2016

Formal foundations for hybrid effect analysis

Yuheng Long
Iowa State University

Formal foundations for hybrid effect analysis

by

Yuheng Long

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Hridesh Rajan, Major Professor

Samik Basu

Steven M. Kautz

Andrew S. Miner

Gurpur M. Prabhu

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGEMENTS ix

ABSTRACT xi

CHAPTER 1. INTRODUCTION 1

1.1 Static Approach for Type Reasoning 1

1.2 This Thesis 2

1.3 Outline 3

CHAPTER 2. RELATED WORK 4

2.1 Static Effect Systems 4

2.2 Dynamic Effect Inspection 5

2.3 Gradual Effect 6

2.4 Dynamic Effect Analyses 7

CHAPTER 3. INTENSIONAL EFFECT POLYMORPHISM 8

3.1 Motivating Examples 11

3.1.1 Safe Parallelism 11

3.1.2 Information Security 12

3.1.3 Consistent Graphical User Interface (GUI) Access 13

3.1.4 Program Optimization – Memoization 14

3.2 λ_{ie} Abstract Syntax 15

3.3 The Type System 18

3.3.1 Definitions . . . 18
3.3.2 Subsumption and Entailment . . . 20
3.3.3 Typing Judgment Overview . . . 23
3.3.4 Static Typing for Dynamic Intensional Analysis . . . 23
3.4 Dynamic Semantics . . . 28
3.5 Meta-Theories . . . 32
3.5.1 Type Soundness . . . 33
3.5.2 Soundness of Intensional Effect Polymorphism . . . 33
3.5.3 Differential Alignment Optimization . . . 35
3.6 Related Work . . . 37
3.7 Summary . . . 38
CHAPTER 4. FIRST-CLASS EFFECTS REFLECTION . . . 39
4.1 Motivating Applications . . . 42
4.1.1 Custom Effect-Aware Schedulers . . . 42
4.1.2 Version-Consistent Dynamic Software Update . . . 46
4.1.3 Data Zeroing . . . 48
4.1.4 Monotonicity and Polarity . . . 50
4.2 λ_{fc}: a Calculus with First-Class Effects . . . 52
4.3 A Base Type System with Double-Bounded Effects . . . 53
4.3.1 Subtyping . . . 53
4.3.2 Type Checking . . . 54
4.4 The Full-Fledged System . . . 56
4.4.1 Polarity Support . . . 57
4.4.2 Refinement Type Checking . . . 59
4.5 Dynamic Semantics . . . 62
4.6 Meta-theory . . . 65
4.6.1 Type Soundness . . . 65
4.6.2 Query-Realize Correspondence . . . 66
4.6.3 Trace Consistency . . . 67

4.7 Related Work . . . 68
4.8 Summary . . . 70
CHAPTER 5. AN EFFECT SYSTEM FOR ASYNCHRONOUS, TYPED EVENTS . . . 71
5.1 Asynchronous, Typed Events . . . 71
5.2 Technical Highlights . . . 73
5.3 A Calculus with Asynchronous Typed Events . . . 76
5.3.1 Expressions . . . 76
5.3.2 Declarations . . . 78
5.4 Type and Static Effect Computation . . . 78
5.4.1 Effects Reasoning for Mutable Handler Queue . . . 78
5.4.2 Type and Effect Attributes, and Effect Interference . . . 78
5.4.3 Expressions . . . 79
5.4.4 Top-Level Declarations . . . 81
5.5 Semantics with Effect-Guided Scheduling . . . 81
5.5.1 Domains . . . 83
5.5.2 Registration-Time Specialization & Dynamic Typing . . . 83
5.5.3 Event Announcement & Safe Implicit Concurrency . . . 85
5.5.4 Yielding Control & Interference Points . . . 86
5.6 Meta-Theories . . . 86
5.6.1 Preliminary Definitions . . . 87
5.6.2 Livelock Freedom . . . 87
5.6.3 Type Soundness . . . 88
5.6.4 Sequential Semantics . . . 89
5.6.5 Modular Reasoning . . . 91
5.7 Related Work . . . 92
5.8 Summary . . . 93
CHAPTER 6. CONCLUSION . . . 94

CHAPTER 7. FUTURE WORK 95

7.1 Empirical Evaluation on the Impact of Hybrid Effect Analysis 95

7.2 Exploratory Study of the Design Impact of Asynchronous, Typed Events 95

7.3 Effect Analysis on Mutable Data Structure 96

BIBLIOGRAPHY 97

LIST OF TABLES

4.1 An Example λ_{fc} Client Domain: The Menagerie of Scheduling Strategies . . 40

4.2 A Summary of λ_{fc} Features 42

4.3 Representative Information Security Policies in First-Class Effects. 50

4.4 Effect Operators and their Corresponding Polarities. 51

4.5 Polarities for Client Predicates ($\mathbb{V}(\mathbb{R}, j)$). 58

LIST OF FIGURES

3.1 Example illustrating λ_{ie} and its usage for safe parallelism. 12

3.2 An application of λ_{ie} in preventing security vulnerabilities. 12

3.3 Example showing how UI effect discipline can be enforced by λ_{ie}. 13

3.4 A proof-of-concept memoization technique. 14

3.5 λ_{ie} Abstract Syntax (Throughout the thesis, notation $\overline{\bullet}$ represents a set of $\bullet$ elements, and notation $\overrightarrow{\bullet}$ represents a sequence of $\bullet$ elements.) 16

3.6 Client implementation of $\mathbb{R}$ and $\mathbb{SAFE}$ `e e`. 17

3.7 λ_{ie} Type System Definitions 19

3.8 λ_{ie} Subsumption and Entailment. 21

3.9 λ_{ie} Typing Rules 22

3.10 Client Implementation of Predicate *clientT* 23

3.11 Definitions for $\forall$ and $\exists$ Introduction and Elimination 24

3.12 λ_{ie} Operational Semantics 29

3.13 Optimized λ_{ie} with Differential Alignment 31

4.1 A Producer-First Scheduler (The new notations introduced by first-class effects are explained on the table below the listing.) 43

4.2 Dynamic Software Updating in First-Class Effects to Preserve Consistency [77]. 47

4.3 Data Zeroing in First-Class Effects Against Leakage of Sensitive Data. 49

4.4 λ_{fc} Abstract Syntax. 51

4.5 The Subtyping Relation. 54

4.6 Typing Rules. 54

4.7 Typing Rules for Standard Expressions. 56

4.8 λ_{fc} Extension with Refinement Types. 57

4.9 Polarity Lattice. 58

4.10 Predicate Implication $|->$. 59

4.11 Typing Rules for Checking Refinement Types. 60

4.12 Functions for Computing Effect Polarity. 61

4.13 λ_{fc} Operational Semantics. 63

5.1 ATE's Abstract Syntax. 77

5.2 Type-and-effect Attributes. 79

5.3 Effect Noninterference. 79

5.4 Type-and-effect Rules. 80

5.5 Type-and-effect Rules for Top Level Declarations. 81

5.6 Semantics Domains and Dynamic Typing. 82

5.7 Operational Semantics. Auxiliary functions are defined in Figure 5.8. 84

5.8 Auxiliary Functions for the Semantics. 86

5.9 Sequential Semantics. 88

5.10 Cooperative Semantics. 90

5.11 Trace Projection. 91

ACKNOWLEDGEMENTS

Throughout my graduate study at Iowa State University, Dr Hridesh Rajan was my adviser and guided me with incredible patient. I have been admiring him for his high level ideas for formalizing problems. His ideas are so inspiring that I would work on them right away when these ideas are given to me and both of us are more than willing to work 24 by 7. I am also encouraged to improve my problem solving skill when we developed several compilers for our ideas together. I will never forget the enormous performance gain we obtain after I followed his suggestion on redesigning a small piece of computer program. Besides, his perseverance will never fail to motivate me to refine our ideas. Despite it is not uncommon that our ideas were rejected by top conferences a couple of times, we never give up. We put more deep thinking into our drafts and we have faith that our idea will be welcomed by the world one day. Last but not least, I can always learn a lot from his writing and presentation skill. Our papers are always in good shape after they have been polished over and over again.

I may have only known Dr Yu David Liu for a very short period of time, but he has become like my second advisor. His insightful ideas on type systems design have been a constant inspiration for me. Very quite often, I can learn something news from what I have already known and change the perspective how I view the problem we are approaching. He has great knowledge in many aspects of programming language, software engineering and system design. Our paper will not be accepted without his deep insight into the low level details of our systems. Also, I appreciate his writing. I have been enjoying my time reading the part of our paper written by him several times. He makes me believe that writing is an art, more than just science.

This dissertation would not have been possible without the hard work and dedication of my colleagues Dr Tyler Sondag and Sean Mooney. Dr Sondag is one of the smartest people I know. He has been helping with the writing of several papers. His suggestions on the presentation and insights into our paper are one of the key reasons our paper was so successful. Sean is a very hard working individual. He has been helping with the development of several compilers, which allows us to measure the

performance of our ideas. Several of our paper would not have been possible without his timely action on the development of our ideas.

Over the past several years, Mehdi Bagherzadeh is the closest colleague I have been working with. He sat right next to me and we have co-authored some of the work this dissertation builds on. His enthusiasm for research is unparalleled. Thank to his encouragement to think deeper, we always find something interesting from the seemingly boring materials. His insights and suggestions directly impacted the success of several projects and also this thesis. I am sincerely thankful for the opportunity to work with him.

I would like to thank my colleagues Dr Steve Kautz, Dr Robert Dyer, Ganesha Upadhyaya, Eric Lin, Youssef Hanna, David Johnston, Dr Hoan Nguyen and Cody Hanika in the Laboratory for Software design. They have read and have provided wonderful feedback and discussion on my ideas over the years, which have shaped our papers nicely. I appreciate their tremendous amount of moral support.

I would like to thank my colleagues Nathan Harmata, Ahmad Sharif, Dr Long Fei, Luis Lozano, Caroline Tice, Dr Gary T. Leavens, Adam Zimmerman, John L. Singleton, Dr Karthik Pattabiraman, Laurel Tweed, Dr Tien N. Nguyen, Dr John Tang Boyland and Rex D. Fernando for providing me with insight feedback.

I would also like to thank my committee members Dr. Samik Basu, Dr. Andrew S. Miner, and Dr. Gurpur M. Prabhu. They have read and have provided valuable and insightful feedback on my proposal, which help significantly improve the quality of our papers.

My research described in this dissertation was supported in part by grants from the US National Science Foundation (NSF) under grants CCF-14-23370, CCF-13-49153, CCF-11-17937, CCF-10-17334, CNS-07-09217, CNS-06-27354, and CAREER awards CCF-08-46059. These grants have ensured that I have a wonderful environment to work in.

The majority of content of this dissertation can be found in prior publications. Chapter 3 is based on our ECOOP publication introducing Intensional Effect Polymorphism [68]. Chapter 4 is based on our technical report of first class effects [70], which is also currently under submission. Chapters 3 are based on our GPCE and MODULARITY publications describing our study on applying hybrid effect analysis to event-driven systems to obtain safe concurrency [69, 71].

ABSTRACT

Type-and-effect systems are a powerful tool for program construction and verification. Type-and-effect systems are useful because it help reduce bugs in computer programs, enable compiler optimizations and provide program documentation. As software systems increasingly embrace dynamic features and complex modes of compilation, static effect systems have to reconcile over competing goals such as precision, soundness, modularity, and programmer productivity. In this thesis, we propose the idea of combining static and dynamic analysis for effect systems to improve precision and flexibility.

We describe *intensional effect polymorphism*, a new foundation for effect systems that integrates static and dynamic effect checking. Our system allows the effect of polymorphic code to be intensionally inspected. It supports a highly precise notion of effect polymorphism through a lightweight notion of dynamic typing. When coupled with parametric polymorphism, the powerful system utilizes runtime information to enable precise effect reasoning, while at the same time retains strong type safety guarantees. The technical innovations of our design include a relational notion of effect checking, the use of bounded existential types to capture the subtle interactions between static typing and dynamic typing, and a differential alignment strategy to achieve efficiency in dynamic typing.

We introduce the idea of *first-class effects*, where the computational effect of an expression can be programmatically reflected, passed around as values, and analyzed at run time. A broad range of designs “hard-coded" in existing effect-guided analyses can be supported through intuitive programming abstractions. The core technical development is a type system with a couple of features. Our type system provides static guarantees to application-specific effect management properties through refinement types, promoting “correct-by-design" effect-guided programming. Also, our type system computes not only the over-approximation of effects, but also their under-approximation. The duality unifies the common theme of permission *vs.* obligation in effect reasoning.

Finally, we show the potential benefit of intensional effects by applying it to an event-driven system to obtain safe concurrency. The technical innovations of our system include a novel effect system to

soundly approximate the dynamism introduced by runtime handlers registration, a static analysis to precompute the effects and a dynamic analysis that uses the precomputed effects to improve concurrency. Our design simplifies modular concurrency reasoning and avoids concurrency hazards.

CHAPTER 1. INTRODUCTION

Type-and-effect systems — effect systems, for short — have broad applications [2, 12, 65, 77]. They were originally developed to reason about safe concurrency [72, 113], but have also been shown to help programmers in analyzing locking disciplines [2], dynamic updating mechanisms [77], checked exceptions [12, 65] and detecting race conditions [19].

In a type-and-effect system, the type information of expression $\mathtt{e}$ encodes and approximates the computational effects σ of $\mathtt{e}$. These effects describe how the state of a program will be modified by expressions in the language; for example, a field expression may have a read or write effect to represent reading from or writing into memory [72, 113].

1.1 Static Approach for Type Reasoning

Traditionally, type-and-effect systems are an augmentation of the static type systems [73] and can be viewed as behavioural type systems. Here the type system describes *what* is computed, while the effect system describes *how* the values are computed. Improving the expressiveness and precision of type-and-effect systems through static approaches is a thoroughly explored topic. Examples of well-known hurdles along the path include recursion, flow ordering, branching, higher-order functions in functional languages, and dynamic dispatch in OO languages. Established type system ideas can help express refined effect abstractions, through, for example, polymorphic types [75], flow-sensitive types [49], typestates [109] and conditional types [5]. In the context of program analysis, classic techniques can improve the precision of effect reasoning, through polymorphic type inference, polymorphic recursion [58], nCFA [105], CPA [4], context-sensitive [118], flow-sensitive [26], and path-sensitive [36] analyses, to name a few. Purely static type-and-effect systems are a worthy direction, but looking forward, we believe they are unlikely to sustain future programming practices, for a number of reasons.

First, traditional limitations of static type systems are often amplified in the context of effect reasoning. As an example, consider recursion: monomorphic treatment of recursive calls is often considered "good enough" in real-world practices of polymorphic type inference, evidenced by early versions of most functional languages. The same simplification for effect reasoning, however, would imply that all recursive calls yield monomorphic effects, a much more unrealistic assumption. Second, emerging software systems increasingly rely on dynamic language features that defy static reasoning. Reflections and native code interactions routinely appear in Java applications. Scripting languages, such as JavaScript [27, 28, 29], with flexible meta-programming are on the rise. In the big data era, data and code are often mingled, and the non-determinism introduced by I/O cannot be ignored. Static approaches can be helpful in some of these scenarios (*e.g.*, [25, 114]), but there remains a significant gap between these piecemeal solutions and a practical effect reasoning system that can work with all emerging dynamic features. Third, in an open programming world with third-party libraries, multi-party collaborations, and complex modes of compilation and linking, static effect type systems either require unrealistically verbose type declarations, or require global program analysis, a nemesis for modularity [89, 91].

Purely static effect systems are a worthy direction, but looking forward, we believe that a complementary foundation is also warranted, where the default is a system that can fully account for and exploit runtime information, aided by static approaches for optimization.

1.2 This Thesis

In this thesis, we develop *intensional effect polymorphism*, a system that integrates static and dynamic effect reasoning. The system relies on dynamic typing to compensate for the conservativeness of traditional static approaches and account for emerging dynamic features, while at the same time harvesting the power of static typing to vouch-safe for programs whose type safety is fundamentally dependent on runtime decision making.

We extend the system by describing a novel type-and-effect system where effects of program expressions are available as first-class values to programmers. The resulting calculus has the ability of querying the effect of any program expression, passing it across the modular boundary, storing it in mutable references, and inspecting its structure at runtime to perform expressive analyses. Now, effect-

guided decisions can be made as part of the program itself. The direct benefit of our system is its expressiveness in effect-guided programming. As we shall see, a variety of meta-level designs currently "hidden" behind the compiler and language runtime are now in the hands of programmers.

We showcase the usefulness of our system by applying intensional effect analysis to an event-driven system and show the concurrency benefits obtained by using our system.

1.3 Outline

In the next chapter, we discuss works related to type-and-effect systems. In chapter 3, we detail the theory behind intensional effect polymorphism. In chapter 4, we extend the idea and introduce first-class effects, which treats effects as first class values in the language. In chapter 5, we evaluate intensional effects and provide a case study on an event-driven system to obtain safe implicit concurrency. In chapter 6, we summarize the contributions of the thesis. In chapter 7, we describe our ongoing works and venues for future work.

CHAPTER 2. RELATED WORK

In this chapter, we discuss works related to intensional effects polymorphism. First we discuss purely static effect systems. Then we discuss other effect systems that rely on dynamic effect inspection. Then we discuss other works on hybridizing static and dynamic analyses. We also discuss related approaches utilizing purely dynamic effects analyses.

2.1 Static Effect Systems

Traditionally, effects consider memory reads, assignments and allocations, but can also encode other events such as exceptions [12, 65] and function calls [5].

Talpin and Jouvelot [113] apply a static polymorphic type-and-effect system to a language that supports imperative operations on reference values to reason about concurrency safety. Effects are generated for expressions accessing the memory values. The canonical property, dynamic and static semantics consistency, ensures that once a program written in their language type-checks, the program will not have concurrency errors. Importantly, programs in their system are implicitly typed and the system automatically infers the type-and-effect of the expressions. Because of the type reconstruction, programmers are relieved of the burden of specifying type-and-effect annotations to the programs. The effect reconstruction system infers effects conservatively (overapproximation); for example, if the effects of a method include memory write effects, the runtime execution of the method may or may not write to the memory.

Lucassen [72] also provides a similar type-and-effect system to reason about safe concurrency. In this system, polymorphic type-and-effect annotations can be added by programmers, which increase the flexibility of the system. To improve the precision, the system infers effects that are not visible outside of functions. Such effects can be ignored and safe concurrency can be improved. The system

also guarantees that the effects computed by the static type system are a sound approximation of the actual side-effects an expression will have.

Effect systems are shown to be valuable to reason about properties beyond safe concurrency. For example, they help programmers analyze function calls [5], locking disciplines [2], dynamic updating mechanisms [77], checked exceptions [12, 65], and detecting race conditions [19].

Marion and Millstein [73] observe that the application domains of different effect systems have a lot in common. In the paper, they introduce a generic type-and-effect system in order to unify several effect systems. The system allows programmers to parameterize the syntax of the effects to be tracked. Also, the system lets programmers customize two functions, known as the *privileges* and *capabilities*. These two functions together with the parameterized effects specify the effect discipline to be statically checked. Finally, the system describes how standard type soundness could be ensured by requiring that these two functions provide several monotonic properties.

Compared with these systems, our intensional effect system computes effects at runtime, which increases the precision of effect systems. Our system relies on dynamic typing to compensate for the conservativeness of these static effect systems and accounts for emerging dynamic features while at the same time harvesting the power of static typing to vouchsafe for programs.

2.2 Dynamic Effect Inspection

Recently, several works propose to use effects dynamically. Most of these works use effect systems to reason about safe concurrency.

Recent systems include TWEJava [59] and Legions [117]. In TWEJava, the core unit of work is a task. Programmers write effect specifications to annotate the effects of the tasks. Each task stores the potential effect of the task itself. TWEJava then implements a scheduler which takes into account the effects of the tasks at runtime and coordinates the tasks such that no tasks that have conflicting effects are executed concurrently. Properties of TWEJava include data race freedom and atomicity. At runtime, tasks can suspend themselves, yield control to subtasks, and transfer their effects to their subtasks to improve responsiveness. Legions provides similar features. In addition, it allows data to be dynamically assigned regions, unlike traditional static region assignment. Legions allow programmers

to provide hints to the scheduler to suggest whether tasks could be reordered, that is, run in a different order than the order in which they were presented to the scheduler. This reordering can have great performance benefits.

On the highest level, our system shares the philosophy with these systems hybridizing static and dynamic effect checking. To the best of our knowledge, however, this is the first time that intensional type analysis is applied to effect reasoning. This combination is powerful, because not only can effect reasoning rely on run-time type information, but also parametric polymorphism is fully retained. We showcase the benefits of generalizing the ideas to applications beyond safe concurrency, such as information security. Such generalizations introduce a subtle interaction between static typing and dynamic typing, which poses a unique challenge for maintaining type soundness.

2.3 Gradual Effect

Bañados *et al.* [10] developed a gradual effect (GE) type system based on gradual typing [106], by extending Marino and Millstein [73] with `?` ("unknown") types. Later Toro and Tanter [116] provided an implementation, which allows programmers to customize effect domains. These systems allow programmers to incrementally annotate the effects. That is, part of the program could be annotated with effect discipline, which will be checked statically. The rest of the program will be unannotated and will be checked and verified by the system at runtime. To check whether the effects respect the discipline, the systems run the application program and monitor the runtime effects, also known as *traces*, and check that the traces respect the discipline. On the other hand, our system lets programmers inspect and query the effect via dynamic typing. In other words, we provide the effects to programmers before running the program. As a gradual typing system, GE excels in scenarios such as prototyping. The system is also unique in its insight by viewing `?` type concretization as an abstract interpretation problem. Our work shares the high-level philosophy of GE — mixing static and dynamic typing for effect reasoning — but the two systems are orthogonal in their approaches. For example, GE programs may run into runtime type errors, whereas our programs do not. Foundationally, the power of intensional effect polymorphism lies upon how parametric polymorphism and intensional type analysis interact — a System F framework on the lambda cube — whereas frameworks based on gradual typing are not.

2.4 Dynamic Effect Analyses

We are unaware of other type-and-effect systems where the (pre-evaluation) effect of an expression is treated as a first-class value. The more established route is to treat the post-evaluation effect (in our terms, the trace) as a first-class value. In Leory and Pessaux [65], exceptions raised through program execution are available to the programmers. This work has influenced many exception handling systems such as Java, where `Exception` objects are also values. Bauer and Pretnar [11] extends the first-class exception idea and allows the programmer to annotate an arbitrary expression as an effect, and upon the evaluation of that expression, control is transferred to a matching `catch`-like handling expression as a first-class value. Similar designs also exist in implicit invocation and aspect-oriented systems [8, 9, 87, 88, 93, 95, 96, 97, 111]. Although much of the work cited in this section uses similar terminology to ours, in fact it is only indirectly related.

CHAPTER 3. INTENSIONAL EFFECT POLYMORPHISM

Improving the expressiveness and precision of type-and-effect systems through static approaches is a worthy direction, but looking forward, we believe these purely static approaches are unlikely to sustain future programming practices. In this chapter, we describe *intensional effect polymorphism*, a system that integrates static and dynamic effect reasoning.

The system relies on dynamic typing to compensate for the conservativeness of traditional static approaches and account for emerging dynamic features, while at the same time harvesting the power of static typing to vouchsafe for programs whose type safety is fundamentally dependent on runtime decision making. Consider the following example:

EXAMPLE 3.0.1 (Conservativeness of Static Typing for Race-Free Parallelism) *Imagine we would like to design a type system to guarantee race freedom of parallel programs. Let expression* $e||e'$ *denote running* e *and* e' *in parallel, whose typing rule requires that* e *and* e' *have disjoint effects. Further, let* `r1` *and* `r2` *be disjoint regions. The following program is race free, even though a purely static effect system is likely to reject it:*

$$
\begin{aligned}
&(\lambda x.\lambda y.(x := 1)||!y) \\
&\qquad (\textbf{\textit{if}}\ 1 > 0\ \textbf{\textit{then ref}}_{\texttt{r1}} 0\ \textbf{\textit{else ref}}_{\texttt{r2}} 0) \\
&\qquad (\textbf{\textit{if}}\ 0 > 1\ \textbf{\textit{then ref}}_{\texttt{r1}} 0\ \textbf{\textit{else ref}}_{\texttt{r2}} 0)
\end{aligned}
$$

Observe that parametric polymorphism is not helpful here: x and y can certainly be typed as region-polymorphic, but the program remains untypable. The root cause of this problem is that race freedom only depends on the *runtime* behaviors of $(x := 1)||!y$, which only depends on what x and y are *at runtime*.

Inspired by Harper and Morrisett [57], we propose an effect system where polymorphic code may intensionally inspect effects at run time. Specifically, expression $\mathbf{assuming}\ e\ \mathbb{R}\ e'\ \mathbf{do}\ e_1\ \mathbf{else}\ e_2$

inspects whether the runtime (effect) type of e and that of e' satisfy binary relation $\mathbb{R}$, and evaluates e_1 if so, or e_2 otherwise. Our core calculus leaves predicate $\mathbb{R}$ abstract, which under different instantiations can support a family of concrete type-and-effect language systems. To illustrate the example of race freedom, let us consider $\mathbb{R}$ being implemented as region disjointness relation $\#$. The previous example can be written in our calculus as follows.

EXAMPLE 3.0.2 (Intensional Effect Polymorphism for Race-Free Parallelism) *The following program type checks, with the static system and the dynamic system interacting in interesting ways. Static typing can guarantee that the lambda abstraction in the first line is well-typed regardless of how it is applied, good news for modularity. Dynamic typing provides precise typing for expression* $(x := 0)$ *and expression* $!y$ *— exploiting the runtime type information of* x *and* y *— allowing for a more precise disjointness check.*

$$(\lambda x.\lambda y.\textbf{\textit{assuming}}\ (x := 0)\#!y\ \textbf{\textit{do}}\ (x := 1)||!y)$$
$$(\textbf{\textit{if}}\ 1 > 0\ \textbf{\textit{then}}\ \textbf{\textit{ref}}_{r1}0\ \textbf{\textit{else}}\ \textbf{\textit{ref}}_{r2}0)$$
$$(\textbf{\textit{if}}\ 0 > 1\ \textbf{\textit{then}}\ \textbf{\textit{ref}}_{r1}0\ \textbf{\textit{else}}\ \textbf{\textit{ref}}_{r2}0)$$

Technical Innovations On the highest level, our system shares the philosophy with a number of type system designs hybridizing static checking and dynamic checking (*e.g.*, [47, 56, 106]), and some in the contexts of effect reasoning [10, 59]. To the best of our knowledge however, this is the first time intensional type analysis is applied to effect reasoning. This combination is powerful, because not only effect reasoning can rely on run-time type information, but also parametric polymorphism is fully retained. For example, observe that in the example above, the types for x and y are parametric, not just "unknowns" or "dynamic". Let us look at another example:

EXAMPLE 3.0.3 (Parametric Polymorphism Preservation) *Here the parallel execution in the second line is statically guaranteed to be type-safe in our system. Programs written with intensional effect polymorphism do not have run-time type errors.*

$$\textbf{\textit{let}}\ s = \lambda x.\lambda y.\textbf{\textit{assuming}}\ (x := 0)\#!y\ \textbf{\textit{do}}\ (x := 1)||!y\ \textbf{\textit{in}}$$
$$(s\ \textbf{\textit{ref}}_{r1}0\ \textbf{\textit{ref}}_{r2}0)\ ||\ (s\ \textbf{\textit{ref}}_{r3}0\ \textbf{\textit{ref}}_{r4}0)$$

In addition, intensional effect polymorphism goes beyond a mechanical adaptation of Harper-Morrisett, with several technical innovations we now summarize. The most remarkable difference is that the intensionality of our type system is enabled through *dynamic typing*. At run time, the evaluation of expression **assuming** $e\ \mathbb{R}\ e'$ **do** e_1 leads to the dynamic typing of e and e'. In contrast, the classic intensional type analysis performs a `typecase`-like inspection on the runtime instantiation of the polymorphic type. Our strategy is more general, in that it not only subsumes the former — indeed, a type derivation conceptually constructed at runtime must have leaf nodes as instances of value typing — but also allows (the effect of) arbitrary expressions to be inspected at run time. We believe this design is particularly relevant for effect reasoning, because it has less to do with the effect of polymorphic variables, and more with *where* the polymorphic variables appear in the program at run time.

Second, we design the runtime type inspection through a *relational* check. In the **assuming** expression, the dynamically verified condition is whether $\mathbb{R}$ holds, instead of what the effect of e or e' is. The relational design does not require programmers to explicitly provide an "effect specification/-pattern" of the runtime type — a task potentially daunting as it may either involve enumerating region names, or expressing conditional specifications such as "a region that some other expression does not touch." Many safety properties reasoned about by effect systems are relational in nature, such as thread interference.

Third, the subtle interaction between static typing and dynamic typing poses a unique challenge on type soundness in the presence of effect subsumption. We elaborate on this issue in §3.3.4. We introduce a notion of bounded existential types to differentiate but relate the types assumed by the static system and those by the dynamic system.

Finally, a full-fledged construction of type derivations at run time for dynamic typing would incur significant overhead. We design a novel optimization to allow for efficient runtime effect computation, eliminating the need for dynamic derivation construction, while producing the same result. The key insight is we could align the static type derivation and the (would-be-constructed) dynamic type derivation, and compute the effects of the latter simply by substituting the difference of the two, a strategy we call *differential alignment*. We will detail this design in §3.4.

3.1 Motivating Examples

In this section, we demonstrate the applicability of λ_{ie} in reasoning about safe parallelism, information security, consistent UI access and program optimization. In each of these applications, the type safety is fundamentally dependent on runtime decision making, that is, whether the relation $\mathbb{R}$ is satisfied. We instantiate the effect relation operator $\mathbb{R}$ with different concrete relations between effects of expressions.

As in previous work [16, 53], we optionally extend standard Java-like syntax with *region declarations* when the client language deems them necessary. In that case, a variable declaration may contain both type and region annotations, *e.g.*, `JLabel j` **in** `ui` declares a variable `j` in region `ui`. For client languages where regions are not explicitly annotated, different abstract locations (such as different fields of an object) are treated as separate regions.

3.1.1 Safe Parallelism

We demonstrate the application of λ_{ie} in supporting safe parallelism, where safety in this context refers to the conventional notion of thread non-interference (race freedom) [72]. Concretely, Figure 3.1 is a simplified example of "operation-agnostic" data parallelism, where the programmer's intention is to apply some statically unknown operation (encapsulated in an `Op` object) — here implemented through reflection — to a data set, here simplified as a pair of data `ft` and `sd`. The programmer wishes to "best effort" leverage parallelism to process `ft` and `sd` in parallel, without sacrificing thread non-interference. The tricky problem of this notion of safety depends on what `Op` object is. For instance, parallel processing of the pair with the `Hash` object is safe, but not when the operation at concern is the prefix sum operator [15], encapsulated as `Pref`.

Static reasoning about the correctness of the parallel composition could be challenging in this example, because the `Op` object remains unknown until $applyTwice$ is invoked at runtime.

The **assuming** expression (line 5) helps the program retain strong type safety guarantees for parallel composition (line 6), while utilizing the runtime information to enable precise reasoning. At runtime, the **assuming** expression intensionally inspects the effects of the expressions $ft = f.\,op(0)$ and $sd = f.\,op(5)$. If they satisfy the binary relation $\#$, parallelism will be enabled. If f points to a `Hash` object,

```
1  class Pair {
2    int ft = 1, sd = 2;

4    int applyTwice(Op f) {
5      assuming ft = f.op(0)   #   sd = f.op(5)
6      do ft = f.op(f.op(ft)) || sd = f.op(f.op(sd));
7    else ft = f.op(f.op(ft)) ;   sd = f.op(f.op(sd));
8    }
9  }

11 Pair pr = new Pair();
12 Op o = (Op) newInstance(readFile("filePath"));
13 pr.applyTwice(o);
```

```
14 interface Op {int op(int i);}

16 class Pref implements Op {
17   int sum = 0;
18   // effect:  write sum
19   int op(int i) { sum += i; }
20 }

22 class Hash implements Op {
23   // effect:  pure, no effect
24   int op(int i) { hash(i); }
25 }
```

Figure 3.1 Example illustrating λ_{ie} and its usage for safe parallelism.

```
1  class Page {
2    String searchBox = "";
3    String url = "wsj.com/search?";
4    String location = "";

6    String load_adv(ThirdParty adv) {
7      assuming url ◊ adv.show(this)
8      do exec url adv.show(this);
9      else "no advertisement";
10   }

12   int search(ThirdParty adv) {
13     load_adv(adv);
14     location = url + searchBox;
15   }
16 }
```

```
17 interface ThirdParty {String show(Page p);}

19 class Good implements ThirdParty {
20   String show(Page p) { "404"; }
21 }

23 class Evil implements ThirdParty {
24   String show(Page p) {
25     p.url = "evil.com";
26   }
27 }

29 ThirdParty adv =
30   (ThirdParty) newInstance(readFile("filePath"));
31 new Page().render(adv);
```

Figure 3.2 An application of λ_{ie} in preventing security vulnerabilities.

the $\#$ relation will be true and the program enjoys safe concurrency (line 6). On the other hand, if f points to a `Pref` object, the program will be run sequentially, desirable for race freedom safety.

3.1.2 Information Security

As another application of intensional effects, consider its usage in preventing security vulnerabilities. Figure 3.2 presents an adapted (`wsj.com`) example of a real-world security vulnerabilities [28]. The page allows users to search information within the site. Once the *search* is called, the page will redirect to a web page corresponding to the *url* and *searchBox* strings (the redirection is represented as changing the *location* variable for simplicity). The page, when created, inserts a third party advertisement, line 8.

The third party code can be malicious, *e.g.*, it can modify the search *url* and redirects the search to a malicious site, from which the whole system could be compromised, *e.g.*, the *Evil* third party code. Ensuring the key security properties becomes challenging with the dynamic features because the third party code is only available at runtime, loaded using reflection. The expression **exec** e_1 e_2 (line 7)

```
1  class UIThread {
2    JLabel global in ui = new JLabel();
3    void eventloop(Runnable closure) {
4      assuming global ∅ closure.run()
5        do spawn global closure.run();
6      else closure.run();
7    }
8  }

10 Runnable closure;
11 if (1 > 0) closure = new NonUI();
12 else closure = new UIAccess();
13 new UIThread().eventloop(closure);

14 region ui;

16 interface Runnable { String run(); }

18 class NonUI implements Runnable {
19   String run() { "does nothing"; }
20 }

22 class UIAccess implements Runnable {
23   JLabel j in ui = new JLabel();
24   String run() { j.val = "UI"; }
25 }
```

Figure 3.3 Example showing how UI effect discipline can be enforced by λ_{ie}.

encodes a *check-then-act* programming pattern. It executes e_2 only if it does not read nor write any object accessible by e_1 and otherwise it gets stuck. The **exec** expression does not execute e_1.

With λ_{ie}, users can intensionally inspect a third party code e whenever e is dynamically loaded. The intensional inspection, accompanied with a relational policy check, ensures that e does not access any sensitive data (the url), specified using the relation $\Diamond$. It also ensures that the **exec** expression does not get stuck.

3.1.3 Consistent Graphical User Interface (GUI) Access

We show how λ_{ie} can be used to reason about the correctness of a GUI usage pattern, common in Subclipse, JDK, Eclipse and JFace [51]. Typically, GUI has a single *UI thread* handling events in the "event loop". This UI thread often spawns separate *background threads* to handle time-consuming operations. Many frameworks enforce a single-threaded GUI policy: only the UI thread can access the GUI objects [51]. If this policy is violated, the whole application may abort or crash. Figure 3.3 shows a simplified example of a UI thread that pulls an event from the $eventloop$ and handles it. In the application, all UI elements reside in the ui region (declared on line 14), *e.g.*, the field j on line 23.

The safety here refers to no UI access in any background thread. The tricky problem here is that the events arrive at runtime with different event handlers. Some handlers may access UI objects while the others do not. Therefore, the correctness of spawning a thread to handle a new event, depends heavily on what objects the corresponding event handler has. For instances, the handler containing a $NonUI$ object can be executed in a background thread, while $UIAccess$ should not. The expression **spawn** e_1 e_2, executes e_2 in a background thread only if it does not allocate, read or write any object in the region specified by e_1, otherwise it gets stuck. The **spawn** expression does not execute e_1.

```
1  class Mem {
2    Integer input = new Integer();

4    int comp(Mutate m, Integer x) {
5      int cache = heavy(input);
6      assuming m.mutate(x) ⫫ heavy(input)
7      do lookup m.mutate(x) (cache=heavy(input));
8      else m.mutate(x); heavy(input);
9    }

11   int heavy(Integer i) { /* ...  */ }
12 }
```

```
13 class Integer { int i = 0; }

15 class Mutate {
16   int mutate(Integer input) {
17     input.i = 101;
18   }
19 }

21 Memo mm = new Mem();
22 Mutate mu = new Mutate();
23 if (1>0) mm.comp(mu, mm.input);
24 else mm.comp(mu, new Integer());
```

Figure 3.4 A proof-of-concept memoization technique.

The **assuming** expression, used by the UI thread, statically guarantees strong type safety for the **spawn** expression, so it wont get stuck. It also utilizes precise runtime information to distinguish handlers with no UI accesses from other handlers. If a handler satisfies the no UI access relation $\varnothing$, it can be safely executed by a background thread. The relation $\varnothing$ is satisfied if the RHS expression does not allocate, read/write any region denoted by the LHS expression.

3.1.4 Program Optimization – Memoization

We utilize λ_{ie} to implement a proof-of-concept memoization technique in a sequential program. Memoization is an optimization technique where the results of expensive function calls are cached and these cached results are returned when the inputs and the environment of the function are the same.

Figure 3.4 presents a simplified application where repeated tasks, here the $heavy$ method calls on line 5 and 8, are performed. These two tasks are separated by a small computation $mutate$, forming a *compute-mutate* pattern [22]. We leave the body of the method $heavy$ intentionally unspecified, which could represent a set of computationally expensive operations. It could, *e.g.*, generate the power set ps of a set of $input$ elements and return the size of ps or do the Bogosort.

The second $heavy$ task needs not be recomputed in full if the $mutate$ invocation does not modify the input nor the environment of $heavy$. If so, the cached result of the first call can be reused and the repeated computation can be avoided. The expression **lookup** e_1 $(e_2 = e_3)$ executes the expressions e_1 and e_2 as a sequence expression $e_1; e_2$ only if e_1 does not write to objects in the regions read by e_3. Otherwise it gets stuck.

Ensuring that the **lookup** expression does not get stuck is challenging. This is because the validity of cache depends on the runtime value of both the mutation m and its input x. For example, if the

parameter x is a new object as the one created on line 24, the cache is valid, while the one alias with the $input$ (line 23) is not valid.

The **assuming** expression solves the problem: the safety of the **lookup** expression is statically guaranteed. At runtime, with precise dynamic information, the intensional binary $\natural$ relational check ensures that the write accesses of the LHS do not affect the RHS expression. If this relation is satisfied, the cache is valid and can be reused.

Other optimizations Intensional effect polymorphism can be used for other similar optimizations, *e.g.*, record and reply style memoization, common sub-expression elimination, redundant load elimination and loop-invariant code motion. In all these applications, if the mutation, *e.g.*, $m.\,mutate(x)$, is infrequent or does not modify a large portion of the heap, the cached results can avoid repeated expensive computations.

Summary The essence of intensional effect polymorphism lies in the *interesting interplay between static typing and dynamic typing*. Static typing guarantees that the potentially unsafe expressions are only used under runtime "safe" contexts (that is, those that pass the relational effect inspection), in highly dynamic scenarios such as parallel composition, loading third party code, handling I/O events, and data reuse. Dynamic typing exploits program runtime type information to allow for more precise effect reasoning, in that "safe" contexts can be dynamically decided upon based on runtime type/effect information.

3.2 λ_{ie} Abstract Syntax

To highlight the foundational nature of intensional effect polymorphism, we build our ideas on top of an imperative region-based lambda calculus. The abstract syntax of λ_{ie} is defined in Figure 4.4. Expressions are standard for an imperative λ calculus, except the last two forms which we will soon elaborate. We do not model integers and unit values, even though our examples may freely use them. Since **if** e **then** e **else** e plays a non-trivial role in our examples, we choose to model it explicitly. As a result, boolean values $\mathsf{b} \in \{true, false\}$ are also explicitly modeled.

Our core syntax is expressive enough to encode the examples in §3.1. However, it does not model objects for simplicity without the loss of generality. Addition of objects is mostly standard [85].

$$
\begin{array}{lcll}
v & ::= & \mathtt{b} \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} & \textit{values} \\
\mathtt{e} & ::= & v \mid \mathtt{x} \mid \mathtt{e}\ \mathtt{e} \mid \textbf{let}\ \mathtt{x} = \mathtt{e}\ \textbf{in}\ \mathtt{e} \mid \textbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} \mid !\mathtt{e} \mid \mathtt{e} := \mathtt{e} \mid \textbf{if}\ \mathtt{e}\ \textbf{then}\ \mathtt{e}\ \textbf{else}\ \mathtt{e} & \textit{expressions} \\
 & \mid & \textbf{assuming}\ \mathtt{e}\ \mathbb{R}\ \mathtt{e}\ \textbf{do}\ \mathtt{e}\ \textbf{else}\ \mathtt{e} \mid \mathbb{SAFE}\ \mathtt{e}\ \mathtt{e} & \\
\mathtt{T} & ::= & \alpha \mid \textbf{Bool} \mid \mathtt{T} \xrightarrow{\sigma} \mathtt{T}' \mid \textbf{Ref}_{\rho}\ \mathtt{T} & \textit{types} \\
\rho & ::= & \overline{\zeta} & \textit{region} \\
\zeta & ::= & \mathrm{r} \mid \gamma & \textit{region element} \\
\sigma & ::= & \overline{\omega} & \textit{effect} \\
\omega & ::= & \varsigma \mid acc_{\rho}\mathtt{T} & \textit{effect element} \\
acc & ::= & \textbf{init} \mid \textbf{rd} \mid \textbf{wr} & \textit{access right}
\end{array}
$$

Figure 3.5 λ_{ie} Abstract Syntax (Throughout the thesis, notation $\overline{\bullet}$ represents a set of $\bullet$ elements, and notation $\overrightarrow{\bullet}$ represents a sequence of $\bullet$ elements.)

We introduced expression **assuming** $\mathtt{e}\ \mathbb{R}\ \mathtt{e}'$ **do** $\mathtt{e}_0$ **else** $\mathtt{e}_1$, which from now on we call $\mathtt{e}$ and $\mathtt{e}'$ the condition expressions, and $\mathtt{e}_0$ the **do** expression. In this more general form, programmers also define the behaviors when the effect check does not hold, specified by the **else** expression $\mathtt{e}_1$. At runtime, this expression retrieves the concrete effects of $\mathtt{e}$ and $\mathtt{e}'$ through dynamic typing, *i.e.*, evaluating neither $\mathtt{e}$ nor $\mathtt{e}'$. The timing of gaining this knowledge is important: the conditions will not be evaluated and the **do** expression is not evaluated yet. In other words, even though our system relies on runtime information, it is not an a posteriori effect monitoring system.

A General Framework Effect reasoning has diverse applications, such as enforcing thread non-interference, immutability, purity, to name a few. We aimed to design a general framework for effect reasoning, which can be concretized to different "client" languages. To achieve this goal, we choose to (1) leave the definition of the binary relation $\mathbb{R}$ abstract; (2) include an abstract $\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}'$ expression, which is type-safe iff $\mathtt{e}\ \mathbb{R}\ \mathtt{e}'$ holds. The $\mathbb{R}$ relation and the $\mathbb{SAFE}$ expression can be concretized to different "client" languages to capture different application domain goals. For example, possible $\mathbb{R}$ implementations are effect non-interference, effect disjointness, or degenerate unary properties such as purity and immutability. When $\mathbb{R}$ is concretized to thread non-interference, one possible concretization of $\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}'$ is parallel composition $\mathtt{e}||\mathtt{e}'$. The instantiations of $\mathbb{R}$ of the applications in §3.1 are shown in Figure 3.6.

Safe Parallel Composition, §3.1.1

$\mathtt{e}\ \mathbb{R}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathtt{e}\#\mathtt{e}'$ "Two effects do not interfere."

$\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad (\mathtt{e}\ ||\ \mathtt{e}')$ "Run the two expressions in parallel."

$\#$ is defined as:

$$\emptyset \# \sigma \qquad \frac{\sigma \# \sigma'' \quad \sigma' \# \sigma''}{\sigma \cup \sigma' \# \sigma''} \qquad \frac{\sigma' \# \sigma}{\sigma \# \sigma'} \qquad \mathbf{rd}_\rho \mathtt{T} \# \mathbf{rd}_{\rho'} \mathtt{T}' \qquad \frac{\rho \neq \rho'}{\mathbf{rd}_\rho \mathtt{T} \# \mathbf{wr}_{\rho'} \mathtt{T}'} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathtt{T} \# \mathbf{wr}_{\rho'} \mathtt{T}'}$$

Information Security, §3.1.2

$\mathtt{e}\ \mathbb{R}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathtt{e}\Diamond\mathtt{e}'$ "Expression $\mathtt{e}'$ does not read/write regions accessible by $\mathtt{e}$."

$\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathbf{exec}\ \mathtt{e}\ \mathtt{e}'$ "Execute $\mathtt{e}'$ if it does not read/write the regions by $\mathtt{e}$."

$\Diamond$ is defined as:

$$\sigma \Diamond \emptyset \qquad \frac{\sigma \Diamond \sigma'' \qquad \sigma' \Diamond \sigma''}{\sigma \cup \sigma' \Diamond \sigma''} \qquad \frac{\sigma'' \Diamond \sigma \qquad \sigma'' \Diamond \sigma'}{\sigma'' \Diamond \sigma \cup \sigma'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathtt{T} \Diamond \mathbf{rd}_{\rho'} \mathtt{T}'} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathtt{T} \Diamond \mathbf{wr}_{\rho'} \mathtt{T}'}$$

UI Access, §3.1.3

$\mathtt{e}\ \mathbb{R}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathtt{e}\varnothing\mathtt{e}'$ "Expression $\mathtt{e}'$ does not access regions accessible by $\mathtt{e}$."

$\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathbf{spawn}\ \mathtt{e}\ \mathtt{e}'$ "Execute $\mathtt{e}'$ in another thread if it accesses no region by $\mathtt{e}$."

$\varnothing$ is defined as:

$$\sigma \varnothing \emptyset \qquad \frac{\sigma'' \varnothing \sigma \qquad \sigma'' \varnothing \sigma'}{\sigma'' \varnothing \sigma \cup \sigma'} \qquad \frac{\sigma \varnothing \sigma'' \qquad \sigma' \varnothing \sigma''}{\sigma \cup \sigma' \varnothing \sigma''} \qquad \frac{\rho \neq \rho'}{acc_\rho \mathtt{T} \varnothing acc_{\rho'} \mathtt{T}'}$$

Memoization, §3.1.4

$\mathtt{e}\ \mathbb{R}\ \mathtt{e}' \quad \stackrel{\text{def}}{=} \quad \mathtt{e}\natural\mathtt{e}'$ "RHS's read has no dependcy on the LHS's write"

$\mathbb{SAFE}\ \mathtt{e}\ (\mathtt{e}_0 = \mathtt{e}_1) \quad \stackrel{\text{def}}{=} \quad \mathbf{lookup}\ \mathtt{e}\ (\mathtt{e}_0 = \mathtt{e}_1)$ "Execute $\mathtt{e}$;$\mathtt{e}_0$ if $\mathtt{e}$ writes no region read by $\mathtt{e}_1$."

$\natural$ is defined as:

$$\emptyset \natural \sigma \qquad \frac{\sigma \natural \sigma'' \qquad \sigma' \natural \sigma''}{\sigma \cup \sigma' \natural \sigma''} \qquad \mathbf{rd}_\rho \mathtt{T} \natural \sigma \qquad \sigma \natural \mathbf{wr}_\rho \mathtt{T} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}_\rho \mathtt{T} \natural \mathbf{rd}_{\rho'} \mathtt{T}'}$$

Figure 3.6 Client implementation of $\mathbb{R}$ and $\mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}$.

Types, Regions, and Effects Programmer types are either primitive types, reference types $\mathbf{Ref}_\rho\ \mathtt{T}$ for store values of type $\mathtt{T}$ in region ρ, or function types $\mathtt{T} \xrightarrow{\sigma} \mathtt{T}'$, from $\mathtt{T}$ to $\mathtt{T}'$ with σ as the effect of the function body. Last but not least, as a framework with parametric polymorphism, types may be type variables α.

Our notion of regions is standard [72, 113], an abstract collection of memory locations. A region in our language can either be demarcated as a constant r, or parametrically as a region variable γ.

An effect is a set of effect elements, either an effect variable ς, or $acc_\rho\mathtt{T}$, representing an access *acc* to region ρ whose stored values are of type $\mathtt{T}$. Access rights **init**, **rd**, **wr** represent allocation, read, and write, respectively.

As the grammar suggests, our framework is a flexible system where a type, a region, or an effect may all be parametrically polymorphic.

3.3 The Type System

This section describes the static semantics for our type-and-effect system. Overall, the type system associates each expression with effects, a goal shared by all effect systems. The highlight is how to construct a *precise* and *sound* effect system to support dynamic-typing-based intensionality. The precision of this type system is rooted at the $\mathbb{R}$ relation enforcement, at **assuming** time, based on effects computed by dynamic typing over runtime values and their types. Our static type system is designed so that any $\mathbb{SAFE}$ expression appearing in the **do** branch does not need to resort to runtime enforcement and the $\mathbb{R}$ relation is guaranteed to hold by the static type system. As we shall see, this leads to non-trivial challenges to soundness, as static typing and dynamic typing make related — yet different — assumptions on effects.

3.3.1 Definitions

Relevant structures of our type system are defined in Figure 3.7.

Type Environment and Type Scheme Type environment Γ maps variables to type schemes, and we use notation $\Gamma(x)$ to refer to $\mathtt{T}$ where the rightmost occurrence of $\mathtt{x} : \mathtt{T}'$ for any $\mathtt{T}'$ in Γ is $\mathtt{x} : \mathtt{T}$.

$$
\begin{array}{lcll}
\Gamma & ::= & \overline{\mathtt{x} \mapsto \tau} & \textit{type environment} \\
\tau & ::= & \forall \overline{g}.\exists \Sigma.\mathtt{T} & \textit{type scheme} \\
g & ::= & \alpha \mid \gamma \mid \varsigma & \textit{generic variable} \\
gs & ::= & \mathtt{T} \mid \rho \mid \sigma & \textit{generic structure} \\
\Phi & ::= & \overline{\Lambda} & \textit{relationship set} \\
\Sigma & ::= & \overline{g \preceq: gs} & \textit{subsumption set} \\
\Lambda & ::= & \sigma \ \mathbb{R} \ \sigma \mid \forall \overline{g}.\Sigma & \textit{relationship}
\end{array}
$$

Figure 3.7 λ_{ie} Type System Definitions

A type scheme is similar to the standard notion where names may be bound through quantification [34]. Our type scheme, in the form of $\forall \overline{g}.\exists \Sigma.\mathtt{T}$, supports both universal quantification and existential quantification. Our use of universal quantification is mundane: the same is routinely used for parametric polymorphism systems. Observe that in our system, type variables, region variables, and effect variables may all be quantified, and we use a metavariable g for this general form, and call it a *generic variable*. Similarly, we use a unified variable gs to represent either a type, a region, or an effect, and call it a *generic structure* for convenience. Existential quantification is introduced to maintain soundness, a topic we will elaborate in a later subsection. For now, only observe that existentially quantified variables appear in the type scheme as a sequence of $g \preceq: gs$, each of which we call a *subsumption relationship*. Here we also informally say g is existentially quantified, with *bound* gs. When $\overline{g}$ is a sequence of 0 and Σ is empty, we also shorten the type scheme $\forall \overline{g}.\exists \Sigma.\mathtt{T}$ as $\mathtt{T}$. Type schemes are alpha-equivalent.

Relationship Set Another crucial structure to construct our type system is the *relationship set* Φ. On the high level, this structure captures the relationships between generic structures. Concretely, it is represented as a set whose element may either be an *abstract effect relationship* $\sigma \ \mathbb{R} \ \sigma'$ — denoting two effects σ and σ' conform to the $\mathbb{R}$ relation — or a *subsumption context relationship*. The latter is represented as $\forall \overline{g}.\Sigma$. Intuitively, a subsumption context relationship is a collection of subsumption relationships, except some of its generic variables may be universally quantified. Subsumption context relationships are alpha-equivalent.

As we shall see, the relationship set plays a pivotal role during type checking. At each step of derivation, this structure represents what one can assume about effects. For example, the interplay

between **assuming** and $\mathbb{SAFE}$ is represented through whether the relationship set constructed through typing **assuming** can entail the relationship that makes the $\mathbb{SAFE}$ expression type-safe. Our relationship set may have a distinct structure, but effect system designers should be able to find conceptual analogies in existing systems, such as privileges in Marino *et al.* [73].

Notations and Convenience Functions We use (overloaded) function ftv to compute the set of free (*i.e.*, neither universally bound nor existentially bound) variables in $\mathtt{T}$, ρ and σ. We use $fv(\mathtt{e})$ to compute the set of free variables in expression e. We use *dom* and *ran* to compute the domain and the range of a function. All definitions are standard. Substitution θ is a mapping function from type variables α to types $\mathtt{T}$, region variables γ to regions ρ and effect variables ς to effects σ. The composition of substitutions, written $\theta\theta'$, if $\theta\theta'(g) = \theta(\theta'(g))$. We further use notation Θ to denote a substitution from variables to values.

We use comma for sequence concatenation. For example, $\Gamma, \mathtt{x} \mapsto \tau$ denotes appending sequence Γ with an additional binding from $\mathtt{x}$ to τ.

3.3.2 Subsumption and Entailment

Figure 3.8 defines subsumption relations for types, effects, and regions. All three forms of subsumption are reflexive and transitive. For function types, both return types and effects are covariant, whereas argument types are contra-variant. For **Ref** types, the regions are covariant, whereas the types for what the store holds must be invariant [115].

(EFF-INST) and (REG-INST) capture the instantiation of universal variables in subsumption context relationship. After all, the latter is a collection of "parameterized" subsumption relationships which can be instantiated.

Finally, we define a simple relation $\Phi \vdash_{ar} \Lambda$ to denote that relationship set Φ can entail Λ. (REL-IN) says any relationship set may entail its element. (REL-CLOSED) intuitively says that $\mathbb{R}$ is closed under the operation subset.

Subtyping: $\Phi \vdash \mathtt{T} \preceq: \mathtt{T}'$

$$\text{(TYPE-REFL)}\ \Phi \vdash \mathtt{T} \preceq: \mathtt{T} \qquad \text{(TYPE-TRAN)}\ \frac{\Phi \vdash \mathtt{T} \preceq: \mathtt{T}_0 \quad \Phi \vdash \mathtt{T}_0 \preceq: \mathtt{T}'}{\Phi \vdash \mathtt{T} \preceq: \mathtt{T}'} \qquad \text{(TYPE-REF)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash \mathbf{Ref}_{\rho}\ \mathtt{T} \preceq: \mathbf{Ref}_{\rho'}\ \mathtt{T}}$$

$$\text{(TYPE-FUN)}\ \frac{\Phi \vdash \mathtt{T}_0' \preceq: \mathtt{T}_0 \qquad \Phi \vdash \mathtt{T}_1 \preceq: \mathtt{T}_1' \qquad \Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'}{\Phi \vdash \mathtt{T}_0 \xrightarrow{\sigma} \mathtt{T}_1 \preceq: \mathtt{T}_0' \xrightarrow{\sigma'} \mathtt{T}_1'}$$

Effect Subsumption: $\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'$

$$\text{(EFF-REFL)}\ \Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma \qquad \text{(EFF-TRAN)}\ \frac{\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma_0 \qquad \Phi \vdash_{\mathit{eff}} \sigma_0 \preceq: \sigma'}{\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'} \qquad \text{(EFF-SUB)}\ \frac{\sigma \subseteq \sigma'}{\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'} \qquad \text{(EFF-CONS)}\ \frac{\sigma \preceq: \sigma' \in \Phi}{\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'}$$

$$\text{(EFF-ACC)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho'}{\Phi \vdash_{\mathit{eff}} \{\mathit{acc}_{\rho}\mathtt{T}\} \preceq: \{\mathit{acc}_{\rho'}\mathtt{T}\}} \qquad \text{(EFF–INST)}\ \frac{\forall \overline{g}.\Sigma \in \Phi \qquad \sigma \preceq: \sigma' \in \theta\Sigma \text{ for some } \theta \quad \mathit{dom}(\theta) = \overline{g} \qquad \mathit{ran}(\theta) \cap \mathit{ftv}(\Sigma) = \emptyset}{\Phi \vdash_{\mathit{eff}} \sigma \preceq: \sigma'}$$

Region Subsumption: $\Phi \vdash_{reg} \rho \preceq: \rho'$

$$\text{(REG-REFL)}\ \Phi \vdash_{reg} \rho \preceq: \rho \qquad \text{(REG-TRANS)}\ \frac{\Phi \vdash_{reg} \rho \preceq: \rho_0 \qquad \Phi \vdash_{reg} \rho_0 \preceq: \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(REG-SUB)}\ \frac{\rho \subseteq \rho'}{\Phi \vdash_{reg} \rho \preceq: \rho'} \qquad \text{(REG-CONS)}\ \frac{\rho \preceq: \rho' \in \Phi}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

$$\text{(REG–INST)}\ \frac{\forall \overline{g}.\Sigma \in \Phi \qquad \rho \preceq: \rho' \in \theta\Sigma \text{ for some } \theta \qquad \mathit{dom}(\theta) = \overline{g} \qquad \mathit{ran}(\theta) \cap \mathit{ftv}(\Sigma) = \emptyset}{\Phi \vdash_{reg} \rho \preceq: \rho'}$$

Relationship Entailment: $\Phi \vdash_{ar} \Lambda$

$$\text{(REL–IN)}\ \frac{\Lambda \in \Phi}{\Phi \vdash_{ar} \Lambda} \qquad \text{(REL-CLOSED)}\ \frac{\Phi \vdash_{ar} \sigma\ \mathbb{R}\ \sigma' \qquad \Phi \vdash_{\mathit{eff}} \sigma_0 \preceq: \sigma \qquad \Phi \vdash_{\mathit{eff}} \sigma_1 \preceq: \sigma'}{\Phi \vdash_{ar} \sigma_0\ \mathbb{R}\ \sigma_1}$$

Figure 3.8 λ_{ie} Subsumption and Entailment.

Typing: $\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma$

(T-BOOL)
$$\Phi; \Gamma \vdash \mathtt{b} : \mathbf{Bool}, \emptyset$$

(T-VAR)
$$\frac{\mathtt{T} \preceq \Gamma(x)}{\Phi; \Gamma \vdash \mathtt{x} : \mathtt{T}, \emptyset}$$

(T-LET)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \qquad \Phi; \Gamma, \mathtt{x} \mapsto \mathit{Gen}(\Gamma, \sigma)(\mathtt{T}) \vdash \mathtt{e}' : \mathtt{T}', \sigma'}{\Phi; \Gamma \vdash \mathbf{let}\ \mathtt{x} = \mathtt{e}\ \mathbf{in}\ \mathtt{e}' : \mathtt{T}', \sigma \cup \sigma'}$$

(T-SUB)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \qquad \Phi \vdash \mathtt{T} \preceq : \mathtt{T}' \qquad \Phi \vdash_{\mathit{eff}} \sigma \preceq : \sigma'}{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}', \sigma'}$$

(T-APP)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T} \xrightarrow{\sigma} \mathtt{T}', \sigma' \qquad \Phi; \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma''}{\Phi; \Gamma \vdash \mathtt{e}\ \mathtt{e}' : \mathtt{T}', \sigma \cup \sigma' \cup \sigma''}$$

(T-SET)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathbf{Ref}_{\rho}\ \mathtt{T}, \sigma \qquad \Phi; \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma'}{\Phi; \Gamma \vdash \mathtt{e} := \mathtt{e}' : \mathtt{T}, \sigma \cup \sigma' \cup \mathbf{wr}_{\rho}\mathtt{T}}$$

(T-REF)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma}{\Phi; \Gamma \vdash \mathbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} : \mathbf{Ref}_{\rho}\ \mathtt{T}, \sigma \cup \mathbf{init}_{\rho}\mathtt{T}}$$

(T-GET)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathbf{Ref}_{\rho}\ \mathtt{T}, \sigma}{\Phi; \Gamma \vdash ! \mathtt{e} : \mathtt{T}, \sigma \cup \mathbf{rd}_{\rho}\mathtt{T}}$$

(T-ABS)
$$\frac{\emptyset; \Gamma, \mathtt{x} \mapsto \mathtt{T} \vdash \mathtt{e} : \mathtt{T}', \sigma}{\Phi; \Gamma \vdash \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} : \mathtt{T} \xrightarrow{\sigma} \mathtt{T}', \emptyset}$$

(T-IF-THEN-ELSE)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathbf{Bool}, \sigma \qquad \Phi; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}, \sigma_0 \qquad \Phi; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}, \sigma_1}{\Phi; \Gamma \vdash \mathbf{if}\ \mathtt{e}\ \mathbf{then}\ \mathtt{e}_0\ \mathbf{else}\ \mathtt{e}_1 : \mathtt{T}, \sigma \cup \sigma_0 \cup \sigma_1}$$

(T-ASSUME)
$$\frac{\begin{array}{c}\overline{\mathtt{x}} = \mathit{fv}(\mathtt{e}) \cup \mathit{fv}(\mathtt{e}') \qquad \Gamma(\overline{\mathtt{x}}) = \overline{\tau} \qquad \Phi'' \vdash \mathit{EGen}(\overline{\tau}) \Rightarrow \overline{\tau}' \qquad \Gamma' = \Gamma, \overline{\mathtt{x} \mapsto \tau'} \qquad \Phi' = \Phi, \Phi'' \\ \Phi'; \Gamma' \vdash \mathtt{e} : \mathtt{T}, \sigma \qquad \Phi'; \Gamma' \vdash \mathtt{e}' : \mathtt{T}', \sigma' \\ \Phi', \sigma\ \mathbb{R}\ \sigma'; \Gamma' \vdash \mathtt{e}_0 : \mathtt{T}''', \sigma_2 \qquad \Phi \vdash \mathtt{T}''' \uparrow \mathtt{T}'' \qquad \Phi \vdash \sigma_2 \uparrow \sigma_0 \qquad \Phi; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}'', \sigma_1\end{array}}{\Phi; \Gamma \vdash \mathbf{assuming}\ \mathtt{e}\ \mathbb{R}\ \mathtt{e}'\ \mathbf{do}\ \mathtt{e}_0\ \mathbf{else}\ \mathtt{e}_1 : \mathtt{T}'', \sigma_0 \cup \sigma_1}$$

(T-SAFE)
$$\frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}_0, \sigma_0 \qquad \Phi; \Gamma \vdash \mathtt{e}' : \mathtt{T}_1, \sigma_1 \qquad \Phi \vdash_{ar} \sigma_0\ \mathbb{R}\ \sigma_1 \qquad \mathit{clientT}(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)}{\Phi; \Gamma \vdash \mathbb{SAFE}\ \mathtt{e}\ \mathtt{e}' : \mathtt{T}, \sigma}$$

Figure 3.9 λ_{ie} Typing Rules

Parallelism: ‖	$clientT(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathtt{T} = \mathtt{T}_0 = \mathtt{T}_1) \wedge (\sigma = \sigma_0 \cup \sigma_1)$
Security: **exec**	$clientT(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathtt{T} = \mathtt{T}_1) \wedge (\sigma = \sigma_1)$
UI: **spawn**	$clientT(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathtt{T} = \mathbf{void}) \wedge (\sigma = \emptyset)$
Memoization: **lookup**	$clientT(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)$	$\stackrel{\text{def}}{=}$	$(\mathtt{T} = \mathtt{T}_1) \wedge (\sigma = \sigma_0)$

Figure 3.10 Client Implementation of Predicate *clientT*

3.3.3 Typing Judgment Overview

Typing judgment in our system takes the form of $\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma$, which consists of a type environment Γ, a relationship set Φ, an expression e, its type $\mathtt{T}$ and effect σ. When the relationship set and the type environment are empty, we further shorten the judgment as $\vdash \mathtt{e} : \mathtt{T}, \sigma$ for convenience. The judgment is defined in Figure 3.9, with auxiliary definitions related to universal and existential quantification deferred to Figure 3.11.

The rules (T-LET) and (T-VAR) follow the familiar *let-polymorphism* (or Damas-Milner polymorphism [34]). Universal quantification is introduced at **let** boundaries, through function $Gen(\Gamma, \sigma)(\mathtt{T})$. Its elimination is performed at (T-VAR), via $\preceq$. Both definitions are standard, and appear in Figure 3.11. The let-polymorphism in **let** $\mathtt{x} = \mathtt{e}$ **in** $\mathtt{e}'$ expression is sound because of the Gen function in the rule (T-LET). The Gen function enforces the standard value restriction [113]. That is, if $\mathtt{e}$ is a value, its type could be generalized and thus be polymorphic, otherwise its type will be monomorphic.

(T-SUB) describes subtyping, where both (monomorphic) type subsumption and effect subsumption may be applied. Rules (T-REF), (T-GET) and (T-SET) for store operations produce the effects of access rights **init**, **rd** and **wr**, respectively. All other rules other than (T-ASSUME) and (T-SAFE) carry little surprise for an effect system.

3.3.4 Static Typing for Dynamic Intensional Analysis

To demonstrate how intensional effect analysis works, let us first consider an unsound but intuitive notion of **assuming** typing in (T-ASSUME-UNSOUND):

$$\text{(T-ASSUME-UNSOUND)}\quad \frac{\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \qquad \Phi; \Gamma \vdash \mathtt{e}' : \mathtt{T}', \sigma' \qquad \Phi, \sigma \mathbb{R} \sigma'; \Gamma \vdash \mathtt{e}_0 : \mathtt{T}'', \sigma_0 \qquad \Phi; \Gamma \vdash \mathtt{e}_1 : \mathtt{T}'', \sigma_1}{\Phi; \Gamma \vdash \textbf{assuming } \mathtt{e} \mathbb{R} \mathtt{e}' \textbf{ do } \mathtt{e}_0 \textbf{ else } \mathtt{e}_1 : \mathtt{T}'', \sigma_0 \cup \sigma_1}$$

$\forall$ Introduction: **Gen**	$Gen(\Gamma, \sigma)(\mathtt{T}) = \forall \overline{g}.\mathtt{T}$ where $\overline{g} = ftv(\mathtt{T}) \backslash (ftv(\Gamma) \cup ftv(\sigma))$
$\forall$ Elimination: $\preceq$	$\mathtt{T}' \preceq \forall \overline{g}.\mathtt{T}$ if $\mathtt{T}' = \theta\mathtt{T}$ for some θ
$\exists$ Introduction: **EGen**	

$$
\begin{array}{rcll}
\mathbb{P} & ::= & - \mid \mathbf{Ref}_{\mathbb{PR}}\ \mathtt{T} \mid \mathtt{T} \xrightarrow{\mathbb{PE}} \mathtt{T} \mid \mathtt{T} \xrightarrow{\sigma} \mathbb{P} & \textit{pack context} \\
\mathbb{PE} & ::= & - \mid \overline{\omega}, \mathbb{PE}, \overline{\omega}' \mid acc_{\mathbb{PR}}\ \mathtt{T} \mid acc_{\rho}\ \mathbb{P} & \\
\mathbb{PR} & ::= & - \mid \overline{\zeta}, \mathbb{PR}, \overline{\zeta}' &
\end{array}
$$

$$
\begin{array}{rcll}
\textit{EGen}(\forall \overline{g}.\mathtt{T}) & \triangleq & \forall \overline{g}.\textit{EGenM}(\mathtt{T}, \emptyset) & \\
\textit{EGenM}(\mathbb{PE}[\sigma], \overline{g}) & \triangleq & \exists \varsigma \preceq: \sigma.\textit{EGenM}(\mathbb{PE}[\varsigma \preceq: \sigma], \overline{g} \cup \{\varsigma\}) & \text{if } \sigma \notin \overline{g}, ftv(\sigma) \subseteq \overline{g}, \varsigma \text{ fresh} \\
\textit{EGenM}(\mathbb{PR}[\rho], \overline{g}) & \triangleq & \exists \gamma \preceq: \rho.\textit{EGenM}(\mathbb{PR}[\gamma \preceq: \rho], \overline{g} \cup \{\gamma\}) & \text{if } \rho \notin \overline{g}, ftv(\rho) \subseteq \overline{g}, \gamma \text{ fresh} \\
\textit{EGenM}(\mathtt{T}, \overline{g}) & \triangleq & \mathtt{T} & \text{if } \sigma \in \overline{g} \text{ for any } \mathtt{T} = \mathbb{PE}[\sigma] \\
 & & & \quad \rho \in \overline{g} \text{ for any } \mathtt{T} = \mathbb{PR}[\rho]
\end{array}
$$

$\exists$ Elimination: $\Rightarrow$	$\forall \overline{g}.(\theta\Sigma) \vdash \forall \overline{g}.\exists \Sigma.\mathtt{T} \Rightarrow \forall \overline{g}.\theta\mathtt{T}$ for some $\theta \wedge dom(\theta) \subseteq \overline{g}$
Lifting: $\uparrow$	

$$
\begin{array}{c}
\Phi \vdash \mathbb{P}[\varsigma \preceq: \sigma] \uparrow \mathbb{P}[\sigma'] \text{ if } \forall \overline{g}.\Sigma \in \Phi, \varsigma \preceq: \sigma \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \sigma \uparrow \sigma' \\
\Phi \vdash \mathbb{P}[\gamma \preceq: \rho] \uparrow \mathbb{P}[\rho'] \text{ if } \forall \overline{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \mathtt{T} \uparrow \mathtt{T} \text{ if } \forall \overline{g}.\Sigma \in \Phi, ftv(\mathtt{T}) \cap (\forall \overline{g}.\Sigma) = \emptyset
\end{array}
$$

$$
\begin{array}{c}
\Phi \vdash \mathbb{PE}[\varsigma \preceq: \sigma] \uparrow \mathbb{PE}[\sigma'] \text{ if } \forall \overline{g}.\Sigma \in \Phi, \varsigma \preceq: \sigma \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \sigma \uparrow \sigma' \\
\Phi \vdash \mathbb{PE}[\gamma \preceq: \rho] \uparrow \mathbb{PE}[\rho'] \text{ if } \forall \overline{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \sigma \uparrow \sigma \text{ if } \forall \overline{g}.\Sigma \in \Phi, ftv(\sigma) \cap (\forall \overline{g}.\Sigma) = \emptyset
\end{array}
$$

$$
\begin{array}{c}
\Phi \vdash \mathbb{PR}[\gamma \preceq: \rho] \uparrow \mathbb{PR}[\rho'] \text{ if } \forall \overline{g}.\Sigma \in \Phi, \gamma \preceq: \rho \in \theta\Sigma \text{ for some } \theta, \Phi \vdash \rho \uparrow \rho' \\
\Phi \vdash \rho \uparrow \rho \text{ if } \forall \overline{g}.\Sigma \in \Phi, ftv(\rho) \cap (\forall \overline{g}.\Sigma) = \emptyset
\end{array}
$$

Figure 3.11 Definitions for $\forall$ and $\exists$ Introduction and Elimination

To type check the **do** expression e_0, the static type system takes advantage of the fact that expressions e and e' satisfy the relation $\mathbb{R}$, *i.e.*, in the third condition of the rule, we strengthen the current Φ with $\sigma\ \mathbb{R}\ \sigma'$. The (T-SAFE) rule in Figure 3.9 says that the expression type checks iff Φ entails the abstract effect relationship $\mathbb{R}$. As a result, a $\mathbb{SAFE}$ expression whose safety happens to rely on $\sigma\ \mathbb{R}\ \sigma'$ can be statically verified to be safe by the static system.

Albeit tempting, the rule above is unsound. To illustrate, consider the safe parallelism discipline in the following example, *i.e.*, we instantiate the $\mathbb{R}$ relation with noninterference relation $\#$ and the $\mathbb{SAFE}$ expression with parallel expression $\|$.

EXAMPLE 3.3.1 (Soundness Challenge) *In the following example, the variables x and y have the same static (but different dynamic) type. Thus, the expression x 3 and y 3 have the same static effect.*

Should the parallel expression at line 5 typecheck with the assumption expression at line 4, there would be a data race at run time.

```
1 let buff = ref 0 in
2  let x = if 1 > 0 then λz. !buff else λz. buff := z in
3   let y = if 0 > 1 then λz. !buff else λz. buff := z in
4    assuming !buff  # x 3
5          do !buff || y 3
6        else !buff  ; x 2
```

In this example, we have an imperative reference *buff*, and two structurally similar but distinct functions x and y. The code intends to perform parallelization, *i.e.*, $!buff \,||\, y\ 3$, line 5. Let us review the types of the variables and the effects of the expressions:

$$\begin{array}{rcl} \texttt{buff} & : & \mathbf{Ref}_\rho\ \mathbf{Int} \\ \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int},\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \end{array} \qquad \begin{array}{rcl} \texttt{!buff} & : & \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{x 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\} \\ \texttt{y 3} & : & \{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\} \end{array}$$

According to the static system, the types of x and y are exactly the same. Thus, by the third condition of (T-SAFE), the expression $!\ buff \,||\, y\ 3$ in line 5, is well-formed. This is because the **assuming** expression has placed $\{\mathbf{rd}_\rho\mathbf{Int}\} \,\#\, \{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\}$ as an element of the relationship set, after typechecking $!\ buff \,\#\, x\ 3$ on line 4.

At runtime, the initialization expression of the **`let`** expressions will be first evaluated before being assigned to the variables (*call-by-value*, details in §3.4). Therefore, x becomes $\lambda\ z.!\ buff$ and y becomes $\lambda\ z.\ buff := z$ before the **assuming** expression. Informally, we refer to the effect computed at runtime through dynamic typing (*e.g.*, right before the **assuming** expression) as *dynamic effect*, as opposed to the *static effect* computed at compile time. The dynamic types and effects of the relevant expressions are:

$$\begin{array}{rcl} \texttt{x} & : & \mathbf{Int} \xrightarrow{\{\mathbf{rd}_\rho\mathbf{Int}\}} \mathbf{Int} \\ \texttt{y} & : & \mathbf{Int} \xrightarrow{\{\mathbf{wr}_\rho\mathbf{Int}\}} \mathbf{Int} \end{array} \qquad \begin{array}{l} \texttt{!buff} : \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{x 3} : \{\mathbf{rd}_\rho\mathbf{Int}\} \\ \texttt{y 3} : \{\mathbf{wr}_\rho\mathbf{Int}\} \end{array}$$

Clearly, the dynamic effect computed for $!\ buff$ and that for $x\ 3$ on line 4 do not conflict. Therefore, the **do** expression $!\ buff \,||\, y\ 3$ will be evaluated. However, the effects of the expressions $!\ buff$ and $y\ 3$ on line 5 do conflict, which causes unsafe parallelism.

The root cause of the problem is that the static and the dynamic system make decisions based on two related but different effects: one with the static effect, and the other with the dynamic effect. A sound type system must be able to differentiate the two.

A Sound Design with Bounded Existentials The key insight from the discussion above is that the static system must be able to express the *dynamic effect* that the **assuming** expression makes decision upon. Before we move on, let us first state several simple observations:

(i) (Dynamic effect refines static effect) The static effect of an expression $\mathtt{e}$ is a conservative approximation of the dynamic effect.

(ii) (Free variables determine effect difference) Improved precision of intensional effect polymorphism is achieved by using the more precise types for the free variables, see *e.g.*, dynamic and static type of the variable `x` in Example 3.3.1.

Observation *(i)* indicates the possibility of referring to the dynamic effect as "there exists some effect that is subsumed by the static effect." Observation *(ii)* further suggests that dynamic effect can be computed by treating all free variables existentially: "there exists some type `T` which is a subtype of the static type `T′` for each free variable, to help mimic the type environment while dynamic effect is computed". Bounded existential types provide an ideal vehicle for expressing this intention.

In (T-ASSUME), the type checking of an **assuming** expression, we substitute the type of each free variable with (an instance of) its existential counterpart. Let us revisit Example 3.3.1, this time with (T-ASSUME). The free variables of the **assuming** expression on line 4, in Example 3.3.1 are `x` and `buff`. The original types of the free variables are:

$$\texttt{buff} \ : \ \mathbf{Ref}_{\rho}\ \mathbf{Int} \qquad \text{and} \qquad \texttt{x} \ : \ \mathbf{Int} \xrightarrow{\{\mathbf{rd}_{\rho}\mathbf{Int},\mathbf{wr}_{\rho}\mathbf{Int}\}} \mathbf{Int}$$

The existential types used to type check the **assuming** expressions are:

$$\texttt{buff} \ : \ \mathbf{Ref}_{\rho}\ \mathbf{Int} \ \text{ and } \ \texttt{x} \ : \ \exists \varsigma_1 \preceq: \{\mathbf{rd}_{\rho}\mathbf{Int}\}, \varsigma_2 \preceq: \{\mathbf{wr}_{\rho}\mathbf{Int}\}.\mathbf{Int} \xrightarrow{\varsigma_1,\varsigma_2} \mathbf{Int}$$

The relationship set is:

$$\Phi = \varsigma_1 \preceq: \mathbf{rd}_{\rho}\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_{\rho}\mathbf{Int} \tag{3.1}$$

The effects of the condition expressions are:

$$\texttt{!buff} \ : \ \mathbf{rd}_{\rho}\mathbf{Int} \qquad \text{and} \qquad \texttt{x 3} \ : \ \varsigma_1 \preceq: \mathbf{rd}_{\rho}\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_{\rho}\mathbf{Int}$$

To type check the **do** expression, Φ is strengthened as:

$$\Phi' = \{\mathbf{rd}_\rho\mathbf{Int} \# \{\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\}, \varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}\} \tag{3.2}$$

When type checking the expression on line 5, $y\ 3$ has effect $\{\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}\}$. We cannot establish $\vdash_{ar}$ (as Figure 3.8). A type error is correctly induced against the potential unsafe parallel expression.

Rule (T-ASSUME) first computes the free variables from the two condition expressions, written $\overline{\mathtt{x}} = fv(\mathtt{e}) \cup fv(\mathtt{e}')$. With assumption $\Gamma(\overline{\mathtt{x}}) = \overline{\tau}$, all free variables $\overline{\mathtt{x}}$ are considered for type environment strengthening. It then applies the existential *introduction* function *EGen* to strengthen τ', the bounded existential with the original type τ as the bound. The definition of *EGen* is in Figure 3.11. It then *eliminates* (or *open*) the existential quantification using $\Rightarrow$. In a nutshell, this predicate $\Phi'' \vdash \mathit{EGen}(\overline{\tau}) \Rightarrow \overline{\tau}'$ introduces an existential type and eliminates it right away (a common strategy in building abstract data types [85]). Subsumption relationship information is placed into the relationship set, $\Phi' = \Phi, \Phi''$. The new environment Γ' has the new types $\overline{\tau}'$ for the free variables, an instantiation of the bounded existential type.

Function *EGen* uses the *EGenM* function to quantify effects and regions. Here, to produce the existential type, function *EGen* maintains the structure of the original type, *e.g.*, if the original type is a function type, it produces a new function type with all *covariant* types/effects/regions quantified. Observe that *contravariant* types/effects/regions are harmless: their dynamic counterpart (which also refines the static one) does not cause soundness problems. To facilitate the quantification, three *pack contexts*, $\mathbb{P}$, $\mathbb{PE}$, $\mathbb{PR}$, are defined, representing the contexts to contain a type, an effect, or a region, respectively.

Finally, the type of the **do** expression needs to be *lifted*, weakening types that may potentially contain refreshed generic variables of existential types, through a self-explaining $\uparrow$ definition in Figure 3.11. For example, the effect computed for the expression $x\ 3$ is $\varsigma_1 \preceq: \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \preceq: \mathbf{wr}_\rho\mathbf{Int}$. The $\uparrow$ function applies the substitution of $\{\varsigma_1 \mapsto \mathbf{rd}_\rho\mathbf{Int}, \varsigma_2 \mapsto \mathbf{wr}_\rho\mathbf{Int}\}$ on the precomputed effect and produces static effect $\mathbf{rd}_\rho\mathbf{Int}, \mathbf{wr}_\rho\mathbf{Int}$.

The typing of (T-SAFE) relies on the client function *clientT*. $\mathit{clientT}(\mathtt{T}, \sigma, \mathtt{T}_0, \sigma_0, \mathtt{T}_1, \sigma_1)$ defines the conditions where a safe expression should typecheck, as shown in Figure 3.10.

3.4 Dynamic Semantics

This section describes the dynamic semantics of λ_{ie}. The highlight is to support a highly precise notion of effect polymorphism via a lightweight notion of dynamic typing, which we call *differential alignment*.

Operational Semantics Overview The λ_{ie} runtime configuration consists of a *store* s, the to be evaluated expression e, and a *trace* f, defined in Figure 3.12. The store maps references (or locations) l to values v. In addition to booleans and functions, locations themselves are values as well. Each store cell also records the region (ρ) and type ($\mathtt{T}$) information of the reference. A trace informally can be viewed as "realized effects," and it is defined as a sequence of accesses to references, with **init**(l), **rd**(l), and **wr**(l), denoting the instantiation, read, and write to location l respectively. Traces are only needed to demonstrate the properties of our language. This structure and its runtime maintenance is unnecessary in a λ_{ie} implementation.

The small-step semantics is defined by relation $s; \mathtt{e}; f \rightarrow s'; \mathtt{e}'; f'$, which says that the evaluation of an expression e with the store s and trace f results in a value e', a new store s', and trace f'. We use notation $[\mathtt{x} \mapsto v]\mathtt{e}$ to define the substitution of x with v of expression $\mathtt{e}$. We use $\rightarrow^*$ to represent the reflexive and transitive closure of $\rightarrow$.

Dynamic Effect Inspection Most reduction rules are conventional, except *(asm)* and *(safe)*. The *(asm)* rule captures the essence of the **assuming** expression, which relies on dynamic typing to achieve dynamic effect inspection. Dynamic typing is defined through type derivation $s; \Phi; \Gamma \underset{\mathtt{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$, defined in the same figure, which extends static typing with one additional rule for reference value typing.

At runtime, the **assuming** expression retrieves the more precise dynamic effect of expression $\mathtt{e}_1$ and $\mathtt{e}_2$, and checks whether relation $\mathbb{R}$ holds. Observe that at run time, $\mathtt{e}_1$ and $\mathtt{e}_2$ in the **assuming** expression is not identical to their respective forms when the program is written. Now, the free variables in the static program has been substituted with values, which carries more precise information on types, regions, and effects. This is the root cause why intensional effect polymorphism can achieve higher precision than a purely static effect system.

Definitions:

s	$::=$	$\overline{l \rightarrow_{\langle \rho, \mathtt{T} \rangle} v}$	*store*
f	$::=$	$\overline{acc(l)}$	*trace*
v	$::=$	$\ldots \mid l$	*(extended) values*
$\mathbb{E}$	$::=$	$- \mid \mathbb{E}\ \mathtt{e} \mid v\ \mathbb{E} \mid$ **let** $\mathtt{x} = \mathbb{E}$ **in** $\mathtt{e} \mid$ **let** $\mathtt{x} = v$ **in** $\mathbb{E} \mid$ **ref** $\rho\ \mathtt{T}\ \mathbb{E}$	*evaluation context*
		$\mid !\mathbb{E} \mid\ \mathbb{E} := \mathtt{e} \mid v := \mathbb{E} \mid$ **if** $\mathbb{E}$ **then** $\mathtt{e}$ **else** $\mathtt{e}$	

Dynamic Typing: $s; \Phi; \Gamma \underset{\mathtt{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$

$$\text{(DT-LOC)}\quad \frac{\{l \mapsto_{\langle \rho, \mathtt{T} \rangle} v\} \in s}{s; \Phi; \Gamma \underset{\mathtt{D}}{\vdash} l : \mathbf{Ref}_{\rho}\ \mathtt{T}, \emptyset}$$

For all other (DT-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of judgment $\Phi; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma$ in the latter rule should be substituted with $s; \Phi; \Gamma \underset{\mathtt{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$ in the former.

Evaluation relation: $s; \mathtt{e}; f \rightarrow s'; \mathtt{e}'; f'$

(cxt)	$s; \mathbb{E}[\mathtt{e}]; f$	$\rightarrow$	$s'; \mathbb{E}[\mathtt{e}']; f, f'$	if	$s; \mathtt{e} \Rightarrow s'; \mathtt{e}'; f'$
(asm)	$s;$ **assuming** $\mathtt{e}_1\ \mathbb{R}\ \mathtt{e}_2$ **do** $\mathtt{e}$ **else** $\mathtt{e}'$	$\Rightarrow$	$s; \mathtt{e}_0; \emptyset$	if	$s; \emptyset; \emptyset \underset{\mathtt{D}}{\vdash} \mathtt{e}_i : \mathtt{T}_i, \sigma_i$ for i = 1, 2 and $\mathtt{e}_0 = \begin{cases} \mathtt{e} & \text{if } \sigma_1\ \mathbb{R}\ \sigma_2 \\ \mathtt{e}' & \text{otherwise} \end{cases}$
$(safe)$	$s; \mathbb{SAFE}\ \langle \mathsf{e} \rangle\ \langle \mathsf{e} \rangle'$	$\Rightarrow$	$clientR(s, \mathtt{e}, \mathtt{e}')$		
(set)	$s; l := v$	$\Rightarrow$	$s, \{l \mapsto_{\langle \rho, \mathtt{T} \rangle} v\}; v;$ **wr**(l)	if	$\{l \mapsto_{\langle \rho, \mathtt{T} \rangle} v'\} \in s$
(ref)	$s;$ **ref** $\rho\ \mathtt{T}\ v$	$\Rightarrow$	$s, \{l \mapsto_{\langle \rho, \mathtt{T} \rangle} v\}; l;$ **init**(l)	if	l *fresh*
(get)	$s; !l$	$\Rightarrow$	$s; s(l);$ **rd**(l)		
(app)	$s; \lambda \mathtt{x} : \mathtt{T}.\mathtt{e}\ v$	$\Rightarrow$	$s; [\mathtt{x} \mapsto v]\mathtt{e}; \emptyset$		
(let)	$s;$ **let** $\mathtt{x} = v$ **in** $\mathtt{e}$	$\Rightarrow$	$s; [\mathtt{x} \mapsto v]\mathtt{e}; \emptyset$		
(ifT)	$s;$ **if** *true* **then** $\mathtt{e}$ **else** $\mathtt{e}'$	$\Rightarrow$	$s; \mathtt{e}; \emptyset$		
(ifF)	$s;$ **if** *false* **then** $\mathtt{e}$ **else** $\mathtt{e}'$	$\Rightarrow$	$s; \mathtt{e}'; \emptyset$		

Figure 3.12 λ_{ie} Operational Semantics

It should be noted that we evaluate neither $\mathtt{e}_1$ nor $\mathtt{e}_2$ at the evaluation of the **assuming** expression. In other words, λ_{ie} is not an *a posteriori* effect monitoring system.

The reduction of *(safe)* relies on an abstract function *clientR*. *clientR*$(s, \mathtt{e}, \mathtt{e}')$ computes the runtime configuration after the one-step evaluation of the the $\mathbb{SAFE}$ expression. The abstract treatment of this function allows λ_{ie} to be defined in a highly modular fashion, similar to previous work [73]. We will come back to this topic, especially its impact on soundness, in §3.5.

Optimization: Efficient Effect Introspection through Differential Alignment The reduction system we have introduced so far may not be efficient: it requires full-fledged dynamic typing, which may entail dynamic construction of type derivations to compute the dynamic effects. In this section, we introduce one optimization.

As observed in §3.3.4, the (sub)expressions that do not have free variables will have the same static effects (*i.e.*, via computed static typing) and dynamic effects (*i.e.*, computed via dynamic typing). Our key insight is that, the only "difference" between the two forms of effects for the same expression lies with those introduced by free variables in the expression. As a result, we define a new dynamic effect computation strategy with two steps:

1. At compile time, we compute the static effects of the two expressions used for the effect inspection of each **assuming** expression in the program. In the meantime, we record the type (which contains free type/effect/region variables) of each free variable that appears in these two expressions.

2. At run time, we "align" the static type of each free variable with the dynamic type associated with the corresponding value that substitutes for that free variable. The alignment will compute a substitution of (static) type/effect/region variables to their dynamic counterparts. The substitution will then be used to substitute the effect we computed in Step 1 to produce the dynamic effect.

For Step 1, we define a transformation from expression $\mathtt{e}$ to an *annotated expression* $\mathtt{d}$, defined in Figure 3.13. The two forms are identical, except that the the **assuming** expression in the "annotated expression" now takes the form of **assuming** $(\overline{\mathtt{x} : \tau})$ $\mathtt{e}_1 : \sigma_1$ $\mathbb{R}$ $\mathtt{e}_2 : \sigma_2$ **do** $\mathtt{e}$ **else** $\mathtt{e}'$, which records the free variables of expressions $\mathtt{e}_1$ and $\mathtt{e}_2$ and their corresponding static types, denoted as $\overline{\mathtt{x} : \tau}$. The

Abstract Syntax in Optimized λ_{ie}

$$\mathtt{d} ::= v \mid \mathtt{x} \mid \mathtt{d}\ \mathtt{d} \mid \textbf{let}\ \mathtt{x} = \mathtt{d}\ \textbf{in}\ \mathtt{d} \mid \textbf{ref}\ \rho\ \mathrm{T}\ \mathtt{d} \mid !\mathtt{d} \mid \mathtt{d} := \mathtt{d} \mid \textbf{if}\ \mathtt{d}\ \textbf{then}\ \mathtt{d}\ \textbf{else}\ \mathtt{d} \quad \textit{annotated expressions}$$
$$\mid \textbf{assuming}\ (\overline{\mathtt{x} : \tau})\ \mathtt{d} : \sigma\ \mathbb{R}\ \mathtt{d} : \sigma\ \textbf{do}\ \mathtt{d}\ \textbf{else}\ \mathtt{d} \mid \mathbb{SAFE}\ \mathtt{d}\ \mathtt{d}$$

Transformation: $\mathtt{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}$

$$\mathtt{x} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{x}$$

$$\mathtt{e}\ \mathtt{e}' \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}\ \mathtt{d}' \qquad \text{if } \mathtt{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}, \mathtt{e}' \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}'$$

$$\vdots$$

$$\textbf{assuming}\ \mathtt{e}_1\ \mathbb{R}\ \mathtt{e}_2\ \textbf{do}\ \mathtt{e}_3\ \textbf{else}\ \mathtt{e}_4 \overset{\Phi,\Gamma}{\rightsquigarrow} \textbf{assuming}\ (\overline{\mathtt{x} : \tau})\ \mathtt{d}_1 : \sigma_1\ \mathbb{R}\ \mathtt{d}_2 : \sigma_2\ \textbf{do}\ \mathtt{d}_3\ \textbf{else}\ \mathtt{d}_4$$
$$\text{if } \overline{\mathtt{x}} = \mathit{fv}(\mathtt{e}_1) \cup \mathit{fv}(\mathtt{e}_2),\ \ \Gamma(\overline{\mathtt{x}}) = \overline{\tau}'$$
$$\Phi'' \vdash \mathit{EGen}(\overline{\tau}') \Rightarrow \overline{\tau},\ \ \Phi' = \Phi, \Phi''$$
$$\Phi'; \Gamma, \overline{\mathtt{x} \mapsto \tau} \vdash \mathtt{d}_i : \mathtt{T}_i, \sigma_i \text{ for } i = 1, 2$$
$$\mathtt{e}_i \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}_i \text{ for } i = 1, 2, 3, 4$$

Operational Semantics in Optimized λ_{ie}: $s; \mathtt{d}; f \to_O s; \mathtt{d}; f$

$$(\mathit{Ocxt}) \quad s; \mathbb{E}[\mathtt{d}]; f \to_O s'; \mathbb{E}[\mathtt{d}']; f, f' \quad \text{if } s; \mathtt{d}; f \Rightarrow_O s'; \mathtt{d}'; f'$$

$$(\mathit{Oasm}) \quad s; \textbf{assuming}(\overline{v : \tau})\ \mathtt{d}_1 : \sigma_1\ \mathbb{R}\ \mathtt{d}_2 : \sigma_2\ \textbf{do}\ \mathtt{d}\ \textbf{else}\ \mathtt{d}' \Rightarrow_O s; \mathtt{d}_0; \emptyset$$
$$\text{if } s; \emptyset; \emptyset \vdash_{\mathtt{DO}} \overline{v : \mathtt{T}}, \emptyset$$
$$\text{and } \theta\overline{\tau} = \mathit{Gen}(\emptyset, \emptyset)(\overline{\mathtt{T}})$$
$$\text{and } \mathtt{d}_0 = \begin{cases} \mathtt{d} & \text{if } \theta\sigma_1\ \mathbb{R}\ \theta\sigma_2 \\ \mathtt{d}' & \text{otherwise} \end{cases}$$

For all other $\Rightarrow_O$ rules, each is isomorphic to its counterpart $\Rightarrow$ rule, except that every occurrence of metavariable $\mathtt{e}$ in the latter rule should be substituted with $\mathtt{d}$ in the former.

Figure 3.13 Optimized λ_{ie} with Differential Alignment

same expression also records the statically computed effects σ_1 and σ_2 for $\mathtt{e}_1$ and $\mathtt{e}_2$. The free variable computation function *fv* and variable substitution function are defined for $\mathtt{d}$ elements in an analogous fashion as for $\mathtt{e}$ elements. We omit these definitions.

Considering all the annotated information is readily available while we perform static typing of the the **assuming** expression— as in (T-Assume) — the transformation from expression $\mathtt{e}$ to annotated expression $\mathtt{d}$ under Φ and Γ, denoted as $\mathtt{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}$, is rather predictable, defined in the same Figure.

The most interesting part of our optimized system is its dynamic semantics. Here we define a reduction system $\rightarrow_O$, at the bottom of the same figure. We further use $\rightarrow_O^*$ to represent the reflexive and transitive closure of $\rightarrow_O$. Upon the evaluation of the annotated **assuming** expression, the types associated with the free variables — now substituted with values — are "aligned" with the types associated with the corresponding values. The latter is computed by judgment $s; \Phi; \Gamma \vdash_{DO} \mathtt{d} : \mathtt{T}, \sigma$, defined as $s; \Phi; \Gamma \underset{\mathtt{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$ where $\mathtt{e} \overset{\Phi,\Gamma}{\rightsquigarrow} \mathtt{d}$. In other words, we only need to dynamically type *values* in the optimized λ_{ie}. The alignment is achieved through the computation of the substitution θ. As we shall see in the next section, such a substitution always exists for well-typed programs.

3.5 Meta-Theories

In this section, we establish formal properties of λ_{ie}. We first show our type system is sound relative to sound customizations of the client effect systems (§3.5.1). We next present important soundness results for intensional effect polymorphism in §3.5.2. We next present a soundness and completeness result on differential alignment in §3.5.3. The proofs of these theorems and lemmas can be found in the accompanying technical report. Before we proceed, let us first define two simple definitions that will be used for the rest of the section.

Definition 3.5.1 *[Redex Configuration] We say* $< s; e; f >$ *is a redex configuration of program* e'*, written* $e' \trianglerighteq <s, e, f>$*, iff* $\emptyset; e'; \emptyset \rightarrow^* s; \mathbb{E}[e]; f$.

Next, let us define relation $s \vdash f : \sigma$, which says that dynamic trace f *realizes* static effect σ under store s:

Definition 3.5.2 *[Effect-Trace Consistency]* $s \vdash f : \sigma$ *holds iff* $\mathrm{acc}(l) \in f$ *implies* $\mathrm{acc}_\rho T \in \sigma$ *where* $\{l \mapsto_{\langle \rho, T \rangle} v\} \in s$.

3.5.1 Type Soundness

Our type system leaves the definition of $\mathbb{R}$ and $\mathbb{SAFE}\ e\ e'$ abstract, both in terms of syntax and semantics. As a result, the soundness of our type system is conditioned upon how these definitions are concretized. Now let us explicitly define the sound concretization condition:

Definition 3.5.3 *[Sound Client Concretization] We say a λ_{ie} client is sound if under that concretization, the following condition holds: if $s;\Phi;\Gamma \underset{\flat}{\vdash} e_0 : \tau_0, \sigma_0$, $s;\Phi;\Gamma \underset{\flat}{\vdash} e_1 : \tau_1, \sigma_1$, $clientT(\tau, \sigma, \tau_0, \sigma_0, \tau_1, \sigma_1)$ and $(s', e, f) = clientR(s, e_0, e_1)$, then $s';\Phi;\Gamma \underset{\flat}{\vdash} e : \tau, \sigma$ and $s' \vdash f : \sigma$.*

All lemmas and theorems for the rest of this section are implicitly under the assumption that Definition 3.5.3 holds, which we do not repeatedly state.

Our soundness proof is constructed through subject reduction and progress:

Lemma 3.5.4 [Type Preservation] *If $s;\Phi;\Gamma \underset{\flat}{\vdash} e : \tau, \sigma$ and $s; e; f \rightarrow s'; e'; f'$, then $s';\Phi;\Gamma \underset{\flat}{\vdash} e' : \tau', \sigma'$ and $\tau' \preceq: \tau$ and $\sigma' \subseteq \sigma$.*

Proof: The proof is by cases on the reduction step applied and the typing derivation of $s;\Phi;\Gamma \underset{\flat}{\vdash} e : \mathtt{T}, \sigma$. □

Lemma 3.5.5 [Progress] *If $s;\Phi;\Gamma \underset{\flat}{\vdash} e : \tau, \sigma$ then either e is a value, or $s; e; f \rightarrow s'; e'; f'$ for some s', e', f'.*

Proof. The proof is by cases on the reduction step applied and the typing derivation of $s;\Phi;\Gamma \underset{\flat}{\vdash} e : \mathtt{T}, \sigma$. □

Theorem 3.5.6 (Type Soundness) *Given an expression e, if $\emptyset;\emptyset \vdash e : \tau, \sigma$, then either the evaluation of e diverges, or there exist some s, v, and f such that $\emptyset; e; \emptyset \rightarrow^* s; v; f$.*

Proof. Immediately followed from Lemma 3.5.4 and Lemma 3.5.5. □

3.5.2 Soundness of Intensional Effect Polymorphism

The essence of intensional effect polymorphism lies in the fact that through intensional inspection (dynamic typing at the **assuming** expression), every instance of evaluation of the $\mathbb{SAFE}\ e_0\ e_1$ ex-

pression in the reduction sequence must be "safe," where "safety" is defined through the $\mathbb{R}$ relation concretized by the client language. To be more concrete:

Definition 3.5.7 (Effect-based Soundness of Intensional Effect Polymorphism) *We say e is effect-sound iff for any redex configuration such that $e \trianglerighteq \langle s, e', f\rangle$ and $e' = \mathbb{SAFE}\ e_0\ e_1$, it must hold that $s; \emptyset; \emptyset \vdash_{\mathcal{D}} e_0 : \tau_0, \sigma_0$ and $s; \emptyset; \emptyset \vdash_{\mathcal{D}} e_1 : \tau_1, \sigma_1$ and $\sigma_0\ \mathbb{R}\ \sigma_1$.*

Effect-based soundness is a corollary of type soundness:

Corollary 3.5.8 (λ_{ie} Effect-based Soundness) *If $\emptyset; \emptyset \vdash e : \tau, \sigma$, then e is effect-sound.*

There remains a gap between this property and why one *intuitively* believes the $\mathbb{SAFE}\ e_0\ e_1$ execution is "safe": ultimately, what we hope to enforce is at runtime, the "monitored effect" — *i.e.* the trace through the evaluation of e_1 and that of e_2 — do not violate what $\mathbb{R}$ represents. The definition above falls short because it relies on the dynamic typing of e_1 and e_2. To rigorously define the more intuitive notion of soundness, let us first introduce a trace-based relation induced from $\mathbb{R}$:

Definition 3.5.9 (Induced Trace Relation) *$\mathbb{R}^{\mathcal{TR}}$ is a binary relation defined over traces. We say $\mathbb{R}^{\mathcal{TR}}$ is induced from $\mathbb{R}$ under store s iff $\mathbb{R}^{\mathcal{TR}}$ is the smallest relation such that if $\sigma_1\ \mathbb{R}\ \sigma_2$, then $f_1\ \mathbb{R}^{\mathcal{TR}} f_2$ where $s \vdash f_1 : \sigma_1$ and $s \vdash f_2 : \sigma_2$.*

One basic property of our reduction system is the trace sequence is monotonically increasing:

Lemma 3.5.10 (Monotone Traces) *If $s; e; f \rightarrow s'; e'; f'$, then $f' = f, f''$ for some f''.*

Proof. The proof is by considerationing each of the semantic rules in Figure 3.12. Clearly, in each rule, $f' = f, f''$ for some f'' □

Given this, we can now define the more intuitive flavor of soundness over traces:

Definition 3.5.11 (Trace-based Soundness of Intensional Effect Polymorphism) *We say e is trace-sound iff for any redex configuration such that $e \trianglerighteq \langle s, e', f\rangle$ and $e' = \mathbb{SAFE}\ e_0\ e_1$, it must hold that for any s_0, e_0', and f_0 where $s; e_0; f \rightarrow^* s_0; e_0'; f, f_0$ and any s_1, e_1', and f_1 where $s; e_1; f \rightarrow^* s_1; e_1'; f, f_1$, then condition $f_0\ \mathbb{R}^{\mathcal{TR}} f_1$ holds.*

To prove trace-based soundness, the crucial property we establish is:

Lemma 3.5.12 (Effect-Trace Consistency Preservation) *If* $s; \Phi; \Gamma \underset{\mathcal{D}}{\vdash} e : T, \sigma$ *and* $s \vdash f : \sigma$ *and* $s; e; f \to s'; e'; f'$ *then* $s' \vdash f' : \sigma'$.

Proof. The proof is by cases on the reduction step applied. □

Finally, we can prove the intuitive notion of soundness of intensional effect polymorphism:

Theorem 3.5.13 (λ_{ie} Trace-Based Soundness) *If* $\emptyset; \emptyset \vdash e : T, \sigma$*, then* e *is trace-sound.*

Proof. The proof is by cases on the reduction step applied, by Lemma 3.5.12 and by Lemma 3.5.4. □

3.5.3 Differential Alignment Optimization

In §3.4, we defined an alternative "optimized λ_{ie}" to avoid full-fledged dynamic typing, centering on differential alignment. We now answer several important questions: (1) *static completeness*: every typable program in λ_{ie} has a corresponding program in optimized λ_{ie}. (2) *dynamic completeness*: for every typable program in λ_{ie}, its corresponding program at run time cannot get stuck due to the failure of finding a differential alignment. (3) *soundness*: for every program in λ_{ie}, its corresponding program in optimized λ_{ie} should behave "predictably" at run time. We will rigorously define this notion shortly; intuitively, it means that "optimized λ_{ie}" is indeed an *optimization* of λ_{ie}, *i.e.*, without altering the results computed by the latter.

Optimization static completeness is a simple property of $\overset{\Phi,\Gamma}{\rightsquigarrow}$:

Theorem 3.5.14 (Static Completeness of Optimization) *For any* e *such that* $\Phi; \Gamma \vdash e : T, \sigma$*, there exists* d *such that* $e \overset{\Phi,\Gamma}{\rightsquigarrow} d$.

Proof. The proof is by cases of the typing derivation of the expression e under consideration and induction on the typing derivation of the subexpressions. □

To correlate the dynamic behaviors of λ_{ie} and optimized λ_{ie}, first recall that the $\to$ reduction system and $\to_O$ reduction system are identical, except for how the **assuming** expression is reduced. The progress of *(Oasm)* relies on the existence of substitution θ that aligns the dynamic type associated with values and the static type. Dynamic completeness of differential alignment thus can be viewed as

the "correspondence of progress" for the two reduction systems to reduce the corresponding **assuming** expressions. This is indeed the case, which can be generally captured by the following lemma:

Theorem 3.5.15 (Dynamic Completeness of Optimization) *If* $s; \Phi; \Gamma \underset{\mathcal{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$ *and* $\mathtt{e} \overset{\emptyset,\emptyset}{\rightsquigarrow} \mathtt{d}$*, then given some* s *and* f*, the following two are equivalent:*

- *there exists some* s'*,* $\mathtt{e}'$ *and* f*' such that* $s; \mathtt{e}; f \to s'; \mathtt{e}', f'$*.*
- *there exists some* s''*,* $\mathtt{d}'$ *and* f'' *such that* $s; \mathtt{d}; f \to_O s''; \mathtt{d}', f''$*.*

Proof. The proof is by cases of the expression $\mathtt{e}$ under consideration and induction on the number of reduction applied. □

Finally, we wish to study soundness. The most important insight is that the transformation relation $\overset{\Phi,\Gamma}{\rightsquigarrow}$ can be preserved through the corresponding reductions of λ_{ie} and optimized λ_{ie}. In other words, one can view the reduction of optimized λ_{ie} as a *simulation* of λ_{ie}:

Lemma 3.5.16 ($\to_O$ Simulates $\to$ with $\overset{\Phi,\Gamma}{\rightsquigarrow}$ Preservation) *If* $s; \Phi; \Gamma \underset{\mathcal{D}}{\vdash} \mathtt{e} : \mathtt{T}, \sigma$ *and* $\mathtt{e} \overset{\emptyset,\emptyset}{\rightsquigarrow} \mathtt{d}$ *and* $s; \mathtt{e}; f \to s'; \mathtt{e}', f'$ *and* $s; \mathtt{d}; f \to_O s''; \mathtt{d}', f''$*, then* $s' = s''$*, and* $f' = f''$*, and* $\mathtt{e}' \overset{\emptyset,\emptyset}{\rightsquigarrow} \mathtt{d}'$*.*

Proof. The proof is by cases of the expression $\mathtt{e}$ under consideration and induction on the number of reduction applied. □

Finally, let us state our soundness of differential alignment:

Theorem 3.5.17 (Soundness of Optimization) *Given some expression* $\mathtt{e}$ *such that* $\emptyset; \emptyset \vdash \mathtt{e} : \mathtt{T}, \sigma$*, and* $\mathtt{e} \overset{\emptyset,\emptyset}{\rightsquigarrow} \mathtt{d}$ *then*

- *there exists a reduction sequence such that* $\emptyset; \mathtt{e}; \emptyset \to^* s; v; f$ *iff there exists a reduction sequence such that* $\emptyset; \mathtt{d}; \emptyset \to_O^* s; v; f$*.*
- *there exists a reduction sequence such that the evaluation of* $\mathtt{e}$ *diverges according to* $\to$ *iff there exists a reduction sequence such that the evaluation of* $\mathtt{d}$ *diverges according to* $\to_O$*.*

Proof. The proof is by cases of the expression $\mathtt{e}$ under consideration, induction on the number of reduction applied and by Lemma 3.5.16. □

Observe that we are careful by not stating the two reduction systems must diverge at the same time, or reduce to the same value at the same time. That would be unrealistic if the client instantiations of our calculus introduce non-determinism.

3.6 Related Work

Static type-and-effect systems are well-explored. Earlier work includes Lucassen [72], and Talpin *et al.* [113], and more recent examples such as Marino *et al.* [73] , Task Types [62], Bocchino *et al.* [16] and Rytz *et al.* [101]. There are well-known language design ideas to improve the precision and expressiveness of static type systems, and many may potentially be applied to effect reasoning, such as flow-sensitive types [49], typestates [109] and conditional types [5]. Classic program analysis techniques such as polymorphic type inference, nCFA [105], CPA [4], context-sensitive, flow-sensitive, and path-sensitive analyses, are good candidates for effect reasoning of programs written in existing languages.

Bañados *et al.* [10] developed a gradual effect (GE) type system based on gradual typing [106], by extending Marino *et al.*[73] with `?` ("unknown") types. As a gradual typing system, GE excels in scenarios such as prototyping. The system is also unique in its insight by viewing `?` type concretization as an abstract interpretation problem. Our work shares the high-level philosophy of GE — mixing static typing and dynamic typing for effect reasoning — but the two systems are orthogonal in approaches. For example, GE programs may run into runtime type errors, whereas our programs do not. Foundationally, the power of intensional effect polymorphism lies upon how parametric polymorphism and intensional type analysis interact — a System F framework on the famous lambda cube — whereas frameworks based on gradual typing are not. Other than gradual typing, other solutions to mix static typing and dynamic typing include the `Dynamic` type [1], soft typing [24] and Hybrid Type Checking [47]. From the perspective of the lambda cube, their expressiveness is on par with gradual typing.

Intensional type analysis by Harper and Morrisett [57] is a framework with many extensions (*e.g.*, [32]). We apply it in the context of effect reasoning, and the intentionality in our system is achieved through dynamic typing, instead of `typecase`-style inspection on polymorphic types. To the best of our knowledge, our system is the first hybrid effect system built on top of the intensional type analysis.

Existential types are commonly used for type abstraction and information hiding. They are also suggested [57, 100] to capture the notion of `Dynamic` type [1]. Our use of existential types are closer to the latter application, except that we aim to differentiate (and connect) the types at compile time and the types at run time, instead of pessimistically viewing the former as `Dynamic`. We are unaware of the use of *bounded* existential types to connect the two type representations.

Effect systems are an important reasoning aid with many applications. For example, beyond the application domains we described in §3.1, they are also known to be useful for safe dynamic updating [77] and checked exceptions [12, 65].

3.7 Summary

In this chapter, we develop a new foundation for type-and-effect systems, where static effect reasoning is coupled with intensional effect analysis powered by dynamic typing. We describe how a precise, sound, and efficient hybrid reasoning system can be constructed, and demonstrate its applications in concurrent programming, memoization, information security and UI access.

CHAPTER 4. FIRST-CLASS EFFECTS REFLECTION

Type-and-effect systems, either purely static [72, 80, 113], dynamic [11, 63, 86] or hybrid [10, 59], have proven to be useful for program construction, reasoning, and verification. In existing approaches, the logic of accessing effects and making decisions over them — is defined by the language designer, and supported by the compiler or the runtime system. The end-user programmer is generally a consumer of the "hardcoded" logic for effect management.

The inspiration of this chapter arose from our endeavor to improve flexibility and precision for effect analyses. We explore two fundamental questions. First, are there benefits of empowering programmers with application-specific effect management? Second, is there a principled design for the effect management, so that programmers are endowed with powerful abstractions while in the meantime provided with strong correctness guarantees?

In this chapter, we develop *first-class effects*, a novel type-and-effect system where the effects of program expressions are available as first-class values to programmers. The resulting calculus, λ_{fc}, is endowed with powerful programming abstractions and a novel type system.

λ_{fc} Programming Abstractions With λ_{fc}, the lifecycle of effect management over program expressions becomes part of the program itself. Effects become first-class citizens, supported by three interconnected programming abstractions:

[EFFECT REFLECTION] Programmers can *query* the effect of any expression $\in$ through a λ_{fc} primitive, **query** $\in$, and the resulting value is a first-class value we call an *effect closure*.

[EFFECT INSPECTION] The structure and memory access details represented by an effect closure can be analyzed through a λ_{fc} effect pattern matching expression, enabling effect-based dispatch to naturally support effect-guided programming.

category	scheduling strategy
ordering strategy	FIFO [117]
	LIFO [63]
	random scheduling [59]
	inherit from previous decisions [63]
	write before read chapter 5
	tasks with less effects first chapter 5
	tasks with more effects first chapter 5
	concurrent read, exclusive write [59, 117]
	conflicting tasks in same thread [63, 84]
	task fusion [37, 86]
	divide into no conflicting groups of tasks [86]
	execute conflicting tasks concurrently [21, 23, 43]
	suspend conflicting tasks [121]
conflict detection strategy	latent effect conflict detection [16]
	pairwise tasks conflict detection[59]
	conflict detection with only the last task chapter 5
	task group conflict detection [117]
custom conflict model	tolerate write/write conflict [43, 76]
	tolerate read/write conflict [43, 76]
	privatization [20, 86, 98]
	speculation [98]

Table 4.1 An Example λ_{fc} Client Domain: The Menagerie of Scheduling Strategies

[EFFECT REALIZATION] The dual of effect reflection is effect realization: the λ_{fc} **realize** $\in$ expression allows an effect closure $\in$ to be *realized*, *i.e.*, the expression, where the effect abstracts from, to be evaluated.

Foundationally, effect reflection and effect realization are the introduction and elimination of first-class effects, in the form of the effect closure. Each effect closure is not only comprised of type-based encoding of the effects of the computation, but also the computation itself. As effect closures may cross modularity boundaries, effect closure can be intuitively viewed as "effect-carrying code."

The direct benefit of first-class effects is its support in flexible effect-guided programming. For example, thread scheduling in concurrent programs is an active and prolific area of research, known to have diverse strategies on schedule ordering, conflict detection, and conflict modeling (see Figure 4.1 for some examples). With first-class effects, programmers can flexibly develop different strategies, such as "if the effects of the tasks commute, then execute them concurrently, otherwise sequentially." We will demonstrate some λ_{fc} programming examples for effect-aware scheduling — and applications beyond scheduling — in §4.1. Overall, a variety of meta-level designs currently "hidden" behind the compiler and language runtime are now in the hands of programmers.

λ_{fc} Type System Design The grand challenge of designing an expressive and flexible programming model lies in principled and precise reasoning. Several hurdles exist in first-class effects.

First, dynamically querying the effect of an expression is tantamount to dynamic typing in a type-and-effect system. Compared with traditional effect systems [72, 113], enforcing soundness and maintaining efficiency are both non-trivial tasks when dynamic typing is mixed with static typing.

Second, despite the dynamic features inherent in first-class effects, it should remain a high priority to provide programmers *static* guarantees over effects. In addition to the more mundane goal of providing precise type structures to track first-class effect values, a feature highly desirable in effect-guided programming is to provide static guarantees to custom effect management. For example, programmers would wish to know whether the scheduling strategies implemented by their program — such as write-before-read — are enforced statically.

Third, programming with first-class effects may introduce unique challenges on program understanding if the underlying language is poorly designed. For instance, a programmer may wish to express that if the effect `x` of expression `e` is a subset of the effect `y` of expression `e'`, then make a run-time decision (*e.g.*, going to Mars). In practice however, the programmer in effect-guided programming indeed means that, if the *trace* — informally, the "post-evaluation effect" — of expression `e` is a subset of the *trace* of expression `e'`, then go to Mars. Unfortunately, effects are pre-evaluation and traces are post-evaluation, and the two do not always correspond. An overly relaxed language design may allow a surprising program behavior where the spaceship is Mars-bound when the traces of `e` and `e'` do not conform to subsetting but the effects do.

We meet the first challenge through a hybrid type system, an instance of type systems hybridizing static and dynamic analyses (*e.g.*, [106]), and more concretely, a member of small but growing family of hybrid type-and-effect systems [10, 59, 68]. We further come up with an optimized operational semantics where full-fledged dynamic typing is unnecessary. We address the second challenge through a refinement type system design. Custom, refined, and application-specific predicates over first-class effects may be statically verified. To address the third challenge, we introduce a novel notion of *trace consistency*, which intuitively says that any dynamic decision made over effects also holds for their corresponding traces. We are able to establish trace consistency for all programs written in first-class effects.

feature	benefit	reason
effect reflection	precision	employs run-time type information
	flexibility	allows dynamic effect computation
predicated effect inspection	flexibility	allows custom effect inspection
	program correctness	facilitates refinement typing
effect closure	flexibility	implements first-class effects
	soundness	connects expression and effect
refinement type	program correctness	provides refined static guarantees
double-bounded effects	flexibility	provides dual views of effects
	soundness	enforces trace consistency
polarity support	flexibility	enables non-monotone operators
	soundness	enforces trace consistency

Table 4.2 A Summary of λ_{fc} Features

The by-products of enforcing trace consistency are two type system features that may be interesting in its own right. First, effects in λ_{fc} have both upper and lower bound, which correspond to the duality of "*may-effect*", *i.e.*, over-approximation and "*must-effect*", *i.e.*, under-approximation. Double-bounded effects not only play a pivotal role in enforcing trace consistency, but also can be viewed independently as capturing the duality of permission [53] *vs.* obligation [18] in effect reasoning. Second, we introduce a notion of *polarity* to predicates defined over effects, providing a general solution to a long-standing problem in type-and-effect systems: reasoning about *non-monotone* effect operators [16, 73].

4.1 Motivating Applications

In this section, we motivate first-class effects through a number of applications ranging from custom effect-aware scheduling, to version-consistent dynamic software updating, to data security. We are aimed at demonstrating the benefits of first-class effects in two folds. First, it provides flexible and expressive abstractions to address challenging patterns of effect-guided programming. Second, it helps programmers design programs where effect analysis and manipulation are "correct-by-design," with refined guarantees specific to individual applications.

4.1.1 Custom Effect-Aware Schedulers

We start by illustrating how first-class effects may help programmers implement a simple ordering strategy in Figure 4.1, write-before-read. The resulting program in first-class effects can be found in

```
---------------------------------- Server ----------------------------------
1  let scheduler = λ x1:exact, x2:exact, buf: Ref_r Int.
2                   let (p, c) = effcase x1, x2:
3                                | _, EC(l1 ∼ u1) where wr_r /<<: u1 ⇒(x1, x2)
4                                | EC(l0 ∼ u0), _ where wr_r <<: l0 ⇒(x1, x2)
5                                | default                          ⇒(x2, x1) in
6                    {wr_r <<: c |-> wr_r <<: p} realize p; realize c
---------------------------------- Client ----------------------------------
7  let client = λ buf: Ref_r Int. let reader = (query !buf) in
8                                 let writer = (query buf := 1) in
9                                  scheduler reader writer buf in
10  client (if 1 > 0 then ref_r1 0 else ref_r2 0)
```

notation	meaning
query e	effect reflection
effcase x : $\overline{\text{T}}$ **where** $P \Rightarrow$ e	predicated effect dispatch
realize e	effect realization
x $\sim$ y	lower bound x effect & upper bound y
EC(x $\sim$ y)	effect closure type
x <<:y	effect x is a subset of y
x /<<:y	x is not a subset of y
|->	logical implication
{P}e	refinement type
wr$_r$	write effect to region r
exact	lower and upper bounds are equal

Figure 4.1 A Producer-First Scheduler (The new notations introduced by first-class effects are explained on the table below the listing.)

Figure 4.1, where the `scheduler` executes the "producer" task (*i.e.*, the one that writes to the region `r`) first, given the two input tasks `x1` and `x2`. The two tasks are `!buf` and `buf := 1` respectively, created by the `client`. Both tasks are parameterized by region parameter `r`, which at run time, may either be constant region `r1` or constant region `r2`.

Under the backdrop of purely static effect systems, this simple program highlights a number of programming challenges in effect-guided programming:

- [DYNAMIC EFFECT SUPPORT] The `scheduler` and the `client` could be deployed across modularity boundaries, such as on different OS domains or different machines. Even though it is easy to precisely specify the effects of the two tasks `!buf` and `buf := 1` on the `client` side, any practical `scheduler` should make no assumption on what the effects of `x1` and `x2` are. (The more general case is the `scheduler` takes a set of tasks as arguments.)

- [REFLECTION DESIGN] It requires principled design where the program fragment to produce the effect (such as the two tasks `!buf` and `buf := 1`) and that to inspect and manipulate the effect (such as checking whether said effect is a read or write) are unified under one program.

- [CUSTOM CORRECTNESS-BY-DESIGN] It is not obvious whether the program indeed has implemented the write-before-read strategy.

Dynamic Effect Support We address the first challenge through two programming abstractions: the **query** expression for dynamic effect querying, and the **effcase** expression for dynamic effect inspection. The **query** expression plays an interesting role in effect reasoning: it enables dynamic effect reasoning, which allows runtime information to be used in effects computation, and hence improves the precision of effect reasoning. For example, on line 7, λ_{fc} is capable of computing the effect as reading from region `r1`, instead of the more conservative "the union of `r1` and `r2`." The **effcase** expression is evocative of the `typecase` design to allow programmers to pattern-match the effect, lines 2-5. For example, the `scheduler` is able to tell which task(s) will write to `r` and execute `writer` before `reader`. Indeed, should we replace line 9 with `scheduler writer reader buf`, it can still schedule the producer first.

With the dual **query/effcase** design, effect querying is decoupled from effect inspection and effect-based decision making. This is useful in practice, because querying (dynamic typing) may incur runtime overhead, and the decoupling allows programmers to decide when the query should happen. For example, on line 7 and 8, the programmer says that the client should shoulder the overhead of effect query, not the server. In a similar vein, such a design allows effect of an expression to be queried once and used multiple times.

Reflection Design A core design question is what constitutes an effect value in a first-class effect system. One obvious choice is to represent the effect value just as a type. The opportunity such a design misses out on is a common idiom in effect-guided programming: the reason why programmers wish to query and inspect an effect in the first place is to *evaluate the expression the aforementioned effect abstractly represents*. For example, the reason we perform effect analysis over the tasks in this example is to schedule and run the tasks at the opportune moment. In first-class effects, we represent the first-class effect value as an *effect closure*, a combination of the effect type and the expression the effect abstractly represents. An effect closure can be passed across modular boundary, *e.g.*, line 9, which can now be viewed as a form of "effect-carrying code." Ultimately, we use the **realize** `e` expression to evaluate the expression the effect closure e abstractly represents.

In this light, the **query** expression in first-class effects can also be viewed as a simple form of reflection whereas the **realize** is analogous to reification. To the best of our knowledge, this is a novel design in hybrid/dynamic effect reasoning systems.

Custom Correctness-By-Design We provide static guarantees for custom-defined predicates over first-class effects through a decidable refinement type system. For example, on line 6, the intuition that the `scheduler` should be "producer-first" is captured by the refinement type $\{\mathbf{wr}_r$ `<<: c` $|{-}{>}$ $\mathbf{wr}_r$ `<<: p`$\}$, which reads if `c` writes `r`, then (logical implication $|{-}{>}$) `p` must also write `r`. Our refinement type system design is intimately linked to our programming abstraction of effect inspection: the case analysis of the **effcase** expression is *predicated* [44, 74]. For example, we are capable of typechecking the program with the refinement type above, thanks to the predicates associated with the **effcase** cases from lines 3-5.

A crucial question in this design space is whether the static guarantees represented in the form of refinement types matters for the program run-time behavior, and if so, *what they say about the run-time behavior*. We define the notion of *trace consistency*: for any well-typed expression whose refinement type has a predicate defined over effects, the "corresponding" predicate — identical except that every occurrence of the effect is replaced with corresponding memory accesses when said effect is realized — still holds. For example, if the refinement type, $\{\mathbf{wr}_r$ <<: `c` |-> $\mathbf{wr}_r$ <<: `p`$\}$, type checks, the memory traces of evaluating `c` and `p` are f_c and f_p respectively, then the formula $\{\mathbf{wr}_r$ <<: f_c |-> $\mathbf{wr}_r$ $f_p\}$ will always be true.

Effects in λ_{fc} have both *lower* and *upper* bounds, *e.g.*, line 3 says `x2` has must effect of `l1` and may effect of `u1`. We will motivate the design of double-bounded effects in §4.1.3 and §4.1.4.

Additional Examples on Scheduling First-class effects increase the flexibility of effect systems by allowing programmers design and customize effect-aware scheduling strategies, such as producer prioritized scheduling. As shown in Figure 4.1, other strategies are possible.

4.1.2 Version-Consistent Dynamic Software Update

Dynamic software update (DSU) [78] allows software to be updated to a new version for evolution without halting or restarting the software. DSU patches running software with new code on-the-fly. An important property of DSU is *version consistency* (VC) [77]. For VC, programmers specify program points where updates could be applied. Code within two immediate update points is viewed as a *transaction*, *i.e.*, we execute either the old version of the transaction or the new version completely.

The listing in Figure 4.2 defines a piece of `data` and two functions `fun1`, an empty function, and `fun0`, which increments the input by 1. Programmers would like to let a list of three blocks of code in `transaction`, lines 16-18 to be a transaction. The three blocks invoke the functions `fun0` and `fun1` and finally read the `data`. Because one of the functions increases `data` by one, the final result should be `!data + 1`. An example update, lines 14-15, swaps the bodies of the functions `fun0` and `fun1`. One approach is to delay the update until the end of `transaction`, *i.e.*, after the last block `!data` finishes execution. In this case, the transaction executes the old version in the current invocation and will execute the new version in the next invocation.

```
---------------------------------- Server ----------------------------------
1 let run1 = λ prologue, epilogue, update.
2   case epilogue of
3   | [] = realize update
4   | h:t = effcase prologue, epilogue, update:
5           |EC(xl ∼ xu), EC(yl ∼ yu), EC(zl ∼ zu) where zu # xu ∨ zu # yu
6                             =>{prologue # update ∨ epilogue
                                 # update} realize update; realize epilogue
7           | default         => h; run1 (query (realize prologue); h) t in
8 let run = λ transaction, update.
9             run1 (query 0) transaction update

---------------------------------- Client ----------------------------------
11 let data = ref_u 0 in
12 let fun0 = ref_r λx. x := !x + 1 in
13  let fun1 = ref_w λx. 1 in
14   let update = query (fun0:= (λx. 1);
15                       fun1:= (λx. x:= !x + 1)) in
16    let transaction = query [fun0 data,
17                             fun1 data,
18                             !data] in
19     run transaction update
```

Figure 4.2 Dynamic Software Updating in First-Class Effects to Preserve Consistency [77].

To increase update availability while ensuring VC, we would like to apply the update when it is available instead of at the end of `transaction`, *e.g.*, the swapping update happens correctly if it is patched after the second block of code `fun1 data`. Here we have executed the two functions whose bodies are to be swapped. The transaction executes the old version completely and the final value of `data` is 1. In contrast, assuming that the updated is patched after `fun0 data`, where we have executed the old version of `fun0` and increased `data` by 1. We will execute the new version of `fun1`, which also increases `data` by 1. The final result is 2, which is unintuitive to programmers.

The second update violates VC, we execute part old code `fun0`, part new code `fun1` and the final result is not correct.

Observe that first-class effects could help reason about whether a patch violates VC at any specific program point. If the effects σ_i of the patch do not conflict ($\#$) with the effects σ_p of code of the transaction before the update, noted as prologue, or effects σ_e of the code after, noted as epilogue, the immediate update (IU) respects VC (details see [77]). In the nutshell, if $\sigma_i \# \sigma_p$, IU is equivalent to applying the update at the beginning of the transaction. On the other hand, if $\sigma_i \# \sigma_e$, IU is equivalent to applying the update at the end of the transaction. This logic of the effects checking is implemented in method `run1` on lines 1-7. It first checks whether the transaction is done, line 3, *i.e.*, the `epilogue`

is an empty list `[]`. If so, the `update` could be applied immediately. Otherwise, effect inspection is used to analyze whether the effects of the `update` conflict with both `prologue` and `epilogue`. If there are no conflicts, line 5, update could also be applied at this point. Otherwise, the `update` needs to be delayed.

For example, the effects of the patch on line 15 is writing to the two functions, **wr**$_{\texttt{r}}$ **Int**, **wr**$_{\texttt{w}}$ **Int**, the effects of the there blocks of the transaction are below, and ✓ and ✗ represent the effects of the block conflict and do not conflict with the swapping update, respectively:

`!fun0 data`	**rd**$_{\texttt{r}}$ **Int** , **rd**$_{\texttt{u}}$ **Int**	✗
`!fun1 data`	**rd**$_{\texttt{w}}$ **Int** , **rd**$_{\texttt{u}}$ **Int**	✗
`!data`	**rd**$_{\texttt{u}}$ **Int**	✓

The update is problematic after the first block, because σ_i conflicts with both σ_p and σ_e, while it is okay after the second block because σ_i only conflicts with σ_p, but not σ_e.

First-class effects are very well-suited for this application. First, it allows programmers to query the effect of the update, which is important because the update is not available until at runtime. Second, it allows programmers to define their custom conflicting ($\#$) function, such as the ones shown in the *custom conflict model* section in Figure 4.1, that can go beyond the standard definition "two effects conflict if they access the same memory region" [77], *e.g.*, conflicts on statistical data does not affect the final results of a program and thus could be ignored [43, 76]. Programmers let the type system know this important fact by writing custom effect analyses in predicated pattern matching, line 5. Finally, first-class effects allow programmers to define custom VC correctness criteria, *e.g.*, on line 6, it says the `update` could only be applied if its effects do not conflict with both `prologue` and `epilogue`. This refinement type will be statically verified by λ_{fc}'s type system.

4.1.3 Data Zeroing

Information security is of growing importance in applications which interact with third-party library, available only at runtime. Consider an example of a bank account in Figure 4.3. It stores the password in the variable `pw`. It has a method `close`, which will be invoked when a client closes the account. This method accepts a third-party library `x` which displays advertisements when the account is closed [28].

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Bank account - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1 let pw = ref_r 12 in
2  let close = λx.effcase x:
3                  |EC(l∼u) where wr_r <:: l   => {wr_r <:: x} realize x
4                  |default                    => pw := -9 in
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Third party libraries - - - - - - - - - - - - - - - - - - - - - - - - - - - -
5   close (query pw := -9);              // Safe Library
6   close (query (if 0 then pw := -9)) // Unsafe
```

Figure 4.3 Data Zeroing in First-Class Effects Against Leakage of Sensitive Data.

The library could be malicious (*e.g.*, line 6), thus enforcing the security policy that no sensitive data are leaked by the library is vital in protecting the system [28]. We use the zeroing strategy [120]. At runtime, we programmatically analyze the effects of the library and execute it only if it destroys (overwrite) the password in the must effect $\mathbf{wr}_r$<<:y, to avoid the recovery of the original password.

Double-Bounded Effects The must-may effect distinction in λ_{fc} is crucial for program correctness. In traditional type-and-effect systems [72, 113], effects are conservative approximations of expressions. This "may-effect" (*i.e.*, over-approximation) may not be expressive enough for a number of applications, including data zeroing. Image the program that would be identical to the one in Figure 4.3 except that the **effcase** expression where the case on line 3 is predicated by the may effect, *i.e.*, $\mathbf{wr}_r$ <<:u. The may-effect view would allow the case to be selected as long as u may write to region r, such as the problematic client on line 6. Since the may-effect is an over approximation, the evaluation of the expression may not write to r at all! As a consequence, the pw is not overridden and could be leaked in the future! With double-bounded effects, the distinction is explicit, and programmers use the must effects on line 3 to ensure that the pw must be overridden. We will explain how λ_{fc} prevents the misuse of the must and may effect in §4.1.4.

The general-purpose zeroing policy is useful in detecting malicious library in many systems, but clients may desire other special-purpose policies, such as the password is not directly read by the library, *a.k.a confidentiality*. In first-class effects, by substituting line 3 with EC(l ∼ u) **where** $\mathbf{wr}_r$ <<:l ∧ $\mathbf{rd}_r$ /<<: u, we ensure that the password is not read in the current execution, it is destroyed and thus can not be read in the future, through the combination of confidentiality and zeroing. Other applicable policies are shown in Figure 4.3.

how	why
zeroing [120]	destroy/overwrite sensitive data
confidentiality [28]	can not read sensitive data
integrity [28]	can not write sensitive data
multiple accesses [104]	can access a subset of data, but not all
negative authorization [14]	can only read non-sensitive data
weak authorization [14]	overridable policy

Table 4.3 Representative Information Security Policies in First-Class Effects.

Summary The flexibility of λ_{fc} is on par with other security systems and λ_{fc} shares the philosophy of improving the flexibility of meta-level designs "hidden" behind effect system by allowing programmers to define custom policy.

4.1.4 Monotonicity and Polarity

Before we delve into the formal development, let us illustrate one dimension of first-class effects design critical for establishing trace consistency. Consider a simple example:

EXAMPLE 4.1.1 *Imagine we only have a traditional may-effect system. (This can be written in λ_{fc} where the lower-bound effect is not used.) The intention of the programmer is still to follow the strategy of write-before-read. However, the following program, if it were to typecheck, would execute the* `reader` *first, because the may-effects represented by both* `reader` *and* `writer` *are* $\mathbf{wr}_r$.

- Client2 -

```
1  let scheduler = λ x1, x2, buf: Ref_r Int.
2                  let (p, c) = effcase x1, x2:
3                                | _, EC(_ ∼ u1) where wr_r /<<: u1 => (x1, x2)
4                                | EC(_ ∼ u0), _ where wr_r <<: u0 => (x1, x2)
5                                | default                         => (x2, x1) in
6                   {wr_r <<: c |–> wr_r <<: p} realize p; realize c
7  let buf = ref_r 0 in
8   let reader = (query if 0 < 1 then buf := 1) in
9    let writer = (query buf := 1) in
10    scheduler reader writer
```

This program will fail type checking in λ_{fc}, through a polarity-based type system design. In λ_{fc}, each n-arity custom predicate is labeled with n polarities, one for each argument. Intuitively, each polarity indicates how effect subsumption affects predicate implication at that argument. To illustrate,

polarity	name	example	source
$-$	monotone decreasing	LHS of <<:	may effect
$+$	monotone increasing	RHS of <<:	must effect
i	invariant	==	may equals must

Table 4.4 Effect Operators and their Corresponding Polarities.

$\mathtt{e}$ ::=	$\mathtt{b} \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} \mid \mathtt{x} \mid \mathtt{e}\ \mathtt{e}$	*expressions*
	$\mid$ **let** $\mathtt{x} = \mathtt{e}$ **in** $\mathtt{e} \mid \mathtt{e}\|\|\mathtt{e}$	
	$\mid$ **ref** $\rho\ \mathtt{T}\ \mathtt{e} \mid\ !\mathtt{e} \mid \mathtt{e} := \mathtt{e}$	*reference*
	$\mid$ **if** $\mathtt{e}$ **then** $\mathtt{e}$ **else** $\mathtt{e}$	*branching*
	$\mid$ **effcase** $\mathtt{x} : \overline{\mathtt{T}}$ **where** $P \Rightarrow \mathtt{e}$	*effect dispatch*
	$\mid$ **query** $\mathtt{e}$	*effect reflection*
	$\mid$ **realize** $\mathtt{e}$	*effect realization*
P ::=	$\mathtt{b} \mid P \wedge P \mid P \vee P \mid \neg P \mid \mathbb{R}\,\overline{\sigma}$	*predicate*
g ::=	α	*type variable*
	$\mid \gamma$	*region variable*
	$\mid \varsigma$	*effect variable*
$\mathtt{T}$ ::=	**Bool** $\mid \alpha \mid$ **Ref**$_\rho\ \mathtt{T}$	*type*
	$\mid\ \mathtt{T} \xrightarrow{\sigma \sim \sigma'} \mathtt{T}'$	*function type*
	$\mid$ **EC**$(\mathtt{T}, \sigma \sim \sigma')$	*effect type*
ρ ::=	$\mathtt{r} \mid \gamma \mid \overline{\rho}$	*region*
σ ::=	$\pi_\rho \mathtt{T} \mid \varsigma \mid \overline{\sigma}$	*effect*
π ::=	**init** $\mid$ **rd** $\mid$ **wr**	*allocation, read and write*
$\mathtt{b}$ ::=	*true* $\mid$ *false*	*boolean*

Figure 4.4 λ_{fc} Abstract Syntax.

consider the operator <<:, λ_{fc} assigns the RHS of <<: with a $+$ polarity to indicate the RHS follows *monotone increasing reasoning*: x_0<<:x_1 entails x_0<<:x_2, given x_1<<:x_2. Examples for other polarities, such as $-$ and i, are shown in the table below.

When effects are applied to the predicate <<:, only the must-effect can be applied to the $+$ polarity position, and only the may-effect can be applied to the $-$ polarity position. By carefully regulating the interaction between may-must effects and predicate polarity, our type system is capable of maintaining trace consistency. This is the key reason why the program above fails to type check, on line 4. We introduce a full-fledged description of polarity support in §4.4.1.

4.2 λ_{fc}: a Calculus with First-Class Effects

The abstract syntax of λ_{fc} with first-class effects, but without refinement types, is defined in Figure 4.4. We defer the discussion of refinement type with effect polarities to §4.4. Our calculus is built on top of an imperative region-based λ calculus. Expressions are mostly standard, except the constructs for effect analyses. As parallel programs serve as an important application domain of first-class effects, we support the parallel composition expression $\mathtt{e}||\mathtt{e}$. We model branching and boolean values explicitly, because they are useful to highlight features such as double bounded effects. The sequential composition $\mathtt{e}; \mathtt{e}'$ in the examples is the sugar form of $\mathbf{let}\ \mathtt{x} = \mathtt{e}\ \mathbf{in}\ \mathtt{e}'$.

Effect management consists of the key abstractions described in §4.1.1. Expression **query** e dynamically computes the effect of expression e. The result of a query is an effect closure. Programmers can inspect the closure x with **effcase** $\mathtt{x} : \overline{\mathtt{T}}\ \mathbf{where}\ \overline{P \Rightarrow \mathtt{e}}$. The expression pattern-matches the closure against the type patterns $\overline{\mathtt{T}}$. P is type constraint to further refine pattern matching. It supports connectives of proposition logic, together with the atomic n-ary form $\mathbb{R}\ \overline{\sigma}$, left abstract, which can be concretized into different forms for different concrete languages.

Effects and Effect Types. Effects are region accesses and have the form $\pi_\rho \mathtt{T}$, representing an access right π to values in region ρ. Access rights include allocation **init**, read **rd** and write **wr**.

Compared with existing effect systems, our system supports both must- and may-effects. Function types $\mathtt{T} \xrightarrow{\sigma \sim \sigma'} \mathtt{T}'$ specifies a function from T to T′ with must-effect σ and may-effect σ' as the effects of the function body.

Effect closure type has the form $\textsc{ec}(\mathtt{T}, \sigma \sim \sigma')$. The value it represents produces must-effect σ, may-effect σ' upon realization, and the realized expression has type T. When T is not used (*e.g.*, in Figure 4.1), we shorten the closure as $\textsc{ec}(\sigma \sim \sigma')$. With λ_{fc}, programmers can inspect the structure of effects, allowing latent effect associated with higher-order function to be analyzed or more generally any effects nested in the types, *e.g.*, for a *single instruction multiple data* (SIMD) application, with the **effcase** expression, programmers can inspect the type, as well as the latent effects of the instruction to verify the concurrency safety. In contrast, traditional systems [59, 68, 117] will create a task for each data, and check that the effects of each pair of tasks do not interfere, that can lead to $O(n^2)$ checks, where n is the size of the data. In a similar vein, such a design eases the effects reasoning of the *map*, *filter*, *select* functions in the *MapReduce* and *ParallelArray* framework [35, 64].

Regions. The domain of regions is the disjoint union of a set of constants r. The region abstracts memory locations in which it will be allocated at runtime. Our notion of region is standard [72, 113]. In λ_{fc}, allocation sites are explicitly labelled with regions. Region inference is feasible [49, 53], an issue orthogonal to our interest.

In λ_{fc}, type α, effect ς, and region γ variables are cumulatively referred to as "pattern variables", and we use a metavariable g for them. A type variable α can be used in the **effcase** expressions to match any type $\mathtt{T}$, given that the constraint P is satisfied if α is substituted with $\mathtt{T}$, similar for effect and region variables.

Before we proceed, let us provide some notations and convenience functions used for the rest of the chapter. Functions *dom* and *rng* are the conventional domain and range functions. Substitution θ maps type variables α to types $\mathtt{T}$, region variables γ to regions ρ, and effect variables ς to effects σ. Comma is used for sequence concatenation.

4.3 A Base Type System with Double-Bounded Effects

The key innovations of our type system design are twofold. First, it uses double bounded types to capture must-may effects. Second, it employs refinement types to fulfill hybrid effect reasoning. We present double-bounded effects in this section, and delay refinement types to §4.4.

4.3.1 Subtyping

Relation $\mathtt{T} <: \mathtt{T}'$ says $\mathtt{T}$ is a subtype of $\mathtt{T}'$ defined in Figure 4.5. The subtyping relation is reflexive and transitive. Reference **ref** types follow invariant subtyping, except that the regions in the **ref** types follow covariant subtyping.

The highlight of the subtyping relation lies in the treatment of the must-may effects. In (sub-FUN), observe that must-effects and may-effects follow opposite directions of subtyping: may-effects are covariant whereas must-effects are contravariants. Intuitively, for a program point that expects a function that *must* produce effect σ, it is always OK to be provided with a function that *must* produce a "superset effect" of σ. On the flip side, for a program point that expects a function that *may* produce effect σ, it is always OK to be provided with a function that *may* produce a "subset effect" of σ.

Subtyping: $\mathtt{T} <: \mathtt{T}'$

$$(sub\text{-REFL})\quad \mathtt{T} <: \mathtt{T}$$

$$(sub\text{-TRANS})\quad \frac{\mathtt{T} <: \mathtt{T}'' \qquad \mathtt{T}'' <: \mathtt{T}'}{\mathtt{T} <: \mathtt{T}'}$$

$$(sub\text{-REF})\quad \frac{\rho \subseteq \rho'}{\mathbf{Ref}_{\rho}\ \mathtt{T} <: \mathbf{Ref}_{\rho'}\ \mathtt{T}}$$

$$(sub\text{-FUN})\quad \frac{\mathtt{T}'_x <: \mathtt{T}_x \qquad \mathtt{T} <: \mathtt{T}' \qquad \sigma_2 \subseteq \sigma_0 \qquad \sigma_1 \subseteq \sigma_3}{\mathtt{T}_x \xrightarrow{\sigma_0 \sim \sigma_1} \mathtt{T} <: \mathtt{T}'_x \xrightarrow{\sigma_2 \sim \sigma_3} \mathtt{T}'}$$

$$(sub\text{-EC})\quad \frac{\mathtt{T} <: \mathtt{T}' \qquad \sigma_2 \subseteq \sigma_0 \qquad \sigma_1 \subseteq \sigma_3}{\mathbf{EC}(\mathtt{T}, \sigma_0 \sim \sigma_1) <: \mathbf{EC}(\mathtt{T}', \sigma_2 \sim \sigma_3)}$$

Figure 4.5 The Subtyping Relation.

Type Checking: $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$

(T-SWITCH)

$$\frac{\Gamma \vdash \mathtt{x} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset \qquad \exists \theta\ .(\theta\mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i) <: \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')), \text{for all } i \in \{1 \ldots n\} \qquad \exists \varsigma\ .\ \mathtt{T}_n = \varsigma \wedge P_n = \mathit{true} \qquad \Gamma, \mathtt{x} \mapsto \mathbf{EC}(\mathtt{T}_i, \sigma_i \sim \sigma'_i) \vdash \mathtt{e}_i : \mathtt{T}, \sigma''_i \sim \sigma'''_i, \text{for all } i \in \{1 \ldots n\}}{\Gamma \vdash \mathbf{effcase}\ \mathtt{x} : \overline{\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')}\ \mathbf{where}\ P \Rightarrow \mathtt{e} : \mathtt{T}, \cap_{i \in \{1 \ldots n\}} \sigma''_i \sim \cup_{i \in \{1 \ldots n\}} \sigma'''_i}$$

(T-QUERY)

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash \mathbf{query}\ \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset}$$

(T-REALIZE)

$$\frac{\Gamma \vdash \mathtt{e} : \mathbf{EC}(\mathtt{T}, \sigma_0 \sim \sigma_1), \sigma_2 \sim \sigma_3}{\Gamma \vdash \mathbf{realize}\ \mathtt{e} : \mathtt{T}, \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3}$$

Figure 4.6 Typing Rules.

As expected, effect subsumption in our system — the "superset effect" and the "subset effect" — is supported through set containment over σ elements. Effect closure types follow a similar design, as seen in (sub-EC).

Our covariant design of may-effects and contravariant design of must-effects on the high level is aligned with the intuition that along the data flow path, the dual bounds of a function, or those of a first-class effect closure may potentially be "loosened."

4.3.2 Type Checking

Type environment Γ maps variables to types:

$$\Gamma \quad ::= \quad \overline{\mathtt{x} \mapsto \mathtt{T}}$$

Notation $\Gamma(\mathtt{x})$ denotes $\mathtt{T}$ if the rightmost occurrence of $\mathtt{x} : \mathtt{T}'$ for any $\mathtt{T}'$ in Γ is $\mathtt{x} : \mathtt{T}$.

Type checking is defined through judgment $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$, defined in Figure 4.6. The judgment says under type environment Γ, expression $\mathtt{e}$ has type $\mathtt{T}$, must-effect σ and may-effect σ'. Subtyping

is represented in the type checking process through (T-SUB), which follows the same pattern to treat must-may effects as in function subtyping and effect closure subtyping.

Effect Bound Reasoning. If effect bounds are "loosened" along the data flow path as we discussed, the interesting question is when the bounds are "tightened". To answer this question, observe that traditional effect systems can indeed be viewed as a (degenerate) double-bounded effect system, where the must-effect is always the empty set.

Our type system on the other hand computes the must-effect along the type checking process. Note that in (T-IF), the must-effect of the branching expression is the *intersection* of the must-effects of the **then** branch and the **else** branch, unioned with the must-effect of the conditional expression. For example, given an expression as follows and that `val` is in region `r`, the must- and may-effects are $\{\mathbf{rd}_r\}$ and $\{\mathbf{rd}_r, \mathbf{wr}_r\}$ respectively:

```
if x > 0 then !val else val := !val + 1
```

The must-effect of the **effcase** expression is computed in an analogous fashion, as shown in the (T-EFFCASE) rule.

Typing Effect Operators. (T-QUERY) shows the expression to introduce an effect closure — the **query** expression — is typed as an effect closure type, including both the *static* type of the to-be-dynamically-typed expression, and its double-bounded effects as reasoned by the static system. In other words, even though first-class effects will be computed at runtime based on information garnered from (the more precise) dynamic typing, our static type system still makes its best effort to type this first-class value, instead of viewing it as an opaque "top" type of the effect closure kind.

The dual of the **query** expression is the **realize** expression. Intuitively, this expression "eliminates" the effect closure, and evaluates the expression carried by the closure, which for convenience we call the *passenger* expression. (T-REALIZE) is defined to be consistent with this view. It says that the expression should have the type of the passenger expression, and the effects should include both those of the expression that will evaluate to the effect closure, and those of the passenger expression.

The (T-EFFCASE) rule shows that the **effcase** expression predictably follows the pattern matching semantics. The variable to inspect must represent an effect closure. To avoid unreachable patterns, the type system ensures every type pattern is indeed satisfiable through substitution and subtyping. In

addition, λ_{fc} requires that the last pattern be a pattern variable, which matches any type, serving as the explicit "**`default`**" clause [1].

Type Checking: $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$

(T-ABS)
$$\frac{\Gamma, \mathtt{x} \mapsto \mathtt{T} \vdash \mathtt{e} : \mathtt{T}', \sigma \sim \sigma'}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} : \mathtt{T} \xrightarrow{\sigma \sim \sigma'} \mathtt{T}', \emptyset \sim \emptyset}$$

(T-APP)
$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \xrightarrow{\sigma_0 \sim \sigma_1} \mathtt{T}', \sigma_2 \sim \sigma_3 \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma_4 \sim \sigma_5}{\Gamma \vdash \mathtt{e}\ \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \cup \sigma_4 \sim \sigma_1 \cup \sigma_3 \cup \sigma_5}$$

(T-BOOL)
$$\Gamma \vdash \mathtt{b} : \textbf{Bool}, \emptyset \sim \emptyset$$

(T-SUB)
$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \qquad \mathtt{T} <: \mathtt{T}' \qquad \sigma_0 \subseteq \sigma \qquad \sigma' \subseteq \sigma_1}{\Gamma \vdash \mathtt{e} : \mathtt{T}', \sigma_0 \sim \sigma_1}$$

(T-LET)
$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma, \mathtt{x} \mapsto \mathtt{T} \vdash \mathtt{e}' : \mathtt{T}', \sigma_2 \sim \sigma_3}{\Gamma \vdash \textbf{let}\ \mathtt{x} = \mathtt{e}\ \textbf{in}\ \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3}$$

(T-GET)
$$\frac{\Gamma \vdash \mathtt{e} : \textbf{Ref}_\rho\ \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash ! \mathtt{e} : \mathtt{T}, \sigma \cup \textbf{rd}_\rho \mathtt{T} \sim \sigma' \cup \textbf{rd}_\rho \mathtt{T}}$$

(T-REF)
$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}{\Gamma \vdash \textbf{ref}\ \rho\ \mathtt{T}\ \mathtt{e} : \textbf{Ref}_\rho\ \mathtt{T}, \sigma \cup \textbf{init}_\rho \mathtt{T} \sim \sigma' \cup \textbf{init}_\rho \mathtt{T}}$$

(T-SET)
$$\frac{\Gamma \vdash \mathtt{e} : \textbf{Ref}_\rho\ \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e}' : \mathtt{T}, \sigma_2 \sim \sigma_3}{\Gamma \vdash \mathtt{e} := \mathtt{e}' : \mathtt{T}, \sigma_0 \cup \sigma_2 \cup \textbf{wr}_\rho \mathtt{T} \sim \sigma_1 \cup \sigma_3 \cup \textbf{wr}_\rho \mathtt{T}}$$

(T-VAR)
$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}, \emptyset \sim \emptyset}$$

(T-PARA)
$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e} : \mathtt{T}', \sigma_2 \sim \sigma_3}{\Gamma \vdash \mathtt{e} || \mathtt{e}' : \mathtt{T}', \sigma_0 \cup \sigma_2 \sim \sigma_1 \cup \sigma_3}$$

(T-IF)
$$\frac{\Gamma \vdash \mathtt{e} : \textbf{Bool}, \sigma_0 \sim \sigma_1 \qquad \Gamma \vdash \mathtt{e}_0 : \mathtt{T}, \sigma_2 \sim \sigma_3 \qquad \Gamma \vdash \mathtt{e}_1 : \mathtt{T}, \sigma_4 \sim \sigma_5}{\Gamma \vdash \textbf{if}\ \mathtt{e}\ \textbf{then}\ \mathtt{e}_0\ \textbf{else}\ \mathtt{e}_1 : \mathtt{T}, \sigma_0 \cup (\sigma_2 \cap \sigma_4) \sim \sigma_1 \cup \sigma_3 \cup \sigma_5}$$

Figure 4.7 Typing Rules for Standard Expressions.

Standard Expressions. Other typing rules are mostly conventional and are shown in Figure 4.7. Store operations (T-REF), (T-GET) and (T-SET) compute initialization **init**, read **rd** and write **wr** effects, respectively. The typing of parallel composition is standard.

4.4 The Full-Fledged System

The type system in Figure 4.6 does not provide any static guarantees for expressions guarded by the predicate P in the **effcase** expression. For example, in resource-aware scheduling (Figure 4.1), the programmer may wish to be provided with the *static* guarantee that the order of buffer access is preserved. We support this refined notion of reasoning through refinement types.

We extend the grammar of our language, in Figure 4.8, to allow the programmers to associate an expression with a refinement type, denoting that the corresponding expression must satisfy the predicate in the refinement type through static type checking.

$$
\begin{array}{rcll}
\mathtt{e} & ::= & \ldots \mid \mathfrak{T}\ \mathtt{e} & \textit{extended expression} \\
\mathfrak{T} & ::= & \{\mathtt{T}, \sigma \sim \sigma' | P\} & \textit{refinement type} \\
\mathtt{T} & ::= & \ldots \mid \mathfrak{T} & \textit{type} \\
\Delta & ::= & \overline{\varsigma \mapsto \mathscr{V}} & \textit{polarity environment} \\
\mathscr{V} & ::= & + \mid - \mid * \mid \mathfrak{i} & \textit{polarity}
\end{array}
$$

Subtyping: $\tau <: \tau'$

$$
(subr\text{-REFINE})\ \frac{P \mid\!-\!> P'}{\Gamma \vdash \{\mathtt{T}, \sigma \sim \sigma' | P\} <: \{\mathtt{T}, \sigma \sim \sigma' | P'\}}
$$

For all other ($subr$-*) rules, each is isomorphic to its counterpart (sub-*) rule in Figure 4.5.

Figure 4.8 λ_{fc} Extension with Refinement Types.

A refinement type $\mathfrak{T}$ takes the form of $\{\mathtt{T}, \sigma \sim \sigma' | P\}$, where predicate P is used to refine the base type $\mathtt{T}$ and effects $\sigma \sim \sigma'$, a common notation in refinement type systems [27, 55, 82]. When $\mathtt{T}$ is not referred to in other parts in the refinement type, we shorten the refinement as $\{\sigma \sim \sigma' | P\}$, *e.g.*, in the zeroing example in Figure 4.3, $\{\mathtt{ef} \sim \mathtt{ef}' | \mathbf{wr}_r <<: \mathtt{ef}\}$. We extend the subtyping relation with one additional rule, ($subr$-REFINE). Here, subtyping of refinement types is defined as the logical implication $|\!-\!>$ of the predicates of the two types.

4.4.1 Polarity Support

Polarity environment Δ, which will be used in type checking, maps effect variables ς to polarities $\mathscr{V}$. $\mathscr{V}$ can either be contravariant $+$, covariant $-$, invariant $\mathfrak{i}$ and bivariant $*$. Intuitively, the $+$ comes from the must-effect, $-$ comes from the may-effect. If must- and may-effects are exactly the same, it induces invariant $\mathfrak{i}$, *e.g.*, the predicate ==, which requires the effects of its LHS and RHS to be equal. If ς appears in both must- and may-effects, but the effects are not the same, ς will be bivariant. The subsumption relations of the variances form a lattice, defined in Figure 4.9, with the join $\sqcup$ going "up". Intuitively, an $\mathfrak{i}$ can appear in a position where $+$ is required, thus $\mathfrak{i}$ is a "subtype" of $+$.

For a predicate of arity n, we say its position j $(1 \leq j \leq n)$ to have contravariant polarity $+$ when effect subsumption of argument j is aligned with predicate implication, *i.e.*, an application of this predicate with argument j being σ always implies a predicate application identical with the former except argument j being σ' and $\sigma \subseteq \sigma'$, shown in (MONO-$\boxed{\uparrow}$) in Figure 4.10, where the $\mathbb{V}(\mathbb{R}, j)$

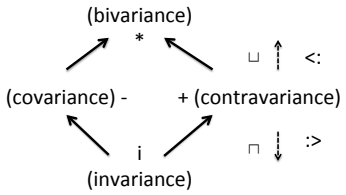

Figure 4.9 Polarity Lattice.

(Figure 4.5) notation gets the j^{th} polarity of the predicate $\mathbb{R}$, *e.g.*, $\mathbb{V}(<<:, 0)$ will return the polarity of LHS of <<:, *i.e.*, $-$ and $\mathbb{V}(<<:, 1)$ will return RHS, *i.e.*, $+$. Similarly, we say the position j of a predicate to have covariant polarity $-$ if an application of this predicate with argument j being σ always implies a predicate application identical with the former except argument j being σ' and $\sigma' \subseteq \sigma$, as (MONO-$\boxed{\downarrow}$). For non-monotone predicates (*e.g.*, ==), the polarities $+$ and $-$ fall short:

EXAMPLE 4.4.1 (Effect Invariant for Non-monotone Effect Predicate) *Programmers wish to check that the effects of two expressions are equal, line 4. The non-monotone (for both LHS and RHS) predicate == is satisfied for the call on line 5, but challenging for a system with co- or contra-variant polarities alone.*

```
1 let buf = ref_r -1 in
2  let fun = λ x:exact, y:exact.
3              effcase x, y:
4              | EC(α0, ς0 ∼ ς0), EC(α1, ς1 ∼ ς1)  where ς0 == ς1 =>{y == x} x; y in
5   fun (query !buf) (query !buf);
6   fun (query buf := 0) (query !buf)
```

$\mathbb{R}$	LHS ($j=0$)	RHS ($j=1$)
<<:	$-$(may)	$+$(must)
/<<:	$+$(must)	$-$(may)
#	$-$(may)	$-$(may)
==	i(may and must)	i(may and must)

Table 4.5 Polarities for Client Predicates ($\mathbb{V}(\mathbb{R}, j)$).

Note that the equivalent of the may-effects (or must-effect) of both sides does not guarantee the equivalent of the traces (runtime memory accesses), *e.g.*, for the following code, the two parameters (line 8) are the same, but their runtime traces are not the same.

```
7 let same = (query if !buf < 0 then buf := 0) in
8  fun same same
```

Predicate Implication: $P \mid\!\!-\!\!> P$

$$\text{(MONO-}\uparrow\text{)}\quad \frac{\mathbb{V}(\mathbb{R}, j) = + \qquad \sigma_j \subseteq \sigma'_j}{\mathbb{R}\ \overline{\sigma}\sigma_j\overline{\sigma'} \mid\!\!-\!\!> \mathbb{R}\ \overline{\sigma}\sigma'_j\overline{\sigma'}}$$

$$\text{(MONO-}\downarrow\text{)}\quad \frac{\mathbb{V}(\mathbb{R}, j) = - \qquad \sigma'_j \subseteq \sigma_j}{\mathbb{R}\ \overline{\sigma}\sigma_j\overline{\sigma'} \mid\!\!-\!\!> \mathbb{R}\ \overline{\sigma}\sigma'_j\overline{\sigma'}}$$

$$\text{(IMP-}\wedge\text{0)}\quad P \wedge P' \mid\!\!-\!\!> P \qquad \text{(IMP-}\vee\text{0)}\quad P \mid\!\!-\!\!> P \vee P'$$

$$\text{(IMP-REFL)}\quad P \mid\!\!-\!\!> P \qquad \text{(IMP-}\wedge\text{1)}\quad P \wedge P' \mid\!\!-\!\!> P' \qquad \text{(IMP-}\vee\text{0)}\quad P' \mid\!\!-\!\!> P \vee P'$$

$$\text{(IMP-TRANS)}\quad \frac{P \mid\!\!-\!\!> P' \qquad P' \mid\!\!-\!\!> P''}{P \mid\!\!-\!\!> P''}$$

Figure 4.10 Predicate Implication $\mid\!\!-\!\!>$.

The **exact** annotation solves this problem. This annotation requires that the may- and must-effects of the expression are the same, inducing invariant. Since the trace is bounded by the same lower and upper bound effects, it is tight. Given that `x==y` and that both `x` and `y` have tight bounds, their traces must be equal.

The must-effect of the expression on line 7 is $\{\mathbf{rd}_r\}$, the may-effect is $\{\mathbf{rd}_r, \mathbf{wr}_r\}$, and thus its effects are not **exact** and the function call, on line 8, fails static type checking. The effects of the four queried expressions on lines 5-6 are **exact**. The calls on those two lines will type check statically. The call on line 5 will satisfy the runtime predicate `==` and execute the code on line 4, while the call on line 8 will not satisfy the predicate and thus not execute the code on line 4. Similarly, the **exact** annotation fulfills a similar task in Figure 4.1. Without **exact**, the program will not be sound.

4.4.2 Refinement Type Checking

Refinement type checking is defined through judgment $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ shown in Figure 4.11, which extends the rules in Figure 4.6 with one additional rule (R-REFINE) for refinement typing and one adaptation rule (R-EFFCASE) for typing predicated effect inspection. The rules ensure that the pattern variables are properly used, *e.g.*, a variable ς with $-$ polarity should not appear in the position where the predicate requires $+$, such as `u0` on line 4 in Example 4.1.1. (R-REFINE) requires that the predicate in the refinement type to be entailed from the predicates in the type environment. Function $[\![\Gamma]\!]$ computes the conjunction of all predicates that appear in the refinement types of Γ [29], defined in Figure 4.12.

The differences between (R-EFFCASE) and (T-EFFCASE) (in §4.3) are highlighted. We compute, via the polar function defined in Figure 4.12, the polarity for each new effect variable appears in the pattern matching types. An effect variable ς appearing in the must-effect, will have $+$ polarity. If ς appears

Refinement Type Checking: $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$

$$\text{(R-REFINE)}\ \frac{\Delta \vdash P \qquad \Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \qquad [\![\Gamma]\!] \mid\!-\!> P}{\Delta; \Gamma \vdash \{\mathtt{T}, \sigma \sim \sigma' | P\}\ \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'}$$

(R-EFFCASE)

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \mathtt{x} : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset \qquad \exists \theta\ .(\theta\mathbf{EC}(\tau_i, \sigma_i \sim \sigma_i') <: \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')), \text{for all } i \in \{1 \ldots n\} \\ \exists \varsigma\ .\ \tau_n = \varsigma \wedge P_n = \emptyset \qquad \Delta_i = \Delta \sqcup \text{polar}_t(\tau_i) \sqcup \text{polar}_e(\sigma_i \sim \sigma_i') \\ \Delta_i \vdash P_i \qquad \Delta_i; \Gamma, \mathtt{x} \mapsto \{\mathbf{EC}(\tau_i, \sigma_i \sim \sigma_i'), \emptyset \sim \emptyset | P_i\} \vdash \mathtt{e}_i : \mathtt{T}, \sigma_i'' \sim \sigma_i''', \text{for all } i \in \{1 \ldots n\}\end{array}}{\Delta; \Gamma \vdash \textbf{effcase}\ \mathtt{x} : \overline{\mathbf{EC}(\mathtt{T}, \sigma \sim \sigma')}\ \textbf{where}\ P \Rightarrow \mathtt{e} : \mathtt{T}, \cap_{i \in \{1 \ldots n\}} \sigma_i'' \sim \cup_{i \in \{1 \ldots n\}} \sigma_i'''}$$

For all other (R-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of judgment $\Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the latter rule should be substituted with $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the former.

Type Checking Predicate: $\Delta \vdash P$

$$\frac{\neg\Delta \vdash P}{\Delta \vdash \neg P} \qquad \frac{\Delta \vdash P \quad \Delta \vdash P'}{\Delta \vdash P \wedge P'} \qquad \frac{\Delta \vdash P}{\Delta \vdash P \vee P'} \qquad \frac{\forall \sigma_j \in \overline{\sigma} \forall \varsigma \in \sigma_j \text{ s.t. } \varsigma \in dom(\Delta).\Delta(\varsigma) <: \mathbb{V}(\mathbb{R}, j)}{\Delta \vdash \mathbb{R}\ \overline{\sigma}}$$

Figure 4.11 Typing Rules for Checking Refinement Types.

Predicate Combination: $[\![\Gamma]\!] = P$

$$[\![\overline{\varsigma \mapsto \{\mathtt{T}, \sigma \sim \sigma' | P\}}]\!] \quad = \quad \bigwedge_{j=1}^{n} P_j$$

Environment Negation: $\neg\Delta = \Delta$

$$\neg\overline{\varsigma \mapsto \mathscr{V}} \quad = \quad \overline{\varsigma \mapsto \neg\mathscr{V}}$$

Polarity Negation: $\neg\mathscr{V} = \mathscr{V}$

$$\begin{array}{rcl} \neg + & = & - \\ \neg - & = & + \\ \neg * & = & * \\ \neg \mathfrak{i} & = & \mathfrak{i} \end{array}$$

Computing Δ from Type: $\text{polar}_t(\mathtt{T}) = \Delta$

$$\begin{array}{rl} \text{polar}_t(\textbf{Bool}) & = \emptyset \\ \text{polar}_t(\alpha) & = \emptyset \\ \text{polar}_t(\textbf{Ref}_\rho\ \mathtt{T}) & = \text{polar}_t(\mathtt{T}) \\ \text{polar}_t(\mathtt{T} \xrightarrow{\sigma \sim \sigma'} \mathtt{T}') & = \text{polar}_t(\mathtt{T}) \sqcup \text{polar}_t(\mathtt{T}') \sqcup \text{polar}_e(\sigma \sim \sigma') \\ \text{polar}_t(\textbf{EC}(\mathtt{T}, \sigma \sim \sigma')) & = \text{polar}_t(\mathtt{T}) \sqcup \text{polar}_e(\sigma \sim \sigma') \end{array}$$

Computing Δ from Effect: $\text{polar}_e(\sigma \sim \sigma) = \Delta$

$$\begin{array}{rcl} \text{polar}_e(\varsigma \sim \varsigma) & = & \varsigma \mapsto \mathfrak{i} \\ \text{polar}_e(\pi_\rho \mathtt{T} \sim \pi'_{\rho'} \mathtt{T}') & = & \emptyset \\ \text{polar}_e(\varsigma \cup \sigma \sim \sigma') & = & \varsigma \mapsto + \sqcup \text{polar}_e(\sigma \sim \sigma') \\ \text{polar}_e(\sigma \sim \varsigma \cup \sigma') & = & \varsigma \mapsto - \sqcup \text{polar}_e(\sigma \sim \sigma') \end{array}$$

Figure 4.12 Functions for Computing Effect Polarity.

in the may-effect, it will have $-$ polarity. If the must- and may-effects are the same ς, ς has $\mathtt{i}$ polarity, otherwise ς has $*$ polarity. We use the computed polarities to check the proper use of the effect variables in each predicate, which is defined in the bottom of the same figure. For example, an effect variable ς with $+$ polarity should not appear in the position where the predicate requires $-$. Most of the checking rules are rather predictable except for the predicate negation. To check the negation, we negate the polarity environment $\neg\Delta$. For example, we require the may-effect of the LHS and must-effect of the RHS of the predicate $<<:$, and for its negation $/<<:$, we require the must-effect of the LHS and may-effect of the RHS. The custom predicates and their polarities specifications are shown in Figure 4.5. The rules associate $\mathtt{x}$ with a refinement type that carries the guarded predicate, P_i in the typing environment, which will be used to check the (R-REFINE) rule.

Effect closures in λ_{fc} are immutable. This language feature significantly simplifies the design of refinement types in λ_{fc}, as the interaction between refinement types and mutable features would otherwise be challenging [27].

4.5 Dynamic Semantics

This section describes λ_{fc}'s dynamic semantics. The highlight is to support runtime effects management and highly precise effects reasoning through dynamic typing.

Semantics Objects. λ_{fc}'s configuration consists of a *store* s, an expression $\mathtt{e}$ to be evaluated, and an effects *trace* f, defined in Figure 4.13. These definitions are conventional. The domain of the store consists of a set of references l. Each reference cell in s records a value, as well as the region r and type $\mathtt{T}$ of the reference. The trace records the runtime accesses to regions along the evaluation, with **init**(r), **rd**(r), and **wr**(r), denoting the initialization, read, and write to r, respectively. Traces only serve a role in the soundness proofs, and thus are unnecessary in a λ_{fc} implementation. More specifically, we will show that the trace is the "realized effects" of the effects computed by λ_{fc}.

The small-step semantics is defined as transition $s;\mathtt{e};f \rightarrow s';\mathtt{e}';f'$. Given a store s and a trace f, the evaluation of an expression $\mathtt{e}$ results in another expression $\mathtt{e}'$, a (possibly updated) store s', and a trace f'. The notation $[\mathtt{x}\backslash v]\mathtt{e}$ substitutes $\mathtt{x}$ with $\mathtt{v}$ in expression $\mathtt{e}$. The notation $\rightarrow^*$ represents the reflexive-transitive closure of $\rightarrow$.

Definitions:

$\mathtt{tc}$	$::=$	$\langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle$	*effect closure*
s	$::=$	$\overline{l \rightarrow_{\langle \mathtt{r},\mathtt{T} \rangle} v}$	*store*
f	$::=$	$\overline{acc(\rho)}$	*trace*
v	$::=$	$\mathtt{b} \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{e} \mid l \mid \mathtt{tc}$	*value*
$\mathbb{E}$	$::=$	$-\mid \mathbb{E}\ \mathtt{e} \mid v\ \mathbb{E} \mid$ **let** $\mathtt{x} = \mathbb{E}$ **in** $\mathtt{e} \mid$ **ref** $\rho\ \mathtt{T}\ \mathbb{E} \mid !\mathbb{E} \mid \mathbb{E} := \mathtt{e} \mid v := \mathbb{E}$	*evaluation context*
	$\mid$	**if** $\mathbb{E}$ **then** $\mathtt{e}$ **else** $\mathtt{e} \mid$ **realize** $\mathbb{E}$	

Dynamic Typing: $s; \Phi; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma$

$$\text{(DT-LOC)}\ \frac{\{l \mapsto_{\langle \rho,\mathtt{T} \rangle} v\} \in s}{s; \Delta; \Gamma \vdash_{\mathtt{D}} l : \mathbf{Ref}_{\mathtt{r}}\ \mathtt{T}, \emptyset \sim \emptyset} \qquad \text{(DT-TYPE)}\ \frac{s; \Phi; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma}{s; \Delta; \Gamma \vdash_{\mathtt{D}} \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle : \mathbf{EC}(\mathtt{T}, \sigma \sim \sigma'), \emptyset \sim \emptyset}$$

For all other (DT-*) rules, each is isomorphic to its counterpart (R-*) rule, except that every occurrence of judgment $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$ in the latter rule should be substituted with $s; \Phi; \Gamma \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma$ in the former.

Evaluation relation: $s; \mathtt{e}; f \rightarrow s'; \mathtt{e}'; f'$

(cxt)	$s; \mathbb{E}[\mathtt{e}]; f \rightarrow s'; \mathbb{E}[\mathtt{e}']; f, f'$ if $s; \mathtt{e} \rightsquigarrow s'; \mathtt{e}'; f'$
(app)	$s; \lambda \mathtt{x} : \mathtt{T}.\mathtt{e}\ v \Rightarrow s; [\mathtt{x} \backslash v]\mathtt{e}; \emptyset$
(let)	$s;$ **let** $\mathtt{x} = v$ **in** $\mathtt{e} \Rightarrow s; [\mathtt{x} \backslash v]\mathtt{e}; \emptyset$
(if)	$s;$ **if** $\mathtt{b}$ **then** $\mathtt{e}_0$ **else** $\mathtt{e}_1 \Rightarrow s; \mathtt{e}; \emptyset$ if $\mathtt{e} = \mathtt{b}\ ?\ \mathtt{e}_0 : \mathtt{e}_1$
(get)	$s; !l \Rightarrow s; v;$ **rd**(r) if $\{l \mapsto_{\langle \mathtt{r},\mathtt{T} \rangle} v\} \in s$
(set)	$s; l := v \Rightarrow s, \{l \mapsto_{\langle \mathtt{r},\mathtt{T} \rangle} v\}; v;$ **wr**(r) if $\{l \mapsto_{\langle \mathtt{r},\mathtt{T} \rangle} v'\} \in s$
(ref)	$s;$ **ref** r T $v \Rightarrow s, \{l \mapsto_{\langle \rho,\mathtt{T} \rangle} v\}; l;$ **init**(r) if $l = freshloc()$
(par)	$s; \mathtt{e} \| \mathtt{e}' \Rightarrow s; \mathtt{e}_0; \emptyset$ if $\mathtt{e}_0 = (\mathtt{e}; \mathtt{e}')$ or $(\mathbf{let}\ \mathtt{x} = \mathtt{e}'\ \mathbf{in}\ \mathtt{e}; \mathtt{x})$
(qry)	$s;$ **query** $\mathtt{e} \Rightarrow s; \langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle; \emptyset$ if $s; \emptyset; \emptyset \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$
$(real)$	$s;$ **realize** $\langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle \Rightarrow s; \mathtt{e}; \emptyset$
$(effc)$	$s;$ **effcase** $\langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle : \overline{\mathtt{T}}$ **where** $\overline{P \Rightarrow \mathtt{e}} \Rightarrow s; \theta\mathtt{e}_i; \emptyset$ if $\mathtt{T} <: \theta\mathtt{T}_i \wedge \theta P_i \wedge\ \forall j < i, \nexists \theta'\ .\ \mathtt{T} <: \theta'\mathtt{T}_j \wedge \theta' P_j$
$(rfmt)$	$s; \{\mathtt{T}, \sigma_0 \sim \sigma_1 \mid P\}\ \mathtt{e} \Rightarrow s; \mathtt{e}; \emptyset$ if $s; \emptyset; \emptyset \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma' \wedge \sigma_0 \subseteq \sigma \wedge \sigma' \subseteq \sigma_1 \wedge P$

Figure 4.13 λ_{fc} Operational Semantics.

We highlight the first-class effects expressions.

Effect Querying as Dynamic Typing. The (QRY) rule illustrates the essence of λ_{fc}'s effect querying. An effect query produces an effect closure, which encapsulates the queried expression and its runtime type-and-effect.

Dynamic typing, defined in Figure 4.13, is used to compute the effects of the queried expression. The dynamic type derivation has the form $s; \Phi; \Gamma \vdash_{\mathrm{D}} \mathtt{e} : \mathtt{T}, \sigma$, which extends static typing with two new rules for reference value and effect closure typing.

At runtime, the free variables of the expressions will be substituted with values, *e.g.*, in (LET) and (APP). Thus, $\mathtt{e}$ in **query** $\mathtt{e}$ is no longer the same as what it was in the source program. These substituted values carry more precise types, regions, and effects information, bringing to λ_{fc} a highly precise notion of effects reasoning comparing to a static counterpart. Also the dynamic typing does not evaluate the queried expression to compute effects, *i.e.*, λ_{fc} is not an *a posteriori* effect monitoring system.

Effect Realization. Dual to the effect querying rule is the realization (REAL) rule, which "eliminates" the effect closure $\langle \mathtt{e}, \mathtt{T}, \sigma, \sigma' \rangle$ and evaluates the passenger expression $\mathtt{e}$. To show the *validity* of the effect query in first-class effects, we will prove in Theorem 4.6.2, that if $\mathtt{e}$ is evaluated, its effects will fall within the lower σ and upper bound σ' effects. When the evaluation terminates, it reduces to a value of type $\mathtt{T}$, specified in the closure.

Effect-Guided Programming via Predicated Effect Inspection. The (EFFC) rule inspects the type-and-effect of the closure. It searches the first matching target types and returns the corresponding branch expression $\mathtt{e}_i$. Such type must be refined by the inner type of the effect closure, with proper "alignment" by substituting the pattern effect variables in the target type with the corresponding effects. It also requires that the substitution satisfies the target programmer-defined effect analyses predicate P_i.

Refinement expressions. The (RFMT) rule models refinement expressions. It retrieves the dynamic effects of the subexpression and checks that they are the same as specified in the refinement type and that the predicate is satisfied. The trace soundness property of our system guarantees that refinement type checking at runtime (see Theorem 4.6.16) always succeeds. Therefore, these runtime types checking can be treated as no-op.

Parallelism. The (PAR) rule simulates parallelism. Its treatment is standard, it nondeterministically reduces to the sequential compositions $\mathtt{e}; \mathtt{e}'$ or $\mathbf{let}\ \mathtt{x} = \mathtt{e}'\ \mathbf{in}\ \mathtt{e}; \mathtt{x}$. This treatment lets the result of $\mathtt{e}'$

be the final result. Due to the safety guarantee from §4.4, these two forms will reduce to the same result [16] upon termination.

4.6 Meta-theory

This section proves the formal properties of λ_{fc}. We first show the standard soundness property (§4.6.1). Next, we prove that the effect carried in the effect closure is valid (§4.6.2). Finally, we present important trace consistency results for λ_{fc} in §4.6.3.

Before we proceed, let us first define two terms that will be used for the rest of the section.

Definition 4.6.1 *[Redex Configuration] We say* $\langle s; e; f\rangle$ *is a redex configuration of program* e'*, written* $e' \trianglerighteq <s, e, f>$*, iff* $\vdash e' : T, \sigma \sim \sigma'$, $\emptyset; e'; \emptyset \rightarrow^* s; \mathbb{E}[e]; f$. *When* e' *is irrelevant, we shorten it as* $\trianglerighteq <s, e, f>$.

Next, let us define relation $s \vdash f : \sigma$, which says that dynamic trace f *realizes* static effect σ and σ' under store s:

Definition 4.6.2 *[Effect-Trace Consistency]* $s \vdash f : \sigma$ *holds iff the following two conditions hold.* ❶ $\mathrm{acc}(r) \in f$ *implies* $\pi_r T \in \sigma'$*; and* ❷ $\pi_r T \in \sigma$ *implies* $\mathrm{acc}(r) \in f$.

That is, the runtime trace falls within the computed must-may effects.

4.6.1 Type Soundness

Our soundness proof is constructed through standard subject reduction and progress:

Lemma 4.6.3 (Type Preservation) *If* $\triangleright s$, $s; \emptyset; \emptyset \vdash_D e : T, \sigma_0 \sim \sigma_1$ *and* $s; e; f \rightarrow s'; e'; f'$*, then* $\triangleright s'$, $s'; \emptyset; \emptyset \vdash_D e' : T', \sigma_2 \sim \sigma_3$ *and* $T' <: T$.

Proof. The proof is by case on the reduction step applied, and by the typing derivation of $s; \emptyset; \emptyset \vdash_D e : T, \sigma_0 \sim \sigma_1$. □

Lemma 4.6.4 (Progress) *If* $s; \emptyset; \emptyset \vdash_D e : T, \sigma \sim \sigma'$ *then either* e *is a value, or* $s; e; f \rightarrow s'; e'; f'$ *for some* s', e', f'.

Proof. The proof is by case on the reduction step applied, and by the typing derivation of $s; \emptyset; \emptyset \vdash_{\mathtt{D}} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$. □

Theorem 4.6.5 (Type Soundness) *Given an expression* $\mathtt{e}$*, if* $\vdash \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$*, then either the evaluation of* $\mathtt{e}$ *diverges, or there exist some* s*,* v*, and* f *such that* $\emptyset; \mathtt{e}; \emptyset \rightarrow^* s; v; f$.

Proof. Immediately followed from Lemma 4.6.3 and Lemma 4.6.4. □

4.6.2 Query-Realize Correspondence

In this section, we establish the validity of effect closures, *i.e.*, the passenger expression always has the effect carried by the closure, regardless of how the closure has been passed around or stored. The key insight for the proof is the nature of dynamic typing: given an expression $\mathtt{e}$, its type and effect depend upon the environment Γ and the types of the store s, but not the values in s. Ground expressions do not have free variables [85], whose typings are independent of Γ. At runtime, the types in s does not change, and all the redex expressions are ground expressions, thus the dynamic typing of an expression is valid throughout the program.

Lemma 4.6.6 (Dynamic Typing Preservation) *If* $s; \emptyset; \emptyset \vdash_{D} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$*, and* $s; \mathtt{e}; f \rightarrow^* s'; \mathtt{e}'; f'$ *then* $s'; \emptyset; \emptyset \vdash_{D} \mathtt{e} : \mathtt{T}, \sigma \sim \sigma'$.

Proof. The proof is by induction on the number of steps applied. □

We can establish the *uniformity* of effect closure typing:

Lemma 4.6.7 (Uniform Effect Closure Typing) *Given a store* s*, a trace* f*, an expression* $\mathtt{e} = \mathbb{E}[\langle \mathtt{e}', \mathtt{T}, \sigma, \sigma' \rangle]$*, such that* $\trianglerighteq <s, \mathtt{e}, f>$*, then* $s; \emptyset; \emptyset \vdash_{D} \mathtt{e}' : \mathtt{T}, \sigma \sim \sigma'$.

Proof. The proof is by induction on the number of reduction steps applied and Lemma 4.6.6. □

A corollary of this lemma is that the effect carried in the closure is valid throughout the entire lifespan from its introduction (effect query) to elimination (effect realization):

Corollary 4.6.8 (Query-Realize Correspondence) *Given a store* s*, a trace* f*, an expression* $\mathtt{e} = \mathbb{E}[\textbf{\textit{query}}\ \mathtt{e}']$*, such that* $\trianglerighteq <s, \mathtt{e}, f>$*. If*

$$s; \mathtt{e}; f \rightarrow s; \mathbb{E}[\langle \mathtt{e}', \mathtt{T}, \sigma, \sigma' \rangle]; f \rightarrow^* s'; \mathbb{E}'[\textbf{\textit{realize}}\ \langle \mathtt{e}', \mathtt{T}, \sigma, \sigma' \rangle]; f'$$

then $s; \emptyset; \emptyset \vdash_{D} \mathtt{e}' : \mathtt{T}, \sigma \sim \sigma'$ *and* $s'; \emptyset; \emptyset \vdash_{D} \mathtt{e}' : \mathtt{T}, \sigma \sim \sigma'$.

4.6.3 Trace Consistency

First, let us state a simple property of our trace, in that it is monotonically increasing over the reduction sequence:

Lemma 4.6.9 (Monotone Traces) *If* $s; e; f \to s'; e'; f'$ *then* $f' = f, f''$ *for some* f''.

Proof. By the definition of the semantics and (CXT). □

Next we show that the must-effect is always a subset of the may-effect.

Lemma 4.6.10 (Normality of Double-Bounded Effects) *If* $\vdash e : T, \sigma_0 \sim \sigma_1$, $e \trianglerighteq \langle s, e', f \rangle$ *and* $s; \emptyset; \emptyset \vdash_{\mathtt{D}} e' : T, \sigma \sim \sigma'$ *then* $\sigma \subseteq \sigma'$.

Proof. The proof proceeds by structural induction on the derivation of $s; \emptyset; \emptyset \vdash_{\mathtt{D}} e : \mathtt{T}, \sigma \sim \sigma'$ and by cases based on the last step in that derivation. □

The essence of refinement type for potential unsafe expression lies in the fact that through runtime effect reflection (dynamic typing), every instance of evaluation of the $\mathfrak{T}\ e$ expression in the reduction sequence must be "safe," where "safety" is defined through the $\mathfrak{T}$ refinement type given by programmers. To be more concrete:

Definition 4.6.11 (Effect Soundness of Refinement Type) *We say* e *is effect-sound iff for any redex configuration such that* $\trianglerighteq \langle s, \mathfrak{T}\ e, f \rangle$, $\mathfrak{T} = \{T', \sigma_0 \sim \sigma_1 | P\}$, $s; \emptyset; \emptyset \vdash_{\mathtt{D}} e : T, \sigma \sim \sigma'$ *it must hold that* $T <: T'$, $\sigma_0 \subseteq \sigma$, $\sigma' \subseteq \sigma_1$, *and* P.

Effect-based soundness is a corollary of type soundness:

Corollary 4.6.12 (λ_{fc} Effect-based Soundness) *If* $\vdash e : T, \sigma \sim \sigma'$, *then* e *is effect-sound.*

The above property only ensures that the dynamic effects, computed by the dynamic typing, are sound. Our ultimate goal is to enforce, at runtime, the trace through the evaluation of e, respects the refinement type $\mathfrak{T}$.

To achieve this goal, we define *trace for expression*:

Definition 4.6.13 (Trace from Effect Closure) *We say* f *is a trace for expression* e *under store* s, *written* $f \propto \langle e, s \rangle$, *iff* $s; e; f' \to^* s'; v; f', f$.

We now define the consistency over traces:

Definition 4.6.14 (Trace Consistent) *We say* e *is trace-consistent if for any* $\trianglerighteq <s, \mathfrak{T}\ e, f'>$, $\mathfrak{T} = \{T, \sigma \sim \sigma' | P\}$, *and* $f \propto <e, s>$, *then* $[\sigma \backslash f][\sigma' \backslash f]P$ *holds.*

To prove trace-based consistency, the crucial property we establish is:

Lemma 4.6.15 (Effect-Trace Consistency Preservation) *If* $s; \emptyset; \emptyset \vdash_D e : T, \sigma \sim \sigma'$ *and* $s; e; f \rightarrow^* s'; v; f, f'$ *then* $s' \vdash f' : \sigma\sigma'$.

Proof. The proof is by induction on the number of reduction steps applied. □

Finally, we can prove the intuitive notion of soundness of first-class effects:

Theorem 4.6.16 (λ_{fc} Trace-Based Consistency) *If* $\vdash e : T, \sigma \sim \sigma'$, *then* e *is trace-consistent.*

Proof. The proof is by Lemma 4.6.15 and by Definition 4.6.14. □

4.7 Related Work

We are unaware of type-and-effect systems where the (pre-evaluation) effect of an expression is treated as a first-class citizen. The more established route is to treat the post-evaluation effect (in our terms, trace) as a first-class value. In Leory and Pessaux [65], exceptions raised through program execution are available to the programmers. This work has influenced many exception handling systems such as Java, where `Exception` objects are also values. Bauer *et al.* [11] extends the first-class exception idea and allows the programmer to annotate an arbitrary expression as effect and upon the evaluation of that expression, control is transferred to a matching `catch`-like handling expression as a first-class value. Similar designs also exist in implicit invocation and aspect-oriented systems [93]. In general, the work cited here, albeit bearing similar terms, has a distant relationship with our work.

Bañados *et al.* [10] extends the idea of gradual typing [106] to develop a gradual effect system and later Toro *et al.* [116] provides an implementation, which allows programmers customize effect domains. A program element in their system may carry unknown effects, which may become gradually known at runtime. Compared with these systems, effect reflection is a first-class programming

abstraction in our system, and effect closures are first-class values. This leads to a number of unique contributions we summarized at the beginning of this chapter.

There is a large body of work of purely static or dynamic systems for effect reasoning. Examples of the former include Lucassen [72], Talpin *et al.* [113], Marino *et al.* [73], Task Types [62] and DPJ [16]. The latter is exemplified by TWEJava [59], and Legion [117]. Along these lines, Nielson *et al.* [80] is among the most well known foundational system.

The may/must-effect duality may be unique to our system, but double-bounded types is not new. For example, in Java generics, Type bound declarations `super` and `extends` are available for generic type variables [31, 107], the dual bounds in the Java nominal type hierarchy. Unique to our (effect) system is that the type checking process actively computes and tightens the bounds of effects — *e.g.*, the type checking rules of branching and effect inspection — suitable for constructing a precise and intuitive effect programming and reasoning system. The use of must-effects to enhance expressiveness and in combination with the may-effects to enable non-monotone effect operator in our system may be unique, but type system designers should be able to find conceptual analogies in existing systems, such as liveness [52] and obligation [18].

The `typecase`-style runtime type inspection by Harper and Morrisett [57] and Crary *et al.* [32] is an expressive approach to perform type analyses at runtime. Our **effcase** $\texttt{x} : \overline{\texttt{T}\ \textbf{where}\ P \Rightarrow \texttt{e}}$ expression shares the same spirit, except that it works on effect closures. In addition, our pattern matching may be guarded by a predicate, which on the high level can be viewed as an (unrolled/explicit) form of predicate dispatch [44, 74]. In general, supporting expressive pattern matching has a long tradition for data types in functional languages, with many developments in object-oriented languages as well (*e.g.*, [1]). Our work demonstrates how predicated pattern matching interacts with refinement types, dynamic typing, double-bounded effect analysis, and first-class effect support.

Refinement types and their variants [29, 50, 55, 82] have received significant attention, with much progress on their expressiveness and decidability. Effect closures in first-class effects are immutable. This language feature significantly simplifies the design of refinement types in λ_{fc}, as the interaction between refinement types and mutable features is known to be challenging [27]. λ_{fc} demonstrates the opportunity of bridging predicated effect inspection and refinement types.

4.8 Summary

We describe a new foundation for effect programming and reasoning, where effects are available as first-class values to programmers. Our system is powered by the subtle interaction among powerful features such as dynamic typing, double-bounded effects, polarity support in predicates, and refinement types. We demonstrate the applications of first-class effects in designing effect-aware scheduler, data zeroing, and version consistent software update.

CHAPTER 5. AN EFFECT SYSTEM FOR ASYNCHRONOUS, TYPED EVENTS

Event-driven systems, or implicit invocation (II) systems, are popular because of their flexibility and modularity benefits [45, 81, 93, 112]. In these systems, there are two major sets of modules: subjects and handlers. Subjects dynamically announce events, and handlers are implicitly invoked when these events are announced.

Exposing concurrency between handlers in II systems is important because it helps improve responsiveness and scalability [102]. Implicit concurrency between handlers can also help reconcile modularity and concurrency goals [69, 89, 91]. However, implicit concurrency remains a challenge fraught with perils such as data races [99, 102]. Data races, which compromise concurrency safety, happen when two concurrent handlers access the same memory location and at least one of them is a write [48]. A type-and-effect system, or effect system for short, may help understand and avoid these problems because it provides information that encodes and approximates the memory accesses of the handlers [72, 113].

Improving the precision of purely static effect systems is a worthy direction, but looking forward, we believe that they are unlikely to benefit II systems. Our belief is shaped by two insights. First, the configuration of handlers is statically unknown because the decoupling mechanism in II systems allows handlers to be *oblivious* [93] and typically, II systems allow handlers to be dynamically registered [99]. Second, taking the effects of all handlers as an approximation for effect analysis could be over-conservative, as a handler will not execute until after it has registered with the announced event. Therefore, considering the effect of a handler before it registers can be imprecise.

5.1 Asynchronous, Typed Events

In this chapter, we introduce *asynchronous, typed events*, ATE in short, a system that can analyze the effects of the handlers at runtime to compensate for the conservativeness of static effect analyses. Asynchronous, typed events build on the notion of quantified, typed events in the Ptolemy language

[92, 93]. Ptolemy language was in turn based on the notion of crosscutting programming interfaces (XPIs) [54, 110, 111, 119] and the unified programming model [87, 95, 96, 97]. It greatly improves implicit concurrency. To illustrate, consider the following example:

EXAMPLE 5.1.1 (Effect inspection for implicit concurrency) *In the following program, we would like to decide whether the handlers of the event* `Ev` *can be run concurrently, when* `Ev` *is announced at line 3. In ATE, the announcement at line 15 triggers the event handling mechanism, which will run the two handlers* `r1` *and* `r2` *concurrently. The event announcement at line 17 will run* `r1` *and* `r2` *concurrently and, upon termination, execute* `w`*. Concurrency is improved since* `r1` *and* `r2` *are run concurrently. Concurrency safety is preserved because the conflicting handler* `w` *will not run until* `r1` *and* `r2` *are done.*

Server

```
1  event Ev { Number i; }
2  class Number { int val; }
3  class S { void s(Number k) { announce Ev(k); } }
```

Client

```
4  class Read {
5    void reg() { register this.r with Ev; }
6    int r(Number i) { return i.val; } // read effect
7  }
8  class Write {
9    void reg() { register this.w with Ev; }
10   void w(Number i) { i.val = 1; }   // write effect
11 }
12 S s = new S(); Number k = new Number();
13 Read r1 = new Read(); Read r2 = new Read();
14 r1.reg(); r2.reg();
15 s.s(k);                          // line 3, announce
16 Writer w = new Write(); w.reg();
17 s.s(k);                          // line 3, announce
```

Static effect systems The (tricky) concurrency decision of whether to run the handlers concurrently at line 3 depends on the handler configuration, *i.e.* which handlers are registered and in what order, and whether their effects conflict. Due to the following reasons, a static effect system is likely to execute all the handlers sequentially. First, the handler configuration may not be known when the

announce expression at line 3 is compiled, due to the decoupling offered by the event type `Ev` [93]. Second, even if all the handlers are known (probably using a whole program analysis, a nemesis for modularity), using the effects of all the handlers as an approximation could be overly conservative for choosing between a concurrent execution and a serial execution. For example, the handlers of the event `Ev` are `r1`, `r2` and `w`. The effects of both `r1` and `r2` are reading `i.val` and the effect of `w` is writing `i.val`, creating a read-write conflict [48]. Therefore, a serial execution has to be used at line 3.

Asynchronous, typed events The serial execution could be over-conservative because at line 15, the conflicting handler `w` has not registered yet and nonconflicting handlers `r1` and `r2` can be run concurrently. The root cause of the conservativeness is that the concurrency decision depends on the handler configuration, which is not known until runtime.

This is precisely where ATE is more effective compared to existing techniques. It can analyze the effects of the handlers at runtime, which enables precise effect computation.

Internally, ATE maintains a *handler queue* for each event. The queue will be mutated and expanded with handler registrations. For instance, before the announcement at line 15, the handler queue has two handlers, `r1` and `r2`. Upon announcement, through dynamic typing, ATE computes the effects of each handler in the queue — gaining highly precise runtime information. Concurrency decisions are guided by these effects, *i.e.*, handlers are run concurrently if the effects do not conflict, *e.g.*, `r1` and `r2`, or else sequentially, *e.g.*, `r1` and `w`. Concurrency is improved with no loss of soundness, as `r1` and `r2` are run concurrently and conflicting handler `w` will not run until `r1` and `r2` are done.

5.2 Technical Highlights

It turns out that it is challenging to integrate dynamic typing into an event-driven system that allows handlers to register dynamically. We now explain the technical difficulties.

Intensional effect inspection At the highest level, ATE shares the philosophy with other dynamic effect systems [59, 117]. However, these systems may not improve the concurrency for II systems where handlers can register dynamically. The reason is that they do not inspect *mutable data structures* (*i.e.*, the mutable handler queue) and thus miss concurrency opportunities. Mechanically extending

these systems could have undesirable consequences, because of the long-standing problem in type-and-effect systems: reasoning about polymorphic mutable data structures is notoriously difficult [85, 113]. Consider the following example:

EXAMPLE 5.2.1 (Post inspection modification) *In the example below, right before the event (`Ev`) announcement at line 15, there are two handlers, an instance `v` of `Subtlety` (line 13) and an instance `r` of `Read` (line 14), in the queue for `Ev`. It may be tempting to conclude that `r` and `v` can be run concurrently, because `v` (lines 9-10) does not have any (direct) read/write effect. A careful reader may say that `v` will* ***announce*** *`Ee` and its handlers could conflict with `r`. Observe that the queue for `Ee` is still empty. Thus, no registered handler interferes with `r`. So intuitively, it is "safe".*

Client2

```
1  event Ee { Number i; }
2  class Hide {
3    void reg() { register this.h with Ee; }
4    void h(Number i) { i.val = 1; }          // write effect
5  }
6  class Subtlety {
7    void reg() { register this.v with Ev; }
8    void v(Number i) {
9      Hide h = new Hide().reg(); // register hidden handler
10     announce Ee(i);
11   }
12 }
13 Subtlety v = new Subtlety().reg();
14 Read r = new Read().reg();
15 new S().s(new Number());
```

Although tempting, the intuition is unsound. When `v` executes, before announcing `Ee` at line 10, it registers a handler `h` (line 9) for `Ee` — dynamically modifying the mutable handlers queue. The **announce** will now execute `h`, which conflicts with `r`, causing unsafe parallelism. This contradicts our belief that there is no conflicting handler.

The root cause of the problem is unsound reasoning about mutable handler queues and their modification post runtime effect inspection.

To solve the problem, ATE introduces two kinds of effects, namely **`reg`** for the **register** expression and **`ann`** for the **announce** expression. The **`ann`** effect approximates the effects of potential handlers

for the to-be-announced event. The **reg** effect captures the modification of handler queues and comes with the (latent) effects [113] of the to-be-register handler, and is the effect incurred when the handler is invoked, *i.e.*, at event announcements. When these two effects combine, the (latent) effects of potential handlers will also be included (detailed in §5.4), *e.g.*, the combination of the effects of line 9 and 10 will include the effects of `h`, *i.e.*, the effects of `v` include the **reg**, **ann** and the write effect from `h`. Now, the effects of `v` and `r` conflict and ATE runs them serially, which is desirable for concurrency safety.

Here ATE's static and dynamic systems interact in interesting ways. Dynamic typing provides precise effects — exploiting runtime information — allowing more concurrency opportunities. Static typing precomputes the effects of the handlers — avoiding potential expensive effect computation at runtime — and soundly captures the dynamic modification of the queues, via the **reg** and **ann** effects, which is good news for reasoning about mutable queues.

Modular reasoning Next, we show an important modularity benefit of ATE's design. Modular reasoning about concurrency [89, 91] could be challenging, due to the well known *pervasive interference* problem [7, 62, 121], defined below.

Definition 5.2.2 *[Pervasive interference] Pervasive interference in a concurrent II program means that between* any two consecutive expressions *of a handler* `h`*, interleaving expressions of another handler could change the states of* `h` *and influence* `h`*'s subsequent behavior.*

To illustrate, consider the following program. The handler `addThenAnnounce` increases the input `Number` by 1 (line 3, using the standard read-increase-set expressions [121]) and announces an event with the modified `Number`.

EXAMPLE 5.2.3 (Low interference density) *Any two consecutive reads of the variable* `j` *at line 3 could result in two completely different integers because of the potential interference of other handlers. This problem is manifested by adding the* interference points *($^{\alpha}$) to the source program.*

```
1  int j;
2  void addThenAnnounce (Number i) {
3    ^α j = i^α.val; j ^α= j^α + 1; i.val ^α= ^α j;
4    announce[i] Ee(i);
5  }            // the less interference points, the better
```

In ATE, the number of interference points is 1 (ℹ), instead of 7 ($^{\alpha}$ and ℹ), i.e.*, between every consecutive expressions. Code that lies within any pair of interference points is a* transaction *or an* atomic *block and thus can be reasoned about sequentially [121]. With ATE, programmers reap the benefits of atomicity when reasoning about handlers.*

In a naive extension of an II language with concurrency but without safety guarantees, programmers must consider all other handlers to determine whether their interleavings would be harmful at every program point. This is illustrated by all $^{\alpha}$ interference points which show global reasoning is required to analyze this program, rather than the modular reasoning we desire. Thanks to the concurrency safety guarantees, ATE controls the interference points ℹ to only after the **announce** expression, instead of every expression.

5.3 A Calculus with Asynchronous Typed Events

The abstract syntax of our calculus that supports asynchronous, typed events is defined in Figure 5.1. Our calculus is built on top of an imperative object-oriented calculus, and Ptolemy [30, 92, 93]. Key language features are highlighted in blue, which support safe implicit concurrency, and in red, which are challenging for a concurrent II language.

5.3.1 Expressions

The syntax includes conventional OO expressions. The highlight of ATE is a few interconnected features:

Dynamic event registration The **register** expression registers handlers with events dynamically (*e.g.*, line 5 in Figure 5.1.1). As shown in Example 5.2.1, reasoning about the concurrency safety of an II language with dynamic event registration is challenging. The tricky problem is that the concurrency safety depends on the configuration of the handlers, which is not known until event registration at runtime. The registration-time specialization in §5.5 solves this problem.

Implicit concurrency via event announcement The **announce** expression is the source of implicit concurrency. At runtime, it inspects the effects of each handler and schedules nonconflicting

prog	::=	$\overline{\texttt{decl}}$ e	program
decl	::=	**class** c **extends** d {$\overline{\texttt{fld}}$ $\overline{\texttt{meth}}$}	class
	|	event p {$\overline{\texttt{form}}$}	event
fld	::=	c f **in** ρ	field
meth	::=	c m($\overline{\texttt{form}}$){e}	method
t	::=	c | **void**	type
form	::=	c x, **where** $x \neq$ **this**	parameter
e	::=	form=e;e | x | **null** | e.m($\overline{\texttt{e}}$)	expression
	|	e.f | e.f=e | **new** c()	reference
	|	**yield** e	cooperation
	|	**register this**.m **with** p	registration
	|	**announce** p($\overline{\texttt{e}}$)	announcement

where

c, d	$\in$	$\mathcal{C}$,	the set of class names
p	$\in$	$\mathcal{P}$,	the set of event names
f	$\in$	$\mathcal{F}$,	the set of field names
m	$\in$	$\mathcal{M}$,	the set of method names
x	$\in$	$\{\textbf{this}\} \cup \mathcal{V}$,	the set of variable names
ρ	$\in$	$\mathcal{R}$,	the set of region names

Figure 5.1 ATE's Abstract Syntax.

handlers to run concurrently. Two handlers may interfere, referred to as *conflicting handlers*, if their effects access the same memory location and at least one of the accesses is a write [48]. The runtime manages the details of concurrency to relieve programmers from the burden of explicitly managing threads and locks.

Modeling concurrency via cooperative handlers To model concurrency and rigorously prove the properties of ATE, we introduce the **yield** expression to simulate *cooperative handlers* [3, 121]. It may not be used in source programs but serves as an intermediate expression in the semantics (§5.5), which is used to allow other handlers to run, *i.e.*, a handler can explicitly yield control to other handlers.

The introduction of cooperative handlers could complicate modular reasoning due to the well known *pervasive interference* problem [7, 62, 121] (see Example 5.2.3). This problem is manifested by adding the **yield** expression (shown as $^{\alpha}$ in Example 5.2.3), referred to as *interference points*, to the source program. We will show in §5.6.5 that ATE controls and limits the interference points to only after the **announce** expression, instead of every expression.

5.3.2 Declarations

A program consists of a sequence of declarations followed by an expression, which can be thought of as the body of a "main" method.

The event type (`event`) declaration facilitates the implicit invocation design style [17, 45, 81, 93, 108], whose intention is to provide a named abstraction for a set of events.

Class declarations are standard except that each field is associated with a *region name* [53, 72, 113], a common way of abstracting memory locations for effect systems to reason about memory accesses. For the examples where regions are not explicitly annotated, different region names suffice.

5.4 Type and Static Effect Computation

We now describe ATE's type system, which computes precise effects for handlers. The dynamic semantics (§5.5) will use these effects to determine a safe order for handler invocation and to improve concurrency. The highlight is new effects to approximate the modification of handler queues.

5.4.1 Effects Reasoning for Mutable Handler Queue

Compared with previous work on static effect reasonings [72, 113], effects of handlers are constantly changing in event-driven systems (Example 5.1.1 and 5.2.1), due to runtime event registrations. The handlers of an event are statically unknown. To tackle the problem, ATE introduces two new effects, the *announce* and *register* effects, expressed as **`ann`** and **`reg`**. An **`ann`** effect serves as a place holder for the concrete effects of zero or more registered handlers and is made concrete during handler registration at runtime (§5.5).

5.4.2 Type and Effect Attributes, and Effect Interference

The type attributes used by the type system are defined in Figure 5.2. The type attributes for expressions are represented as (t, σ): the type t of an expression and its effect set σ.

The interference relation is shown in Figure 5.3. Read effects do not conflict with each other. Write effects conflict with read and write effects accessing the same region. Event registration **register** will modify the event queue to append the new handler and the **announce** expression will read the queue

$$
\begin{array}{lcll}
\theta & ::= & \mathrm{OK} & \textit{decl type} \\
 & | & \overline{t} \xrightarrow{\sigma} t \ \mathbf{in}\ c & \textit{method type} \\
 & | & t, \sigma & \textit{expression type} \\
\sigma & ::= & \overline{\epsilon} & \textit{program effect} \\
\epsilon & ::= & \mathbf{rd}\ \rho & \textit{read effect} \\
 & | & \mathbf{wr}\ \rho & \textit{write effect} \\
 & | & \texttt{ann}\ p & \textit{announce effect} \\
 & | & \texttt{reg}\ p\ \sigma & \textit{register effect} \\
\Pi & ::= & \overline{x : t} & \textit{type environment}
\end{array}
$$

Figure 5.2 Type-and-effect Attributes.

to execute the registered handlers. Similar to read/write effects, `reg` conflicts with each other and `ann` accessing the same event p.

Noninterfering Effects, $\sigma \sharp \sigma$:

$$
\emptyset \sharp \sigma \qquad \frac{\sigma \sharp \sigma'' \qquad \sigma' \sharp \sigma''}{(\sigma \cup \sigma') \sharp \sigma''} \qquad \frac{\sigma \sharp \sigma'}{\sigma' \sharp \sigma} \qquad \mathbf{rd}\ \rho \sharp \mathbf{rd}\ \rho' \qquad \frac{\rho \neq \rho'}{\mathbf{rd}\ \rho \sharp \mathbf{wr}\ \rho'} \qquad \frac{\rho \neq \rho'}{\mathbf{wr}\ \rho \sharp \mathbf{wr}\ \rho'}
$$

$$
(\texttt{reg}\ p\ \sigma / \texttt{ann}\ p) \sharp \mathbf{wr}/\mathbf{rd}\ \rho \qquad \frac{p \neq p'}{\texttt{reg}\ p\ \sigma \sharp \texttt{ann}\ p'} \qquad \frac{p \neq p'}{\texttt{reg}\ p\ \sigma \sharp \texttt{reg}\ p'\ \sigma'} \qquad \texttt{ann}\ p \sharp \texttt{ann}\ p'
$$

Figure 5.3 Effect Noninterference.

Notations The notation $t' <: t$ means t' is a subtype of t. It is the reflexive-transitive closure of the declared subclass relationships. We state the type checking rules using a fixed class table (list of declarations CT [60]). The typing rules for expressions use a type environment, Π, which is a finite partial mapping from variable names x to types t.

5.4.3 Expressions

The rules for expressions are rather conventional, shown in Figure 5.4. Rules (T-GET) and (T-SET) for store operations produce the read and write effects, respectively. We highlight the interesting rules. The (T-YIELD) says that a **yield** expression has same type and effect as the expression e.

The (T-REGISTER) says that the effect of a register expression is a register effect `reg` associated with the effects of the to-be-register handler, to model handler queue modification, *e.g.*, in Example 5.2.1,

Typing: $\Pi \vdash e : t, \sigma$

(T-YIELD)
$$\frac{\Pi \vdash e : t, \sigma}{\Pi \vdash \textbf{yield}\ e : t, \sigma}$$

(T-ANNOUNCE)
$$\frac{\textbf{event}\ p\ \{\overline{t\,x}\} \in CT \qquad \forall\, t_i\, x_i \in \overline{t\,x} \text{ s.t. } \Pi \vdash e_i : t_i', \sigma_i \ \wedge\ t_i' <: t_i}{\Pi \vdash \textbf{announce}\ p\ (\overline{e}) : \textbf{void}, \overline{\sigma} \sqcup \textbf{ann}\ p}$$

(T-REGISTER)
$$\frac{\Pi \vdash \textbf{this} : t, \emptyset \quad (c, t'', m\ (\overline{t'\,x})\ \{e\}, \sigma) = \mathit{find}(t, m) \qquad \textbf{event}\ p\ \{\overline{t\,x}\} \in CT \qquad \forall t \in \overline{t}\ .\ t' <: t}{\Pi \vdash \textbf{register this}.m\ \textbf{with}\ p : t, \textbf{reg}\ p\ \sigma}$$

(T-VAR)
$$\frac{\Pi(\langle x \rangle) = t}{\Pi \vdash \langle x \rangle : t, \emptyset}$$

(T-CALL)
$$\frac{(c_1, t, m\ (\overline{t\,x})\ \{e_{n+1}\}, \sigma) = \mathit{find}(c_0, m) \qquad \Pi \vdash e_0 : c_0, \sigma_0 \qquad (\forall\, t_i\, x_i \in \overline{t\,x},\ \Pi \vdash e_i : t_i', \sigma_i \ \wedge\ t_i' <: t_i)}{\Pi \vdash e_0.m(\overline{e}) : t, \sigma \sqcup \overline{\sigma}}$$

(T-NEW)
$$\frac{\mathit{isClass}(c)}{\Pi \vdash \textbf{new}\ c() : c, \emptyset}$$

(T-NULL)
$$\Pi \vdash \texttt{null} : c, \emptyset$$

(T-DEF)
$$\frac{\Pi \vdash e : c', \sigma \qquad c' <: c \qquad \Pi \oplus \{\langle x \rangle \mapsto c\} \vdash e' : t, \sigma'}{\Pi \vdash c\ \langle x \rangle = e; e' : t, \sigma \sqcup \sigma'}$$

(T-GET)
$$\frac{\Pi \vdash e : c, \sigma \qquad \mathit{type}(c, f) = (\rho, t)}{\Pi \vdash e.f : t, \sigma \sqcup \textbf{rd}\ \rho}$$

(T-SET)
$$\frac{\Pi \vdash e : c, \sigma \qquad \mathit{type}(c, f) = (\rho, t) \qquad \Pi \vdash e' : t', \sigma' \qquad t' <: t}{\Pi \vdash e.f = e' : t', \sigma \sqcup \sigma' \sqcup \textbf{wr}\ \rho}$$

"Latent" Handler Effects and its Realization, $\sigma \sqcup \sigma = \sigma$:

$$\sigma \sqcup \sigma' = \sigma \cup \sigma' \cup \{\epsilon \mid \textbf{ann}\ p \in \sigma \wedge \textbf{reg}\ p\ \sigma'' \in \sigma' \wedge \exists \epsilon \in \sigma''\}$$

Figure 5.4 Type-and-effect Rules.

the effect of the expression at line 9 is **reg** `Ee` **wr** ρ, where **wr** ρ is the effect of `h`.

The (T-ANNOUNCE) says that the effects of the expression are the union of all the parameters' effects plus one announcement effect, **ann**. This effect serves as a place holder which will be used by registration-time specialization in §5.5 to fill up more precise effect information at runtime.

The communication of the effects from handler registration to a handler invocation is best viewed in the effect operator $\sqcup$ used in the rules and defined at the bottom of Figure 5.4. Via the **register** expression, the effect of a handler is put inside the **reg** effect while with the (T-ANNOUNCE) and $\sqcup$, this embedded effect is extracted from the **reg** effect to be exercised at the point of announcement; effects flow from the points where handlers are registered to the points where they are invoked, *e.g.*, in Example 5.2.1, `h` will run as the result of 1) its registration at line 9 and 2) the **announce** at line 10. Therefore, the effects of `v` include the effects of `h`, when combining the effects **reg** `Ee` **wr** ρ and **ann**

Typings for Declarations:

$$\text{(T-EVENT)}\ \frac{\forall (t\ x) \in \overline{t\ x}, \mathit{isClass}(t)}{\vdash \mathbf{event}\ p\ \{\overline{t\ x}\} : \text{OK}} \qquad \text{(T-PROGRAM)}\ \frac{\forall \mathit{decl} \in \overline{\mathit{decl}},\ \vdash \mathit{decl} : \text{OK} \qquad \vdash e : t, \sigma}{\vdash \overline{\mathit{decl}}\ e : t, \sigma}$$

$$\text{(T-CLASS)}\ \frac{\mathit{validF}(\overline{t\ f}, d) \qquad \forall \mathit{meth} \in \overline{\mathit{meth}},\ \vdash \mathit{meth} : t\ \mathbf{in}\ c}{\vdash \mathbf{class}\ c\ \mathbf{extends}\ d\ \{\overline{t\ f}\ \mathbf{in}\ \rho\ \ \overline{\mathit{meth}}\} : \text{OK}}$$

$$\text{(T-METHOD)}\ \frac{\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t) \qquad \forall t_i\ x_i \in \overline{t\ x},\ \mathit{isClass}(t_i) \qquad \mathit{isType}(t) \qquad (\overline{x : t}, \mathbf{this} : c) \vdash e : t', \sigma \qquad t' <: t}{\vdash t\ m(\overline{t\ x})\{e\} : \overline{t} \xrightarrow{\sigma} t\ \mathbf{in}\ c}$$

Auxiliary Functions:

$$\begin{array}{rll} \mathit{isClass}(t) & \mathbf{if} & \mathbf{class}\ t \ldots \in CT \\ \mathit{isType}(t) & \mathbf{if} & \mathit{isClass}(t) \vee t = \mathbf{void} \\ \mathit{validF}(\overline{t\ f}, c) & \mathbf{if} & \forall (t\ f) \in \overline{t\ f}, \mathit{isClass}(t) \wedge f \notin \mathit{dom}(\mathit{flds}(c)) \\ \mathit{flds}(c) = fs & \mathbf{if} & \mathbf{class}\ c\ \mathbf{extends}\ d\ \{\overline{t\ f}\ \mathbf{in}\ \rho\ \ldots\} \in CT \\ & \wedge & fs = \mathit{flds}(d) \cup \overline{f \mapsto_\rho t} \end{array}$$

Valid Method Overriding: $\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t)$

$$\mathit{override}(m, c, \overline{t} \xrightarrow{\sigma} t)\ \mathbf{if}\ (c', t, m\ (\overline{t\ x})\ \{e\}, \sigma') = \mathit{find}(c, m) \wedge \sigma \subseteq \sigma'$$

Method Lookup: $\mathit{find}(c, m) = (c', t, m(\overline{t\ x})\{e\}, \sigma)$

$$\mathit{find}(c, m) = \begin{cases} (c, t, m(\overline{tx})\{e\}, \sigma) & \mathbf{if}\ \mathbf{class}\ c\ \mathbf{extends}\ d\{\ldots\overline{\mathtt{meth}}\} \in CT \\ & \wedge\ (t, \sigma, m(\overline{t\ x})\{e\}) \in \overline{\mathtt{meth}} \\ \mathit{find}(d, m) & \mathbf{otherwise} \end{cases}$$

Type Lookup for Field: $\mathit{type}(c, f) = (\rho, t)$

$$\mathit{type}(c, f) = \begin{cases} (\rho, t) & \mathbf{if}\ \mathbf{class}\ c\ \mathbf{extends}\ d\ \{\overline{field}\ldots\} \in CT \\ & \wedge\ t\ f\ \mathbf{in}\ \rho \in \overline{field} \\ \mathit{type}(d, f) & \mathbf{otherwise} \end{cases}$$

Figure 5.5 Type-and-effect Rules for Top Level Declarations.

$\mathtt{E}e.\ \sqcup$ can be viewed as a special form of effect set union $\cup$, which will union the effects of its LHS and RHS, a typical way of merging the effects of subexpressions, *e.g.*, (T-GET).

5.4.4 Top-Level Declarations

The rules for declarations are standard, shown in Figure 5.5.

The (T-METHOD) uses the function *override* (Figure 5.5) to check overriding, which enforces that the effect of an overriding method is a subset of the overridden method [53].

5.5 Semantics with Effect-Guided Scheduling

Here we give a small-step operational semantics for ATE. The main novelty is to support precise reasoning of the dynamically changing effects of handlers via registration-time specialization, dynamic typing and the integration of the effect system with a scheduling algorithm that produces safe execution, while improving concurrency for II programs.

Definitions:

$$
\begin{array}{lcll}
\Sigma & ::= & \langle \psi, \mu, \gamma, f \rangle & \textit{program configuration} \\
\psi & ::= & \overline{\langle e, id \rangle} & \textit{task queue} \\
id & ::= & id.id \mid 0 \mid 1 \mid \ldots & \textit{thread id} \\
\mu & ::= & \overline{loc \mapsto [c\,\texttt{.}\,\overline{f \mapsto_\rho v}]} & \textit{store} \\
v & ::= & \texttt{null} \mid loc & \textit{value} \\
\gamma & ::= & \overline{p \mapsto \langle b, \overline{loc.m} \rangle} & \textit{event map} \\
b & ::= & true \mid false & \textit{boolean value} \\
f & ::= & \overline{\langle id, \sigma \rangle} & \textit{trace} \\
\mathbb{E} & ::= & - \mid \mathbb{E}\,\texttt{.}\,m(\overline{e}) \mid v\,\texttt{.}\,m(\overline{v}\mathbb{E}\overline{e}) \mid \mathbb{E}.f \mid \mathbb{E}\,\texttt{.}\,f{=}e \mid v\,\texttt{.}\,f{=}\mathbb{E} \mid t\ x{=}\mathbb{E}; e & \textit{evaluation context} \\
 & & \mid \textbf{announce}\ p(\overline{v}\mathbb{E}\overline{e}) \mid \textbf{register}\ \mathbb{E}.m\ \textbf{with}\ p &
\end{array}
$$

Dynamic Typing: $\gamma, \mu, \Pi \vdash_D e : t, \sigma$

$$
\frac{\mu(loc) = [c.\overline{f \mapsto_\rho v}]}{\gamma, \mu, \Pi \vdash_D loc : c, \emptyset}\ (\gamma\text{-}loc)
\qquad
\frac{\Pi \vdash \textbf{announce}\ p\ (\overline{e})\ : \textbf{void}, \sigma \qquad \gamma, \mu, \Pi \vdash_D p : \langle b, \sigma' \rangle}{\gamma, \mu, \Pi \vdash_D \textbf{announce}\ p\ (\overline{e})\ : \textbf{void}, \sigma \sqcup \sigma'}\ (\gamma\text{-ANNOUNCE})
$$

$$
\frac{\gamma(p) = \langle b', \overline{loc.m} \rangle \qquad b = \bigwedge \forall \sigma_i, \sigma_j \in \overline{\sigma}\ \text{s.t.}\ i \neq j\ \text{s.t.}\ \sigma_i \# \sigma_j \qquad \forall loc.m \in \overline{loc.m}\ .\ dispatch(\mu, loc, m) = m(\overline{t\ x})\{e\}\ \wedge\ \gamma, \mu, \overline{x : t} \vdash_D [loc/\texttt{this}]e : t', \sigma}{\gamma, \mu, \Pi \vdash_D p : \langle b, \sqcup\overline{\sigma} \rangle}\ (\gamma\text{-EVENT})
$$

For all other (γ-*) rules, each is isomorphic to its counterpart (T-*) rule, except that every occurrence of the judgment $\Pi \vdash e : t, \sigma$ in the latter rule should be substituted with $\gamma, \mu, \Pi \vdash_D e : t, \sigma$ in the former.

Figure 5.6 Semantics Domains and Dynamic Typing.

5.5.1 Domains

The small steps taken in the semantics are defined as transitions from one configuration to another. These configurations are shown in Figure 5.6. A configuration consists of a task queue ψ, a store μ, an event map γ and a *trace* f. Each reference cell in μ records an object $c.F$, consisting of a class name c and a field record. A field record $\overline{f \mapsto_\rho v}$ maps field names f to values v in region ρ. A value v may either be **`null`** or a location loc. The map γ maps an event p to a configuration. This configuration consists of a (mutable) queue of handlers $\overline{loc.m}$ and a boolean flag b, indicating whether the handlers can be run concurrently.

The task queue ψ consists of an ordered list of task configurations $\langle e, id \rangle$. Each task configuration (called simply a task) consists of the task identifier id and an expression e serving as the remaining evaluation to be done for the task.

A trace f is the "realized effects". It is defined as a sequence of accesses to references, with read/write to regions and event registration and announcement. Traces are only needed to demonstrate the soundness (§5.6), but are unnecessary in the implementation.

ATE uses a call-by-value evaluation strategy. The operator $\oplus$ is an overriding operator for finite functions, *i.e.*, if $\mu' = \mu \oplus \{loc \mapsto o\}$, then $\mu'(loc') = o$ if $loc' = loc$, otherwise $\mu'(loc') = \mu(loc')$. The rest of this section highlights the rules for the expressions **announce**, **register** and **yield**.

5.5.2 Registration-Time Specialization & Dynamic Typing

The (REG) rule appends the new handler to the mutable queue $p \mapsto \langle b, \overline{loc.m} + loc.m \rangle$ for event p. Concurrency decisions can now be made because the previously unknown handlers become known. If none of the handlers conflicts, indicated by the flag b, they can be run concurrently. *Dynamic typing* is used to compute the effect of the new handler.

Dynamic typing provides more precise effects because of two reasons: 1) at runtime, the variables of the source expression $\in$ will be substituted with values (*e.g.*, (DEF) and (CALL)), which carries more precise runtime information; and 2) the previously unknown handlers are known (*registration-time specialization*), by inspecting the queue. Dynamic typing is defined through type derivation $\gamma, \mu, \Pi \vdash_D e : t, \sigma$ in Figure 5.7, which extends static typing, defined in Figure 5.4, with one additional rule (γ-*loc*)

Evaluation Relation: $\psi, \mu, \gamma, f \hookrightarrow \psi', \mu', \gamma', f'$

$$(cont)\ \langle \mathbb{E}[e], id\rangle + \psi, \mu, \gamma, f \hookrightarrow \langle \mathbb{E}[\textbf{yield}\ e'], id\rangle + \psi + \psi', \mu', \gamma', f + \langle id, \sigma\rangle \quad \textbf{if} \quad \mathrm{e}, id, \mu, \gamma \Rightarrow \mathrm{e}', \psi', \mu', \gamma', \sigma$$

Local Reduction: $e_{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma$, **where** $_{\mathbb{C}} = id, \mu, \gamma$

$$(reg)\ \textbf{register}\ loc.m\ \textbf{with}\ p_{\mathbb{C}} \Rightarrow loc, \emptyset, \mu, \gamma', \texttt{reg}\ p\ \sigma\ \textbf{if}\ \mu(loc) = [c.\overline{f \mapsto_\rho v}] \wedge find(c,m) = (\ldots, \sigma)$$
$$\wedge\ \gamma'' = \gamma \oplus \{p \mapsto \langle b, \overline{loc.m} + loc.m\rangle\}$$
$$\wedge\ \gamma' = \{p' \mapsto \left\langle b', \overline{loc'.m'}\right\rangle \mid \gamma''(p') = \left\langle b'', \overline{loc'.m'}\right\rangle \wedge \gamma'', \mu, \emptyset \vdash_{\mathbb{D}} p' : \langle b', \overline{\sigma}\rangle\}$$

$$(ann)\quad \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \Rightarrow e, \psi, \mu, \gamma, \texttt{ann}\ p\quad \textbf{if}\ \gamma(p) = \langle b, \overline{loc.m}\rangle \wedge \psi = \overline{\langle e, id\rangle}$$
$$\wedge\ e = \vec{join}(\overline{id})$$
$$\wedge \forall loc_i.m_i \in \overline{loc.m}\ id_i = id.fresh()$$
$$\wedge\ e_i = \begin{cases} dyn(\mu, loc_i, m_i, \overline{v}) & \textbf{if}\ b \\ \vec{join}(\overline{id_{i-1}}); dyn(\mu, loc_i, m_i, \overline{v}) & \textbf{if}\ !b \end{cases}$$

$$(join)\quad \vec{join}(\overline{id})_{\mathbb{C}} \Rightarrow e, \emptyset, \mu, \gamma, \texttt{join}\quad \textbf{if}\ \nexists id_i \in \overline{id}\ \text{s.t.}\ \langle e_i, id_i\rangle \in \psi \wedge e = \texttt{null}$$

$$(call)\quad loc.m(\overline{v})_{\mathbb{C}} \Rightarrow e, \emptyset, \mu, \gamma, \emptyset\quad \textbf{if}\ dyn(\mu, loc, m, \overline{v}) = e$$
$$(def)\quad c\ x = v; e_{\mathbb{C}} \Rightarrow e', \emptyset, \mu, \gamma, \emptyset\quad \textbf{if}\ e' = [v/x]e$$
$$(new)\quad \textbf{new}\ c()_{\mathbb{C}} \Rightarrow loc, \emptyset, \mu', \gamma, \emptyset\quad \textbf{if}\ loc \notin dom(\mu) \wedge flds(c) = \overline{f \mapsto_\rho t}$$
$$\wedge\ \mu' = \mu \oplus \{loc \mapsto [c\,.\,\overline{f \mapsto_\rho \texttt{null}}]\}$$
$$(set)\quad loc.f = v_{\mathbb{C}} \Rightarrow v, \emptyset, \mu', \gamma, \textbf{wr}\ \rho\quad \textbf{if}\ \mu' = \mu \oplus (loc \mapsto [c.\overline{f \mapsto_\rho v} \oplus (f \mapsto_\rho v)])$$
$$(get)\quad loc.f_{\mathbb{C}} \Rightarrow v, \emptyset, \mu, \gamma, \textbf{rd}\ \rho\quad \textbf{if}\ \mu(loc) = [c.\overline{f \mapsto_\rho v}]$$

Cooperative Handling: $\psi, \mu, \gamma, f \hookrightarrow \psi', \mu', \gamma', f'$

$$(yield)\quad \langle\langle \mathbb{E}[\textbf{yield}\ e], id\rangle + \psi, \mu, \gamma, f\rangle \hookrightarrow \langle active(\psi + \langle \mathbb{E}[e], id\rangle), \mu, \gamma, f\rangle$$
$$(end)\quad \langle\langle v, id\rangle + \psi, \mu, \gamma, f\rangle \hookrightarrow \langle active(\psi), \mu, \gamma, f\rangle$$

Figure 5.7 Operational Semantics. Auxiliary functions are defined in Figure 5.8.

for reference *loc* value typing. In previous work, effects do not change at runtime. In ATE, the effects could change due to dynamic event registration, *e.g.*, the effects of a subject that may announce p, could change, with more handlers registered with p. To account for this, dynamic typing inspects the handlers in the event map γ (the (γ-EVENT) rule) and provides precise effects for the **announce** expressions. The (γ-EVENT) checks for event p whether the handlers for p can be run concurrently, and what the effects of all these handlers are. Note that the dynamic typing rule may recurse on the events when checking an **announce** expression and thus the *fixed point* operator is used.

Note that a handler h can register (other) handlers h′ for an event p when handling an event and later announce the event p (*e.g.*, v in Example 5.2.1). The effects of h should include the registration, announce effects and the effects of h′. This scenario is handled by $\sqcup$ (Figure 5.4). An alternative sound solution will let the effects of h to conservatively be top, *i.e.*, read/write the entire store [16].

5.5.3 Event Announcement & Safe Implicit Concurrency

The (ANN) retrieves the handlers registered for the corresponding event p. The dynamic typing used in the **register** provides precise effects and analyzes whether the handlers can be run concurrently. If their effects conflict, each handler has to wait until the completion of the previous registered handler using the expression $\vec{join}(\overline{id_{i-1}})$ to avoid concurrency errors. Otherwise, the handlers can all be run concurrently. The **announce** waits for its handlers to complete.

Note that if any pair of handlers conflict, the formalism executes the handlers sequentially. A better implementation is possible, *e.g.*, executing nonconflicting handlers concurrently or executing a handler as soon as all its conflicting handlers are done [59], or executing handlers with less effects before handlers with more effects to promote modular reasoning [6]. These schedulings maintain concurrency safety because conflicting handlers can never be run concurrently. There are many scheduling techniques from which our work can learn, but the simplification suffices to illustrate the soundness.

The expression $\vec{join}(\overline{id})$ can only process after all the tasks $\overline{id}$ are done, *i.e.*, no longer in the queue ψ. The expression $\vec{join}$ is joining other handlers and known as a *right mover* [48], indicated by the head symbol $\rightarrow$. As interference points only exist after the right mover [121], **announce** is the only interference points in ATE (see Example 5.2.3).

Dynamic Dispatch, $dispatch(\mu, loc, m) = m(\overline{t\ x})\{e\}$:

$$dispatch(\mu, loc, m) = m(\overline{t\ x})\{e\} \quad \textbf{if } \mu(loc) = [c.\overline{f \mapsto_\rho v}] \wedge find(c, m) = (c', t, m(\overline{t\ x})\{e\}, \sigma)$$

$$dyn(\mu, loc, m, \overline{v}) = [loc/\texttt{this}, \overline{v/x}]e \quad \textbf{if } dispatch(\mu, loc, m) = m(\overline{t\ x})\{e\}$$

Cooperative Handlers Management, $active(\psi) = \psi$:

$$active(\langle e, id\rangle + \psi) = \begin{cases} \langle e, id\rangle + \psi & \textbf{if } e \neq \vec{join}(\overline{id}) \vee \nexists id' \in \overline{id} \text{ s.t. } \langle e', id'\rangle \in \psi \\ active(\psi + \langle e, \tau\rangle) & \textbf{otherwise} \end{cases}$$

Figure 5.8 Auxiliary Functions for the Semantics.

5.5.4 Yielding Control & Interference Points

To model concurrency, we use preemptive interleaving [121], like Abadi and Plotkin [3], *i.e.*, the running handlers will relinquish control (interference points) to other handlers at each step (see the (CONT)). We will prove in §5.6.5 that, in ATE, this preemptive semantics is equivalent to the cooperative semantics, where the only interference points appear after the **announce** expression.

The (YIELD) puts the current handler to the end of the queue ψ and starts the next active task from this queue. Finding an active task is done by the function $active$ (Figure 5.8). It returns the top most task in ψ that can be run. A task is ready to run if it is not waiting on other tasks, *i.e.*, not a $\vec{join}$ expression, or all the tasks it is waiting on are done.

The (END) rule says that the current running task is done (it evaluates to a single value v), thus it will be removed from the queue and the next active task will be scheduled.

5.6 Meta-Theories

We now show the key properties of ATE. The properties include the standard type soundness (§5.6.3), liveness (§5.6.2), sequential semantics (§5.6.4) and sparse interference points (§5.6.5). In previous works [72, 113], the exact set of concurrent tasks that will be spawned are known statically. A technical challenge for proving the soundness of our work is that concurrent tasks spawned as a result of an event announcement are unknown statically due to dynamic registration.

5.6.1 Preliminary Definitions

Before we proceed, we first give some simple definitions that will be used for the rest of the section.

Definition 5.6.1 *[Redex configuration] We say Σ is a* redex configuration *of program $\overline{\langle decl \rangle}\ e$, written $e \trianglerighteq \Sigma$, iff $\langle \langle e, 0 \rangle, \emptyset, \emptyset, \emptyset \rangle \overset{*}{\hookrightarrow} \Sigma$. We say Σ is a proper redex configuration, written $\trianglerighteq\Sigma$, if $\exists e$ such that $\vdash \overline{decl}\ e : t, \sigma$ and $e \trianglerighteq \Sigma$.*

Definition 5.6.2 *[Well-typed queue] A queue ψ is* well-typed *in μ and γ, written $\gamma, \mu \vdash \psi$, if and only if $\forall \langle e, id \rangle \in \psi, \gamma, \mu, \emptyset \vdash_D e : t, \sigma$ for some t and σ.*

Definition 5.6.3 *[Well-typed event map] A handler $loc.m$ is* well-typed *in μ for event p, written as $loc.m \simeq (\mu, p)$, if $(m\ (\overline{t'\ x})\ \{e\}) = dispatch(\mu, loc, m)$, $\texttt{event}\ p\ \{\overline{t\ x}\} \in CT$, $(\forall\ t_i\ x_i \in \overline{t\ x},\ t'_i <: t_i)$. An event map γ* is well-typed *in μ, written as $\mu \vdash \gamma$, if $\forall p \in dom(\gamma)$ s.t.$(\gamma(p) = \langle b, \overline{loc.m} \rangle) \Rightarrow (\forall loc.m \in \overline{loc.m}$ s.t. $loc.m \simeq (\mu, p))$.*

Definition 5.6.4 *[Local reduction] Let two configurations $\Sigma = \langle \langle e, id \rangle + \psi, \mu, \gamma, f \rangle$ and $\Sigma' = \langle \langle e', id \rangle + \psi', \mu', \gamma', f' \rangle$. A reduction $\Sigma \hookrightarrow^* \Sigma'$, is called a task* local reduction*, denoted as $\Sigma \rightarrowtail \Sigma'$, if $\nexists e''$ s.t. $\Sigma \hookrightarrow^* \langle \langle e'', id \rangle + \psi'', \mu'', \gamma'', f'' \rangle \hookrightarrow^* \Sigma'$.*

Definition 5.6.5 *[Well-typed configuration] A configuration $\Sigma = \langle \psi, \mu, \gamma, f \rangle$ is* well-typed*, written as $\vdash \Sigma$, if $\gamma, \mu \vdash \psi$ and $\mu \vdash \gamma$.*

5.6.2 Livelock Freedom

In the semantics, ATE lets some tasks be inactive, *i.e.*, wait for conflicting tasks to maintain concurrency safety. We prove that at any configuration, there exists a task that is active and thus can make progress. Intuitively, each task can only wait for its predecessor handlers, *i.e.*, conflicting handlers that registered earlier and its handlers if it announces an event. Such waiting relationship forms a tree, not a circle (circular wait). Therefore, ATE is livelock free.

Definition 5.6.6 *[Blocked configurations] Let a configuration $\Sigma = \langle \psi, \mu, \gamma, f \rangle$. The task $\langle e, id \rangle$ in Σ blocks, written $\uparrow \langle e, id \rangle$, if $e == \mathbb{E}[\vec{join}(\overline{id})]$ and $\exists \langle e', id' \rangle \in \psi$ s.t. $id' \in \overline{id}$. Otherwise, $\langle e, id \rangle$ is*

active, written $\downarrow\langle e, id\rangle$. *A configuration* Σ *blocks, written* $\uparrow \Sigma$, *if* $\forall \langle e, id\rangle \in \psi, \uparrow \langle e, id\rangle$, *otherwise,* Σ *can make progress, written* $\downarrow \Sigma$.

Theorem 5.6.7 [Liveness] *If* $\trianglerighteq\Sigma$, *then* $\downarrow \Sigma$.

Proof: The proof is by induction on the number of reduction steps (Figure 5.7) applied. □

Evaluation Relation: $\psi, \mu, \gamma, f \hookrightarrow_S \psi', \mu', \gamma', f'$

$(cont) \quad \langle \mathbb{E}[e], id\rangle + \psi, \mu, \gamma, f \quad \hookrightarrow_S \quad \langle \mathbb{E}[e'], id\rangle + \psi + \psi', \mu', \gamma', f + \langle id, \sigma\rangle$ **if** $\langle \mathrm{e}\rangle, id, \mu, \gamma \Rightarrow_S \langle \mathrm{e}\rangle', \psi', \mu', \gamma', \sigma$

Sequential Reduction: $e_{\mathbb{C}} \Rightarrow_S e', \psi, \mu', \gamma', \sigma$ **where** $_{\mathbb{C}} = id, \mu, \gamma$

(ann_S) **announce** $p\,(\overline{v})_{\mathbb{C}} \quad \Rightarrow_S \quad \overline{e}, \emptyset, \mu, \gamma,$ **ann** p **if** $\gamma(p) = \langle b, \overline{loc.m}\rangle \wedge \forall loc_i.m_i \in \overline{loc.m}.\ dyn(\mu, loc_i, m_i, \overline{v}) = e_i$

For all other ($*_S$) rules, each is isomorphic to its counterpart (*) rule, except that every occurrence of the judgment $e_{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma$ in the latter rule should be substituted with $e_{\mathbb{C}} \Rightarrow_S e', \psi, \mu', \gamma', \sigma$ in the former.

Figure 5.9 Sequential Semantics.

5.6.3 Type Soundness

In this section, we prove the standard type soundness. First we prove that with more items in the store μ and event map γ, the typing of the same expression remain unchanged.

Definition 5.6.8 *[Store enlargement] Let* μ *and* μ' *be two stores. We write* $\mu \lessdot \mu'$, *if:*

1. $dom(\mu) \subseteq dom(\mu')$;
2. $\forall loc \in dom(\mu)$, *if* $\mu(loc)=[c.\overline{f \mapsto_\rho v}]$, *then* $\mu'(loc)=[c.\overline{f \mapsto_\rho v'}]$.

Lemma 5.6.9 [Store extension] *If* $\gamma, \mu, \Pi \vdash_D e : t, \sigma$ *and* $\mu \lessdot \mu'$, *then* $\gamma, \mu', \Pi \vdash_D e : t, \sigma$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$. □

Definition 5.6.10 *[Event map enlargement] Let* p *be an event type,* γ *and* γ' *be two event maps. We write* $\gamma \lessdot_{\langle p, loc.m\rangle} \gamma'$, *if all the following hold:*

1. $dom(\gamma) = dom(\gamma')$;
2. $\forall p' \in dom(\gamma)$, *if* $\gamma(p')=\langle b, \overline{loc.m}\rangle$, *then* $\gamma'(p')=\langle b', \overline{loc.m}\rangle$;
3. *if* $\gamma(p) = \langle b, \overline{loc.m}\rangle$, *then* $\gamma(p) = \langle b', \overline{loc.m} + loc.m\rangle$.

Lemma 5.6.11 [Event map extension I] *If* $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, $\gamma \lessdot_{\langle p, loc.m \rangle} \gamma'$, *and* **ann** $p \notin \sigma$ *then* $\gamma', \mu, \Pi \vdash_D e : t, \sigma$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$. □

Lemma 5.6.12 [Event map extension II] *If* $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, $\gamma \lessdot_{\langle p, loc.m \rangle} \gamma'$, $\mu(loc) = [c.\overline{f \mapsto_\rho v}]$, $find(c, m) = (\ldots, \sigma')$, *and* **ann** $p \in \sigma$ *then* $\gamma', \mu, \Pi \vdash_D e : t, post(\sigma \cup \sigma')$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$. □

Our soundness proof is constructed through standard subject reduction and progress:

Lemma 5.6.13 [Preservation] *Let* $\Sigma = \langle\langle e, id\rangle + \psi, \mu, \gamma, f\rangle$. *If* $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $\Sigma \rightarrowtail \langle\langle e', id\rangle + \psi', \mu', \gamma'\rangle$, *then there is some* t' *and* σ' *such that* $\gamma', \mu', \emptyset \vdash e' : t', \sigma' \wedge t' <: t$.

Proof: The proof proceeds by structural induction on the derivation of $\gamma, \mu, \Pi \vdash_D e : t, \sigma$, Lemma 5.6.9, 5.6.12, and 5.6.12. □

Lemma 5.6.14 [Progress] *Let* $\Sigma = \langle\langle e, id\rangle + \psi, \mu, \gamma, f\rangle$. *If* $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, *then either* e *is a value* v, *or* $\Sigma \rightarrowtail \langle\langle e', id\rangle + \psi', \mu', \gamma'\rangle$.

Proof: By cases on the reduction step applied. □

Theorem 5.6.15 [Type Soundness] *Given an expression* e, $\emptyset, \emptyset, \emptyset \vdash_D e : t, \sigma$, *then either the evaluation of* e *diverges, or there exists some* μ, v, γ *and* f *such that* $\langle\langle e, 0\rangle, \emptyset, \emptyset, \emptyset\rangle \hookrightarrow^* \langle\langle v, 0\rangle, \mu, \gamma, f\rangle$.

Proof: By Lemma 5.6.14 and 5.6.13. □

5.6.4 Sequential Semantics

In this section, we will prove that the execution of an ATE program (which runs nonconflicting handlers concurrently in §5.5) is *behaviorally equivalent* to its sequential counterparts, where every **announce** expression will execute the handlers one by one serially. First, let us define relation $f \propto \sigma$, which says a trace f *realizes* a static effect σ:

Definition 5.6.16 *[Static effect contains dynamic effect]* $f \propto \sigma$ *holds iff* $\forall \langle id, \sigma'\rangle \in f \,.\, \sigma' \subseteq \sigma$.

Evaluation Relation: $\psi, \mu, \gamma, f \hookrightarrow_Y \psi', \mu', \gamma', f'$

$$(cont)\ \langle \mathbb{E}[e], id\rangle + \psi, \mu, \gamma, f \hookrightarrow_Y \langle \mathbb{E}[e'], id\rangle + \psi + \psi', \mu', \gamma', f + \langle id, \sigma\rangle \ \textbf{if}\ \langle \mathrm{e}\rangle, id, \mu, \gamma \Rightarrow_Y \langle \mathrm{e}\rangle', \psi', \mu', \gamma', \sigma$$

Cooperative Reduction: $e_{\mathbb{C}} \Rightarrow_Y e', \psi, \mu', \gamma', \sigma$ **where** $_{\mathbb{C}} = id, \mu, \gamma$

$$(ann_Y)\quad \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \Rightarrow_Y \textbf{yield}\ e, \psi, \mu, \gamma, \texttt{ann}\ p \quad \textbf{if}\ \textbf{announce}\ p\ (\overline{v})_{\mathbb{C}} \Rightarrow e, \psi, \mu, \gamma, \texttt{ann}\ p$$

For all other ($*_Y$) rules, each is isomorphic to its counterpart (*) rule, except that every occurrence of the judgment $e_{\mathbb{C}} \Rightarrow e', \psi, \mu', \gamma', \sigma$ in the latter rule should be substituted with $e_{\mathbb{C}} \Rightarrow_Y e', \psi, \mu', \gamma', \sigma$ in the former.

Figure 5.10 Cooperative Semantics.

Next, we state and prove that every pair of the handlers in the queue ψ do not conflict:

Definition 5.6.17 *[Noninterfering tasks] Two task* $\langle e, id\rangle$ *and* $\langle e', id'\rangle$ *are* noninterfering *in* μ, γ, *written* $\gamma, \mu \trianglerighteq \langle e, id\rangle \# \langle e', id'\rangle$, *if:*

1. $e = \vec{join}(\overline{id}) \ \wedge\ id' \in \overline{id}$; *or* $e' = \vec{join}(\overline{id}) \ \wedge\ id \in \overline{id}$;
2. *or* $\gamma, \mu, \emptyset \vdash_D e : \sigma, t$, $\gamma, \mu, \emptyset \vdash_D e' : \sigma', t'$ *and* $\sigma \# \sigma'$.

Definition 5.6.18 *[Noninterfering queue] A queue* ψ *is* noninterfering *in* μ *and* γ, *written* $\gamma, \mu \trianglerighteq \psi$, *if* $\forall \langle e_i, id_i\rangle, \langle e_j, id_j\rangle \in \psi$ *s.t.* $i \neq j, \gamma, \mu \trianglerighteq \langle e_i, id_i\rangle \# \langle e_j, id_j\rangle$.

Lemma 5.6.19 [Noninterfering preservation] *Let* $\Sigma = \langle \psi, \mu, \gamma, f\rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\Sigma \hookrightarrow \Sigma'$ *where* $\Sigma' = \langle \psi', \mu', \gamma', f'\rangle$, *then* $\gamma', \mu' \trianglerighteq \psi'$.

Proof: By cases on the reduction step and Definition 5.6.18. □

Next, we prove that the trace a handler leaves realizes its effects given by the dynamic typing:

Lemma 5.6.20 [Effect subsumption] *Let* $\Sigma = \langle \langle e, id\rangle, \mu, \gamma, f\rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $\Sigma \rightarrowtail \Sigma'$ *where* $\Sigma' = \langle \langle e', id\rangle + \psi', \mu', \gamma', f + f'\rangle$, *then*

(a) $\gamma', \mu', \emptyset \vdash_D e' : t', \sigma' \wedge (t' <: t) \wedge (\sigma' \subseteq \sigma)$;

(b) $f' \propto \sigma$.

Proof: By cases on the reduction step applied. □

Prefix, $pref(id, id') = b$:

$$pref(id, id') = \begin{cases} true \ \textbf{if} \ (id == id') \vee (id == id'.id'' \text{ for some } id'') \\ false \ \textbf{otherwise} \end{cases}$$

Trace Projection, $\pi(id, f) = f$:

$$\pi(id, f) = \begin{cases} \emptyset & \textbf{if } f = \emptyset \\ \langle id', \sigma \rangle + \pi(id, f') & \textbf{if } f = \langle id', \sigma \rangle + f' \ \wedge \ pref(id, id') \\ \pi(id, f') & \textbf{otherwise} \end{cases}$$

Figure 5.11 Trace Projection.

Lemma 5.6.21 [Effect preservation] *If* $\Sigma = \langle \langle e, id \rangle, \mu, \gamma, f \rangle$, *and* $\trianglerighteq \Sigma$. *If* $\gamma, \mu \trianglerighteq \psi$, $\gamma, \mu, \emptyset \vdash_D e : t, \sigma$, $e \neq \mathbb{E}[\vec{join}(\overline{id})]$, $\Sigma \hookrightarrow^* \langle \langle v, id \rangle, \mu', \gamma', f' \rangle$ *then* $\pi(id, f' - f) \propto \sigma$.

Proof: By cases on the reduction step applied. □

With the above, we can prove that any ATE program is race free. There remains a gap between this property and why one *intuitively* believes that ATE is sequentially consistent (SC). To rigorously define the more intuitive notion of SC, let us first introduce the sequential semantics (handlers run sequentially) of ATE, shown in Figure 5.9.

We are ready to prove that an ATE program behaves the same as its sequential counterpart:

Theorem 5.6.22 [Sequential Semantics] *Given an expression* e, $\emptyset, \emptyset, \emptyset \vdash_D e : t, \sigma$, *then either the evaluation of* e *diverges in both the sequential and the parallel semantics, or there exists some* μ, v, γ, f *and* f' *such that* $\langle \langle e, 0 \rangle, \emptyset, \emptyset, \emptyset \rangle \hookrightarrow^* \langle \langle v, 0 \rangle, \mu, \gamma, f \rangle$ *and* $\langle \langle e, 0 \rangle, \emptyset, \emptyset, \emptyset \rangle \hookrightarrow_S^* \langle \langle v, 0 \rangle, \mu, \gamma, f' \rangle$.

Proof: By Lemma 5.6.19, 5.6.20, and 5.6.21. □

5.6.5 Modular Reasoning

In this section, we will prove that the execution of an ATE program (which has preemptive semantics, *i.e.*, yielding control to other active handlers at each step in §5.5) is *behaviorally equivalent* to its cooperative counterparts: a handler will only yield after announcing an event. The cooperative semantics is defined in Figure 5.10.

Theorem 5.6.23 [Cooperative Semantics] *Given an expression* e, $\emptyset, \emptyset, \emptyset \vdash_D e : t, \sigma$, *then either the evaluation of* e *diverges in both the cooperative and the parallel semantics, or there exists some* μ, v, γ, f *and* f' *such that* $\langle \langle e, 0 \rangle, \emptyset, \emptyset, \emptyset \rangle \hookrightarrow^* \langle \langle v, 0 \rangle, \mu, \gamma, f \rangle$ *and* $\langle \langle e, 0 \rangle, \emptyset, \emptyset, \emptyset \rangle \hookrightarrow_Y^* \langle \langle v, 0 \rangle, \mu, \gamma, f' \rangle$.

Proof: As proven in §5.6.4, an ATE program is data race free. Therefore, reference access is both mover [121]. The **announce** expression, which forks concurrent handlers, is a left mover, and the $\vec{join}$ expression, which waits for its children handlers, is a right mover. Interference points only exist at left movers [121], *i.e.*, the **announce** expression. □

5.7 Related Work

Asynchronous typed events are inspired from a large body of work on events, *e.g.*, [17, 33, 45, 46, 61, 66, 81, 83, 93, 108, 112]. This work goes beyond previous work which views events as a design decoupling mechanisms [45, 81, 87, 88, 93, 95, 96, 97, 112] to leverage decoupling for safe concurrency.

There are plenty of works on using static effect systems to reason about safe concurrency. Earlier work includes Lucassen [72], and Talpin et al. [113], and more recent examples such as Task Types [62] and Bocchino et al. [16]. Compared with these works, our system uses the effects dynamically. Effects of the handlers are computed statically by the type system, and the semantics use these effects to compute a safe order for handler invocation at runtime.

There are several works on using effects dynamically, including TWEJava [59], and Legion [117]. In these works, effects do not change at runtime. In ATE, however, effects could change due to dynamic event registration. ATE introduces a novel type-and-effect system to reason about mutable handler queue, which could be challenging for the above systems.

Compared with software transactional memory (STM) and other related ideas [13] and effect monitoring systems [10], our system computes the effects of the handlers by the type system. Concurrency decisions are guided by the effects of handlers at runtime to gain precision. ATE detects potential conflicts before executing the handlers, while STM executes threads speculatively, and detects conflicts afterwards. In case of conflicts, STM rolls back all the changes.

There is a large body of work on the message-passing, and the publish/subscribe paradigms in distributed systems [67, 83, 90, 103]. These works either require programmers to manually account

for data races, or assume disjoint address space between concurrent processes [83, 103]. ATE tackles concurrency problems in shared-memory paradigm. It eases the burden on the programmer by allowing modular reasoning and by providing implicit safe concurrency [89, 91].

5.8 Summary

In this chapter, we pursue the goal of unifying modular reasoning and concurrency in program design. We have developed the notion of *asynchronous, typed events* that are helpful for programs where modules are decoupled using implicit-invocation design style, and where handlers can register dynamically. Event announcements provide implicit concurrency. Registration-time specialization provides precise effects analyses, which improves safe concurrency for II programs. Dynamic typing takes into account the handlers registered to reason about the mutable handler queue. ATE facilitates modular reasoning about concurrency safety.

CHAPTER 6. CONCLUSION

In this thesis, we develop a new foundation for type-and-effect systems, where static effect reasoning is integrated with dynamic effect analyses powered by dynamic typing. Our work allows the effects of arbitrary program expressions to be intensionally inspected and programmatically analyzed at runtime by end-user programmers. Furthermore, this thesis promotes effects as first-class values. Programmers can pass effects across the modular boundary, store them in mutable references, and inspect their structures at runtime to perform expressive effect analyses. Effect-guided decisions can be made as part of the program itself.

This thesis also develops a highly precise notion of effect reasoning through dynamic typing, while at the same time harvesting the power of static typing to retains strong type safety guarantees. We use a differential alignment strategy to achieve efficiency in dynamic typing.

Additionally, this thesis explores the subtle interaction between static and dynamic typing and the interaction among powerful features such as mutable data structure analyses, existential typing, dynamic typing, runtime type test, double-bounded effects, refinement types and polarity analysis.

We describe how a precise, sound, and efficient hybrid reasoning system can be constructed, and demonstrate its applications in effect-aware scheduling, memoization, information security, UI access and consistent software updates.

We showcase the benefit of using hybrid effect analyses in program development by applying it to an event-driven system to obtain safe concurrency. Event-driven systems are popular because of their flexibility and modularity benefits. We show that the precision of the effect analysis can be improved in event-driven systems which allow handlers to be dynamically registered and concurrency opportunities could be obatined via dynamic typing. Our design simplifies modular reasoning about concurrency in the event-driven system and avoids concurrency hazards. In this sense, our system unifies modular reasoning and concurrency in program design.

CHAPTER 7. FUTURE WORK

The techniques introduced in this thesis for intensional effects were foundational and theoretical. We would like to investigate how we could make our system more flexible and compute more precise effects. Overall, there are some cases where our system could still be conservative. In this chapter, we outline areas in which we plan to do research, hoping to provide the benefits of hybrid effects analyses to different domains of program development.

7.1 Empirical Evaluation on the Impact of Hybrid Effect Analysis

Our first venue of future work is to investigate how much impact our hybrid effect system can have on general software systems. The Boa infrastructure [38, 41, 42, 79, 94] provides capabilities for analyzing large-scale software repositories. We would like to use Boa to find out whether the effects of methods of subclasses and their superclasses are different, and in what situations their effects are different. Hybrid effect analysis is well-suited for situations where the effects of the dynamically dispatched methods are different or unknown statically. In such cases, purely static effect analyses are likely to be too conservative. Our system works by using dynamic typing to compensate for the conservativeness of traditional static approaches, in addition, it precomputes the known effects and stores them to avoid future recomputation. We would like to understand whether there are scenarios in real world open source projects where our hybrid effect analysis could have an impact on the precision of effect analysis.

7.2 Exploratory Study of the Design Impact of Asynchronous, Typed Events

We have already shown that asynchronous, typed events are very useful in obtaining safe concurrency for event-driven systems. In the future, we would like to conduct an exploratory study in a manner similar to Dyer *et al.*'s work [39, 40] to examine the design impact of asynchronous, typed events.

7.3 Effect Analysis on Mutable Data Structure

In the future, we would like to improve the flexibility of our system by allowing effect analysis and inspection on mutable data structures. In this thesis, we show the difficulty and subtlety involved in applying effect analyses on mutable data structures. We provide a solution in a simple case for an event-driven system, successfully obtaining safe concurrency, which could be difficult for both static and dynamic effect systems. We would like to formalize a general solution to tackle this notoriously challenging problem. Our initial idea is that new kinds of effects maybe needed, for example, effects similar to the announcement and registration effects. We would also like to measure the efficiency of our approach by analyzing the potential overhead introduced by the dynamic part of our system and by verifying the performance gains obtained, such as speedup and throughput.

BIBLIOGRAPHY

[1] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. (1989). Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 213–227, New York, NY, USA. ACM.

[2] Abadi, M., Flanagan, C., and Freund, S. N. (2006). Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255.

[3] Abadi, M. and Plotkin, G. (2009). A model of cooperative threads. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 29–40, New York, NY, USA. ACM.

[4] Agesen, O. (1996). *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford University, Stanford, CA, USA.

[5] Aiken, A., Wimmers, E. L., and Lakshman, T. K. (1994). Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 163–173, New York, NY, USA. ACM.

[6] Bagherzadeh, M., Dyer, R., Fernando, R. D., Sánchez, J., and Rajan, H. (2015). Modular reasoning in the presence of event subtyping. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 117–132, New York, NY, USA. ACM.

[7] Bagherzadeh, M. and Rajan, H. (2015). Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108, New York, NY, USA. ACM.

[8] Bagherzadeh, M., Rajan, H., and Darvish, A. (2013). On exceptions, events and observer chains. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 185–196, New York, NY, USA. ACM.

[9] Bagherzadeh, M., Rajan, H., Leavens, G. T., and Mooney, S. (2011). In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 141–152, New York, NY, USA. ACM.

[10] Bañados, F., Garcia, R., and Tanter, É. (2014). A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 283–295, New York, NY, USA. ACM.

[11] Bauer, A. and Pretnar, M. (2012). Programming with algebraic effects and handlers. *CoRR*.

[12] Benton, N. and Buchlovsky, P. (2007). Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 15–26, New York, NY, USA. ACM.

[13] Berger, E. D., Yang, T., Liu, T., and Novark, G. (2009). Grace: safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, New York, NY, USA. ACM.

[14] Bertino, E., Jajodia, S., and Samarati, P. (1996). Supporting multiple access control policies in database systems. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, SP '96, pages 94–, Washington, DC, USA. IEEE Computer Society.

[15] Blelloch, G. E. (1993). Prefix sums and their applications.

[16] Bocchino, R. L. and Adve, V. S. (2011). Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 306–332, Berlin, Heidelberg. Springer-Verlag.

[17] Bodden, E., Tanter, E., and Inostroza, M. (2014). Join point interfaces for safe and flexible decoupling of aspects. *ACM Trans. Softw. Eng. Methodol.*, 23(1):7:1–7:41.

[18] Boström, P. and Müller, P. (2015). Modular verification of finite blocking in non-terminating programs. In *ECOOP*, volume 37 of *Leibniz International Proceedings in Informatics*, pages 639–663. Schloss Dagstuhl – Leibniz-Zentrum fur Informatik.

[19] Boyapati, C., Lee, R., and Rinard, M. (2002). Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA. ACM.

[20] Burckhardt, S., Baldassin, A., and Leijen, D. (2010a). Concurrent programming with revisions and isolation types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 691–707, New York, NY, USA. ACM.

[21] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. (2010b). A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA. ACM.

[22] Burckhardt, S., Leijen, D., Sadowski, C., Yi, J., and Ball, T. (2011). Two for the price of one: A model for parallel and incremental computation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 427–444, New York, NY, USA. ACM.

[23] Burnim, J., Elmas, T., Necula, G., and Sen, K. (2012). Concurrit: Testing concurrent programs with programmable state-space exploration. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 16–16, Berkeley, CA, USA. USENIX Association.

[24] Cartwright, R. and Fagan, M. (1991). Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA. ACM.

[25] Chlipala, A. (2010). Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 122–133, New York, NY, USA. ACM.

[26] Choi, J.-D., Burke, M., and Carini, P. (1993). Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA. ACM.

[27] Chugh, R., Herman, D., and Jhala, R. (2012a). Dependent types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 587–606, New York, NY, USA. ACM.

[28] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. (2009). Staged information flow for JavaScript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62, New York, NY, USA. ACM.

[29] Chugh, R., Rondon, P. M., and Jhala, R. (2012b). Nested refinements: A logic for duck typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 231–244, New York, NY, USA. ACM.

[30] Clifton, C. and Leavens, G. T. (2006). MiniMAO$_1$: Investigating the semantics of proceed. *Sci. Comput. Program*, 63(3).

[31] Craciun, F., Chin, W.-N., He, G., and Qin, S. (2009). An interval-based inference of variant parametric types. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 112–127, Berlin, Heidelberg. Springer-Verlag.

[32] Crary, K., Weirich, S., and Morrisett, G. (1998). Intensional polymorphism in type-erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 301–312, New York, NY, USA. ACM.

[33] Cunningham, R. and Kohler, E. (2005). Making events less slippery with eel. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 3–3, Berkeley, CA, USA. USENIX Association.

[34] Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA. ACM.

[35] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

[36] Dillig, I., Dillig, T., and Aiken, A. (2008). Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 270–280, New York, NY, USA. ACM.

[37] Dyer, R. (2013). Task fusion: Improving utilization of multi-user clusters. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 117–118, New York, NY, USA. ACM.

[38] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013a). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA. IEEE Press.

[39] Dyer, R., Rajan, H., and Cai, Y. (2012). An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 143–154, New York, NY, USA. ACM.

[40] Dyer, R., Rajan, H., and Cai, Y. (2013b). Transactions on aspect-oriented software development x. chapter Language Features for Software Evolution and Aspect-oriented Interfaces: An Exploratory Study, pages 148–183. Springer-Verlag, Berlin, Heidelberg.

[41] Dyer, R., Rajan, H., Nguyen, H. A., and Nguyen, T. N. (2014). Mining billions of AST nodes to

study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA. ACM.

[42] Dyer, R., Rajan, H., and Nguyen, T. N. (2013c). Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 23–32, New York, NY, USA. ACM.

[43] Erickson, J., Musuvathi, M., Burckhardt, S., and Olynyk, K. (2010). Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 151–162, Berkeley, CA, USA. USENIX Association.

[44] Ernst, M. D., Kaplan, C. S., and Chambers, C. (1998). Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 186–211, London, UK, UK. Springer-Verlag.

[45] Eugster, P. and Jayaram, K. R. (2009). EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 570–594, Berlin, Heidelberg. Springer-Verlag.

[46] Fischer, J., Majumdar, R., and Millstein, T. (2007). Tasks: language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 134–143, New York, NY, USA. ACM.

[47] Flanagan, C. (2006). Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 245–256, New York, NY, USA. ACM.

[48] Flanagan, C. and Freund, S. N. (2009). Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA. ACM.

[49] Foster, J. S., Terauchi, T., and Aiken, A. (2002). Flow-sensitive type qualifiers. In *Proceedings of*

the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, pages 1–12, New York, NY, USA. ACM.

[50] Freeman, T. and Pfenning, F. (1991). Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA. ACM.

[51] Gordon, C. S., Dietl, W., Ernst, M. D., and Grossman, D. (2013). JavaUI: Effects for controlling UI object access. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 179–204, Berlin, Heidelberg. Springer-Verlag.

[52] Gotsman, A., Cook, B., Parkinson, M., and Vafeiadis, V. (2009). Proving that non-blocking algorithms don't block. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 16–28, New York, NY, USA. ACM.

[53] Greenhouse, A. and Boyland, J. (1999). An object-oriented effects system. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 205–229. Springer-Verlag, London, UK, UK.

[54] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60.

[55] Guha, A., Saftoiu, C., and Krishnamurthi, S. (2011). Typing local control and state using flow analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 256–275, Berlin, Heidelberg. Springer-Verlag.

[56] Hackett, B. and Guo, S.-y. (2012). Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA. ACM.

[57] Harper, R. and Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 130–141, New York, NY, USA. ACM.

[58] Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289.

[59] Heumann, S. T., Adve, V. S., and Wang, S. (2013). The tasks with effects model for safe concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 239–250, New York, NY, USA. ACM.

[60] Igarashi, A., Pierce, B., and Wadler, P. (1999). Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 132–146, New York, NY, USA. ACM.

[61] Krohn, M., Kohler, E., and Kaashoek, M. F. (2007). Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA. USENIX Association.

[62] Kulkarni, A., Liu, Y. D., and Smith, S. F. (2010). Task types for pervasive atomicity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 671–690, New York, NY, USA. ACM.

[63] Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., and Chew, L. P. (2008). Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 217–228, New York, NY, USA. ACM.

[64] Lea, D. (2000). A Java fork/join framework. In *JAVA '00*.

[65] Leroy, X. and Pessaux, F. (2000). Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377.

[66] Li, P. and Zdancewic, S. (2007). Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 189–199, New York, NY, USA. ACM.

[67] Long, Y., Bagherzadeh, M., Lin, E., Upadhyaya, G., and Rajan, H. (2016). On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 54–65, New York, NY, USA. ACM.

[68] Long, Y., Liu, Y. D., and Rajan, H. (2015). Intensional effect polymorphism. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 346–370.

[69] Long, Y., Mooney, S. L., Sondag, T., and Rajan, H. (2010). Implicit invocation meets safe, implicit concurrency. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 63–72, New York, NY, USA. ACM.

[70] Long, Y. and Rajan, H. (2013). First Class Effect. Technical Report 13-10, Iowa State University, Computer Science.

[71] Long, Y. and Rajan, H. (2016). A type-and-effect system for asynchronous, typed events. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 42–53, New York, NY, USA. ACM.

[72] Lucassen, J. M. and Gifford, D. K. (1988). Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA. ACM.

[73] Marino, D. and Millstein, T. (2009). A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, New York, NY, USA. ACM.

[74] Millstein, T. (2004). Practical predicate dispatch. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 345–364, New York, NY, USA. ACM.

[75] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

[76] Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., and Calder, B. (2007). Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA. ACM.

[77] Neamtiu, I., Hicks, M., Foster, J. S., and Pratikakis, P. (2008). Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 37–49, New York, NY, USA. ACM.

[78] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. (2006). Practical dynamic software updating for C. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 72–83, New York, NY, USA. ACM.

[79] Nguyen, H. A., Dyer, R., Nguyen, T. N., and Rajan, H. (2014). Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA. ACM.

[80] Nielson, F. and Nielson, H. R. (1999). Type and effect systems. In *Correct System Design*.

[81] Notkin, D., Garlan, D., Griswold, W. G., and Sullivan, K. J. (1993). Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, London, UK, UK. Springer-Verlag.

[82] Nystrom, N., Saraswat, V., Palsberg, J., and Grothoff, C. (2008). Constrained types for object-oriented languages. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 457–474, New York, NY, USA. ACM.

[83] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The information bus: An architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 58–68, New York, NY, USA. ACM.

[84] Philbin, J., Edler, J., Anshus, O. J., Douglas, C. C., and Li, K. (1996). Thread scheduling for cache locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 60–71, New York, NY, USA. ACM.

[85] Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

[86] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. (2011). The tao of parallelism in algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA. ACM.

[87] Rajan, H. (2005). *Unifying Aspect- and Object-Oriented Program Design*. PhD thesis, The University of Virginia, Charlottesville, Virginia.

[88] Rajan, H. (2007). Design pattern implementations in Eos. In *Proceedings of the 14th Conference on Pattern Languages of Programs*, PLOP '07, pages 9:1–9:11, New York, NY, USA. ACM.

[89] Rajan, H. (2010). Building scalable software systems in the multicore era. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 293–298, New York, NY, USA. ACM.

[90] Rajan, H. (2015). Capsule-oriented programming. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 611–614, Piscataway, NJ, USA. IEEE Press.

[91] Rajan, H., Kautz, S. M., and Rowcliffe, W. (2010). Concurrency by modularity: Design patterns, a case in point. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 790–805, New York, NY, USA. ACM.

[92] Rajan, H. and Leavens, G. T. (2007). Quantified, typed events for improved separation of concerns. Technical Report 07-14, Iowa State University, Department of Computer Science.

[93] Rajan, H. and Leavens, G. T. (2008). Ptolemy: A language with quantified, typed events. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 155–179, Berlin, Heidelberg. Springer-Verlag.

[94] Rajan, H., Nguyen, T. N., Leavens, G. T., and Dyer, R. (2015). Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 579–582, Piscataway, NJ, USA. IEEE Press.

[95] Rajan, H. and Sullivan, K. (2003). Eos: Instance-level aspects for integrated system design. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 297–306, New York, NY, USA. ACM.

[96] Rajan, H. and Sullivan, K. J. (2005). Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 59–68, New York, NY, USA. ACM.

[97] Rajan, H. and Sullivan, K. J. (2009). Unifying aspect- and object-oriented design. *ACM Trans. Softw. Eng. Methodol.*, 19(1):3:1–3:41.

[98] Ravichandran, K. and Pande, S. (2013). Multiverse: Efficiently supporting distributed high-level speculation. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 533–552, New York, NY, USA. ACM.

[99] Raychev, V., Vechev, M., and Sridharan, M. (2013). Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166, New York, NY, USA. ACM.

[100] Rossberg, A. (2008). Dynamic translucency with abstraction kinds and higher-order coercions. *Electron. Notes Theor. Comput. Sci.*, 218:313–336.

[101] Rytz, L., Odersky, M., and Haller, P. (2012). Lightweight polymorphic effects. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 258–282, Berlin, Heidelberg. Springer-Verlag.

[102] Safi, G., Shahbazian, A., Halfond, W. G. J., and Medvidovic, N. (2015). Detecting event anomalies in event-based systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 25–37, New York, NY, USA. ACM.

[103] Schmidt, D. C. (1995). Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern Languages of Program Design*, pages 529–545.

[104] Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

[105] Shivers, O. G. (1991). *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis.

[106] Siek, J. and Taha, W. (2007). Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 2–27, Berlin, Heidelberg. Springer-Verlag.

[107] Smith, D. and Cartwright, R. (2008). Java type inference is broken: Can we fix it? In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 505–524, New York, NY, USA. ACM.

[108] Steimann, F., Pawlitzki, T., Apel, S., and Kästner, C. (2010). Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1):1:1–1:43.

[109] Strom, R. E. and Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171.

[110] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 166–175, New York, NY, USA. ACM.

[111] Sullivan, K. J., Griswold, W. G., Rajan, H., Song, Y., Cai, Y., Shonle, M., and Tewari, N. (2010). Modular aspect-oriented design with XPIs. *ACM Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42.

[112] Sullivan, K. J. and Notkin, D. (1990). Reconciling environment integration and component independence. *SIGSOFT Softw. Eng. Notes*, 15(6):22–33.

[113] Talpin, J.-P. and Jouvelot, P. (1994). The type and effect discipline. *Inf. Comput.*, 111(2):245–296.

[114] Tan, G. and Morrisett, G. (2007). Ilea: inter-language analysis across Java and C. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 39–56, New York, NY, USA. ACM.

[115] Tofte, M. (1990). Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34.

[116] Toro, M. and Tanter, É. (2015). Customizable gradual polymorphic effects for Scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 935–953, New York, NY, USA. ACM.

[117] Treichler, S., Bauer, M., and Aiken, A. (2013). Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 495–514, New York, NY, USA. ACM.

[118] Wilson, R. P. and Lam, M. S. (1995). Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA. ACM.

[119] Xu, J., Rajan, H., and Sullivan, K. (2004). Understanding aspects via implicit invocation. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 332–335, Washington, DC, USA. IEEE Computer Society.

[120] Yang, X., Blackburn, S. M., Frampton, D., Sartor, J. B., and McKinley, K. S. (2011). Why nothing matters: The impact of zeroing. In *Proceedings of the 2011 ACM International Conference*

on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 307–324, New York, NY, USA. ACM.

[121] Yi, J. and Flanagan, C. (2010). Effects for cooperable and serializable threads. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 3–14, New York, NY, USA. ACM.