type of the suspended workflows. It clearly 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 $CW[E,M,?,A]$.⁶

Usually, a data type is necessary to represent fixpoints of functors [Gib07, OG10]. The data type `Fix` is parameterized over functors and the types of values:

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

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

3.3.5 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(c1: F[MM[A]]): MM[A] = c1 match{
    case Comp(c, k) => ...
```

⁶Precisely speaking, $CW[E,M,?,?]$ is a bifunctor and we actually take a fixpoint of the bifunctor.

```

    ...
  }
  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 `PAabort`, `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.

3.3.6 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` [Fre], 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 as is the compensation monad [RV13] to achieve this. 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]
```

⁷The continuation monad transformer must be stack safe, although neither scalaz nor cats (another library similar to scalaz) provide that one. Our Scala implementation employs a workaround that relies on Trampoline [Bja12] in the `IO` monad. In other words, we always use the `IO` monad as the underlying user monad of the `CW` monad.

```

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 wrapping 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 */
    }}
  }
}

```

3.3.7 Atomicity

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

We add a command `CheckA` for active context checking and `CheckI` for inactive context checking, of which definitions are the same to `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 interpretations of `CheckA` is the same to `Check` and that of `CheckI` is just continuing evaluating the subsequent workflow without checking the context.

The two blocks are implemented as two functions as sub for sub-workflows. 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
    }
  })
}

```

3.3.8 Abnormal Termination and Exceptions in Scala

We have already mentioned abnormal termination in Section 3.2. 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 same as `/+`, but absorbs some particular exceptions `AbortE` and `PAAbortE` in the normal action, and raises the interruption `Abort` or `PAAbort`. For example:

```

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

```

When running `cw0`, if the exception `e` is `AbortE` or `PAAbortE`, 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:

```
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) => throwCWEException(Abort) // raise abort
    case Failure(RestartE) => throwCWEException(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 do one of (1) raising `Abort` or `PAAbort` inside `ContextWorkflow`, (2) successfully committing `na`, and (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 explicit type constructors of `M` and `Try`, we use implicit conversions. For further detail, see Section 3.3.9.

3.3.9 Hiding Type Parameters for Simplicity

What we use as `CW[A]` in Section 3.1 is the abbreviation of `CW[Unit,IO,Nothing,A]`. The important point is to fix `M` to `IO` and `S` to `Nothing`.

The monad `IO` is the standard way to treat effectful code in monadic programming, but explicit use of `IO` monad constructors 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` which contains a field of the type `IO[A]` by implicit conversions, and then (2) the method `/+` of the special object is invoked, which 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 `/+` of `CWOps` class using the function `/+` appeared in Section 3.3.

The reason using `Nothing` as the suspended workflow type is that in order to treat `CW` as a monad, type parameters except for `A` must be fixed or parameterized. Although the latter approach looks good, it becomes redundant in Scala. For example, let `CWS[S,A]` be `CW[Unit,IO,S,A]`, and let's 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 have to use `def` and then type parameters `S` are appeared around, since Scala's value is not polymorphic. While such definitions can be treated well in Haskell, we have to manually parameterize it one by one in Scala.

Therefore, we fix `S` to `Nothing` and cast `Nothing` to a proper suspended workflow type at run-time. However, why does this work? This is because that when `S` is actually used is at the run-time. At the run-time of `CW[A]`, it is enough to cast `Nothing` to `Fix[CW[Unit,IO,?,?],A]`.

3.4 Summary

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

Chapter 4

Modularizing Context-Dependent Behaviors in ContextWorkflow

In this chapter, we try to modularize CDBs in ContextWorkflow. Though we use contexts as an execution status in ContextWorkflow, we want to use mechanisms of behavioral changes of COP in ContextWorkflow too. For example, we would like to program CAAs such as energy-aware computing [SDF⁺11, ZLL15], where several modes relate to energy consumption and are changed dynamically. In this work, we develop a stand-alone library to modular reactive behavioral adaptation in Scala, called GEAR. GEAR is supposed to be used to write atomic actions of ContextWorkflow.

The rest of this chapter is organized as follows. First, we look overview of this work in Section 4.1. In Section 4.2, we briefly review FRP and COP and explain the motivation of our FRP-based implementation. Section 4.3 describes our library briefly and explains how to control reactions of context-dependent computations. In Section 4.4, we show a part of the implementation of our library. The rest summarizes this chapter.

4.1 Overview

Contexts are, very roughly speaking, anything a (software) system interacts with. They usually change continuously as time passes and affects the system's behavior through some internal states that reflect the context. For example, the battery status of a mobile device can be a context and can have effects on the behavior of an application program through a variable that stands for the power modes, e.g., high- and low-battery modes. Here, we call (descriptions of) behavior affected by contexts *context-dependent computations*.

However, language constructs to control how reactive values influence context-dependent computations have not been studied enough. If a context changes during an evaluation of its dependent computations, how should a program behave? Flute language [BVDR⁺12] partially addressed this problem, by advocating *interruptible context-dependent executions*; an evaluation of a context-dependent computation can be interrupted according to implicit context changes.

In this paper, we propose another technique to control reactions of context-dependent computations; suppressing reactions using dynamic binding. Our approach ensures consistency of an evaluation of a chain of context-dependent computations, and prevent unexpected reactions.

As a proof of concept, we develop a new Scala library GEAR, which provides some COP features. It uses the existing Functional Reactive Programming (FRP) library

REScala [SHM14] to make the implementation simple. Our main idea is to treat contexts as reactive values using FRP, substituting for many existing layer activation mechanisms of COP. We also show that *proceed* calls found in many COP languages can be implemented by delimited continuations [DF90].

4.2 Contexts as Reactive Values

In this section, we describe the reason why we consider contexts as reactive values. First, we briefly explain FRP and COP.

4.2.1 Functional Reactive Programming

Programming with reactive values are well studied in Functional Reactive Programming (FRP) [EH97, MO12, BLCVC⁺13]. In FRP, reactive values are declaratively constrained with other values, and they are automatically changed according to changes of other values.

REScala is a library of FRP and treats reactive values as *signals*. A signal can depend on other signals. If a signal value is changed, the values of dependent signals change automatically. Here is an example of a REScala program. `Var` is a variable containing a constant and is also a signal.

```
val a = Var(3)
val b = Signal{a() + 3}
a() = 5 // b also changes
```

Signal `b` depends on the variable `a`. Therefore, if the value of variable `a` changes, the value of `b` also changes automatically (Line 3). REScala provides many functions to treat various kinds of information, such as discrete events and time-varying values, as signals.

4.2.2 Context-Oriented Programming

COP is a relatively new programming paradigm to modularize context-dependent computations. COP provides layers to factor out computations related to the same kind of contexts. A layer is a set of partial method declarations, which override original definitions. A layer can be activated and deactivated at run-time, making contained partial declarations effective and ineffective, respectively.

In ContextL [CH05], *layered functions*, which represent base function definitions, are composed of *layered methods*, which represent fragmented, partial definitions of the layered function. Every layered method belongs to one layer; when a layered function is called, it calls appropriate layered methods corresponding to the currently activated layers. Here is an example of a ContextL program. Suppose that a power mode changes according to the battery level of a device, and according to power modes, how an application displays objects changes.

```
(deflayer high-performance)
(deflayer low-performance)

(define-layered-function display-object (object))

(define-layered-method display-object
  :in-layer high-performance
  ... containing code ...)
```

```
(with-active-layers (high-performance)
  ... (display-object obj) ...)
```

Here, the first two lines declare two layers. Line 4 declares that a layered function `display-object` takes one parameter. Line 6 defines a layered method, which relates to a layer `high-performance`. Line 10 means a *per-control-flow* layer activation, that is, the selected layer is activated within the block and the behavior of a layered function during the execution of the block is influenced by activated layers.

4.2.3 Characteristics of Contexts

According to [KAM15], existing layer activation mechanisms of COP are classified as *per-control-flow*, *imperative*, *event-based* and *implicit*. *Per-control-flow* activation, shown above, is also known as block-style activation. Imperative activation achieves activation of layers by a special statement; the activation continues until it is deactivated by another special statement. Event-based activation is a special form of imperative activation which is triggered by some asynchronous events. Implicit activation is specified by a condition; while the condition is true, corresponding layers become active. For example, implicit activation is written as follows in a hypothetical Java-like language.

```
layer HighPerformance{
  boolean active() {
    return bat_level > 50;
  }
}
```

This code means that while the value of variable `bat_level` is larger than 50, layer `HighPerformance` becomes active.

4.2.4 Layer Activation as Reactive Values

In this section, we insist that reactive values can generalize many layer activation mechanisms. Since our idea is based on the Flute language, we describe it first.

Flute is a COP language with implicit activation (by the classification of [KAM15]). In Flute, layered functions (called *modal* in it) are associated with *context sources*, which are the same as reactive values; Flute also allows several partial definitions of a layered function; each of which has a predicate on the associated context sources. When a layered function is called, it calls a layered method whose predicate evaluates to true using the current values of context sources. In Flute, the notion of layers is implicit; it is expressed in terms of context sources and predicates of layered methods.

As Flute does, we also introduce context sources, using signals of REScala. Our key insight is that by the representation of context sources, most layer activation mechanisms explained in the last section can be expressed by only signals. Here, let's express a layer activation of a `PowerMode`, using REScala.

```
trait PowerMode
object HighPerformance extends PowerMode
object LowPerformance extends PowerMode

val pow_layer: Var[PowerMode] = Var(HighPerformance)
```

Suppose that singleton objects `High-` and `LowPerformance` represent layers. In addition, we can define a signal `pow_layer` that contains an activated layer as its value (Line 5).

Computation depending on `PowerMode` will check the value of `pow_layer` to choose appropriate behavior. Then, an imperative activation is done by just changing the value of the signal.

```
pow_layer() = LowPerformance
```

Per-control-flow activation is expressed by dynamic binding (see Section 4.3.3). Event-based and implicit activations are just expressed by constraints on signals. For example, implicit activation becomes as follows.

```
val pow_layer = Signal{
  if (bat_level > 50) HighPerformance
  else LowPerformance }
```

Since REScala can treat events as signals, event-based activation is simulated in the same way.

4.2.5 Problem of Asynchronous Context Change

Suppose that two objects `obj1` and `obj2` have a method that displays a picture and the resolution of the displayed picture depends on the performance mode: in the low performance mode, the resolution of the picture becomes low and in the high performance mode, the result becomes opposite. Here, each objects has a member `display` which realizes such behavior using layered function.

```
obj1.display(); obj2.display(); /* layered functions */
```

In our setting, the performance mode is asynchronously changeable since it depends on battery status. Then, the evaluation of the functions above may result in two pictures with different resolutions. The problem here is that we have no way to prevent such a situation from happening. We propose a mechanism to suppress reactions to context source, by using dynamic binding in the next section.

4.3 Context-Dependent Computation with Reactive Values

Here, we show constructs of our library GEAR. We describe core classes, how to use them and a technique using dynamic binding to suppress reactions to asynchronous change of reactive values.

4.3.1 Core Classes of GEAR

A *context source* (class `CS`) represents a reactive value.

```
class CS[S](src: Signal[S])
```

Class `CS` (ContextSource) has a field of `Signal`. It also supports dynamic binding (see Section 4.3.3). Here, we see the battery level of a device as a context source.

```
val bat_level: Signal[Int] = // Signal of battery level
val bat_cs = new ContextSource(bat_level)
bat_cs.value() /* the current value of Int */
```

The current value of a context source can be obtained by `value()` as shown on line 3.

A *layered function* (class `LF`) represents context-dependent computations.

```
class LF[P,R](var plfs: List[PLF[P,R]])
```

Class LF has an ordered list of PLF (partial layered function) of same parameter types P and return types R. In our library, a layered function is an executable unit of a context-dependent computation. Since the field `plfs` is mutable, we can modify some behavior in a LF dynamically.

A *context-dependent function* (class CDF) is a composable unit of a context-dependent computation.

```
class CDF[S,P,R] (src: CS[S], p: S => Boolean,
  f: S => Cop[P,R] => P => R )
  extends PLF[P,R]
```

Class CDF extends the trait PLF; so a CDF can be a part of a LF. An instance of CDF connects to a context source (`src`). The field `p` is a predicate, which is evaluated at the beginning of evaluation of this CDF, and if it returns true, the method body will be evaluated. The field `f` represents a method body, where P is a parameter type, R is a return type, S is the current value of the context source `src` at an evaluation of this function, and Cop provides special constructs such as `proceed` and `thisfun`.

```
class Cop[P,R](thisfun: P => R, proceed: P => R)
```

The field `proceed` is a special function to call another PLF within an evaluation of a LF, and `thisfun` means its containing layered function, which is used for recursive calls. Finally, we can use the value of a context source and a `proceed` call in the method body.

```
new CDF[Int,Unit,Int](bat_cs, (_: Int) > 50,
  s => cop => _ => { cop.proceed(); } )
```

Here, the second parameter `cop` is an instance of class Cop. Then, we can write `cop.proceed()` in the body for a `proceed` call.

Although we have introduced LFs and CDFs, context-dependent computations do not have to be defined as CDFs and LFs. In some sense, every computation that uses a context source is considered context-dependent! CDFs and LFs are organizing tools for context-dependent computation, which is made extensible by them, since a LF is dynamically composable.

4.3.2 Semantics

Here, we describe the evaluation mechanism of our layered function. Basically, at each call of a context-dependent function, the dispatch mechanism considers the current values of context sources and whether the predicate is satisfied or not.

- Firstly, evaluate the predicate of the head of the list of partial layered functions with the current values of context sources, and if it returns true, then the body is evaluated. Otherwise, evaluate the next partial layered function.
- If a `proceed` call occurs in a body, also evaluate the next partial layered function. If the rest of the list is Nil, it raises an exception.

Here is an example of how whole constructs of our library are used and work.

```
def do_high = new CDF[Int,Unit,Int](bat_cs,
  (_:Int) > 50,
  s => cop => _ => { /* high-power mode behavior */ })

def do_low = new CDF[Int,Unit,Int](bat_cs,
  (_:Int) <= 50,
```

```
s => cop => _ => { /* low-power mode behavior */}
```

```
def display_object = new LF(List(do_high, do_low))
```

```
display_object() /* the result depends on bat_cs*/
```

In the example, the call of `display_object` (line 11) will first try to call `do_high` and if the predicate returns false (that is, the battery level is smaller than or equal to 50), it will try to call `do_low`.

4.3.3 Dynamic Binding of Context Sources

Class `CS` provides a method `withValue` like `DynamicVariable`, which is a class to realize dynamically scoped variables in Scala.

```
def withValue[T](newval: Signal[S])(thunk: => T): T
```

During the evaluation of the `thunk` (equals to a block), the field `src` of the receiver of `CS` is bound to another signal `newval`; each CDF depending on the `CS` connects to the new signal within the block. This is used to control reactions of context-dependent functions. Then, `withValue` provides two interesting applications *suppression of reactions* and *context replacement*.

Suppression of Reactions As described in Section 4.2.5, evaluating two methods that depend on the same context source may cause an unexpected result; mixing both behavior of low-power mode and behavior of high-power mode. (Assume that the member `display` is realized like `display_object` above). To escape such a problem, we can express such an intention by using dynamic binding.

```
bat_cs.withValue(Var(bat_cs.value())){
  obj1.display(); obj2.display();
}
```

This expresses caching of the context source `bat_cs` by binding `bat_cs` to a `Var` containing the current value of `bat_cs`. Then, we can ensure that `obj1.display` and `obj2.display` are evaluated under the same context. Context caching can be regarded as a kind of per-control flow activation of COP.

Context Replacement Suppose that some functions are dependent on UTC time, and we want to use JST time instead in some program context. Then, we can express such a program by rebinding to other reactive values.

```
val utc = /* Signal of UTC time */
val jst = Signal{ utc() + 9 }
val time_cs = new ContextSource(utc)
time_cs.withValue(jst){ /* functions dependent on time_cs will refer to jst
  time */ }
```

4.4 Implementation

In this section, we show implementation details of GEAR. GEAR is implemented on top of the existing FRP library `REScala`, making it possible to use `Signal`. The interesting point of our implementation is that we use delimited continuations library of Scala [RMO09] in order to realize proceed calls.

Class CS is just a container for an instance of `Signal` and an instance of `DynamicVariable`. Method `value` returns the current signal and `withValue` is just using that of `DynamicVariable`.

```
class CS[T](val src: Signal[T]) {
  val enclosing = new DynamicVariable[Signal[T]](src)
  def value = enclosing.value
  def withValue[S](newval: T)(thunk: => S): S = enclosing.withValue(newval){
    thunk}
}
```

Class CDF below makes a special closure which connects to the CS given as the field `src`.¹ The field `f` is the actual function body of this CDF. As we have described in the last section, `f` should be of the type $S \rightarrow \text{Cop}[P, R] \rightarrow P \rightarrow R$, where `S` is the type of context source values, `Cop[P, R]` is the class of COP language constructs, `P` is a parameter type, and `R` is a return type.² The `apply` method (Lines 4–10) has two parameters: `v` as an argument to the body `f` and `thisfun` as a LF that contains this CDF. What it does is to bind `proceed` to the captured delimited continuation (Lines 5–6), and apply `f` to the current value of the signal of `src` (Line 7) and a newly created instance of `Cop` (Line 8) if the predicate `p` is evaluated to true; otherwise (Line 9) call `proceed`. The `apply` has answer type `R` (denoted by the annotation `@cps[R]`) (Line 4). As we see later, this method is invoked from a loop to invoke `apply` on all PLFs held by a LF. So, the delimited continuation captured by `shift` at the beginning of `apply` represents the rest of invocations of `apply`. As a result, a call to `proceed` results in the execution of the rest of partial layered functions in the LF that this CDF is associated with.

```
class CDF[S,P,R] (src: CS[S], p: S => Boolean,
  f: S => Cop[P,R] => P => R)
  extends PLF[P,R]{
  def apply(v: P, thisfun: LF[P,R]): P @cps[R] =
    shift{
      proceed: (P => R) =>
      val t: S = src.value()
      if (p(t)) f(t)(new Cop(thisfun, proceed))(v)
      else proceed(v)
    }
}
```

Class LF has a list of PLFs. The method `dispatch` (Lines 3–13) invokes containing CDFs in order, and is used within a reset block (Line 14). Since CDF's body is `shift`, a `proceed` call is a continuation delimited by a dispatch of a LF, and will call the next CDF by the recursive call of `dispatch` (Line 8). If there is no CDF to proceed, an exception is thrown (Line 11).³

```
class LF[P,R](plfs: List[PLF[_ ,P,R]])
  extends Function1[P,R]{
  def dispatch(plfs: List[PLF[P,R]], v: P): R @cps[R] = {
    var param = v
    plfs match {
      case plf :: rest => {
        param = plf(v, this)
        dispatch(rest, param)
      }
    }
  }
}
```

¹The CDF has variations that connect to more than one CS, but we omit them for simplicity.

²A current limitation is that multiple parameters must be written as pairs.

³We could enforce a programmer to specify a default, proceed-less PLF and call it, instead of throwing an exception.

```

    }
    case Nil =>
      throw new ProceedingFunctionNotFoundException
    }
  }
  def apply(v: P): R = reset{ dispatch(plfs, v) }
}

```

Trait PLF represents partial method functions. The reason why LF has a list of type of not CDFs but PLFs is that we can convert a normal function into a PLF. The definition `fToPLF` provides implicit conversion of a function `f` to a PLF, creating an instance of PLF that discards a captured delimited continuation in `apply`.

```

trait PLF[P,R]{
  def apply(v: P, thisfun: LF[P,R]): P @cps[R]
}

implicit def fToPLF[P,R](f: P => R) =
  new PLF[P,R]{
    def apply(v: P, thisfun: LF[P,R]): P @cps[R] =
      shift{ k: (P => R) => f.apply(v) }
  }

```

4.5 Summary

We have described our new library GEAR, which is an attempt to combine COP features and FRP features. Our key ideas are regarding contexts as reactive values, and using dynamic binding for context caching and context replacement, which enable programmers to control reactions of context-dependent computations easily. While this work realizes COP using FRP, we believe that COP can provide FRP a good modularity mechanism. We also assume that we use this library to write atomic actions of ContextWorkflow in a compositional way.

Chapter 5

Related Work

5.1 Foundation of Context-Oriented Programming

This section compares our $\text{ContextFJ}_{\leq}$ in Chapter 2 with work on foundation of COP.

Our work is a direct descendant of Igarashi, Hirschfeld, and Masuhara [HIM11, IHM12], where a tiny COP language ContextFJ is developed and its type system is proved to be sound. ContextFJ is not equipped with layer inheritance, layer subtyping, or first-class layers but allows baseless partial methods to be declared in the second type system [IHM12], in which `requires` declarations are first introduced into COP.

Our swappable layers resemble atomic layers in ContextL [CD08], in which mutual exclusion between layers can be specified and activation of an atomic layer automatically deactivates another layer in conflict. Our syntax is a little verbose in that the swappable layer name such as `Weather` has to be explicit because a layer may have more than one swappable layer in its superlayers. It may be a reasonable idea to disallow a sublayer of a swappable layer to be swappable for the sake of syntactic conciseness.

Similarly to our swappable layers for layer deactivation, Kamina et al. [KAI14, KAMI] also show another approach to safe layer deactivation mechanism and formalized its semantics and type safety with an extension of ContextFJ . Their approach is also based on `requires` clauses. The key idea is to modify the method lookup so that it searches not only activated layers but all layers that are required by those activated layers.

Besides block-style layer activation mechanisms as in JCop , there are other mechanisms such as imperative activation of Subjective-C [GCM⁺10], event-based activation of EventCJ [KAM11], and implicit activation of Flute [BVDR⁺12]. The original JCop also supports implicit layer activation [AHL13], but currently we omit it from our formalization. ServalCJ [KAM15] provides a generalized layer activation mechanism that can treat the layer activation mechanisms above uniformly. Although some of them [AKM11, KAM16] study formal semantics, they do not discuss type soundness of languages with baseless partial methods; e.g., ServalCJ does not support baseless partial methods.

There are several studies to enrich description of relationships between contexts. Subjective-C [GCM⁺10], an extension of Objective-C with COP, adopts imperative context activation with imperative context relationship description, which supports various kinds of declarations of dependency between layers, such as implication, requirement, and exclusion. $\text{Context Petri Nets}$ [CGM⁺13, CGM⁺15] is a context-oriented extension of Petri Nets, and helps formalization of description of context dependencies in Subjective-C. ML_{CoDa} [DFG14, DFG16] provides two kinds of components; one for

declarative description of context dependencies and the other for functional computation. It also provides a type and effect system and a loading-time verification mechanism that detects failures in adaptation.

5.2 Advanced Modularization Mechanisms

This section compares our $\text{ContextFJ}_{<}$ in Chapter 2 with work on advanced modularization mechanisms.

Dynamic Software Product Line Software product line (SPL) is a paradigm of industrial software development that enables to create various variations of software by mostly reusing common modules. Researchers have proposed programming languages for SPL, such as Feature-Oriented Programming [Pre97, BSR04] and Delta-Oriented Programming (DOP) [SBB⁺10]. They provide modules that refine existing classes and combine them according to a given configuration at compile time or build time.

Recent studies [HHPS08, RSAS11] reveal that SPL also needs dynamic reconfiguration of software, and so dynamic DOP [DS11, DPSS17] is proposed. Dynamic DOP provides mainly three kinds of modules; a delta module for describing refinement of classes (similar to a layer of COP), a product-line declaration for describing valid configurations, and a dynamic reconfiguration graph for replacing heap objects dynamically. Unlike COP, the composition order of delta modules is determined uniquely by a given product line declaration; this property is called *unambiguity*. A type system of dynamic DOP also ensures that all valid reconfigurations lead to type-safe products.

Type systems for advanced composition mechanisms of OOP There are many type systems proposed for advanced composition mechanisms such as mixins [BPS99, FKF98], traits [LS08], open classes (a.k.a. inter-type declarations) [CMLC06], and revisers [CIZ10]. A common idea is to let programmers declare dependency between modules as required interfaces; our *requires* declarations basically follow it. In most work, however, composition is done at compile or link time unlike COP languages. We think that it is interesting that the same idea works even for dynamic composition found in COP languages.

Kamina and Tamai [KT04] propose McJava, in which mixin-based composition can be deferred to object instantiation. In fact, new expressions can specify a class and mixins to instantiate an object. So, the type of an object also consists of a class name and a sequence of mixin names. Whereas composition is per-instance basis in McJava, it is global in $\text{ContextFJ}_{<}$. However, in McJava, composition cannot be changed once an object is instantiated.

Drossopoulou et al. [DDDCG02] proposed *Fickle_{II}*, a class-based object-oriented language with *dynamic reclassification*, which allows an object to change its class at run time. Their idea of root classes, which serve as interface, is similar to our swappable layers; their restriction that state classes cannot be used as type for fields is similar to ours that a sublayer of a swappable cannot be required by any other layer.

Bettini et al. [BCD13] developed a type system for *dynamic trait replacement*, which allows methods in an object to be exchanged at run time. They introduce the notion of *replaceable* to describe the signatures of replaceable methods; a replaceable appears as part of the type of an object and the trait to replace methods of the object has to provide the methods in that replaceable. The roles of replaceables and traits are somewhat

similar to those of swappable layers, which provide interfaces common to swapped layers, and sublayers of swappable.

Although not a type system, Burton and Sekerinski [BS15] studies interference problem of dynamic mixin composition, in which some order of mixin composition breaks required specification of class methods. They develop a refinement calculus in order to formalize dynamic mixin composition.

5.3 Ancestors and Cousins of ContextWorkflow

In this section, we compare our ContextWorkflow in Chapter 3 with some related work.

Context-Oriented Programming In the literature of context-oriented programming [HCN08], which advocates the use of layers to modularize context-dependent behavior, there are several studies on behavioral change in response to asynchronous context changes [VLDN07, BVDR⁺12]. Among them, closest to the present work is Flute [BVDR⁺12] 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 [RW02] argue that in language runtime systems such as JVM which 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 *soft termination* to make mobile code being 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 realize suspensions in a easy way, such as first-class continuations [HFW84, DF90], which are supported in languages such as Scheme [SDF⁺09] and Scala [RMO09], and *coroutines* [Con63]. Coroutines are a generalization of subroutines in the sense that they do not exit but calling another coroutine as the caller coroutine suspends, and is supported in languages such as Lua [DMRI04]. We expect that these facilities are also useful for implementing ContextWorkflow.

Asynchronous Exception Asynchronous exception, found in, e.g., Haskell [MJMR01], Ruby and OCaml [DEH⁺17], is also used to realize interruption. Java and Scala threads take a so-called semi-asynchronous approach [], 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 [GMS87, CP13] and is provided in several languages such as Windows Workflow Foundation [wwf] in .NET and Windows PowerShell [pow]. PowerShell also supports checkpointing for fault tolerance. There

are many studies for the formalization of workflow [BMM05, BHF05, LZPH07]. Among them, our core ContextWorkflow is based on the Bruni et al.'s formalization [BMM05].

Scientific workflow [LAB⁺06] is an adaptation of the workflow to scientific computations, where a series of heavy computations are executed. In the scientific workflow, checkpoints are also useful to avoid wasteful recomputation [CA08]. We suppose that ContextWorkflow can be used to develop these applications.

Software Transactional Memory Software Transactional Memory (STM) [ST97], provided, e.g., by Scala [Sca11] and Haskell [HMPJH05], is a language-level approach to concurrency control, which is similar to a database transaction. STM provides an *atomic block* which means a critical section that has loads and stores for heap memories. 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 in order to realize partial aborts without using nested atomic block and gain efficiency [KH08]. 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).

Compensation and Asynchronous Exception Monads Although our CW monad is mainly composed of free, continuation and exception monads, its initial idea is from the compensation monad [RV13], which is comprised of continuation and exception monads. We also got the idea that asynchronous exception can be given as free monads from resumption monads [Har06, HAGP08], which is known to be structurally the same as free monad [PG14].

Modular Exception Handling Modularization of exception handling code has been a significant concern in Aspect-Oriented Programming [KLM⁺97, CFGF08] because the separation of exception handling code from normal code enhances re-usability of each module. The compensation approach [Wei06], 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 [YG07], any programs run forward and backward, and it is ensured that each direction is the exact inverse of the other. In other words, if programmers write a normal action in reversible programming languages, its compensation action is automatically defined. So, integrating reversible programming to ContextWorkflow will be interesting because it can release programmers from the burden of manually specifying compensation actions. Although it is often cumbersome, compensations being programmable have 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 3.1.

5.4 On Modularization of Context-Dependent Behaviors

This section compares our GEAR in Chapter 4 with some related work.

In many COP languages like JCop, the behavior of layered function is usually composed at layer activation time. In our library, however, a layered function always has

the same list of PLFs unless modifying it explicitly, and evaluating their containing predicates at dispatch time. This mechanism may cause a larger overhead than other mechanisms. An alternative implementation can be considered: responding to changes of a signal value, related CDFs are added to or removed from LFs dynamically (like an observer). It would need some additional management of signals, LFs and CDFs.

Flute [BVDR⁺12] greatly influenced our work. As described in Section 4.2.4, their modal is almost the same as our context-dependent functions. The main difference is that ours do not have interruptible/resumable executions, while ours enable dynamic binding of context sources.

ServalCJ [KAM15] has both synchronous and asynchronous layer activation. Its semantics of activation order is that synchronous activation always precedes the asynchronous one. Our approach is similar to this model, when we identify reactive values with asynchronous activation and context caching/context replacement with synchronous activation. The main difference appears in per-control-flow activation; while ServalCJ uses point-cut style, ours uses block-style (`withValue`).

Layered functions are not peculiar to COP. It can be considered an implementation of *Chain of Responsibility* (CoR) pattern [GHJV94]. While each behavior in a chain has to extend the same abstract class, layered functions are not restricted that way. Since CoR has some variations such as Tree of Responsibility, it would be useful to generalize layered function to support such variations.

Chapter 6

Conclusion

6.1 This Thesis

So far, this thesis has studied a programming language to develop robust context-aware applications. The “robust” software means that it does not cause an error or has enough error-handling functions. Therefore, we paid attention to the problem *achieving safety from harmful context changes* which has emerged as context-aware applications become more and more important and many programming language approaches have been evolved. This thesis has focused on two sub-problems; (1) static detection of erroneous run-time adaptation of context-dependent behaviors and (2) run-time recovery of unfinished context-dependent tasks. Then, we have proposed language-level solutions for each of them.

Chapter 2 has focused on developing type-safe version of JCop, which is an advanced Context-Oriented Programming language including layer inheritance, subtyping of layers, first-class layers and layer swapping. For the purpose, we have proposed a formal calculus $\text{ContextFJ}_{<}$ including type system, based on Igarashi et al.’s ContextFJ . We also have proven the type soundness property, which makes the language type-safe.

In Chapter 3, we have investigated error-handling mechanism for context-aware applications and developed ContextWorkflow. The language is based on workflow, which is a well-known fault-tolerance technique for long-running applications, and supports compensation, asynchronous interruption, suspension and checkpoints. To formalize the specification of ContextWorkflow and explain the execution model, we have developed a core calculus for ContextWorkflow and proven some properties. Then, we have proposed the monadic interpreter and implemented it in Scala as an embedded domain-specific language.

Chapter 4 has described an approach to modularize context-dependent behaviors in ContextWorkflow. For this purpose, we have developed a stand-alone library GEAR in Scala, which is supposed to be used to write atomic actions in ContextWorkflow.

In conclusion, although our approach, designing a language to develop safer context-aware applications efficiently, is far from completeness, we believe that the notions brought by our work will help programmers and software designers to capture and model context-aware applications more cleanly.

6.2 Future Work

This thesis has mainly shown static verification via type system and run-time error recovery via workflow. However, we have not fully developed a unified language supporting both facilities. Though we took the simple approach in Chapter 4, there is

some room for improvement. For example, dynamic binding of signal should be combined with atomic constructs of ContextWorkflow. To provide more COP-like mechanisms such as layer is also an important future work. Other integration approaches are to develop ContextWorkflow on COP languages, or to develop COP languages with ContextWorkflow features. However, these would require more complicated work including formalizations, proofs of type soundness, and implementations.

Further verification techniques for our language are also an important future direction. For example, one popular program verification method is Hoare logic, and this was also extended to COP [LNRA16] and workflow [LQQ08]; so we firstly would like to try to apply existing techniques to ours.

We also discuss future work for individual work in the following.

A Type System for JCop In JCop, a layer definition can contain field and (ordinary) method declarations so that a layer instance can act just like an ordinary object. Type-checking accesses to these members of layer instances is the same as ordinary objects. If we model fields of layer instances, we will have to modify the reduction relation so that the sequence of activated layers consists of layer instances (with their field values) rather than layer names.

JCop also provides special variable `thislayer`, which can be used in partial methods and is similar to `this` of classes. It represents the layer instance in which the invoked partial method is found at run time and can be used to access fields and methods of that layer instance. In operational semantics, the layer instance would be substituted for `thislayer`, similarly to `this`. Typing `thislayer` is also similar to `this` in the sense that it is given the name of the layer in which it appears but `thislayer` cannot be used for layer activation because, at run time, it may be bound to an instance of a *weak* subtype.

We have not fully investigated the interaction between our type system with other features in Java, such as concurrency, generics, and lambda, although we expect most of them are orthogonal.

ContextWorkflow 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 suspect that the semantics of suspension, checkpoints, and sub-workflows should 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? Also, 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 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 [KSS13], 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 software transactional memory. Therefore, it is 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 do a kind of verification, e.g., by estimating that ten percent 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 [GMC09].

Bibliography

- [ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.
- [ADB⁺99] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.
- [AFG⁺07] Liliana Ardissono, Roberto Furnari, Anna Goy, Giovanna Petrone, and Marino Segnan. Context-aware workflow management. In *International Conference on Web Engineering*, pages 47–52. Springer, 2007.
- [AH12] Malte Appeltauer and Robert Hirschfeld. *The JCop language specification: Version 1.0, April 2012*. Number 59. Universitätsverlag Potsdam, 2012.
- [AHHM11] Malte Appeltauer, Robert Hirschfeld, Michael Hauptand, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Computer Software*, 28(1):272–292, January 2011.
- [AHL13] Malte Appeltauer, Robert Hirschfeld, and Jens Lincke. Declarative layer composition with the JCop programming language. *Journal of Object Technology*, 12, 2013.
- [AKL08] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th international conference on Generative programming and component engineering*, pages 101–112. ACM, 2008.
- [AKM11] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Featherweight EventCJ: a core calculus for a context-oriented language with event-based per-instance layer transition. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*. ACM, 2011.
- [BCD13] Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.
- [BHF05] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes. The First 25 Years*, pages 133–150. Springer, 2005.
- [Bja12] RO Bjarnarson. Stackless Scala with free monads. *Scala Days*, 2012.
- [BLCVC⁺13] 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.

- [BMM05] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. *ACM SIGPLAN Notices*, 40(1):209–220, 2005.
- [BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *ECOOP’99–Object-Oriented Programming*, pages 43–66. Springer, 1999.
- [BS15] Eden Burton and Emil Sekerinski. The safety of dynamic mixin composition. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1992–1999. ACM, 2015.
- [BSR04] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30(6):355–371, 2004.
- [BVDR⁺12] 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 *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, Onward! 2012*, pages 67–84. ACM, 2012.
- [CA08] Daniel Crawl and Ilkay Altintas. A provenance-based fault tolerance mechanism for scientific workflows. *Provenance and Annotation of Data and Processes*, pages 152–159, 2008.
- [CD08] Pascal Costanza and Theo D’Hondt. Feature descriptions for context-oriented programming. In *Proc. of SPLC (2)*, pages 9–14, 2008. Workshop on Dynamic Software Product Lines (DSPL 2008).
- [CFGF08] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented programming. In *Proc. of the AOSD’08*, pages 72–83, New York, NY, USA, 2008. ACM.
- [CGM⁺13] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, and Theo D’Hondt. Modeling and analyzing self-adaptive systems with context Petri nets. In *Proc. of TASE*, pages 191–198. IEEE, 2013.
- [CGM⁺15] Nicolás Cardozo, Sebastián González, Kim Mens, Ragnhild Van Der Straeten, Jorge Vallejos, and Theo D’Hondt. Semantics for consistent activation in context-oriented systems. *Information and Software Technology*, 58:71–94, 2015.
- [CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10. ACM, 2005.
- [CIZ10] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proc. of the ACM OOPSLA*, pages 539–554, 2010.

- [CMLC06] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.
- [Con63] Melvin E Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.
- [CP13] Christian Colombo and Gordon J. Pace. Recovery within long-running transactions. *ACM Comput. Surv.*, 45(3):28:1–28:35, July 2013.
- [DDDCG02] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle_{II}. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.
- [DEH⁺17] 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.
- [DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Lisp and Functional Programming*, pages 151–160, 1990.
- [DFG14] Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A two-phase static analysis for reliable adaptation. In *International Conference on Software Engineering and Formal Methods*, pages 347–362. Springer, 2014.
- [DFG16] Pierpaolo Degano, Gian-Luigi Ferrari, and Letterio Galletta. A two-component language for adaptation: Design, semantics and program analysis. *IEEE Transactions on Software Engineering*, 42(6):505–529, 2016.
- [DL99] 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.
- [DMRI04] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimsky. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.
- [DPSS17] Ferruccio Damiani, Luca Padovani, Ina Schaefer, and Christoph Seidl. A core calculus for dynamic delta-oriented programming. *Acta Informatica*, pages 1–39, 2017.
- [DS11] Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, page 34. ACM, 2011.
- [EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97*, pages 263–273. ACM, 1997.

- [Fee93] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 179–187. ACM, 1993.
- [FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. of the ACM POPL*, pages 171–183. ACM, 1998.
- [Fre] Phil Freeman. Stack safety for free. <http://functorial.com/stack-safety-for-free/index.pdf>.
- [GCM⁺10] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C. In *International Conference on Software Language Engineering*, pages 246–265. Springer, 2010.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [Gib07] Jeremy Gibbons. Datatype-generic programming. In *Datatype-Generic Programming*, pages 1–71. Springer, 2007.
- [GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proc. of ACM POPL, POPL '09*, pages 127–139, New York, NY, USA, 2009. ACM.
- [GMCC13] Sebastián González, Kim Mens, Marius Colacioiu, and Walter Cazzola. Context traits: dynamic behaviour adaptation through run-time trait re-composition. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 209–220. ACM, 2013.
- [GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. of the ACM SIGMOD*, pages 249–259, New York, NY, USA, 1987. ACM.
- [Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.
- [HAGP08] 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*, pages 153–176. Springer-Verlag, 2008.
- [Har06] William L Harrison. The essence of multitasking. In *International Conference on Algebraic Methodology and Software Technology*, pages 158–172. Springer, 2006.
- [HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
- [HCNP02] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *International School on Advanced Functional Programming*, pages 159–187. Springer, 2002.

- [HFW84] 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.
- [HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.
- [HIM11] Robert Hirschfeld, Atsushi Igarashi, and Hidehiko Masuhara. ContextFJ: A minimal core calculus for context-oriented programming. In *Proc. of Foundations of Aspect-Oriented Languages (FOAL)*, March 2011.
- [HMPJH05] 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.
- [Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.
- [IHM12] A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. In *Proc. of FOOL*, October 2012.
- [IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
- [KAI14] Tetsuo Kamina, Tomoyuki Aotani, and Atsushi Igarashi. On-demand layer activation for type-safe deactivation. In *Proc. of COP’14*, Uppsala, Sweden, July 2014.
- [KAM11] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *Proc. of ACM AOSD*, pages 253–264. ACM, 2011.
- [KAM15] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism through contexts and subscribers. In *Proceedings of the 14th International Conference on Modularity*, pages 14–28. ACM, 2015.
- [KAM16] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Generalized layer activation mechanism for context-oriented programming. In Shigeru Chiba, Mario Südholt, Patrick Eugster, Lukasz Ziarek, and Gary T. Leavens, editors, *Transactions on Modularity and Composition I*, pages 123–166. Springer International Publishing, Cham, 2016.
- [KAMI] Tetsuo Kamina, Tomoyuki Aotani, Hidehiko Masuhara, and Atsushi Igarashi. Method safety mechanism for asynchronous layer deactivation. To appear.
- [KH08] 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.

- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. *Aspect-oriented programming*, pages 220–242. Springer, 1997.
- [KSS13] 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.
- [KT04] Tetsuo Kamina and Tetsuo Tamai. McJava—a design and implementation of Java with mixin-types. In *Proc. of APLAS*, pages 398–414, 2004.
- [LAB⁺06] 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.
- [LNRA16] Gary T Leavens, David Naumann, Hridesh Rajan, and Tomoyuki Aotani. Specifying and verifying advanced control features. In *International Symposium on Leveraging Applications of Formal Methods*, pages 80–96. Springer, 2016.
- [LQQ08] Chenguang Luo, Shengchao Qin, and Zongyan Qiu. Verifying BPEL-like programs with Hoare logic. In *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, pages 151–158. IEEE, 2008.
- [LS08] Luigi Liquori and Arnaud Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Transactions on Programming Languages and Systems*, 30(2):11, 2008.
- [LZPH07] 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.
- [MCDM08] Stijn Mostinckx, Andoni Lombide Carreton, and Wolfgang De Meuter. Reactive context-aware programming. *Electronic Communications of the EASST*, 11, 2008.
- [MJMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proc. of the ACM PLDI*, pages 274–285, New York, NY, USA, 2001. ACM.
- [MO12] Ingo Maier and Martin Odersky. Deprecating the observer pattern with Scala. React. Technical report, EPFL, 2012.
- [Mog89] 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.
- [mon] Monadless. <http://monadless.io/>.
- [NG06] NC 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.

- [OG10] Bruno CDS 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.
- [PG14] Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In *Mathematical Foundations of Programming Semantics Thirtieth Annual Conference*, page 273, 2014.
- [PHE99] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *PADL*, volume 99, pages 91–105. Springer, 1999.
- [pow] Powershell documentation. <https://docs.microsoft.com/powershell/>.
- [Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP’97–Object-Oriented Programming*, pages 419–443. Springer, 1997.
- [RLT78] Brian Randell, Pete Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.
- [RMO09] 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.
- [RSAS11] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Automated Software Engineering*, 18(2):163–197, 2011.
- [RV13] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proc. of the ACM POPL, POPL ’13*, pages 249–262, New York, NY, USA, 2013. ACM.
- [RW02] Algis Rudys and Dan S Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):138–168, 2002.
- [SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.
- [SBB⁺10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91, 2010.
- [sca] scalaz. <https://github.com/scalaz/scalaz>.
- [Sca11] STM Scala. Expert group. scalastm.web, 2011.
- [SDF⁺09] Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised⁶ report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

- [SDF⁺11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanaprasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *Proc. of the ACM PLDI*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.
- [SHM14] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th international conference on Modularity*, pages 25–36. ACM, 2014.
- [SLI08] Sucha Smachet, 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.
- [Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 38–45, New York, NY, USA, 1986. ACM.
- [ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [Swi08] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(4):423–436, 2008.
- [VLDN07] Martin Von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.
- [Wad95] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
- [Wei06] Westley Weimer. Exception-handling bugs in Java and a language extension to avoid them. *Advanced Topics in Exception Handling Techniques*, 4119:22–41, 2006.
- [WF94] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.
- [wwf] Windows workflow foundation. <https://docs.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/>.
- [YG07] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proc. ACM PEPN*, PEPN '07, pages 144–153, New York, NY, USA, 2007. ACM.
- [ZJ10] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ml. *Journal of Functional Programming*, 20(2):137–173, 2010.
- [ZLL15] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *Proc. of the ACM ICSE*, pages 767–777, Piscataway, NJ, USA, 2015. IEEE Press.

Appendix A

Proof of Type Soundness of ContextFJ_<:

A.1 Proofs

We fix CT and LT and assume (CT, LT) ok throughout this section.

Lemma A.1.1 *If $pmtree(m, C, L_2) = \bar{T} \rightarrow T_0$ and $L_1 <_w L_2$, then $pmtree(m, C, L_1) = \bar{T} \rightarrow T_0$.*

Proof : By induction on $L' <_w L$, using *noconflict*(L', L) in the case where $L' \triangleleft L$. $\square$

Lemma A.1.2 *If $mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0$ and $\Lambda_3 <_{sw} \Lambda_1$ and $\Lambda_4 <_{sw} \Lambda_2$ and $\Lambda_1 \subseteq \Lambda_2$ and $\Lambda_3 \subseteq \Lambda_4$, then $mtype(m, C, \Lambda_3, \Lambda_4) = \bar{T} \rightarrow T_0$.*

Proof : By induction on the derivation of $mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0$ with case analysis on the last rule used.

Case MT-CLASS: `class C  $\triangleleft$  D { ...  $T_0$  m( $\bar{T}$   $\bar{x}$ ) { return e; } ... }`

MT-CLASS finishes the case.

Case MT-PMETHOD: $\exists L_1 \in \Lambda_1. pmtree(m, C, L_1) = \bar{T} \rightarrow T_0$

By $\Lambda_3 <_{sw} \Lambda_1$, there exists $L_3 \in \Lambda_3$ such that either (1) $L_3 <_w L_1$ or (2) there exists L such that L swappable and $L_1, L_3 <_w L$. In the case (1), Lemma A.1.1 and MT-PMETHOD finish the case. In the case (2), by T-LAYERSW and *noconflict*(L_1, L_3), it is easy to show $pmtree(m, C, L_3) = \bar{T} \rightarrow T_0$. Then, Lemma A.1.1 and MT-PMETHOD finish the case.

Case MT-SUPER: `class C $\triangleleft$ D { ... $\bar{M}$ } $m \notin \bar{M}$
 $\forall L \in \Lambda_1. pmtree(m, C, L)$ undefined $mtype(m, D, \Lambda_2, \Lambda_2) = \bar{T} \rightarrow T_0$`

If $pmtree(m, C, L)$ is undefined for all $L \in \Lambda_3$, then the induction hypothesis and MT-SUPER finish the case. Otherwise, we have $pmtree(m, C, L) = S_0 \ m(\bar{S} \ \bar{x}) \{ \dots \}$ for some $L \in \Lambda_3$. Then, $mtype(m, C, \Lambda_3, \Lambda_4) = \bar{S} \rightarrow S_0$ holds by MT-PMETHOD and also there exists L' such that $LT(L')(C.m) = S_0 \ m(\bar{S} \ \bar{x}) \{ \dots \}$.

By the induction hypothesis, $mtype(m, D, dom(LT), dom(LT)) = \bar{T} \rightarrow T_0$ (since $dom(LT) <_w \Lambda_2$). By MT-SUPER, $mtype(m, C, \emptyset, dom(LT)) = \bar{T} \rightarrow T_0$. Finally, $\bar{S}, \bar{S}_0 = \bar{T}$, T_0 follows from *override*^{*h*}(L', C), finishing the case. $\square$

Lemma A.1.3 *If $mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0$ and $mtype(m, C, \Lambda_3, \Lambda_4) = \bar{T}' \rightarrow T_0'$, then $\bar{T}, T_0 = \bar{T}', T_0'$.*

Proof : By induction on the derivation of $mtype(m, C, \Lambda_1, \Lambda_2) = \bar{T} \rightarrow T_0$ with case analysis on the last rule used.

Case MT-CLASS: $\text{class } C \triangleleft D \{ \dots \ T_0 \ m(\bar{T} \ \bar{x}) \{ \dots \} \ \dots \}$

Easy. Use $\text{override}^h(L, C)$ for $L \in \text{dom}(LT)$ if $\text{mtype}(m, C, \Lambda_3, \Lambda_4) = \bar{T}' \rightarrow T_0'$ is derived by MT-PMETHOD, in which case there exists L' such that $L <_w L'$ and $LT(L')(C.m) = T_0' \ m(\bar{T}' \ \bar{x}) \{ \dots \}$. (Note that $\text{mtype}(m, C, \Lambda_3, \Lambda_4) = \bar{T}' \rightarrow T_0'$ cannot be derived by MT-SUPER.)

Case MT-PMETHOD: $\exists L_1 \in \Lambda_1. \text{pmttype}(m, C, L_1) = \bar{T} \rightarrow T_0$

There exists L_1' such that $L_1 <_w L_1'$ and $LT(L_1')(C.m) = T_0 \ m(\bar{T} \ \bar{x}) \{ \dots \}$. Further case analysis on $\text{mtype}(m, C, \Lambda_3, \Lambda_4) = \bar{T}' \rightarrow T_0'$.

Subcase MT-CLASS:

Similar to the above case.

Subcase MT-PMETHOD: $\exists L_2 \in \Lambda_3. \text{pmttype}(m, C, L_2) = \bar{T}' \rightarrow T_0'$

There exists L_2' such that $L_2 <_w L_2'$ and $LT(L_2')(C.m) = T_0' \ m(\bar{T}' \ \bar{x}) \{ \dots \}$. Then, $\text{noconflict}(L_1', L_2')$ finishes the case.

Subcase MT-SUPER: $\text{class } C \triangleleft D \{ \dots \ \bar{M} \} \quad m \notin \bar{M}$
 $\forall L \in \Lambda_3. \text{pmttype}(m, C, L) \text{ undefined}$
 $\text{mtype}(m, D, \Lambda_4, \Lambda_4) = \bar{T}' \rightarrow T_0'$

In this case, $\text{mtype}(m, C, \emptyset, \text{dom}(LT)) = \bar{T}' \rightarrow T_0'$ because we can show that

$$\begin{aligned} \text{mtype}(m, D, \Lambda_4, \Lambda_4) &= \text{mtype}(m, D, \text{dom}(LT), \text{dom}(LT)) \\ &= \text{mtype}(m, C, \emptyset, \text{dom}(LT)). \end{aligned}$$

by Lemma A.1.2 and MT-SUPER. Then, $\text{override}^h(L_1', C)$ finishes the case.

Case MT-SUPER: $\text{class } C \triangleleft D \{ \dots \ \bar{M} \} \quad m \notin \bar{M}$
 $\forall L \in \Lambda_1. \text{pmttype}(m, C, L) \text{ undefined} \quad \text{mtype}(m, D, \Lambda_2, \Lambda_2) = \bar{T} \rightarrow T_0$

Further case analysis on $\text{mtype}(m, C, \Lambda_3, \Lambda_4) = \bar{T}' \rightarrow T_0'$.

Subcase MT-PMETHOD:

Similar to the subcase MT-SUPER above.

Subcase MT-CLASS:

Cannot happen.

Subcase MT-SUPER:

By the induction hypothesis, $\text{mtype}(m, D, \Lambda_2, \Lambda_2) = \text{mtype}(m, D, \Lambda_4, \Lambda_4)$. □

Lemma A.1.4 If $\text{fields}(C) = \bar{T} \ \bar{F}$ and $D <: C$, then $\text{fields}(D) = \bar{T} \ \bar{F} \ \bar{S} \ \bar{G}$ for some $\bar{S}, \bar{G}$.

Proof : By induction on $D <: C$. □

Lemma A.1.5 (Weakening, Lemma 2.3.4) If $\mathcal{L}; \Lambda; \Gamma \vdash e : T$, then $\mathcal{L}; \Lambda; \Gamma, x : S \vdash e : T$.

Proof : By straightforward induction on $\mathcal{L}; \Lambda; \Gamma \vdash e : T$. □

Lemma A.1.6 If $\text{mtype}(m, C, \Lambda) = \bar{T} \rightarrow T_0$ and $D <: C$, then $\text{mtype}(m, D, \Lambda) = \bar{T} \rightarrow S_0$ and $S_0 <: T_0$ for some S_0 .

Proof : By induction on $D <: C$. We show only the case where D extends C . If $\text{class } D \triangleleft C \{ \dots \ S_0 \ m(\bar{S} \ \bar{x}) \{ \text{return } e; \} \ \dots \}$, then $\text{mtype}(m, D, \Lambda) = \bar{S} \rightarrow S_0$ for some $\bar{S}$ by MT-CLASS. By $\text{override}^v(D)$, $\bar{S} = \bar{T}$ and $S_0 <: T_0$. If $\exists L \in \Lambda. \text{pmttype}(m, D, L) = \bar{S} \rightarrow S_0$, then we have $\text{mtype}(m, D, \Lambda) = \bar{S} \rightarrow S_0$ by MT-PMETHOD. By $\text{override}^h(L, D)$, we get $\bar{S} = \bar{T}$ and $S_0 = T_0$. Otherwise, $\text{mtype}(m, D, \Lambda) = \bar{T} \rightarrow T_0$ by MT-SUPER. □

Lemma A.1.7 (Narrowing, Lemma 2.3.5) *If $\mathcal{L}; \Lambda; \Gamma \vdash e : T$ and $\Lambda' <_{sw} \Lambda$, then $\mathcal{L}; \Lambda'; \Gamma \vdash e : T$.*

Proof : By induction on $\mathcal{L}; \Lambda; \Gamma \vdash e : T$. We show only some representative cases.

Case T-INVK: $e = e_0.m(\bar{e})$ $\mathcal{L}; \Lambda; \Gamma \vdash e_0 : C_0$
 $mtype(m, C_0, \Lambda) = \bar{D} \rightarrow C$ $\mathcal{L}; \Lambda; \Gamma \vdash \bar{e} : \bar{E}$ $\bar{E} <: \bar{D}$

By Lemma A.1.2, $mtype(m, C_0, \Lambda') = \bar{D} \rightarrow C$. Then, the induction hypothesis and T-INVK finish the case.

Case T-WITH: $e = \text{with } e_l \ e_0$ $\mathcal{L}; \Lambda; \Gamma \vdash e_l : L$ $L \text{ req } \Lambda''$
 $\Lambda <_w \Lambda''$ $\mathcal{L}; \Lambda \cup \{L\}; \Gamma \vdash e_0 : T$

By LSSW-INTRO, we have $\Lambda' \cup \{L\} <_{sw} \Lambda \cup \{L\}$. By the induction hypothesis, $\mathcal{L}; \Lambda' \cup \{L\}; \Gamma \vdash e_0 : T$.

It is easy to show that $<_{sw}$ is transitive and so $\Lambda' <_{sw} \Lambda''$. By the induction hypothesis, we also have $\mathcal{L}; \Lambda'; \Gamma \vdash e_l : L$.

Moreover, in a well-formed program, $L \text{ req } \Lambda''$ means that Λ'' does not contain any sublayer of swappable layers. By these facts and LSS-INTRO, we get $\Lambda' <_w \Lambda''$. Then, by T-WITH, $\mathcal{L}; \Lambda'; \Gamma \vdash \text{with } e_l \ e_0 : T$, finishing the case.

Case T-SWAP: $e = \text{swap } (e_l, L_{sw}) \ e_0$ $\mathcal{L}; \Lambda; \Gamma \vdash e_l : L$
 $L_{sw} \text{ swappable}$ $L <_w L_{sw}$
 $L \text{ req } \Lambda''$ $\Lambda_{rm} = (\Lambda \setminus \{L' \mid L' <_w L_{sw}\})$
 $\Lambda_{rm} <_w \Lambda''$ $\mathcal{L}; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T$

It is easy to show that $(\Lambda' \setminus \{L' \mid L' <_w L_{sw}\}) \cup \{L\} <_{sw} \Lambda_{rm} \cup \{L\}$ from $\Lambda' <_{sw} \Lambda$. By the induction hypothesis, $\mathcal{L}; (\Lambda' \setminus \{L' \mid L' <_w L_{sw}\}) \cup \{L\}; \Gamma \vdash e_0 : T$.

By LSSW-INTRO, we have $(\Lambda' \setminus \{L' \mid L' <_w L_{sw}\}) \cup \{L\} <_{sw} \Lambda''$. Moreover, in a well-formed program, $L \text{ req } \Lambda''$ means that Λ'' does not have any sublayer of swappable layers. By these facts and LSS-INTRO, we get $(\Lambda' \setminus \{L' \mid L' <_w L_{sw}\}) \cup \{L\} <_w \Lambda''$. By the induction hypothesis, we also have $\mathcal{L}; \Lambda'; \Gamma \vdash e_l : L$. Then, by T-SWAP, $\mathcal{L}; \Lambda'; \Gamma \vdash \text{swap } (e_l, L_{sw}) \ e_0 : T$, finishing the case. $\square$

Lemma A.1.8 (Strengthening for values, Lemma 2.3.6) *If $\mathcal{L}; \Lambda; \Gamma \vdash v : T$ then, $\mathcal{L}'; \Lambda'; \Gamma' \vdash v : T$.*

Proof : By straightforward induction on $\mathcal{L}; \Lambda; \Gamma \vdash v : T$. $\square$

Lemma A.1.9 (Substitution, Lemma 2.3.7) *If $\mathcal{L}; \Lambda; \Gamma, \bar{x} : \bar{T} \vdash e : T$ and $\mathcal{L}; \Lambda; \Gamma \vdash \bar{v} : \bar{S}$ and $\bar{S} <: \bar{T}$, then $\mathcal{L}; \Lambda; \Gamma \vdash [\bar{v}/\bar{x}]e : S$ and $S <: T$ for some S .*

Proof : By induction on $\mathcal{L}; \Lambda; \Gamma, \bar{x} : \bar{T} \vdash e : T$ with case analysis on the last rule used. We show main cases of T-WITH and T-SWAP.

Case T-WITH: $e = \text{with } e_l \ e_0$ $\mathcal{L}; \Lambda; \Gamma, \bar{x} : \bar{T} \vdash e_l : L$ $L \text{ req } \Lambda'$
 $\Lambda <_w \Lambda'$ $\mathcal{L}; \Lambda \cup \{L\}; \Gamma, \bar{x} : \bar{T} \vdash e_0 : T$

By the induction hypothesis, $\mathcal{L}; \Lambda; \Gamma \vdash [\bar{v}/\bar{x}]e_l : L_0$ and $L_0 <: L$ for some L_0 . By induction on $L_0 <: L$, it is easy to show that $L_0 \text{ req } \Lambda'$. Since $L_0 <: L$, we also have $L_0 <_w L$ and so it is easy to show $\Lambda \cup \{L_0\} <_w \Lambda \cup \{L\}$. By the induction hypothesis, $\mathcal{L}; \Lambda \cup \{L\}; \Gamma \vdash [\bar{v}/\bar{x}]e_0 : S$ and $S <: T$ for some S . Then, by Lemma A.1.7, $\mathcal{L}; \Lambda \cup \{L_0\}; \Gamma \vdash [\bar{v}/\bar{x}]e_0 : S$; and T-WITH finish the case.

Case T-SWAP: $e = \text{swap } (e_l, L_{sw}) e_0$ $\mathcal{L}; \Lambda; \Gamma, \bar{x}: \bar{T} \vdash e_l : L$
 $L_{sw} \text{ swappable}$ $L <_w L_{sw}$ $L \text{ req } \Lambda'$
 $\Lambda_{rm} = \Lambda \setminus \{L' \mid L' <_w L_{sw}\}$ $\Lambda_{rm} <_w \Lambda'$
 $\mathcal{L}; \Lambda_{rm} \cup \{L\}; \Gamma, \bar{x}: \bar{T} \vdash e_0 : T$

By the induction hypothesis, $\mathcal{L}; \Lambda; \Gamma \vdash [\bar{v}/\bar{x}]e_l : L_0$ and $L_0 < L$ for some L_0 . By induction on $L_0 < L$, it is easy to show $L_0 \text{ req } \Lambda'$. Since $L_0 < L$, we have $L_0 <_w L$ and $L_0 <_w L_{sw}$ and $\Lambda_{rm} \cup \{L_0\} <_w \Lambda_{rm} \cup \{L\}$. By the induction hypothesis, $\mathcal{L}; \Lambda_{rm} \cup \{L\}; \Gamma \vdash [\bar{v}/\bar{x}]e_0 : S$ and $S < T$ for some S . Then, by Lemma A.1.7, $\mathcal{L}; \Lambda_{rm} \cup \{L_0\}; \Gamma \vdash [\bar{v}/\bar{x}]e_0 : S$; and T-WITH finishes the case. $\square$

Lemma A.1.10 *If $L_1 <_w L_2$ and $L_1 \text{ req } \Lambda_1$ and $L_2 \text{ req } \Lambda_2$, then $\Lambda_1 <_w \Lambda_2$.*

Proof : By induction on $L_1 <_w L_2$. Use T-LAYER in the case for LSW-EXTENDS. $\square$

We prove a stronger property than Lemma 2.3.11; in the statement below, $(<_w; \text{req})$ stands for the composition of the two relations $<_w$ and req .

Lemma A.1.11 *If $\Lambda \text{ wf}$, then $\forall L \in \Lambda, \forall L' \text{ s.t. } L (<_w; \text{req}) L', \exists L'' \in \Lambda. L'' <_w L'$.*

Proof : Induction on the derivation of $\Lambda \text{ wf}$.

Case WF-EMPTY:

Trivial.

Case WF-WITH: $\Lambda = \Lambda_0 \cup \{L_a\}$ $\Lambda_0 \text{ wf}$ $L_a \text{ req } \Lambda'$ $\Lambda_0 <_w \Lambda'$

By the induction hypothesis, we have $\forall L \in \Lambda_0. \forall L' \text{ s.t. } L (<_w; \text{req}) L'. \exists L'' \in \Lambda_0. L'' <_w L'$. By $\Lambda_0 <_w \Lambda'$ and Lemma A.1.10, we have $\forall L' \text{ s.t. } L_a (<_w; \text{req}) L'. \exists L'' \in \Lambda_0. L'' <_w L'$. So, $\forall L \in \Lambda. \forall L' \text{ s.t. } L (<_w; \text{req}) L'. \exists L'' \in \Lambda. L'' <_w L'$.

Case WF-SWAP: $\Lambda = \Lambda_{rm} \cup \{L_a\}$ $\Lambda_0 \text{ wf}$ $L_{sw} \text{ swappable}$ $L_a <_w L_{sw}$
 $L_a \text{ req } \Lambda_a$ $\Lambda_{rm} = \Lambda_0 \setminus \{L' \mid L' <_w L_{sw}\}$ $\Lambda_{rm} <_w \Lambda_a$

By the induction hypothesis, we have

$$\forall L \in \Lambda_0. \forall L' \text{ s.t. } L (<_w; \text{req}) L'. \exists L'' \in \Lambda_0. L'' <_w L',$$

and so

$$\forall L \in \Lambda_{rm}. \forall L' \text{ s.t. } L (<_w; \text{req}) L'. \exists L'' \in \Lambda_0 \text{ s.t. } L'' <_w L'.$$

In fact, we can show that

$$\forall L \in \Lambda_{rm}. \forall L' \text{ s.t. } L (<_w; \text{req}) L', \exists L'' \in (\Lambda_{rm} \cup \{L_a\}) \text{ s.t. } L'' <_w L' :$$

if $L'' \in \{L_b \mid L_b <_w L_{sw}\}$ for given L and L' , then it must be the case that $L_{sw} <_w L'$ because L' is required by some weak supertype of L and so must not be a sublayer of a swappable and that $L_a <_w L'$.

By $L_a \text{ req } \Lambda_a$ and $\Lambda_{rm} <_w \Lambda_a$, we finally have

$$\forall L \in \Lambda. \forall L' \text{ s.t. } L (<_w; \text{req}) L', \exists L'' \in (\Lambda_{rm} \cup \{L_a\}) \text{ s.t. } L'' <_w L'.$$

$\square$

Lemma A.1.12 (Lemma 2.3.12) *If $\Lambda \text{ wf}$ and $\text{mtype}(\mathbf{m}, \mathbf{C}, \Lambda)$ defined and $\text{mtype}(\mathbf{m}, \mathbf{D}, \Lambda)$ undefined and $\mathbf{C} \triangleleft \mathbf{D}$, then $(\exists L' \in \Lambda. \text{proceed} \notin \text{pmbody}(\mathbf{m}, \mathbf{C}, L'))$ or $\text{mtype}(\mathbf{m}, \mathbf{C}, \emptyset, \Lambda)$ defined.*

Proof: We prove by induction on the derivation of wf a stronger property: If Λwf and $mtype(m, C, \Lambda)$ defined and $mtype(m, D, \Lambda)$ undefined and $C \triangleleft D$, then $(\exists L' \in \Lambda.\text{proceed} \notin \text{pmbody}(m, C, L') \wedge (\forall L'', L''' \text{ s.t. } L' \prec_w L'' \wedge L'' \text{ swappable} \wedge L''' \prec_w L''.\text{proceed} \notin \text{pmbody}(m, C, L''')))$ or $mtype(m, C, \emptyset, \Lambda)$ is defined.

In what follows, we define predicate $npr(m, C, \Lambda)$ by $(\exists L' \in \Lambda.\text{proceed} \notin \text{pmbody}(m, C, L') \wedge (\forall L'', L''' \text{ s.t. } L' \prec_w L'' \wedge L'' \text{ swappable} \wedge L''' \prec_w L''.\text{proceed} \notin \text{pmbody}(m, C, L''')))$ or $mtype(m, C, \emptyset, \Lambda)$ is defined.

Case WF-EMPTY:

Trivial.

Case WF-WITH: $\Lambda = \Lambda_0 \cup \{L_a\}$ $\Lambda_0 wf$ $L_a \text{ req } \Lambda'$ $\Lambda_0 \prec_w \Lambda'$

If $mtype(m, C, \Lambda_0)$ is defined, by the induction hypothesis, $npr(m, C, \Lambda_0)$ holds. Since $\Lambda = \Lambda_0 \cup \{L_a\}$, $npr(m, C, \Lambda)$ also holds.

Otherwise, it must be the case that $mtype(m, C, \Lambda_0)$ undefined and $mtype(m, C, \{L_a\})$ defined. Since $\Lambda \prec_w \Lambda_0 \prec_w \Lambda'$ and neither $mtype(m, C, \Lambda_0)$ nor $mtype(m, D, \Lambda)$ is defined, $mtype(m, C, \Lambda', \Lambda' \cup \{L_a\})$ is undefined. Then, $\text{proceed} \notin \text{pmbody}(m, C, L_a)$ holds since if the partial method had proceed , it would contradict the fact that L_a is well-typed (in particular, $mtype(m, C, \Lambda', \Lambda' \cup \{L_a\})$ would not be defined, as opposed to what T-PROCEED requires). If L_a is a sublayer of swappable layer L_{sw} , for all $L_b \prec_w L_{sw}$, $\text{proceed} \notin \text{pmbody}(m, C, L_b)$ through the same argument (note that $L_b \text{ req } \Lambda'$). Then, $npr(m, C, \Lambda)$ holds.

Case WF-SWAP: $\Lambda = \Lambda_{rm} \cup \{L_a\}$ $\Lambda_0 wf$ $L_{sw} \text{ swappable}$ $L_a \prec_w L_{sw}$
 $L_a \text{ req } \Lambda_a$ $\Lambda_{rm} = \Lambda_0 \setminus \{L' \mid L' \prec_w L_{sw}\}$ $\Lambda_{rm} \prec_w \Lambda_a$

It is easy to show $\Lambda \prec_w \Lambda_0$ and vice versa. By Lemma A.1.2, $mtype(m, C, \Lambda_0)$ is defined and $mtype(m, D, \Lambda_0)$ is undefined. By the induction hypothesis, $npr(m, C, \Lambda_0)$, that is, either (1) $mtype(m, C, \emptyset, \Lambda_0)$ is defined, or (2) $(\exists L' \in \Lambda_0.\text{proceed} \notin \text{pmbody}(m, C, L') \wedge (\forall L'', L''' \text{ s.t. } L' \prec_w L'' \wedge L'' \text{ swappable} \wedge L''' \prec_w L''.\text{proceed} \notin \text{pmbody}(m, C, L''')))$.

We show $npr(m, C, \Lambda)$ by case analysis. In the case (1), we have $mtype(m, C, \emptyset, \Lambda)$ defined by Lemma A.1.2. The case (2) is also easy: if $L' \in \Lambda_{rm}$, then $L' \in \Lambda$; otherwise, $\text{proceed} \notin \text{pmbody}(m, C, L_a)$ because $L' \prec_w L_{sw}$ and $L_a \prec_w L_{sw}$ and $L_{sw} \text{ swappable}$, hence $npr(m, C, \Lambda)$. □

Lemma A.1.13 (Lemma 2.3.13) If $\{\bar{L}\} wf$ and $mtype(m, C, \{\bar{L}\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$, then $ndp(m, C, \bar{L}, \bar{L})$.

Proof: By induction on the length of $C \prec D \prec \dots \text{Object}$. The case where the length is zero is trivial.

Case: $C \triangleleft D$ $mtype(m, D, \{\bar{L}\})$ undefined

By $\{\bar{L}\} wf$ and Lemma A.1.12, we have $mtype(m, C, \emptyset, \{\bar{L}\})$ is defined or $\exists L_1 \in \{\bar{L}\}.\text{proceed} \notin \text{pmbody}(m, C, L_1)$. If $mtype(m, C, \emptyset, \{\bar{L}\})$ is defined, class C must have the definition of method m since $mtype(m, D, \{\bar{L}\})$ is undefined, and so NDP-CLASS finishes the case. In the other case, NDP-LAYER finishes the case.

Case: $C \triangleleft D$ $mtype(m, D, \{\bar{L}\})$ defined

By the induction hypothesis, $ndp(m, D, \bar{L}, \bar{L})$ holds. Then, NDP-SUPER finishes the case. □

Lemma A.1.14 (Lemma 2.3.14) If $ndp(m, C, \bar{L}', \bar{L})$, then $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ for some $\bar{T}$ and T_0 .

Proof : By induction on $ndp(m, C, \bar{L}', \bar{L})$. □

Lemma A.1.15 *If $L.C.m; \Lambda; \Gamma \vdash e : T$ and $L' <_w L$, then $L'.C.m; \Lambda; \Gamma \vdash e : T$.*

Proof : Suppose that $L \text{ req } \Lambda_0$ and $L' \text{ req } \Lambda_1$. Since L and L' are well-formed, $\Lambda_1 <_w \Lambda_0$. We proceed by induction on $L.C.m; \Lambda; \Gamma \vdash e : T$. We show only main cases.

Case T-SUPERP: $e = \text{super}.m'(\bar{e})$ $\text{class } C \triangleleft D$ $L \text{ req } \Lambda_0$
 $mtype(m', D, \Lambda_0 \cup \{L\}) = \bar{T} \rightarrow T$ $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}$ $\bar{S} <: \bar{T}$

Since $L' <_w L$ and $\Lambda_1 <_w \Lambda_0$, we have $\Lambda_1 \cup \{L'\} <_w \Lambda_0 \cup \{L\}$. Then, by Lemma A.1.2, $mtype(m', D, \Lambda_1 \cup \{L'\}) = \bar{T} \rightarrow T$. The induction hypothesis and T-SUPERP finish the case.

Case T-PROCEED: $e = \text{proceed}(\bar{e})$ $L \text{ req } \Lambda_0$
 $mtype(m, C, \Lambda_0, \Lambda_0 \cup \{L\}) = \bar{T} \rightarrow T$ $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}$
 $\bar{S} <: \bar{T}$

Since $\Lambda_1 <_w \Lambda_0$, we have $\Lambda_1 \cup \{L'\} <_w \Lambda_0 \cup \{L\}$. Then, by Lemma A.1.2, $mtype(m, C, \Lambda_1, \Lambda_1 \cup \{L'\}) = \bar{T} \rightarrow T$. The induction hypothesis and T-PROCEED finish the case.

Case T-SUPERPROCEED: $e = \text{superproceed}(\bar{e})$ $L \triangleleft L''$
 $pmttype(m, C, L'') = \bar{T} \rightarrow T$ $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}$
 $\bar{S} <: \bar{T}$

We have that for some L''' , $L' \triangleleft L'''$. Then, $L''' <_w L''$ and $pmttype(m, C, L''') = \bar{T} \rightarrow T$ by Lemma A.1.1. The induction hypothesis and T-SUPERPROCEED finish the case. □

Lemma A.1.16 (Inversion for partial method body, Lemma 2.3.9) *If $pmbody(m, C, L) = \bar{x}.e_0$ in L' and $L \text{ req } \Lambda$ and $pmttype(m, C, L) = \bar{T} \rightarrow T_0$, then $L.C.m; \Lambda \cup \{L\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0$ for some $S_0 <_w T_0$.*

Proof : By induction on $pmbody(m, C, L) = \bar{x}.e_0$ in L' .

Case PMB-SUPER: $LT(L)(C.m)$ undefined $L \triangleleft L''$
 $pmbody(m, C, L'') = \bar{x}.e_0$ in L'

By $pmttype(m, C, L) = \bar{T} \rightarrow T_0$ and PMT-SUPER, it must be the case that $pmttype(m, C, L'') = \bar{T} \rightarrow T_0$. By the induction hypothesis,

$$L''.C.m; \Lambda \cup \{L''\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0$$

for some $S_0 <_w T_0$. Lemmas A.1.7 and A.1.15 finish the case.

Case PMB-LAYER: $LT(L)(C.m) = T_0$ $C.m(\bar{T} \ \bar{x})\{ \text{return } e; \}$ $L' = L$

By T-PMETHOD, it must be the case that

$$L.C.m; \Lambda \cup \{L\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : S_0$$

for some S_0 s.t. $S_0 <_w T_0$, finishing the case. □

Lemma A.1.17 (Substitution for super, proceed and superproceed, Lemma 2.3.8)

1. *If $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ and $L.C.m; \Lambda; \Gamma \vdash e : T$ and $C_0.m \vdash \langle C, (\bar{L}'; L''), \bar{L} \rangle \text{ ok}$ and $C \triangleleft D$ and $L'' <_w L \triangleleft L'$ and $\Lambda <_{sw} \{L\}$ and $\text{proceed} \in e \implies ndp(m, C, \bar{L}', \bar{L})$, then $\bullet; \Lambda; \Gamma \vdash Se : T$ where*

$$S = \left[\begin{array}{ll} \text{new } C_0(\bar{v}) \langle C, \bar{L}', \bar{L} \rangle . m & / \text{proceed,} \\ \text{new } C_0(\bar{v}) \langle D, \bar{L}, \bar{L} \rangle & / \text{super,} \\ \text{new } C_0(\bar{v}) \langle C, L', (\bar{L}'; L''), \bar{L} \rangle . m / \text{superproceed} \end{array} \right].$$

2. If $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ and $C.m; \Lambda; \Gamma \vdash e : T$ and $C_0.m \vdash \langle C, \bar{L}', \bar{L} \rangle \text{ ok}$ and $C \triangleleft D$ and $\Lambda \triangleleft_{sw} \{\bar{L}\}$, then $\bullet; \Lambda; \Gamma \vdash [\text{new } C_0(\bar{v}) \langle D, \bar{L}, \bar{L} \rangle / \text{super}]e : T$.

Proof : 1. By induction on $L.C.m; \Lambda; \Gamma \vdash e : T$ with case analysis on the last typing rule used. We show main cases below.

Case T-SUPERB:

Cannot happen.

Case T-SUPERP: $e = \text{super}.m'(\bar{e})$ $mtype(m', D, \Lambda' \cup \{L\}) = \bar{T}' \rightarrow T$
 $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}'$ $L \text{ req } \Lambda'$ $\bar{S}' \triangleleft \bar{T}'$

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) \langle D, \bar{L}, \bar{L} \rangle.m'(S\bar{e}) : T$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$. Next, we show $C_0.m \vdash \langle D, \bar{L}, \bar{L} \rangle \text{ ok}$. By $C_0.m \vdash \langle C, \bar{L}', \bar{L} \rangle \text{ ok}$, we have $C_0 \triangleleft C$, from which $C_0 \triangleleft D$ follows, and $\{\bar{L}\} \text{ wf}$. By Lemma A.1.11 and $L'' \in \{\bar{L}\}$ and $L'' \triangleleft_w L$, for any L_1 such that $L \text{ req } L_1$, there exists $L_2 \in \{\bar{L}\}$ such that $L_2 \triangleleft_w L_1$; so, $\{\bar{L}\} \triangleleft_w \Lambda' \cup \{L\}$. Then, by $mtype(m', D, \Lambda' \cup \{L\}) = \bar{T}' \rightarrow T$ and Lemma A.1.2, we have $mtype(m', D, \{\bar{L}\}) = \bar{T}' \rightarrow T$; moreover, by Lemma A.1.13, $ndp(m, D, \bar{L}, \bar{L})$. So, $C_0.m \vdash \langle D, \bar{L}, \bar{L} \rangle \text{ ok}$. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\bar{e} : \bar{S}'$ and, by assumption, $\bar{S}' \triangleleft \bar{T}'$. Finally, T-INVKA finishes the case.

Case T-PROCEED: $e = \text{proceed}(\bar{e})$ $mtype(m, C, \Lambda', \Lambda' \cup \{L\}) = \bar{T}' \rightarrow T$
 $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}'$ $L \text{ req } \Lambda'$ $\bar{S}' \triangleleft \bar{T}'$

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \text{new } C_0 \langle C, \bar{L}', \bar{L} \rangle(\bar{v}).m(S\bar{e}) : T$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$. Since $\text{proceed} \in e$, we have $ndp(m, C, \bar{L}', \bar{L})$, from which $C_0.m \vdash \langle C, \bar{L}', \bar{L} \rangle \text{ ok}$ and follow. By Lemmas A.1.14 and A.1.3, we have $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T}' \rightarrow T$, too. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\bar{e} : \bar{S}'$ and, by assumption, $\bar{S}' \triangleleft \bar{T}'$. Finally, T-INVKA finishes the case.

Case T-SUPERPROCEED: $e = \text{superproceed}(\bar{e})$ $L \triangleleft L'$
 $pmttype(m, C, L') = \bar{T}' \rightarrow T$
 $L.C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}'$ $\bar{S}' \triangleleft \bar{T}'$

It suffices to show that $\bullet; \Lambda; \Gamma \vdash \text{new } C_0 \langle C, L', (\bar{L}'; L'') \rangle.m(S\bar{e}) : T$.

By assumption, we have $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$ and $C_0.m \vdash \langle C, (\bar{L}'; L''), \bar{L} \rangle \text{ ok}$ and $L'' \triangleleft_w L'$. Also, $pmttype(m, C, L') = \bar{T}' \rightarrow T$, by assumption. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\bar{e} : \bar{S}'$ and, by assumption, $\bar{S}' \triangleleft \bar{T}'$. Finally, T-INVKA finishes the case.

Case T-WITH: $e = \text{with } e_l \ e_0$ $L.C.m; \Lambda; \Gamma \vdash e_l : L$ $L \text{ req } \Lambda_0$
 $\Lambda \triangleleft_w \Lambda_0$ $L.C.m; \Lambda \cup \{L\}; \Gamma \vdash e_0 : T$

Since $\Lambda \triangleleft_{sw} \{\bar{L}\}$, we have $\Lambda \cup \{L\} \triangleleft_{sw} \{\bar{L}\}$ by LSSW-INTRO. By the induction hypothesis, $\bullet; \Lambda; \Gamma \vdash Se_l : L$ and $\bullet; \Lambda \cup \{L\}; \Gamma \vdash Se_0 : T$. T-WITH finishes the case.

Case T-SWAP: $e = \text{swap } (e_l, L_{sw}) \ e_0$ $L.C.m; \Lambda; \Gamma \vdash e_l : L$
 $L \text{ req } \Lambda_0$ $L_{sw} \text{ swappable}$
 $L \triangleleft_w L_{sw}$ $\Lambda_{rm} = \Lambda \setminus \{L' \mid L' \triangleleft_w L_{sw}\}$
 $\Lambda_{rm} \triangleleft_w \Lambda_0$ $L.C.m; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T$

Since $\Lambda \triangleleft_{sw} \{\bar{L}\}$, we have $\Lambda_{rm} \cup \{L\} \triangleleft_{sw} \{\bar{L}\}$ by LSSW-INTRO. By the induction hypothesis, $\bullet; \Lambda; \Gamma \vdash Se_l : L$ and $\bullet; \Lambda_{rm} \cup \{L\}; \Gamma \vdash Se_0 : T$. T-SWAP finishes the case.

2. By induction on $C.m; \Lambda; \Gamma \vdash e : T_0$ with case analysis on the last typing rule used. We show only main cases below (note that none of the cases T-PROCEED and T-SUPERP and T-SUPERPROCEED can happen).

Case T-SUPERB: $e = \text{super}.m'(\bar{e})$ $mtype(m', D, \emptyset) = \bar{T}' \rightarrow T_0$
 $C.m; \Lambda; \Gamma \vdash \bar{e} : \bar{S}'$ $\bar{S}' <: \bar{T}'$

Let $S = [\text{new } C_0(\bar{v}) < D, \bar{L}, \bar{L}' > / \text{super}]$. It suffices to show that $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) < D, \bar{L}, \bar{L}' > .m'(S\bar{e}) : T_0$. By assumption, we have $\bullet; \Lambda; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0$. Next, we show $C_0.m \vdash < D, \bar{L}, \bar{L}' > \text{ ok}$. By $C_0.m \vdash < C, \bar{L}', \bar{L} > \text{ ok}$, we have $C_0 <: C$, from which $C_0 <: D$ follows, and $\{\bar{L}\} \text{ wf}$. By $mtype(m', D, \emptyset) = \bar{T}' \rightarrow T_0$ and Lemma A.1.2, we have $mtype(m', D, \{\bar{L}\}) = \bar{T}' \rightarrow T_0$; moreover, by Lemma A.1.13, $ndp(m, D, \bar{L}, \bar{L}')$. So, $C_0.m \vdash < D, \bar{L}, \bar{L}' > \text{ ok}$. By the induction hypothesis, we have $\bullet; \Lambda; \Gamma \vdash S\bar{e} : \bar{S}'$ and, by assumption, $\bar{S}' <: \bar{T}'$. Finally, T-INVKA finishes the case. $\square$

Lemma A.1.18 (Inversion for method body, Lemma 2.3.10) Suppose $\{\bar{L}\} \text{ wf}$ and $mbody(m, C, \bar{L}', \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$ and $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ and $ndp(m, C, \bar{L}', \bar{L})$.

1. If $\bar{L}'' = \bar{L}''' ; L_0$, then $L_0 \text{ req } \Lambda$ and $L_0.C'.m; \Lambda \cup \{L_0\}; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : U_0$ and $C <: C'$ and $U_0 <: T_0$ and $ndp(m, C', \bar{L}'', \bar{L})$ for some Λ and U_0 .
2. If $\bar{L}'' = \bullet$, then $C'.m; \emptyset; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : U_0$ and $C <: C'$ and $U_0 <: T_0$ and $ndp(m, C', \bullet, \bar{L})$ for some U_0 .

Proof: Both 1 and 2 are proved simultaneously by induction on $mbody(m, C, \bar{L}', \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$.

Case MB-CLASS: $\text{class } C \triangleleft D \{ \dots S_0 m(\bar{S} \bar{x}) \{ \text{return } e_0; \} \dots \}$
 $C' = C$ $\bar{L}' = \bullet$ $\bar{L}'' = \bullet$

By T-CLASS, T-METHOD, MT-CLASS, it must be the case that

$$T_0, \bar{T} = S_0, \bar{S} \quad C.m; \emptyset; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : U_0 \quad U_0 <: T_0$$

for some U_0 . We have $ndp(m, C, \bullet, \bar{L})$ by NDP-CLASS, finishing the case.

Case MB-LAYER: $pmbody(m, C, L_0) = \bar{x}.e_0$ in L_1 $C' = C$ $\bar{L}'' = \bar{L}'$

By the definition of $pmbody$, there exists some L_1 such that $LT(L_1)(C.m) = S_0 C.m(\bar{S} \bar{x}) \{ \text{return } e; \}$ and $L_0 <_w L_1$. By T-PMETHOD, it must be the case that

$$T_0, \bar{T} = S_0, \bar{S} \quad L_1 \text{ req } \Lambda_1 \quad L_1.C.m; \Lambda_1 \cup \{L_1\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : U_0 \quad U_0 <: T_0$$

for some U_0 and Λ_1 . It is easy to show by induction on $L_0 <_w L_1$ using Lemma A.1.7 and T-LAYER and T-LAYERSW that

$$L_0.C.m; \Lambda \cup \{L_0\}; \bar{x} : \bar{T}, \text{this} : C \vdash e_0 : U_0$$

for some Λ such that $L_0 \text{ req } \Lambda$. Finally, we have $ndp(m, C', \bar{L}'', \bar{L})$ by assumption, finishing the case.

Case MB-SUPER: $\bar{L}' = \bullet$ $\text{class } C \triangleleft D \{ \dots \bar{M} \}$ $m \notin \bar{M}$
 $mbody(m, D, \bar{L}, \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$

By MT-SUPER, it must be the case that $mtype(m, D, \{\bar{L}\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$. By Lemma A.1.13, we have $ndp(m, D, \bar{L}, \bar{L})$. The induction hypothesis and transitivity of subtyping finish the case.

Case MB-NEXTLAYER: $\bar{L}' = \bar{L}_b; L_1$ $pmbody(m, C, L_1)$ undefined
 $mbody(m, C, \bar{L}_b, \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$

We show $ndp(m, C, \bar{L}_b, \bar{L})$ holds by case analysis on $ndp(m, C, \bar{L}_b; L_1, \bar{L})$. The cases NDP-SUPER and NDP-CLASS are easy. The case NDP-LAYER is easy, too: since $pmbody(m, C, L_1)$ undefined, by NDP-LAYER, we have $ndp(m, C, \bar{L}_b, \bar{L})$. Since $pmbody(m, C, L_1)$ is undefined and $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$, it must be the case that $mtype(m, C, \{\bar{L}_b\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$. Then, the induction hypothesis finishes the case. $\square$

Theorem A.1.1 (Subject Reduction) Suppose $\vdash (CT, LT)$ ok. If $\bullet; \{\bar{L}\}; \Gamma \vdash e : T$ and $\{\bar{L}\}$ wf and $\bar{L} \vdash e \longrightarrow e'$, then $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S$ for some S such that $S <: T$.

Proof: By induction on $\bar{L} \vdash e \longrightarrow e'$ with case analysis on the last reduction rule used. We show only main cases.

Case R-FIELD: $e = \text{new } C_0(\bar{v}).f_i$ $fields(C_0) = \bar{C} \ \bar{f}$ $e' = v_i$

By T-FIELD and T-NEW, it must be the case that

$$\bullet; \{\bar{L}\}; \Gamma \vdash \bar{v} : \bar{D} \quad \bar{D} <: \bar{C} \quad C = C_i$$

Then, we have $\bullet; \{\bar{L}\}; \Gamma \vdash v_i : D_i$ and $D_i <: C_i$, finishing the case.

Case R-INVK: $e = \text{new } C_0(\bar{v}).m(\bar{w})$
 $\bar{L} \vdash \text{new } C_0(\bar{v}) < C_0, \bar{L}, \bar{L} > .m(\bar{w}) \longrightarrow e'$

By T-INVK and T-NEW, it must be the case that

$$\begin{aligned} \bullet; \{\bar{L}\}; \Gamma \vdash \bar{v} : \bar{S} \quad fields(C_0) = \bar{T} \ \bar{f} \quad \bar{S} <: \bar{T} \\ mtype(m, C_0, \{\bar{L}\}) = \bar{T}' \rightarrow T \quad \bullet; \{\bar{L}\}; \Gamma \vdash \bar{w} : \bar{S}' \quad \bar{S}' <: \bar{T}'. \end{aligned}$$

By Lemma A.1.13, $ndp(m, C_0, \bar{L}, \bar{L})$ and so $C_0.m \vdash < C_0, \bar{L}, \bar{L} >$ ok holds. Since $\{\bar{L}\} <_{sw} \{\bar{L}\}$, we have

$$\bullet; \{\bar{L}\}; \Gamma \vdash \text{new } C_0(\bar{v}) < C_0, \bar{L}, \bar{L} > .m(\bar{w}) : T$$

by T-INVKA. By the induction hypothesis, $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S$ for some $S <: T$, finishing the case.

Case R-INVKP: $e = \text{new } C_0(\bar{v}) < C', \bar{L}'', \bar{L}' > .m(\bar{w})$
 $mbody(m, C', \bar{L}'', \bar{L}') = \bar{x}.e_0$ in $C'', (\bar{L}'''; L_0)$
 $C'' \triangleleft D$
 $L_0 \triangleleft L_1$

$$e' = \left[\begin{array}{ll} \text{new } C_0(\bar{v}) & / \text{this,} \\ \bar{w} & / \bar{x}, \\ \text{new } C_0(\bar{v}) < C'', \bar{L}''', \bar{L}' > .m & / \text{proceed,} \\ \text{new } C_0(\bar{v}) < D, \bar{L}', \bar{L}' > & / \text{super,} \\ \text{new } C_0(\bar{v}) < C'', L_1, (\bar{L}'''; L_0), \bar{L}' > .m / \text{superproceed} \end{array} \right] e_0$$

By T-INVKA, it must be the case that

$$\begin{aligned} \bullet; \{\bar{L}\}; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0 \quad C_0.m \vdash < C', \bar{L}'', \bar{L}' > \text{ ok} \quad \{\bar{L}\} <_{sw} \{\bar{L}'\} \\ mtype(m, C', \{\bar{L}''\}, \{\bar{L}'\}) = \bar{T}' \rightarrow T \quad \bullet; \{\bar{L}\}; \Gamma \vdash \bar{w} : \bar{S}' \quad \bar{S}' <: \bar{T}' \end{aligned}$$

for some $\bar{T}'$ and $\bar{S}'$.

By Lemma A.1.18,

$$\begin{aligned} & L_0.C''.m; \Lambda \cup \{L_0\}; \bar{x} : \bar{T}, \text{this} : C'' \vdash e_0 : S \\ & L_0 \text{ req } \Lambda \\ & C' <: C'' \\ & S <: T \\ & \text{ndp}(m, C'', \bar{L}'''; L_0, \bar{L}') \end{aligned}$$

and for some Λ and S .

By S-TRANS, $C_0 <: C''$. From $\text{ndp}(m, C'', \bar{L}'''; L_0, \bar{L}')$ and $C_0.m \vdash \langle C', \bar{L}'', \bar{L}' \rangle \text{ ok}$, it follows that $C_0.m \vdash \langle C'', (\bar{L}'''; L_0), \bar{L}' \rangle \text{ ok}$.

By $\{\bar{L}'\} \text{ wf}$ and Lemma A.1.11 and $L_0 \in \bar{L}'$ and $L_0 \text{ req } \Lambda$, we have $\forall L \in \Lambda, \exists L' \in \bar{L}'. L' <_w L$. So, by LSS-INTRO, we have $\{\bar{L}'\} = \{\bar{L}'\} \cup \{L_0\} <_w \Lambda \cup \{L_0\}$. By this fact and $\{\bar{L}\} <_{sw} \{\bar{L}'\}$, we get $\{\bar{L}\} <_{sw} \Lambda \cup \{L_0\}$. By Lemma A.1.7,

$$L_0.C''.m; \{\bar{L}\}; \bar{x} : \bar{T}, \text{this} : C'' \vdash e_0 : S$$

By $\text{ndp}(m, C'', \bar{L}'''; L_0, \bar{L}')$ and the definition of ndp , $\text{proceed} \in e_0$ implies $\text{ndp}(m, C', \bar{L}''', \bar{L}')$. Then, by Lemmas A.1.8 and Lemma A.1.17(1),

$$\bullet; \{\bar{L}\}; \bar{x} : \bar{T}, \text{this} : C'' \vdash \left[\begin{array}{ll} \text{new } C_0(\bar{v}) \langle C'', \bar{L}''', \bar{L}' \rangle .m & / \text{proceed,} \\ \text{new } C_0(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super,} \\ \text{new } C_0(\bar{v}) \langle C'', L_1, (\bar{L}'''; L_0), \bar{L}' \rangle .m / \text{superproceed} \end{array} \right] e_0 : S$$

By Lemmas A.1.8, A.1.5 and A.1.9, $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S'$ for some $S' <: S$. By S-TRANS, $S' <: T$, finishing the case.

Case R-INVKSP: $e = \text{new } C_0(\bar{v}) \langle C', L_1, (\bar{L}''; L_0), \bar{L}' \rangle .m(\bar{w})$
 $\text{pmbody}(m, C', L_1) = \bar{x}.e_0 \text{ in } L_2$
 $C' <: D$
 $L_2 <: L_3$

$$e' = \left[\begin{array}{ll} \text{new } C_0(\bar{v}) & / \text{this} \\ \bar{w} & / \bar{x} \\ \text{new } C_0(\bar{v}) \langle C'', \bar{L}'', \bar{L}' \rangle .m & / \text{proceed} \\ \text{new } C_0(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super,} \\ \text{new } C_0(\bar{v}) \langle C'', L_3, (\bar{L}''; L_0), \bar{L}' \rangle .m / \text{superproceed} \end{array} \right] e_0$$

By T-INVKAL, it must be the case that

$$\begin{array}{lll} \bullet; \{\bar{L}\}; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0 & C.m \vdash \langle C', (\bar{L}''; L_0), \bar{L}' \rangle \text{ ok} & \{\bar{L}\} <_{sw} \{\bar{L}'\} \\ L_0 <_w L_1 & \text{pmtyp}(m, C', L_1) = \bar{T}' \rightarrow T & \bullet; \{\bar{L}\}; \Gamma \vdash \bar{w} : \bar{S}' \quad \bar{S}' <: \bar{T}' \end{array}$$

for some $\bar{T}'$ and $\bar{S}'$. Let Λ be the layer set such that $L_1 \text{ req } \Lambda$. By Lemma A.1.16,

$$L_1.C'.m; \Lambda \cup \{L_1\}; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : S$$

and $S <: T$ for some S .

Since $L_0 <_w L_1$, L_0 requires all the layers that L_1 requires (including Λ). By $\{\bar{L}'\} \text{ wf}$ and Lemma A.1.11 and $L_0 \in \bar{L}'$, we have $\forall L \in \Lambda, \exists L' \in \{\bar{L}'\}$ such that $L' <_w L$. So, $\{\bar{L}'\} = \{\bar{L}'\} \cup \{L_0\} <_w \Lambda \cup \{L_1\}$. By this and $\{\bar{L}\} <_{sw} \{\bar{L}'\}$, we have $\{\bar{L}\} <_{sw} \Lambda \cup \{L_1\}$. By Lemma A.1.7,

$$L_1.C'.m; \{\bar{L}\}; \bar{x} : \bar{T}, \text{this} : C' \vdash e_0 : S.$$

By $ndp(m, C', (\bar{L}''; L_0), \bar{L}')$ (which follows from $C.m \vdash \langle C', (\bar{L}''; L_0), \bar{L}' \rangle \text{ ok}$) and the definition of ndp , $\text{proceed} \in e_0$ implies $ndp(m, C', \bar{L}'', \bar{L}')$ holds. Then, by Lemmas A.1.8, A.1.5, A.1.9 and A.1.17(1), $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S'$ for some $S' < S$. By S-TRANS, $S' < T$, finishing the case.

$$\begin{aligned} \text{Case R-INVKB: } e &= \text{new } C_0(\bar{v}) \langle C', \bar{L}'', \bar{L}' \rangle . m(\bar{w}) \\ mbody(m, C', \bar{L}'', \bar{L}') &= \bar{x}. e_0 \text{ in } C'', \bullet \\ C'' &\triangleleft D \\ e' &= \left[\begin{array}{ll} \text{new } C_0(\bar{v}) & / \text{this} \\ \bar{w} & / \bar{x} \\ \text{new } C_0(\bar{v}) \langle D, \bar{L}', \bar{L}' \rangle & / \text{super} \end{array} \right] e_0 \end{aligned}$$

By T-INVKA, it must be the case that

$$\begin{aligned} \bullet; \{\bar{L}\}; \Gamma \vdash \text{new } C_0(\bar{v}) : C_0 & \quad C_0.m \vdash \langle C', \bar{L}'', \bar{L}' \rangle \text{ ok} & \quad \{\bar{L}\} <_{sw} \{\bar{L}'\} \\ mtype(m, C', \{\bar{L}''\}, \{\bar{L}'\}) = \bar{T}' \rightarrow T & \quad \bullet; \{\bar{L}\}; \Gamma \vdash \bar{w} : \bar{S}' & \quad \bar{S}' < \bar{T}' \end{aligned}$$

for some $\bar{T}'$ and $\bar{S}'$. By Lemma A.1.18,

$$C''.m; \emptyset; \bar{x} : \bar{T}, \text{this} : C'' \vdash e_0 : S$$

and $C' < C''$ and $S < T$ and $ndp(m, C'', \bullet, \bar{L}')$ for some S . By S-TRANS, $C_0 < C''$. By Lemma A.1.5,

$$C''.m; \{\bar{L}\}; \bar{x} : \bar{T}, \text{this} : C'' \vdash e_0 : S$$

By Lemmas A.1.8, A.1.5, A.1.9 and A.1.17(2), $\bullet; \{\bar{L}\}; \Gamma \vdash e' : S'$ for some $S' < S$. By S-TRANS, $S' < T$, finishing the case.

$$\begin{aligned} \text{Case RC-WITH: } e &= \text{with new } L() \ e_0 & e' &= \text{with new } L() \ e_0' \\ with(L, \bar{L}) &= \bar{L}' & \bar{L}' \vdash e_0 &\longrightarrow e_0' \end{aligned}$$

By T-WITH, it must be the case that

$$\bullet; \{\bar{L}\}; \Gamma \vdash \text{new } L() : L \quad L \text{ req } \Lambda \quad \{\bar{L}\} <_w \Lambda \quad \bullet; \{\bar{L}\} \cup \{L\}; \Gamma \vdash e_0 : T$$

for some Λ . Here, $\{\bar{L}'\} = \{\bar{L}\} \cup \{L\}$ wf by WF-WITH. By the induction hypothesis, $\bullet; \{\bar{L}\} \cup \{L\}; \Gamma \vdash e_0' : S$ for some $S < T$. By T-WITH, $\bullet; \{\bar{L}\}; \Gamma \vdash e_0' : S$, finishing the case.

$$\text{Case RC-WITHARG: } e = \text{with } e_l \ e_0 \quad e' = \text{with } e_l' \ e_0 \quad \bar{L} \vdash e_l \longrightarrow e_l'$$

By T-WITH, it must be the case that

$$\bullet; \{\bar{L}\}; \Gamma \vdash e_l : L \quad L \text{ req } \Lambda \quad \{\bar{L}\} <_w \Lambda \quad \bullet; \{\bar{L}\} \cup \{L\}; \Gamma \vdash e_0 : T$$

for some Λ . By the induction hypothesis, we have $\bullet; \{\bar{L}\}; \Gamma \vdash e_l' : L'$ for some $L' < L$. By LS-EXTENDS, L' and L have the same require clause Λ . Since $L' < L$, we have $L' <_w L$, and $\{\bar{L}\} \cup \{L'\} <_w \{\bar{L}\} \cup \{L\}$. By Lemma A.1.7 and T-WITH, $\bullet; \{\bar{L}\}; \Gamma \vdash e' : T$. Reflexivity of $<$ finishes the case.

$$\text{Case R-WITHVAL: } e = \text{with new } L() \ v_0 \quad e' = v_0$$

By T-WITH, it must be the case that $\bullet; \{\bar{L}\} \cup \{L\}; \Gamma \vdash v_0 : T$. By Lemma A.1.8, $\bullet; \{\bar{L}\}; \Gamma \vdash v_0 : T$, finishing the case.

$$\begin{aligned} \text{Case RC-SWAP: } e &= \text{swap } (\text{new } L(), L_{sw}) \ e_0 & e' &= \text{swap } (\text{new } L(), L_{sw}) \ e_0' \\ swap(L, L_{sw}, \bar{L}) &= \bar{L}' & \bar{L}' \vdash e_0 &\longrightarrow e_0' \end{aligned}$$

By T-SWAP, it must be the case that

$$\begin{array}{lll} \bullet; \{\bar{L}\}; \Gamma \vdash \text{new } L() : L & L_{sw} \text{ swappable} & L <_w L_{sw} \quad L \text{ req } \Lambda \\ \Lambda_{rm} = \{\bar{L}\} \setminus \{L' \mid L' <_w L_{sw}\} & \Lambda_{rm} <_w \Lambda & \bullet; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T \end{array}$$

for some L , Λ , and Λ_{rm} . Here, $\{\bar{L}'\} = \Lambda_{rm} \cup \{L\}$. Then, $\{\bar{L}'\}$ wf by WF-SWAP. By the induction hypothesis, $\bullet; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0' : S$ for some $S < T$. By T-SWAP, $\bullet; \{\bar{L}\}; \Gamma \vdash e_0' : S$, finishing the case.

$$\text{Case RC-SWAPARG:} \quad e = \text{swap}(e_l, L_{sw}) \quad e_0 \quad e' = \text{swap}(e_l', L_{sw}) \quad e_0 \\ \bar{L} \vdash e_l \longrightarrow e_l'$$

By T-SWAP, it must be the case that

$$\begin{array}{lll} \bullet; \{\bar{L}\}; \Gamma \vdash e_l : L & L_{sw} \text{ swappable} & L <_w L_{sw} \quad L \text{ req } \Lambda \\ \Lambda_{rm} = \{\bar{L}\} \setminus \{L' \mid L' <_w L_{sw}\} & \Lambda_{rm} <_w \Lambda & \bullet; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T \end{array}$$

for some L , Λ , and Λ_{rm} . By the induction hypothesis, we have $\bullet; \{\bar{L}\}; \Gamma \vdash e_l' : L'$ for some $L' < L$. By LS-EXTENDS, L' and L have the same require clause Λ . Since $L' < L$, we have $L' <_w L$, $L' <_w L_{sw}$, and $\Lambda_{rm} \cup \{L'\} <_w \Lambda_{rm} \cup \{L\} <_w \Lambda$. By Lemma A.1.7 and T-SWAP, $\bullet; \{\bar{L}\}; \Gamma \vdash e' : T$. Reflexivity of $<$ finishes the case.

Case R-SWAPVAL:

Similar to Case R-WITHVAL.

$$\text{Case RC-INVKRECVAL:} \quad e = e_0.m(\bar{e}) \quad \bar{L} \vdash e_0 \longrightarrow e_0' \quad e' = e_0'.m(\bar{e})$$

By T-INVK, it must be the case that

$$\bullet; \{\bar{L}\}; \Gamma \vdash e_0 : C_0 \quad mtype(m, C_0, \{\bar{L}\}) = \bar{T} \rightarrow T \quad \bullet; \{\bar{L}\}; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} < \bar{T}.$$

for some $\bar{T}$ and $\bar{S}$. By the induction hypothesis, $\bullet; \{\bar{L}\}; \Gamma \vdash e_0' : D_0$ for some $D_0 < C_0$. By Lemma A.1.6, $mtype(m, D_0, \{\bar{L}\}) = \bar{T} \rightarrow S$ and $S < T$ for some S . By T-INVK, $\bullet; \{\bar{L}\}; \Gamma \vdash e_0'.m(\bar{e}) : S$, finishing the case.

$$\text{Case RC-INVKAARG:} \quad e = e_0.m(\dots, e_i, \dots) \quad \bar{L} \vdash e_i \longrightarrow e_i' \\ e' = e_0.m(\dots, e_i', \dots)$$

By T-INVK, it must be the case that

$$\bullet; \{\bar{L}\}; \Gamma \vdash e_0 : C_0 \quad mtype(m, C_0, \{\bar{L}\}) = \bar{T} \rightarrow T \quad \bullet; \{\bar{L}\}; \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} < \bar{T}.$$

for some $\bar{T}$ and $\bar{S}$. By the induction hypothesis, $\bullet; \{\bar{L}\}; \Gamma \vdash e_i' : S_i'$ for some $S_i' < S_i$. By S-TRANS, $S_i' < T_i$. So, by T-INVK, $\bullet; \{\bar{L}\}; \Gamma \vdash e' : T$, finishing the case.

Case RC-NEW, RC-INVKAARG1, RC-INVKAARG2:

Similar to the case above. □

Lemma A.1.19 (Lemma 2.3.15) *If $pmtree(m, C, L) = \bar{T} \rightarrow T_0$, then there exist $\bar{x}$ and e_0 and L' ($\neq \text{Base}$) such that $pmbdy(m, C, L) = \bar{x}.e_0$ in L' and the lengths of $\bar{x}$ and $\bar{T}$ are equal and $L <_w L'$.*

Proof: By induction on $pmtree(m, C, L) = \bar{T} \rightarrow T_0$.

$$\text{Case PMT-LAYER:} \quad LT(L)(C.m) = T_0 \quad C.m(\bar{T} \bar{x}) \{ \text{return } e; \}$$

By T-PMETHOD, the lengths of $\bar{T}$ and $\bar{x}$ are equal. $L <_w L$ by Reflexivity of $<_w$. Then, PMB-LAYER finishes the case.

Case PMT-SUPER: $LT(L)(C.m)$ undefined $L \triangleleft L'$ $pmtree(m, C, L') = \bar{T} \rightarrow T_0$

The induction hypothesis and PMB-LAYER and LSW-EXTENDS and LSW-EXTENDS finish the case. $\square$

Lemma A.1.20 (Lemma 2.3.16) *If $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ and $\bar{L}'$ is a prefix of $\bar{L}$ and $\{\bar{L}\}$ wf, then there exist $\bar{x}$ and e_0 and $\bar{L}''$ and $C' (\neq \text{Object})$ such that $mbody(m, C, \bar{L}, \bar{L}') = \bar{x}.e_0$ in $C', \bar{L}''$ and the lengths of $\bar{x}$ and $\bar{T}$ are equal and, if $\bar{L}''$ is not empty, the last layer name of $\bar{L}''$ is not Base.*

Proof : By lexicographic induction on $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ and the length of $\bar{L}'$.

Case: $\bar{L}' = \bullet$ $\text{class } C \triangleleft D \{ \dots S_0 \ m(\bar{S} \ \bar{x}) \{ \text{return } e_0; \} \dots \}$

By MT-CLASS, it must be the case that $\bar{T}, T_0 = \bar{S}, S_0$ and the lengths of $\bar{S}$ and $\bar{x}$ are equal. Then, by MB-CLASS, $mbody(m, C, \bullet, \bar{L}) = \bar{x}.e_0$ in $C, \bullet$, finishing the case.

Case: $\bar{L}' = \bullet$ $\text{class } C \triangleleft D \{ \dots \bar{M} \} \quad m \notin \bar{M}$

It must be the case that $mtype(m, C, \{\bar{L}'\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$ is derived by MT-SUPER and $mtype(m, D, \{\bar{L}\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$. The induction hypothesis and MB-SUPER finish the case.

Case: $\bar{L}' = \bar{L}''', L_0$ $pmtree(m, C, L_0) = \bar{T} \rightarrow T_0$

By Lemma A.1.19 and MB-LAYER.

Case: $\bar{L}' = \bar{L}'''; L_0$ $pmtree(m, C, L_0)$ undefined

Since $pmtree(m, C, L_0)$ undefined, it must be the case that $mtype(m, C, \{\bar{L}'''\}, \{\bar{L}\}) = \bar{T} \rightarrow T_0$. By the induction hypothesis, there exist $\bar{x}$ and e_0 and $\bar{L}''$ and $C' (\neq \text{Object})$ such that $mbody(m, C, \bar{L}''', \bar{L}') = \bar{x}.e_0$ in $C', \bar{L}''$ and the lengths of $\bar{x}$ and $\bar{T}$ are equal. It follows that $pmbody(m, C, L_0)$ is undefined from $pmtree(m, C, L_0)$ undefined. MB-NEXTLAYER finishes the case. $\square$

Theorem A.1.2 (Progress) *Suppose $\vdash (CT, LT)$ ok. If $\bullet; \{\bar{L}\}; \bullet \vdash e : T$ and $\{\bar{L}\}$ wf, then e is a value or $\bar{L} \vdash e \longrightarrow e'$ for some e' .*

Proof : By induction on $\bullet; \{\bar{L}\}; \bullet \vdash e : T$ with case analysis on the last typing rule used.

Case T-VAR, T-SUPER, T-PROCEED, T-SUPERPROCEED:

Cannot happen.

Case T-FIELD: $e = e_0.f_i$ $\bullet; \{\bar{L}\}; \bullet \vdash e_0 : C_0$ $fields(C_0) = \bar{T} \ \bar{f}$ $C = C_i$

By the induction hypothesis, either e_0 is a value or there exists e_0' such that $\bar{L} \vdash e_0 \longrightarrow e_0'$. In the latter case, RC-FIELD finishes the case. In the former case where e_0 is a value, by T-NEW, we have

$$e_0 = \text{new } C_0(\bar{v}) \quad \bullet; \{\bar{L}\}; \bullet \vdash \bar{v} : \bar{S} \quad \bar{S} \triangleleft \bar{T}.$$

So, we have $\bar{L} \vdash e \longrightarrow v_i$, finishing the case.

Case T-INVK: $e = e_0.m(\bar{e})$ $\bullet; \{\bar{L}\}; \bullet \vdash e_0 : C_0$
 $mtype(m, C_0, \{\bar{L}\}) = \bar{T} \rightarrow T$ $\bullet; \{\bar{L}\}; \bullet \vdash \bar{e} : \bar{S}$ $\bar{S} \triangleleft \bar{T}$

By the induction hypothesis, there exist $i \geq 0$ and e_i' such that $\bar{L} \vdash e_i \longrightarrow e_i'$, in which case RC-INVKRECV or RC-INVKARG finishes the case, or all e_i 's are values $v_0, \bar{v}$. Then, by T-NEW, $v_0 = \text{new } C_0(\bar{w})$ for some values $\bar{w}$. By Lemma A.1.20, there exist $\bar{x}, e_0', \bar{L}''$

and $C' (\neq \text{Object})$ such that $mbody(m, C_0, \bar{L}, \bar{L}) = \bar{x}.e_0$ in $C', \bar{L}''$ and the lengths of $\bar{x}$ and $\bar{T}$ are the same. Since $C' \neq \text{Object}$, there exists D' such that $\text{class } C' \triangleleft D' \{ \dots \}$. We have two subcases here depending on whether $\bar{L}''$ is empty or not. We will show the case where $\bar{L}''$ is not empty; the other case is similar. Let $\bar{L}'' = \bar{L}'''; L_0$ for some $\bar{L}'''$. Since $L_0 \neq \text{Base}$, there exists L_1 such that $\text{layer } L_0 \triangleleft L_1 \{ \dots \}$. Then, the expression

$$e' = \left[\begin{array}{l} \text{new } C_0(\bar{w}) \quad \quad \quad / \text{this} \\ \bar{v} \quad \quad \quad \quad \quad \quad \quad / \bar{x} \\ \text{new } C_0(\bar{w}) \langle C', \bar{L}''', \bar{L} \rangle . m / \text{proceed} \\ \text{new } C_0(\bar{w}) \langle D', \bar{L}, \bar{L} \rangle \quad \quad \quad / \text{super} \\ \text{new } C_0(\bar{w}) \langle C', L_1, \bar{L}, \bar{L} \rangle \quad / \text{superproceed} \end{array} \right] e_0'$$

is well defined (note that the lengths of $\bar{x}$ and $\bar{v}$ are equal). Then, by R-INVKP and R-INVK, $\bar{L} \vdash e \longrightarrow e'$.

Case T-NEW: $e = \text{new } C(\bar{e}) \quad fields(C) = \bar{T} \quad \bullet; \{\bar{L}\}; \bullet \vdash \bar{e} : \bar{S} \quad \bar{S} <: \bar{T}$

By the induction hypothesis, either (1) $\bar{e}$ are all values, in which case e is also a value; or (2) there exists i and e_i' such that $\bar{L} \vdash e_i \longrightarrow e_i'$, in which case RC-NEW finishes the case.

Case T-NEWL:

Trivial.

Case T-WITH: $e = \text{with } e_l \ e_0 \quad \bullet; \{\bar{L}\}; \bullet \vdash e_l : L \quad \bullet; \{\bar{L}\} \cup \{L\}; \bullet \vdash e_0 : T$
 $L \text{ req } \Lambda \quad \{\bar{L}\} <_w \Lambda$

By the induction hypothesis, either e_l is not a value, in which case RC-WITHARG finishes the case; or e_0 is a value, in which case R-WITHVAL finishes the case; or there exists e_0' such that $\text{with}(L, \bar{L}) \vdash e_0 \longrightarrow e_0'$, in which case RC-WITH finishes the case (notice that $\{\text{with}(L, \bar{L})\} wf$, by WF-WITH).

Case T-SWAP: $e = \text{swap } (e_l, L_{sw}) \ e_0 \quad \bullet; \{\bar{L}\}; \bullet \vdash e_l : L$
 $L_{sw} \text{ swappable} \quad L <_w L_{sw}$
 $L \text{ req } \Lambda' \quad \Lambda_{rm} = \{\bar{L}\} \setminus \{L' \mid L' <_w L_{sw}\}$
 $\Lambda_{rm} <_w \Lambda' \quad \mathcal{L}; \Lambda_{rm} \cup \{L\}; \Gamma \vdash e_0 : T_0$

By the induction hypothesis, either e_l is not a value, in which case RC-SWAPARG finishes the case; or e_0 is a value, in which case R-SWAPVAL finishes the case; or there exists e_0' such that $\text{swap}(L, L_{sw}, \bar{L}) \vdash e_0 \longrightarrow e_0'$, in which case RC-SWAP finishes the case (notice that, by WF-SWAP, $\{\text{swap}(L, L_{sw}, \bar{L})\} wf$).

Case T-INVKA:

Similar to the case for T-INVK.

Case T-INVKAL: $e = \text{new } C_0(\bar{v}) \langle D_0, L_1, (\bar{L}''; L_0), \bar{L}' \rangle . m(\bar{e})$
 $\bullet; \{\bar{L}\}; \bullet \vdash \text{new } C_0(\bar{v}) : C_0 \quad C_0.m \vdash \langle D_0, L_1, (\bar{L}''; L_0), \bar{L}' \rangle \text{ ok}$
 $\{\bar{L}\} <_{sw} \{\bar{L}'\} \quad L_0 <_w L_1$
 $pmtyp(m, D_0, L_1) = \bar{T}' \rightarrow T_0$
 $\bullet; \{\bar{L}\}; \bullet \vdash \bar{e} : \bar{S}' \quad \bar{S}' <: \bar{T}'$

By the induction hypothesis, either (1) there exists $i \geq 1$ and e_i' such that $\bar{L} \vdash e_i \longrightarrow e_i'$, in which case RC-INVKARG finishes the case, or (2) all e_i 's are values $\bar{w}$. Then, by Lemma A.1.19, there exist $\bar{x}$, e_0' and $L_2 (\neq \text{Base})$ such that $pmbdy(m, D_0, L_1) = \bar{x}.e_0$ in L_2 and the lengths of $\bar{x}$ and $\bar{T}'$ are the same. Since $L_2' \neq \text{Base}$, there exists L_3 such that $\text{layer } L_2 \triangleleft L_3 \{ \dots \}$.

By Sanity Condition (8), D_0 is not Object and there exists E_0 such that $\text{class } D_0 \triangleleft E_0 \{ \dots \}$. Then, the expression

$$e' = \left[\begin{array}{ll} \text{new } C_0(\bar{v}) & / \text{this} \\ \bar{w} & / \bar{x} \\ \text{new } C_0(\bar{v}) \langle D_0, \bar{L}'', \bar{L} \rangle . m & / \text{proceed} \\ \text{new } C_0(\bar{v}) \langle E_0, \bar{L}, \bar{L} \rangle & / \text{super} \\ \text{new } C_0(\bar{v}) \langle D_0, L_3, (\bar{L}''; L_0), \bar{L} \rangle . m / \text{superproceed} & \end{array} \right] e_0'$$

is well defined (note that the lengths of $\bar{x}$ and $\bar{v}$ are equal). Then, by R-INVKSP, $\bar{L} \vdash e \longrightarrow e'$.

□

Theorem A.1.3 (Type Soundness) *If $\vdash (CT, LT, e) : T$ and e reduces to a normal form under the empty set of layers, then the normal form is $\text{new } S(\bar{v})$ for some $\bar{v}$ and S such that $S <: T$.*

Proof : By T-PROG and Theorems A.1.1 and A.1.2.

□

Appendix B

Proof of Properties of Core ContextWorkflow

B.1 Proofs

In the following theorems, let $p_k = A_k/C_k$ for some k , and we define functions as follow.

- $b(t)$ be a workflow that removes `sub`, `check`, `cp` and `cp#E` from t .
- $includes(t, m, n)$ iff $b(t) = p_m; \dots; p_n$ if $m \leq n$; or t has no primitive workflows otherwise.
- $includes(E, m, n) = includes(E[check], m, n)$.
- $includes(\bar{c}, n, m)$ iff $\bar{c} \setminus \{\text{sub}, \text{ccp}\#E\} = C_n; \dots; C_m$ if $m \leq n$; or $\bar{c}$ has no atomic actions C_* otherwise.
- $nosub(t, m, n)$ iff $includes(t, m, n)$ and t has no `sub`-workflow.

Lemma 1 (Commit) *If $includes(t, m, n)$ and $\langle t, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}' \rangle$, then $\vec{A} = A_m, \dots, A_n$ (when $m \leq n$) or $\vec{A} = \epsilon$ (otherwise).*

Proof : By straightforward induction on the derivation. □

Lemma 2 (Abort) *If $nosub(t, m, n)$ and $\langle t, E, \bar{c} \rangle \Uparrow_{A|P}^{\vec{A}} \langle \bar{c}', [] \rangle$, then $\vec{A} = A_m, \dots, A_i$ and $includes(\bar{c}', i, m)$ for some i such that $m \leq i \leq n$ (when $m \leq n$), or $\vec{A} = \epsilon \wedge includes(\bar{c}', 0, 1)$ (otherwise).*

Proof : By straightforward induction on the derivation. □

Lemma 3 (Compensation) *If $\bar{c} = C_m, \dots, C_n$ and $\langle \bar{c} \rangle \Downarrow^{\vec{c}} \langle \bullet, [] \rangle$, then $\vec{c} = C_m, \dots, C_n$.*

Proof : By straightforward induction on the derivation. □

Lemma 4 (Checkpoint) *Suppose $nosub(t, m, k)$ and t has no `cp#Es` and $\langle t, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}', [] \rangle$ and $includes(E, k+1, n)$ and $includes(\bar{c}, m-1, 1)$ and $1 \leq m$ and $\text{ccp}\#E_s \notin \bar{c}$ and $\text{ccp}\#E_s \in \bar{c}'$ and is just after C_j (or just before C_{j+1} , so $\bar{c}'$ usually be $C_k, \dots, C_{j+1}, \dots, \text{ccp}\#E_s, \dots, C_j, \dots, C_m$) and $m-1 \leq j \leq k$.*

1. If $m-1 \leq k \leq n \wedge m \leq n$, then $includes(E_s, j+1, n)$.
2. If $n \leq k < m$, then $includes(E_s, m, k)$.

Proof : Proof by induction on the derivation of $\langle t, E, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \bar{c}', [] \rangle$. We show only the case of (1), and main rules.

Case CW-CHECKPOINT: $E_s = E[[]; \text{cp}\#E]$ $j = m-1$

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

Case CW-SEQ: $t = t_1; t_2$ $\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}'' \rangle$
 $\langle t_2, E, \bar{c}'' \rangle \Downarrow^{\vec{A}_2} \langle \bar{c}' \rangle$

We get $\text{includes}(t_1, m, i)$ and $\text{includes}(t_2, i+1, k)$ for some i s.t., $m-1 \leq i \leq k$. The induction hypothesis finishes the case. $\square$

Lemma 5 (Partial Abort) Suppose $\text{nosub}(t, m, n_0)$ and t has no $\text{cp}\#E_*$ and $\langle t, [], \bullet \rangle \Uparrow_P^{\vec{A}} \langle \bar{c}', [] \rangle$ and $\vec{A} = A_m, \dots, A_n$ and $\text{includes}(\bar{c}', n, m)$ and $\langle \bar{c}' \rangle \Downarrow^{\vec{C}} \langle \bar{c}'', E_s \rangle$.

- If $m \leq n$, then $\vec{C} = \epsilon$ and $\text{includes}(E_s, m, n)$ and $\text{includes}(\bar{c}'', n, m)$, or $\vec{C} = C_n, \dots, C_{k+1}$ and $\text{includes}(E_s, k+1, n)$ and $\text{includes}(\bar{c}'', k, m)$ for some k ($m-1 \leq k < n$).
- If $m > n$, then $\vec{C} = \epsilon$ and $\text{includes}(E_s, m, n)$ and $\text{includes}(\bar{c}'', n, m)$.

Proof : Proof by induction on the derivation of $\langle \bar{c} \rangle \Downarrow^{\vec{C}} \langle \bar{c}', E_s \rangle$, using Lemma 4. $\square$

Lemma 6 (Suspend) Suppose $\text{includes}(t, m, k)$ and $\langle t, E, \bar{c} \rangle \Uparrow_S^{\vec{A}} \langle \bar{c}', E_s \rangle$ and $\text{includes}(E, k+1, n)$.

1. If $m-1 \leq k \leq n \wedge m \leq n$, then $\vec{A} = A_m, \dots, A_i$ for some i such that $m \leq i \leq k$ and $\text{includes}(E_s, i+1, n)$, or $\vec{A} = \epsilon$ and $\text{includes}(E_s, m, n)$.
2. If $n \leq k < m$, then $\text{includes}(E_s, m, k)$.

Proof : Proof by induction on the derivation. We show only the case of (1), and main rules.

Case CW-CHECK-SUSPEND:

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

Case CW-SUB-INT: $t = \text{sub}(t')/c$

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

Case CW-SEQ-INT1: $t = t_1; t_2$

We get $\text{includes}(t_1, m, j)$ for some j s.t., $m-1 \leq j \leq k$. We also get $\text{includes}(E[[]; t_2], j+1, n)$. Then, the induction hypothesis finishes the case.

Case CW-SEQ-INT2: $t = t_1; t_2$ $\langle t_1, E[[]; t_2], \bar{c} \rangle \Downarrow^{\vec{A}_1} \langle \bar{c}'' \rangle$
 $\langle t_2, E, \bar{c}'' \rangle \Uparrow_S^{\vec{A}_2} \langle \bar{c}', E_s \rangle$

We get $\text{includes}(t_1, m, j)$ for some j s.t., $m-1 \leq j \leq k$. By Lemma 1, $\vec{A}_1 = A_m, \dots, A_{j-1}$ (when $m \leq j$), or $\vec{A}_1 = \epsilon$ (when $j = m-1$). We also get $\text{includes}(t_2, j+1, k)$ from $\text{includes}(t, m, k)$ and $\text{includes}(t_1, m, j)$. We still have $\text{includes}(E, k, n)$.

Then, by the induction hypothesis, $\vec{A}_2 = A_j, \dots, A_i$ for some i such that $j \leq i \leq k$ and $\text{includes}(E_s, i+1, n)$, or $\vec{A}_2 = \epsilon$ and $\text{includes}(E_s, m, n)$.

Finally, we can finish the case concatenating $\vec{A}_1$ and $\vec{A}_2$. $\square$

Theorem 1 (Workflow commits) If $\text{includes}(t, m, n)$ and $\langle t, \bar{c} \rangle \Downarrow^{\vec{A}} \langle \rangle$ and $m \leq n$, then $\vec{A} = A_m, \dots, A_n$.

Proof : By Lemma 1 and CW-PROGRAM-COMMIT. $\square$

Theorem 2 (Workflow aborts (Successful Compensation)) If $\text{nosub}(\mathbf{t}, \mathbf{m}, \mathbf{n})$ and $\langle \mathbf{t}, \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{A}}}^{\bar{\mathbf{A}}}$ and $\mathbf{m} \leq \mathbf{n}$ and $\bar{\mathbf{c}} = \mathbf{C}_k, \dots, \mathbf{C}_l$, then $\bar{\mathbf{A}} = \mathbf{A}_m, \dots, \mathbf{A}_i, \mathbf{C}_i, \dots, \mathbf{C}_m, \mathbf{C}_k, \dots, \mathbf{C}_l$ for some i ($\mathbf{m} \leq i \leq \mathbf{n}$).

Proof : By Lemmas 2 and 3 and CW-PROGRAM-ABORT. $\square$

Theorem 3 (Restarted suspended workflow commits) If $\langle \mathbf{t}, \bullet \rangle \Downarrow_{\bar{\mathbf{A}}}^{\bar{\mathbf{A}}} \langle \rangle$ and $\langle \mathbf{t}, \bullet \rangle \Downarrow_{\bar{\mathbf{S}}}^{\bar{\mathbf{S}}} \langle \bar{\mathbf{c}}, \mathbf{E} \rangle$ and $\langle \mathbf{E}[\text{check}], \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{C}}}^{\bar{\mathbf{C}}} \langle \rangle$, then $\bar{\mathbf{A}} = \bar{\mathbf{C}}, \bar{\mathbf{C}}^{\bar{\mathbf{C}}}$.

Proof : By Theorem 1, Lemma 6 and CW-PROGRAM-SUSPEND. $\square$

Theorem 4 (Workflow partially aborts) If $\text{nosub}(\mathbf{t}, \mathbf{m}, \mathbf{n})$ and $\langle \mathbf{t}, \bullet \rangle \Downarrow_{\bar{\mathbf{P}}}^{\bar{\mathbf{A}}} \langle \bar{\mathbf{c}}, \mathbf{E} \rangle$ and $\mathbf{m} \leq \mathbf{n}$, then either of the followings hold.

- $\bar{\mathbf{A}} = \mathbf{A}_m, \dots, \mathbf{A}_i, \mathbf{C}_i, \mathbf{C}_{i-1}, \dots, \mathbf{C}_j$ and $\text{includes}(\mathbf{E}, \mathbf{j}, \mathbf{n})$ and $\text{includes}(\bar{\mathbf{c}}, \mathbf{j}-1, \mathbf{m})$ for some i and j ($\mathbf{m} \leq j \leq i \leq \mathbf{n}$).
- $\bar{\mathbf{A}} = \mathbf{A}_m, \dots, \mathbf{A}_n$ and $\text{includes}(\mathbf{E}, 1, \mathbf{0})$ and $\text{includes}(\bar{\mathbf{c}}, \mathbf{n}, \mathbf{m})$.

Moreover, the followings hold.

1. (Suspended workflow commits) If $\langle \mathbf{E}[\text{check}], \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{A}}}^{\bar{\mathbf{A}}} \langle \rangle$, then $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \mathbf{A}_j, \dots, \mathbf{A}_n$, or $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \epsilon$ (if $\text{includes}(\mathbf{E}, 1, \mathbf{0})$).
2. (Suspended workflow aborts) If $\langle \mathbf{E}[\text{check}], \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{A}}}^{\bar{\mathbf{A}}} \langle \rangle$, then $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \epsilon$ (if $j = \mathbf{m}$), or $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \mathbf{C}_{j-1}, \dots, \mathbf{C}_m$.

Proof : By Lemma 2, Lemma 5 and CW-PROGRAM-PABORT.

1. By Theorem 1.
2. By Theorem 2.

$\square$

Theorem 5 (Partial abort, checkpoint and nested workflow) Suppose that $\text{includes}(\mathbf{t}, 1, \mathbf{n})$ and $\mathbf{t} \setminus \text{check} = \mathbf{p}_1; \dots; \mathbf{cp}; \mathbf{p}_k; \dots; \mathbf{p}_m; \text{sub}(\mathbf{p}_{m+1}; \dots; \mathbf{cp}; \mathbf{p}_j; \dots; \mathbf{p}_l) / \mathbf{C}_a; \mathbf{p}_{l+1}; \dots; \mathbf{p}_n$ and $\langle \mathbf{t}, \bullet \rangle \Downarrow_{\bar{\mathbf{P}}}^{\bar{\mathbf{A}}} \langle \bar{\mathbf{c}}, \mathbf{E} \rangle$.

1. (Partial abort skips compensations of complete sub-workflow) If $\mathbf{A}_{l+1} \in \{\bar{\mathbf{A}}\}$, then $\bar{\mathbf{A}} = \mathbf{A}_1, \dots, \mathbf{A}_i, \mathbf{C}_i, \dots, \mathbf{C}_{l+1}, \mathbf{C}_a, \mathbf{C}_m, \dots, \mathbf{C}_k, \dots, \mathbf{C}_1$ for some $i (> 1)$.
2. (A suspended workflow remembers checkpoints in a sub-workflow) If $\mathbf{A}_{l+1} \in \{\bar{\mathbf{A}}\}$ and $\langle \mathbf{E}[\text{check}], \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{P}}}^{\bar{\mathbf{A}}} \langle \bar{\mathbf{c}}'', \mathbf{E}' \rangle$ and $\mathbf{A}_j \in \{\bar{\mathbf{A}}^{\bar{\mathbf{A}}}\} \wedge \mathbf{A}_l \notin \{\bar{\mathbf{A}}^{\bar{\mathbf{A}}}\}$, then $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \mathbf{A}_k, \dots, \mathbf{A}_i, \mathbf{C}_i, \dots, \mathbf{C}_j$ for some i ($j \leq i \leq l$).
3. (A suspended workflow remembers checkpoints before a sub-workflow) If $\mathbf{C}_j \in \{\bar{\mathbf{A}}\}$ and $\langle \mathbf{E}[\text{check}], \bar{\mathbf{c}} \rangle \Downarrow_{\bar{\mathbf{P}}}^{\bar{\mathbf{A}}} \langle \bar{\mathbf{c}}'', \mathbf{E}' \rangle$ and $\mathbf{A}_{l+1} \in \{\bar{\mathbf{A}}^{\bar{\mathbf{A}}}\}$, then $\bar{\mathbf{A}}^{\bar{\mathbf{A}}} = \mathbf{A}_j, \dots, \mathbf{A}_i, \mathbf{C}_i, \dots, \mathbf{C}_{l+1}, \mathbf{C}_a, \mathbf{C}_m, \dots, \mathbf{C}_k$ for some $i > 1$.

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

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 C_a .
2. We can get $E = E_0$ from the derivation tree. Then, straightforwardly from the derivation of $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}'', E' \rangle$ using Lemma 1 and Lemma 2.
3. We can get $E = E_1$ from the derivation tree. Then, straightforwardly from the derivation of $\langle E[\text{check}], \bar{c} \rangle \Downarrow_P^{\vec{A}} \langle \bar{c}'', E' \rangle$ using Lemma 1 and Lemma 2.

□