UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GRADUAL TYPING FOR GENERIC TYPE-AND-EFFECT SYSTEMS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FELIPE ANDRÉS BAÑADOS SCHWERTER

PROFESOR GUÍA:
ÉRIC TANTER
PROFESOR CO-GUÍA:
RONALD GARCIA

MIEMBROS DE LA COMISIÓN:
PABLO BARCELÓ BAEZA
ALEXANDRE BERGEL
ISMAEL FIGUEROA PALET

Este trabajo ha sido parcialmente financiado por CONICYT-PCHA Magíster Nacional 2013-22130167, Proyecto FONDECYT 1110051 y por Ayudas para Estadías Cortas de Investigación destinadas a Estudiantes Tesistas de Doctorado y Magíster de la Universidad de Chile, VAA-DPP

SANTIAGO DE CHILE
2014

Resumen

Los sistemas de tipos-y-efectos (type-and-effect systems) permiten a los programadores hacer valer invariantes y restricciones sobre los efectos secundarios que se generan durante la evaluación de un programa. Los sistemas de tipos-y-efectos consideran efectos secundarios tales como estado, excepciones y E/S, entre otros. Desafortunadamente, los sistemas de tipos-y-efectos también obligan al programador a introducir anotaciones de efectos, lo que implica un esfuerzo adicional. En la práctica, los sistemas de tipos-y-efectos no son comúnmente usados. Conjeturamos que una de las razones importantes para la limitada adopción de los sistemas de efectos son las dificultades para realizar la transición desde un sistema donde los efectos secundarios son implícitos hacia una disciplina de efectos totalmente estática.

Los tipos graduales (Gradual typing) permiten a los programadores combinar la flexibilidad de los lenguajes dinámicamente tipados con las garantías provistas por los sistemas de tipos estáticos. En lenguajes con tipos graduales, las anotaciones de tipos son parte del lenguaje, pero no son obligatorias. Un sistema de tipos gradual utiliza la información disponible para proveer garantías estáticas, rechazando los programas claramente incoherentes, e introduce verificaciones en tiempo de ejecución cuando la información estática no es suficiente para aceptar o rechazar definitivamente un programa.

Esta tesis demuestra que las ideas de diseño detrás de los tipos graduales pueden aplicarse a los sistemas de tipos-y-efectos, tanto para aumentar la expresividad de estos sistemas así como para proveer flexibilidad para migrar programas con efectos secundarios implícitos e irrestrictos hacia programas con una disciplina de efectos completamente estática.

Se adaptaron ideas de tipos graduales para introducir *verificación gradual de efectos* para sistemas de tipos-y-efectos. La verificación gradual de efectos habilita al programador para decidir dónde y cuándo introducir anotaciones de efectos, agregando verificaciones en tiempo de ejecución cuando las anotaciones estáticas so ninsuficientes. Para evitar redefinir la verificación gradual de efectos para cada disciplina de tipos-y-efectos, introducimos verificación gradual de efectos para una plataforma genérica de tipos-y-efectos, en la que se puede instanciar cualquier disciplina de efectos monotónica, produciendo un sistema coherente. Presentamos la verificación gradual de efectos basándonos en conceptos de interpretación abstracta para construir la verificación gradual de efectos genérica.

Utilizando verificación gradual de efectos genérica, introducimos tipos graduales para sistemas de tipos-y-efectos: un sistema donde las anotaciones de efectos y de tipos no son obligatorias, y donde se introducen verificaciones en tiempo de ejecución y "casts" cuando la información estática no es suficiente para asegurar la coherencia de un programa. De la manera definida, los tipos graduales para sistemas de tipos-y-efectos permiten migrar desde sistemas carentes de anotaciones de efectos o de tipos hacia una disciplina estática de tipos-y-efectos de manera segura.

Abstract

Type-and-effect systems allow programmers to enforce invariants and restrictions about the side effects generated by the evaluation of a program. Type-and-effect systems consider side effects like state, exceptions, and I/O, among others. Unfortunately, type-and-effect systems also charge programmers with the overhead of introducing effect annotations. In practice, type-and-effects systems are not commonly used. We conjecture that an important reason for the limited adoption of effect systems is the difficulty to transition from a system where side effects are implicit towards a fully static effect discipline.

Gradual typing enables programmers to combine the flexibility of dynamically typed languages with the safety of static type systems. In gradual typing, type annotations are allowed but are not mandatory. Gradual typing uses the available information to provide static guarantees, discarding clearly unsound programs, and introduces runtime checks whenever static information is not sufficient to definitely accept or reject a program.

This thesis demonstrates that design ideas from gradual typing can be applied to type-and-effect systems, both to increase the expressivity of type-and-effect systems, and to provide flexibility for migrating programs with unrestricted and implicit side effects towards programs with a fully static effect discipline.

We adapt ideas from gradual typing to introduce *gradual effect checking* for type-and-effect systems. Gradual effect checking enables the programmer to decide when and where to introduce effect annotations, introducing runtime checks whenever static annotations are insufficient. To avoid redefining gradual effect checking for each type-and-effect discipline, we introduce gradual effect checking for a generic type-and-effect framework, in which any monotonic effect discipline can be instantiated to produce a sound system. We present gradual effect checking in terms of abstract interpretation concepts to build generic gradual effect checking.

Using generic gradual effect checking, we introduce gradual typing for type-and-effect systems: a system where effect and type annotations are not mandatory, and where runtime checks and casts are introduced when static information is not sufficient to ensure safety. As defined, gradual typing for type-and-effect systems permits migration from systems without effect or type annotations towards a static type-and-effect discipline safely.

Agradecimientos

A mi familia, en especial a mis padres Julio y María Luisa, por todo el cariño y apoyo que me han dado a lo largo de la vida. También a mis hermanas, Javiera y Cecilia, que me aguantaron mañas y flojeras, y a mis abuelos Samuel, Gualda, Ulda y Teo.

A mis amigos. A Alfredo, Jorge y Naty, por todas las comidas, conversas y en fin, por estar siempre cerca. A todos los que me acompañaron en el recorrido por la Universidad: a Marcel, Muguette, Valverde, Fernanda, Gustavo y tantos otros que hicieron de Beauchef un lugar memorable.

A la Compañía de Jesús, por su formación, por su confianza, por los Ejercicios y por permitirme ser parte de la familia ignaciana. Gracias en especial a todos los jesuitas que me han acompañado a lo largo de la vida. A Eugenio Valenzuela, Pablo Peña, Gerardo Schmidt, Carlos Vidal, José Tomás Vicuña, Nicolás Oelckers, René Cortínez, Pablo Walker, Pedro Labrín, y a John O'Brien, entre otros. Gracias también a los jesuitas que ya no están, muy especialmente al padre Raúl Combes.

A los profesores que me interesaron en la investigación científica y en distintos campos del saber. Gracias a Éric Tanter por abrir las puertas al área de lenguajes, por su confianza y por la libertad que me dio para trabajar en esta tesis. Gracias a Ronald Garcia por su paciencia, por aceptarme como futuro estudiante de doctorado y por re-enseñarme a escribir. A Alexandre Bergel por enseñarme Smalltalk, por investigar juntos y por los papers que escribimos. A Eden Medina por su confianza y por transformar mi visión de la computación como disciplina. A Claudio Gutiérrez por tantas conversaciones, y en especial por aumentar mi interés en la historia de la computación en Chile. Finalmente, gracias a Ciro Schmidt por su paciencia y consejos.

To the University of British Columbia and St. John's College for being such great hosts, and to all the people and friends I have met there, especially Guillaume, Abhijit, Pat, Sheila, Farhad, and Ian.

And especially and most importantly, this whole work is dedicated to Sarah, without whom this thesis would not have been finished yet[1].

[1]Or might have actually been finished months earlier!

Contents

List of Figures

Chapter 1

Introduction

A type-and-effect system is an advanced kind of type system for a programming language. Like any type system, a type-and-effect system analyses programs without evaluating them, rejecting those programs that do not follow a set of typing rules that enforce semantic invariants for programs. Examples of programs usually rejected by type systems are programs that try to use a number as a function, programs that provide an incorrect argument to a function, and programs that try to use a string instead of a boolean condition for an `if` statement. A type-and-effect system is different from a standard type system in that it also verifies invariants about the side-effects generated when evaluating the program, and rejects programs that neglect the imposed restrictions for side-effects. Common side-effects considered by type-and-effect systems are exceptions, I/O and state. Examples of languages with type-and-effect systems are Scala [18] and Koka [14].

Type-and-effect systems impose strong restrictions for the development process that may end up discouraging developers from using them. To analyze the side effects a program may produce, a type-and-effect system relies on effect annotations. Effect annotations establish a limit on which side effects may be generated when a program is evaluated. These annotations impose an overhead for the programmer. With a type-and-effect system, a programmer must consider the side effect restrictions for her programs early on the programming process, restrictions that may be unknown or may change throughout the development process. In general, programmers are unlikely to write fully-effect annotated programs, though notable exceptions include the Haskell[1] and Clean[2] programming languages.

We conjecture that an important reason for the limited adoption of effect systems is the difficulty of transitioning from a system where side effects are implicit and unrestricted to a system with a fully static effect discipline. Another explanation is that effect systems are necessarily conservative and therefore occasionally reject valid programs.

A similar dilemma had previously arised between statically typed languages (like Haskell, ML, and Java) and dynamically typed languages (like Smalltalk, Scheme, and Javascript). While statically typed languages provide tighter static guarantees and reject semantically

[1]`http://www.haskell.org`
[2]`http://wiki.clean.cs.ru.nl/Clean`

incorrect programs, they also reject some correct programs and force programmers to decide type signatures early on the development process, incurring overhead for prototyping. Dynamically typed languages, on the other hand, ease prototyping by not requiring explicit type signatures, but at the same time do not provide any static guarantees about program safety.

To bridge the gap between both kinds of languages, Siek and Taha [24] proposed Gradual Typing as a flexible kind of type system where type annotations are allowed but are not mandatory. Gradual typing empowers the programmer to decide when and where to introduce type annotations, thus providing the flexibility required for prototyping while still providing static guarantees, using any existing type annotations to check for program inconsistencies. If programs are fully annotated with types at some point of the development process, gradual typing provides the same guarantees as static type systems.

Gradual type systems have been introduced for functional and object oriented languages among others, but no previous work has explored the interactions between gradual typing and type-and-effect systems. In this work we support the thesis that **gradual typing design ideas can be applied to type-and-effect systems, both to increase the expressivity of type-and-effect systems, and to provide flexibility for migrating programs with unrestricted and implicit side effects towards programs with a fully static effect discipline.** The research presented in this work is part of the area of programming languages, particularly on the design and definition of programming language semantics and type systems.

The system we propose provides the programmer with the choice of introducing effect annotations in a gradual way. Gradual effect annotations increase the expressivity of the language, accepting as valid some programs whose effect restrictions could not be accepted by a static type-and-effect system, which must necessarily make conservative estimations. Our system also lifts the requirement of mandatory full effect annotations in type-and-effect systems and verifies that annotated and unannotated sections of a program are coherent, providing the programmer with the flexibility to decide when and where to introduce effect annotations in programs. We develop *gradual effect checking* to generate a type-and-effect system which provides gradual effect annotations, using ideas from gradual typing. Using gradual effect checking, we introduce *gradual typing for type-and-effect-systems*, a framework that fully combines gradual typing and type-and-effect systems, providing the programmer with the flexibility of both gradual types and gradual effects. The contributions of our work are:

- **Introduce gradual effect checking**.We explore the ideas required to define gradual effect checking through a simple language with a type-and-effect system extracted from the literature. In the spirit of gradual typing, we introduce a type system with consistency for the language, an algorithm that translates programs into an intermediate language and inserts explicit runtime checks for the assumptions made by the type system, and a runtime semantics for the intermediate language. The intermediate language strictly checks effect restrictions, so the translation algorithm must insert all the required checks to ensure that every program fullfills the effect discipline.
- **Redefine gradual effect checking for a generic type-and-effect framework**.

We avoid the need to reintroduce gradual effect checking for every particular effect discipline by providing a generic framework for type-and-effect systems. To do so, we reuse the work of Marino and Millstein [15], in which they introduce a generic framework for type-and-effect systems (M&M). We apply concepts from the theory of abstract interpretation [8] to generalize and justify our design choices. With abstract interpretation, we are able to provide gradual effect checking for the generic framework without having to introduce restrictions beyond those already imposed by the original framework: Any effect discipline defined in terms of the generic framework can automatically receive the benefits of gradual effect checking. We provide a type safety [28] proof for the language proposed.

- **Provide alternative evaluation semantics for the gradual effect checking generic system**. The evaluation semantics of the original generic framework preserves some information at runtime only because it is required to prove type safety, information that could therefore be removed in an implementation. In the generic gradual effect checking framework, some of that runtime information is used to enforce the effect discipline. We introduce two different runtime semantics for the generic framework with gradual effect checking. These two semantics give the language implementor a choice between complete fidelity to the effect discipline of the original generic framework and reduced space overhead at runtime. We call the semantics with reduced space footprint a *conservative semantics*, because if a program evaluates to a value in the conservative semantics, it also does in the original semantics modulo a set of value annotations (tags). Both semantics are safe, so the language implementor can decide which semantics to choose.
- **Introduce gradual typing for a generic type-and-effect system**. Our final result is a system that combines gradual effect checking with gradual typing. We provide two separate systems, one for a simplified version of the generic framework (without tags) and one for the full generic framework. We introduce the simplified generic framework to show that, without tag information, combining gradual typing and gradual effect checking is trivial because all the elements required are encapsulated in gradual effect checking. In the full generic framework, gradual typing has to manage uncertainty from the annotations used in values and types, so introducing gradual typing becomes more complicated. We propose a simple system that makes pessimistic assumptions for the annotations while ensuring safety, based on the restrictions imposed by the M&M framework.

The rest of this document is organized as follows: Chapter 2 introduces the ideas that will be assumed familiar to the reader throughout the rest of our work. It explores type systems and language semantics, type safety, type-and-effect systems, and the generic type-and-effect framework. It also introduces a language with a simple type-and-effect system for state related side effects, called the fluent language. Chapter 3 introduces *gradual effect checking* in the context of the fluent language, exploring the path that led to our final design for gradual effect checking. Chapter 4 presents gradual effect checking for the generic M&M framework. Our development revisits the concepts required for gradual effect checking in terms of abstract interpretation, introducing the required concepts as needed.

Chapter 5 introduces the conservative semantics for the generic gradual effect checking framework, and establishes how this new semantics relates to the original semantics introduced in chapter 4. This relation is formalized through what we call a conservative approximation theorem. The chapter also shows an example program that has different behaviors among both semantics.

Chapter 6 introduces *gradual typing for type-and-effect systems*. This chapter explores gradual typing for both a simplified generic type-and-effect system and for the full system introduced in chapter 4. In the simplified language, we show that gradual effect checking encapsulates the effect-related requirements to introduce gradual typing for type-and-effect systems. The full system presents new challenges with respect to handling tags in gradual typing, making pessimistic assumptions for the privileges available.

The appendices in this document present detailed proofs for theorems stated throughout our work. Appendix A revisits the fluent language introduced in chapter 2 and presents in detail how it can be considered as an instantiation of the M&M type-and-effect system, providing a formal example of how to use the M&M framework. Appendix B provides type safety proofs for gradual effect checking as presented in chapter 3, and appendix C provides type safety proofs for generic gradual effect checking as presented in chapter 4.

Part of the research results presented in this work have already been published in:

- Felipe Bañados, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2014.

Chapter 2

Background and Related Work

This chapter introduces several ideas and concepts required to understand our work. We assume familiarity with these ideas throughout the rest of this thesis. We provide a summary of the ideas of the chapter in section 2.6, so that readers with previous knowledge of them may completely skip the rest of this chapter.

2.1 Type systems and language semantics

This section introduces the concepts of type systems and operational semantics. A type system, introduced in section 2.1.2, is a tool used in programming languages to prevent the occurrence of a collection of errors during execution, called type errors. A structural operational semantics, introduced in section 2.1.1, specifies the runtime behavior of a language with a transition relation that represents one step of evaluation. We also present in this section examples of a type system and an operational semantics for two programming languages, the lambda calculus introduced in section 2.1.1 and the simply typed lambda calculus (STLC) introduced in section 2.1.2.

After introducing these concepts, we restrict the valid interaction between the type system and the language semantics by defining *type safety*, concept introduced in section 2.1.3. We finish this section constructing the base language we will use to define type-and-effect systems, by defining a base type `Unit` for the STLC in section 2.1.4, and introducing a model of mutable references in section 2.1.5.

2.1.1 The lambda calculus

The lambda calculus is a formal system introduced by Church [7] that works as a programming language and is used throughout the literature to develop programming languages theory. In the lambda calculus, every valid expression is either a λ abstraction, a variable, or an application of expressions. Abstractions, denoted $(\lambda x\ .\ \mathrm{e})$, consist of a parameter x

and body e. The parameter x is an identifier that can be used as a variable in the body of the function, and will only be replaced by a value when evaluating a function application. Applications are denoted $\mathrm{e}_1\ \mathrm{e}_2$, where e_2 is the argument set to replace every occurrence of the parameter of e_1 in its body. The set of all valid programs[1] in the lambda calculus can be defined in Backus-Naur Form (BNF) as follows:

$$\begin{array}{lcll} v & ::= & (\lambda x\ .\ \mathrm{e}) & \text{Values} \\ \mathrm{e} & ::= & v \mid x \mid \mathrm{e}\ \mathrm{e} & \text{Expressions} \end{array}$$

Different kinds of values (booleans, natural numbers, etc.) can be represented in the lambda calculus by encoding them with abstractions [3], but for simplicity they are usually introduced explicitly. We extend our lambda calculus with natural numbers, n. The procedure to introduce booleans and other basic values in the language is analogous.

$$\begin{array}{lcll} n & ::= & 0 \mid 1 \mid 2 \mid \ldots & \text{Numbers} \end{array}$$

We can also introduce basic operations over the values in the language. Since we have introduced natural numbers, it would also be interesting to include operations like addition or multiplication. These operations are called *primitives*. Though addition and multiplication are both binary operations, there is no restriction on the amount of arguments that a primitive operation uses: the boolean "not" operator receives one argument, and an operation to build empty lists receives none. In our natural numbers example, we only introduce binary operations.

$$\begin{array}{lcll} v & ::= & (\lambda x\ .\ \mathrm{e}) \mid n & \text{Values} \\ p & ::= & + \mid \times & \text{Primitive Binary Operations} \\ \mathrm{e} & ::= & v \mid x \mid \mathrm{e}\ \mathrm{e} \mid \mathrm{e}\ p\ \mathrm{e} & \text{Expressions} \end{array}$$

Operational semantics for the lambda calculus

A structural operational semantics specifies how a language performs steps of evaluation, defining an abstract machine for the language that transforms language expressions. We introduce an operational semantics by presenting a set of evaluation rules. These rules specify the kinds of expressions that can make a step of evaluation, and the new expressions to which they evaluate.

Evaluation rules in an operational semantics define a transition relation. The transition relation $\rightarrow$ associates an expression in the language with a new expression, also in the lan-

[1]We will use indistinguishably the names program, term and expression for syntactically valid elements of a language

guage ($\rightarrow \subseteq \mathrm{e} \times \mathrm{e}$). We use the transitive-reflexive closure of this relation ($\rightarrow^*$) to define evaluation.

The core transition step in the lambda calculus is application, defined for expressions of the form ($\mathrm{e}_1\ \mathrm{e}_2$). If e_1 is an abstraction, an expression of the form $\lambda x\ .\ \mathrm{e}'$, every occurrence of x in the body e' of the function can be replaced by e_2. We use notation $[{}^{\mathrm{e}_2}/_{x}]\,\mathrm{e}_1$ to express that every occurrence of x in e_1 is substituted by e_2.

We follow a call-by-value strategy to evaluate function applications. Call-by-value means that in order to evaluate an application ($\mathrm{e}_1\ \mathrm{e}_2$), both e_1 and e_2 need to be evaluated to a value first. We evaluate expressions from left to right, so we first evaluate e_1 with repeated application of the rule [E-App-1], hoping to reach a value. If we cannot reach a value, evaluation gets stuck. If we reach a value, rule [E-App-2] continues evaluation on e_2. Again, if we do not reach a value in the argument position by repeated use of rule [E-App-2], evaluation gets also stuck, though in a different state than the case for [E-App-1].

If we do reach a value by using [E-App-2], we can try to use rule [E-App]. If the value in the e_1 position of an application is not a function, evaluation gets stuck because rule [E-App] expects an abstraction in that location. Otherwise, evaluation proceeds by substitution.

$$\text{E-App}\ \frac{}{(\lambda x\ .\ \mathrm{e})\ v \rightarrow [{}^{v}/_{x}]\,\mathrm{e}} \qquad \text{E-App-1}\ \frac{\mathrm{e}_1 \rightarrow \mathrm{e}_1'}{\mathrm{e}_1\ \mathrm{e}_2 \rightarrow \mathrm{e}_1'\ \mathrm{e}_2} \qquad \text{E-App-2}\ \frac{\mathrm{e}_2 \rightarrow \mathrm{e}_2'}{v\ \mathrm{e}_2 \rightarrow v\ \mathrm{e}_2'}$$

Rules [E-App-1] and [E-App-2] have a similar structure. Both declare how to make a step of evaluation by recursively evaluating part of the expression. If we follow this style, every time we introduce a new language construct we also need to introduce new rules for recursively evaluating parts of it. Even with a few features, like in the languages we introduce throughout this work, this requires introducing several very similar evaluation rules, adding a lot of overhead when reading the semantics and the formal proofs. We refactor these similar rules, abstracting the recursive evaluation patterns in one rule that uses evaluation frames [17].

An evaluation frame represents which part of an expression has to be evaluated first, marking the position with a hole ($\square$). The hole in a frame f may be replaced by any expression e, and the replacement is denoted $f[\mathrm{e}]$. For example, an expression ($\mathrm{e}_1\ \mathrm{e}_2$) may be represented as a ($\square\ \mathrm{e}_2$) frame, in which the hole is replaced by e_1, $(\square\ \mathrm{e}_2)[\mathrm{e}_1]$. To avoid ambiguity, we require frames to have only one hole. Whenever we introduce a feature that requires to first evaluate a sub expression, we use rule [E-Frame]

$$\text{E-Frame}\ \frac{\mathrm{e} \rightarrow \mathrm{e}'}{f[\mathrm{e}] \rightarrow f[\mathrm{e}']}$$

We then introduce different frames f to encapsulate the evaluation order for sub parts of an expression. In the language introduced so far, there are two kinds of evaluation frames for function application and two for primitive operations: ($\square$ e), (v $\square$), ($\square$ p e) and (v p $\square$).

These evaluation frames encapsulate rules [E-App-1] and rule [E-App-2], respectively.

$$f \quad ::= \quad \square \; \mathrm{e} \mid v \; \square \mid \square \; p \; \mathrm{e} \mid v \; p \; \square \quad \text{Evaluation Frames}$$

2.1.2 The simply typed lambda calculus (STLC)

In this section, we introduce a programming language widely used as a formal foundation, called the simply typed lambda calculus (STLC) [6]. We use this language as a research basis throughout this thesis.

What is a type system

A type system [5] is a syntactical method to analyze programs. Type systems are used to prevent the occurrence of a collection of errors during execution, called type errors. The definition of type errors differs from system to system, but in general it includes the use of a function on arguments for which it is not defined, and trying to use a non-function as a function in an application [28].

Type systems are formally defined by a set of inference rules. An inference rule associates a group of language expressions with a *type*, a range of values to which the expressions may evaluate. Simple examples of types are `Boolean` for booleans and `Nat` for natural numbers.

Inference rules are formed by a set of premises and one conclusion. We create an instance of an inference rule for a particular program by showing that every premise holds. If the set of premises is empty, then the rule has no prerequisites and can always be instantiated for its domain of programs. As an example, we introduce an inference rule to infer a type for natural numbers. We define a type `Nat` for the natural numbers. The inference rule is written as follows:

$$\frac{n \in \mathbb{N}}{n \colon \mathtt{Nat}} \begin{array}{l} \leftarrow \text{Premise(s)} \\ \leftarrow \text{Conclusion} \end{array}$$

This inference rule is read "If $n \in \mathbb{N}$, then n has type `Nat`.". An instance of this rule for the natural number 42 is written

$$\frac{42 \in \mathbb{N}}{42 \colon \mathtt{Nat}}$$

In more complex programs, instances of inference rules can be stacked transitively to fulfill premises of a rule, producing a *type derivation*. A type derivation is an existence proof that an expression e is related with a type T. If there exists a type derivation for a program, the program is called "well-typed".

A type system introduces restrictions over what programs are valid in a particular language. It is usual to use the type system as an execution filter, allowing only well-typed programs to run. When the language allows the introduction of type annotations in programs, inference rules can use type annotations to enforce restrictions over which programs to accept.

A type system approximates the result of a program without evaluating it. Type information may also be used as a form of documentation, used to verify consistency of software, or even used by a compiler to apply certain optimizations.

Introducing types into the lambda calculus

The lambda calculus introduced in section 2.1.1 does not distinguish between abstractions, natural numbers, nor any kind of value. It is a *dynamically typed* language, meaning that values are not classified statically among different types. In a dynamically typed language, the only way to restrict the valid arguments for an abstraction is to insert an explicit verification in the runtime semantics, verification that will need to be performed for every application. To statically discard invalid arguments for abstractions, a type system can be used.

STLC is a type restricted version of the lambda calculus. In STLC, the parameters of λ abstractions are annotated with a type. This type annotation limits the valid arguments for the abstraction. Values for the language are then defined as

$$v \quad ::= \quad \lambda x\colon T \ . \ \mathrm{e} \mid n \quad \text{Values}$$

We can classify values in this language in two types: natural numbers ($\texttt{Nat}$) and abstractions ($T_1 \longrightarrow T_2$), which receive an argument of type T_1 and produce a result of type T_2.

$$T \quad ::= \quad \texttt{Nat} \mid T \longrightarrow T \quad \text{Types}$$

By restricting the type of the parameters of abstractions, function applications that use the abstraction also have to be restricted. So far we have only presented syntactic restrictions, which do not exclude programs like $(\lambda f\colon \texttt{Nat}\longrightarrow\texttt{Nat} \ . \ f\ 1)\ 2$, nor limits application only to functions (4 2 is a syntactically valid program). We express these restrictions in the definition of a type system.

If type restrictions map to restrictions over values, we can reason about valid expressions looking only at their types. For example, if values with a function type are always abstractions and never numbers, we can discard programs like (4 2) by simply limiting valid (e_1 e_2) applications to the cases where e_1 has a function type. A *canonical forms* lemma encodes how restrictions over types map to restrictions over values. We present this lemma for STLC after introducing all the type inference rules available in the type system.

The type system requires inference rules for every language construct available. Typing abstractions requires to type their body under the assumption that the argument has the type declared for the parameter. To make this information available, we use type environments, denoted Γ. A type environment is a partial function from identifiers to types. The type environment is given as context for the typing relation. The typing relation is now denoted $\Gamma \vdash \mathrm{e} \colon T$, which reads as "e has type T under Γ".

To type an identifier x, we need to obtain the type from the type environment. We do this in rule [T-Var] by accessing $\Gamma(x)$. Rule [T-Nat] provides types for numbers. Rule [T-Abs] types abstractions, extending the type environment with the restriction declared in the function parameter to infer the type of the function body. Rule [T-App] only accepts applications where the first element is of function type, and discards applications where the argument is not of the type expected for the function parameter.

$$\text{T-Var}\,\frac{\Gamma(x) = T}{\Gamma \vdash x \colon T} \qquad \text{T-Abs}\,\frac{\Gamma, x \colon T_1 \vdash \mathrm{e} \colon T_2}{\Gamma \vdash \lambda x \colon T_1\ .\ \mathrm{e} \colon T_1 \longrightarrow T_2} \qquad \text{T-App}\,\frac{\begin{array}{c}\Gamma \vdash \mathrm{e}_1 \colon T_1 \longrightarrow T_2 \\ \Gamma \vdash \mathrm{e}_2 \colon T_1\end{array}}{\Gamma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_2}$$

$$\text{T-Nat}\,\frac{n \in \mathbb{N}}{\Gamma \vdash n \colon \mathtt{Nat}} \qquad \text{T-Prim}\,\frac{\begin{array}{c}\Gamma \vdash \mathrm{e}_1 \colon \mathtt{Nat} \\ \Gamma \vdash \mathrm{e}_2 \colon \mathtt{Nat}\end{array}}{\Gamma \vdash \mathrm{e}_1\ p\ \mathrm{e}_2 \colon \mathtt{Nat}}$$

This type system identifies structurally invalid programs before evaluation. We can now present the definition of a canonical forms lemma [19] for STLC:
Lemma 1 (Canonical Forms)**.** This lemma is composed of the following two sub lemmas:

1. *If* $\Gamma \vdash v \colon \mathtt{Nat}$*, then* v *is a natural number* ($v \in \mathbb{N}$).
2. *If* $\Gamma \vdash v \colon T_1 \longrightarrow T_2$*, then* $v = (\lambda x \colon T_1\ .\ \mathrm{e})$*, for some expression* e.

Proof. There is only one typing rule for values of each type (`Nat` and function types). We can then infer the conclusions by inverting the inference rules. Though rule [T-Prim] also assigns type `Nat`, neither $\mathrm{e}_1 + \mathrm{e}_2$ nor $\mathrm{e}_1 \times \mathrm{e}_2$ are values. □

This canonical forms lemma is useful when proving properties of the type system, like type safety as introduced in section 2.1.3. We now move forward to define evaluation for STLC in terms of an operational semantics.

2.1.3 Type safety

As we introduced the operational semantics, we mentioned several cases in which evaluation should not proceed, like the application (4 2). We want to identify statically if a program will evaluate to a value or will get stuck in an intermediate state. Though this is the goal of the type system, we need to verify that the operational semantics and the type system are consistent with each other. To be consistent, the static guarantees provided by the type system must be honored by evaluation, and the type system must produce an estimation of the result of evaluating a program in the semantics. This consistency restriction between the type system and the semantics is called type safety.

Type safety or type soundness is a formalization of the notion that programs accepted by the type system (or that are well-typed) do not "go wrong"[16]. As Pierce mentions in [19], Harper characterized soundness as the sum of two properties, progress and preservation:

- **Progress**: *If an expression is well-typed, then is either a value, or it can perform a step of evaluation in the semantics.* This property links the type system to the semantics, making a well-typed expression a valid starting point for evaluation. It also means that non-final expressions can always perform a new evaluation step.
- **Preservation**: *If a well-typed expression performs a step of evaluation, then the resulting expression is also well-typed, and has the same type.* This property, also called subject-reduction [28], links back the semantics with the type system. With preservation we state that the semantics does not break the typing relation, so the progress property will also hold with the expression resulting from performing an evaluation step.

With these properties, the transitive and reflexive closure of the transition relation $\rightarrow$ can be used to evaluate any well-typed expression, and evaluation will always reach a value. Both progress and preservation properties hold for STLC[2], meaning that the STLC is type safe.

2.1.4 The `Unit` type

We now have a language with natural numbers, abstractions and function application. However, we do not use natural numbers for anything interesting. We may either extend the language with features that make use of natural numbers, like addition, multiplication and others, or for simplicity we remove numbers. We follow the second approach.

If we remove numbers, we can also delete the `Nat` type. Unfortunately, without `Nat` our inductive definition of types becomes not well-founded, since we removed the base case. To create concrete types, at least one base type is needed. For a base type to be of interest, there must be at least one value that holds that type. We replace numbers with a type that fulfills these two minimal requirements. In the literature, this simple base type is called `Unit`, and only represents the value `unit`. Numbers, booleans, and any other interesting type can afterwards be added to the language if so desired, but we keep the language as simple as possible for our analysis.

$$\begin{array}{lcll} v & ::= & \lambda x\colon T\ .\ \mathrm{e} \mid \texttt{unit} & \text{Values} \\ T & ::= & \texttt{Unit} \mid T \longrightarrow T & \text{Types} \end{array}$$

Since `unit` is a value, it does not have any associated evaluation rules, and thus we can keep the operational semantics unaltered. At the type system level, we remove the rule [T-

[2]Formal statements of the progress and preservation properties for an STLC with booleans instead of numbers, and a proof for progress are presented in chapter 9 of Pierce [19], a proof for the preservation property is presented in appendix A.

Nat] that we used for natural numbers, and we introduce a type inference rule for the `unit` value, [T-Unit].

$$\text{T-Unit}\,\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}$$

2.1.5 Extending the simply typed lambda calculus with references

Throughout this work, we extend different languages with new features. In this section we show a first extension for STLC standard in the literature [12, 28]: heap memory allocation or state, a feature present in most programming languages. We introduce this language as the starting point for the type-and-effect system introduced in section 2.4. We follow the syntax presented by Pierce [19]. Both the type system and the operational semantics are presented in full detail in fig. 2.1.

To handle state-related actions we introduce three new language constructs: ($\texttt{ref}$ e) for allocation, (!e) to read the contents of a memory location, and ($\mathrm{e}_1 := \mathrm{e}_2$) to assign a new value in a memory location. To model the memory space, we use a partial function from memory locations to values called a *store* and denoted μ. The value $\mu(l)$ represents the contents stored at location l.

$$\begin{array}{lcll} v & ::= & \ldots \mid l & \text{Values} \\ \mathrm{e} & ::= & \ldots \mid \texttt{ref}\ \mathrm{e} \mid !\mathrm{e} \mid \mathrm{e} := \mathrm{e} & \text{Expressions} \end{array}$$

The reduction relation needs to consider the memory state, and therefore has to carry a particular store throughout evaluation. Rules that do not operate over state like [E-App] just preserve the store without changes. Rule [E-Frame] transfers the recursive changes to the store, where the notation e | μ represents a pair of an expression and a store:

$$\text{E-Frame}\,\frac{\mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{f[\mathrm{e}] \mid \mu \to f[\mathrm{e}'] \mid \mu'}$$

And the new expressions require extending our definition of evaluation frames:

$$\begin{array}{lcll} f & ::= & \ldots \mid \texttt{ref}\ \Box \mid !\Box \mid \Box := \mathrm{e} \mid v := \Box & \text{Evaluation Frames} \end{array}$$

New language constructs that are not values require also new evaluation rules. Allocation rule [E-Ref] adds a value into the store and returns the location value with the new store. Rule [E-Deref] recurs to the μ function to read a value from the store. Rule [E-Asgn] performs assignment, changing the partial function to return the new value. Notation $\mu[l \mapsto v]$

$$
\begin{array}{llll}
T & ::= & \texttt{Unit} \mid T \longrightarrow T \mid \texttt{Ref}\ T & \text{Types} \\
v & ::= & \texttt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} \mid l & \text{Values} \\
\mathrm{e} & ::= & v \mid x \mid \mathrm{e}\ \mathrm{e} \mid \texttt{ref}\ \mathrm{e} \mid !\mathrm{e} \mid \mathrm{e} := \mathrm{e} & \text{Expressions} \\
f & ::= & \square\ \mathrm{e} \mid v\ \square \mid \texttt{ref}\ \square \mid !\square \mid \square := \mathrm{e} \mid v := \square & \text{Evaluation Frame}
\end{array}
$$

Operational Semantics

$$\text{E-Frame}\ \frac{\mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{f[\mathrm{e}] \mid \mu \to f[\mathrm{e}'] \mid \mu'} \qquad \text{E-App}\ \frac{}{(\lambda x\colon T\ .\ \mathrm{e})\ v \mid \mu \to [{}^{v}/_{x}]\,\mathrm{e} \mid \mu}$$

$$\text{E-Ref}\ \frac{l \notin \mathrm{dom}\,(\mu)}{\texttt{ref}\ v \mid \mu \to l \mid \mu[l \mapsto v]} \qquad \text{E-Deref}\ \frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu}$$

$$\text{E-Asgn}\ \frac{l \in \mathrm{dom}\,(\mu)}{l := v \mid \mu \to \texttt{unit} \mid \mu[l \mapsto v]}$$

Type system

$$\text{T-Unit}\ \frac{}{\Gamma; \Sigma \vdash \texttt{unit}\colon \texttt{Unit}} \qquad \text{T-Var}\ \frac{\Gamma(x) = T}{\Gamma; \Sigma \vdash x\colon T} \qquad \text{T-Abs}\ \frac{\Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}\colon T_2}{\Gamma; \Sigma \vdash \lambda x\colon T_1\ .\ \mathrm{e}\colon T_1 \longrightarrow T_2}$$

$$\text{T-App}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathrm{e}_1\colon T_1 \longrightarrow T_2 \\ \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_1\end{array}}{\Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon T_2} \qquad \text{T-Loc}\ \frac{\Sigma(l) = T}{\Gamma; \Sigma \vdash l\colon T} \qquad \text{T-Ref}\ \frac{\Gamma; \Sigma \vdash \mathrm{e}\colon T}{\Gamma; \Sigma \vdash \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T}$$

$$\text{T-Deref}\ \frac{\Gamma; \Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ T}{\Gamma; \Sigma \vdash !\mathrm{e}\colon T} \qquad \text{T-Asgn}\ \frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ T \\ \Gamma; \Sigma \vdash \mathrm{e}_2\colon T\end{array}}{\Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \texttt{Unit}}$$

Figure 2.1: The Simply Typed Lambda Calculus with References and Unit

represents a modified function that returns the value v for location l, and works exactly as μ for any other location.

$$\text{E-Ref}\ \frac{l \notin \mathrm{dom}\,(\mu)}{\texttt{ref}\ v \mid \mu \to l \mid \mu[l \mapsto v]} \qquad \text{E-Deref}\ \frac{\mu(l) = v}{!l \mid \mu \to v \mid \mu}$$

$$\text{E-Asgn}\ \frac{l \in \mathrm{dom}\,(\mu)}{l := v \mid \mu \to \texttt{unit} \mid \mu[l \mapsto v]}$$

The type system also needs to be extended. Just as we used a type environment Γ to map variables to types, we statically model stores with a store typing environment Σ. Σ is a partial function from locations to types. We say that a store is consistent with an environment, written $\Gamma \mid \Sigma \vDash \mu$, if for every location l in μ, the value $\mu(l)$ holds type $\Sigma(l)$

under context Γ and Σ. Since μ holds values and abstractions are values, a Γ is required to verify consistency between a store and an environment. An abstraction stored in $\mu(l)$ may hold in its body a variable x different from its parameter. To type the abstraction in $\mu(l)$ and ensure that its type is compatible with $\Sigma(l)$, we require a Γ from which to obtain the type associated to x.

Since we introduced memory locations as values, we also introduce a new type constructor for memory locations, `Ref` T, denoting a memory location holding values of type T.

$$T \quad ::= \quad \texttt{Unit} \mid T \longrightarrow T \mid \texttt{Ref}\ T \quad \text{Types}$$

Typing rules not directly related to state only carry Σ as structural information, using the same Σ for typing their premises. The type inference rules for the new language features verify consistency between the store typing environment and the types of expressions being allocated, read or assigned. These typing rules, named [T-Loc], [T-Ref], [T-Deref] and [T-Asgn], are introduced in fig. 2.1.

The language using this semantics and type system is type safe, as defined in terms of the following progress and preservation theorems:
Theorem 2 (Progress)**.** *Suppose* e *is a well-typed closed expression* ($\emptyset; \Sigma \vdash \mathrm{e} \colon T$)[3]*. Then* e *is either a value* v *or* $\mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$*.*
Theorem 3 (Preservation)**.** *Let* e *be an expression such that* $\Gamma; \Sigma \vdash \mathrm{e} \colon T$ *and* μ *a store such that* $\Gamma \mid \Sigma \vDash \mu$ *and* $\mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for some* e' *and* μ'*. Then there exists a* Σ' *such that* $\Gamma \mid \Sigma' \vDash \mu'$*,* $\Sigma \subseteq \Sigma'$ *and* $\Gamma; \Sigma' \vdash \mathrm{e}' \colon T$*.*

We do not provide a proof of these theorems for brevity. The interested reader is referred to Chapter 13 of Pierce's Book [19], where Progress and Preservation theorems, differing from our theorems only in presentation details, are introduced and proven for this language.

2.2 Gradual typing

Programming languages like ML, Java, Scala, C# and Haskell have type systems that statically ensure the inexistence of some errors in programs, preventing execution of programs until every error is corrected. The static type checking in these languages provides both documentation and consistency guarantees for programs.

However, these guarantees can also prove too restrictive for software developers. During the first stages of development, there is a lot of prototyping: types are not yet final, thus fulfilling the type restrictions becomes an unnecessary overhead. Similar cases occur when software requirements change, which is a common situation.

Several contemporary and popular programming languages do not perform any static

[3]A *closed expression* e is an expression without free variables, thus e must have a valid type derivation using the empty type environment as context.

verification, giving the programmer the flexibility to avoid defining type restrictions, but at the cost of the lack of static checks and therefore more fragile software. Static checks must be delayed to runtime to verify, for example, that the arguments provided to a function are the expected ones. These required runtime checks introduce a performance overhead. Notable examples of languages without static typing, called dynamically typed languages, are Scheme, Smalltalk, Python, Ruby and Javascript.

Gradual Typing, introduced by Siek and Taha [24], is an attempt to combine the flexibility of dynamic typing with the guarantees of static typing, allowing the programmer to annotate parts of the program with their types and to leave portions of the program unannotated at will. Unannotated programs are associated with a default unknown type (denoted ?). For example, function (λx . e) is automatically transformed into ($\lambda x\colon$? . e). Important goals of gradual typing are to provide static guarantees for fully-annotated parts of the code, to avoid runtime checks and to provide developers with the benefits of both static and dynamic typing in a single language.

2.2.1 The type consistency relation

The type equality restrictions in the inference rules of a static type system are loosened by using a consistency ($\sim$) relation instead of equality. This consistency relation permits using both an exact type and related types with some unknown information, and is defined as follows:

$$\frac{}{T \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T} \qquad \frac{T_1 \sim T_1' \qquad T_2 \sim T_2'}{T_1 \longrightarrow T_2 \sim T_1' \longrightarrow T_2'}$$

The consistency relation ($\sim$) is reflexive and symmetric, but not transitive. If consistency were a transitive relation, we could relate any two types a and b through the unknown type $((a \sim ?) \text{ and } (? \sim b) \Rightarrow a \sim b)$ and the type system would become useless because it could not discard any program, never giving static guarantees.

As an example, we present a typing rule for function application with Gradual Typing:

$$\text{T-App}\frac{\begin{array}{c}\Gamma; \Sigma \vdash \mathrm{e}_1 \colon T_1 \\ \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T_2 \\ T_1 \sim (T_2 \longrightarrow T_3)\end{array}}{\Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3} \tag{2.1}$$

This rule accepts programs where e_1 has a function type, but also programs where the type for e_1 is statically unknown. When a program is completely type annotated, the inference rule works just like in a standard static type system: Programs like (`unit unit`) are statically rejected because $\Gamma; \Sigma \vdash \mathrm{e}_1 \colon$ `Unit` and `Unit` $\not\sim$ `Unit`$\longrightarrow T_3$. Whenever static information is insufficient to prove if a program is either safe or unsafe, it must be checked at runtime. The standard procedure is to introduce type casts.

2.2.2 Casts as runtime checks

The type system presented so far accepts programs that it couldn't prove safe statically. For these programs, safety must be verified at runtime. The type system accepted these programs because it made optimistic assumptions that would make the program safe. These assumptions are explicited using type casts. A cast delays type verification until runtime. If runtime verification fails, the program must throw an error.

For example, consider the program $(\lambda f : ?\ .\ f\ \texttt{unit})\texttt{unit}$. This program is statically accepted, because statically the parameter f could be a function. At runtime, however, the program must clearly fail because the argument `unit` is not a function. In gradual typing, a translation algorithm must convert programs into an intermediate language, inserting on the way enough runtime casts to ensure safety, making the example program fail at runtime.

A cast is denoted $\langle T_2 \Leftarrow T_1\rangle \mathrm{e}$, meaning that even though e statically has type T_1, we can optimistically assume that e also has type T_2. The execution semantics of casts, presented in section 2.2.3, enforces that the assumption always holds: If e does not have type T_2 at runtime, the semantics triggers an error.

Cast insertion is directed by a translation relation ($\Rightarrow$). This relation uses type information to identify when to introduce a cast. To avoid introducing redundant casts, a cast insertion predicate may be used to only introduce a cast when there is a chance that the cast may fail:

$$\langle\!\langle T_2 \Leftarrow T_1 \rangle\!\rangle \mathrm{e} = \begin{cases} \mathrm{e} & \text{if } T_1 = T_2 \\ \langle T_2 \Leftarrow T_1\rangle \mathrm{e} & \text{otherwise} \end{cases}$$

The translation relation is defined in terms of a set of cast insertion rules. A cast insertion rule for the translation relation is introduced for each assumption made by the typing rules in the gradual type system. We now present the rules for function application, and list the full translation relation in fig. 2.2. The gradual type inference rule [T-App] introduced in section 2.2.1, combines two distinct assumptions: First, it assumes that e_1 has a function type. To make this assumption explicit when e_1 has the unknown type we introduce rule [C-App-1]. Second, when e_1 has a function type, we assume that e_2 has a type compatible with the argument type of e_1. This assumption is made explicit by rule [C-App-2].

$$\text{C-App-1}\ \frac{\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' : ? \qquad \Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' : T_2}{\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow (\langle T_2 \longrightarrow ? \Leftarrow ?\rangle \mathrm{e}_1')\ \mathrm{e}_2' : ?}$$

$$\text{C-App-2}\ \frac{\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' : T_1 \longrightarrow T_3 \qquad \Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' : T_2 \qquad T_1 \sim T_2}{\Gamma;\Sigma \vdash \mathrm{e}_1'\ \langle\!\langle T_1 \Leftarrow T_2 \rangle\!\rangle \mathrm{e}_2' : T_3}$$

2.2.3 Operational semantics for casts

We now present a simplified operational semantics for casts. Though it is not the same as defined in the literature, this semantics suffices to present the concepts behind gradual typing. This simplified semantics will also work as a basis for our design of gradual typing for type-and-effect systems as introduced in chapter 3.

A cast $\langle T_2 \Leftarrow T_1 \rangle$e expresses the static assumption that an expression of type T_1 may also have T_2. This assumption will certainly hold if both types are the same. If they are not the same, the assumption can only hold if also the type consistency relation ($T_1 \sim T_2$) holds. Cast assumptions hold with respect to the values permitted in certain positions of a program. Thus a first step for evaluation consists in reducing the expression being casted to a value. This goal is achieved by introducing a new evaluation frame for casts, $\langle T \Leftarrow T \rangle \square$, which allows rule [E-Frame] to reduce a casted expression to a value.

A value might be wrapped with more than one cast. For example, consider the abstraction $(\lambda f \colon ? \ . \ f \ 1)$. By applying f to 1, the abstraction makes the assumption that f is also a function. The cast insertion algorithm makes the assumption explicit by translating the abstraction to $(\lambda f \colon ? \ . \ (\langle \texttt{Int} \longrightarrow ? \Leftarrow ? \rangle f) \ 1)$. If this abstraction is applied to a value like 4, which holds type `Int`, the argument is also wrapped with a cast$\langle ? \Leftarrow \texttt{Int} \rangle$, to hide the information of its type. The translation relation will always insert a cast when a properly typed value is expected to have type ?. This information must be kept after substitution, so that the program

$$(\lambda f \colon ? \ . \ (\langle \texttt{Int} \longrightarrow ? \Leftarrow ? \rangle f) \ 1) \ \langle ? \Leftarrow \texttt{Int} \rangle 4 \tag{2.2}$$

may afterwards proceed to check that the casts are inconsistent. Thus, we also consider casted values as values in the language.

$$v \quad ::= \quad \texttt{unit} \mid \lambda x \colon T \ . \ \text{e} \mid \langle T \Leftarrow T \rangle v \quad \text{Values}$$

Type inconsistencies might arise when composing casts since type consistency is not transitive, thus individually valid assumptions might hide a global assumption that is not valid. An example of this situation arises in program 2.2. After substitution, the program becomes $(\langle \texttt{Int} \longrightarrow ? \Leftarrow ? \rangle \langle ? \Leftarrow \texttt{Int} \rangle 4) \ 1$. We need to evaluate the pair of casts to see that 4 cannot be used as a function.

We can merge nested casts that go through a shared type. For this goal we introduce rule [E-Cast-Merge].

$$\text{E-Cast-Merge}\frac{}{\langle T_3 \Leftarrow T_2 \rangle \langle T_2 \Leftarrow T_1 \rangle v \rightarrow \langle T_3 \Leftarrow T_1 \rangle v}$$

After reducing our program with rule [E-Cast-Merge], we reach a $(\langle \texttt{Int} \longrightarrow ? \Leftarrow \texttt{Int} \rangle 4) \ 1$. This cast would only be valid if type `Int` is consistent with $\texttt{Int} \longrightarrow ?$, a false assumption. To signal an error in these situations, we introduce rule [E-Cast-Err].

$$\text{E-Cast-Err}\frac{T_1 \not\sim T_2}{\langle T_2 \Leftarrow T_1 \rangle v \rightarrow \mathsf{Error}}$$

This error must also be propagated out of the frame, so the full program evaluates to an error. For this goal we introduce rule [E-Error].

$$\text{E-Error}\;\frac{}{f[\mathsf{Error}] \to \mathsf{Error}}$$

These rules are sufficient for evaluating our example, but are not sufficient to remove every cast. Reducing casts by [E-Cast-Merge] leads to identity casts ($\langle T \Leftarrow T \rangle$). An identity cast holds a tautological assumption, so it can always be safely removed. Rule [E-Cast-Id] removes identity casts.

$$\text{E-Cast-Id}\;\frac{}{\langle T \Leftarrow T \rangle v \to v}$$

To be completely comprehensive, the semantics also has to reduce casts for function types, also called higher order casts. A higher order cast for a function can be broken into a cast for the argument and a cast for the result. Higher order casts can be reduced with rule [E-Cast-Higher-Order]. This rule transforms the casted value, which we know is a function, into a new function that is properly applied with the required casts.

$$\text{E-Cast-Higher-Order}\;\frac{x \notin \mathit{freevars}(v)}{\langle T'_1 \longrightarrow T'_2 \Leftarrow T_1 \longrightarrow T_2 \rangle v \to \lambda x \colon T'_1 \;.\; \langle T'_2 \Leftarrow T_2 \rangle (v\ \langle T_1 \Leftarrow T'_1 \rangle x)}$$

Unlike previous rules, our definition of [E-Cast-Higher-Order] does not evaluate the cast immediately. Instead, rule [E-Cast-Higher-Order] delays cast evaluation until application of function v. This strategy makes the language more flexible: if there is an invalid sequence of casts (remember that v is a value, so it may contain more higher order casts), it becomes part of the body of the new function. If the function v is never applied, then the invalid casts inside the new function will never be evaluated and the program will execute correctly, like in a dynamically typed language. If we eagerly evaluated the higher order casts like in the rest of the cast evaluation rules we introduced, a program with an invalid higher order cast would always fail, even when the function is never applied. With these rules, we can reduce all casts in the language. The full semantics is presented in fig. 2.2.

Our simple semantics does not identify which of the casts in a program failed. Though this is sufficient for our theoretical purposes, a reference to the failing cast acts as good feedback for the programmer. Formally, providing information of the failing cast is called *blame tracking* [27, 1, 10]. Another problem in our semantics is that composed casts may incur higher space costs, particularly when casts are inserted in recursive calls of a program. There has been a lot of recent work in avoiding space leaks [13] while managing to identify the failing cast. Identifying the failing cast is not trivial and different strategies and results have been generated ([27], [25], [22]). We do not solve questions about blame tracking in this thesis, but is one of the goals for future work.

$$
\begin{array}{rcll}
T & ::= & \texttt{Unit} \mid ? \mid T \longrightarrow T & \text{Types} \\
v & ::= & \texttt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} & \text{Values} \\
\mathrm{e} & ::= & v \mid \mathrm{e}_1\ \mathrm{e}_2 \mid \langle T \Leftarrow T \rangle \mathrm{e} & \text{Expressions} \\
f & ::= & \square\ \mathrm{e} \mid v\ \square \mid \langle T \Leftarrow T \rangle \square & \text{Evaluation Frames}
\end{array}
$$

$$
\text{C-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x \Rightarrow x\colon T}
\qquad\qquad
\text{C-Unit}\ \frac{}{\Gamma \vdash \texttt{unit} \Rightarrow \texttt{unit}\colon \texttt{Unit}}
$$

$$
\text{C-Abs}\ \frac{\Gamma, x\colon T_1 \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T_2}{\Gamma \vdash \lambda x\colon T_1\ .\ \mathrm{e} \Rightarrow \lambda x\colon T_1\ .\ \mathrm{e}'\colon T_1 \longrightarrow T_2}
$$

$$
\text{C-App-1}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1'\colon ? \\ \Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2'\colon T_2\end{array}}{\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow (\langle T_2 \longrightarrow ? \Leftarrow ? \rangle \mathrm{e}_1')\ \mathrm{e}_2'\colon ?}
\qquad
\text{C-App-2}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1'\colon T_1 \longrightarrow T_3 \\ \Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2'\colon T_2 \\ T_1 \sim T_2\end{array}}{\Gamma;\Sigma \vdash \mathrm{e}_1'\ \langle\!\langle T_1 \Leftarrow T_2 \rangle\!\rangle \mathrm{e}_2'\colon T_3}
$$

$$
\text{E-Frame}\ \frac{\mathrm{e} \to \mathrm{e}'}{f[\mathrm{e}] \to f[\mathrm{e}']}
\qquad
\text{E-Error}\ \frac{}{f[\mathsf{Error}] \to \mathsf{Error}}
\qquad
\text{E-App}\ \frac{}{(\lambda x\colon T\ .\ \mathrm{e})\ v \to [{}^{v}\!/\!{}_{x}]\,\mathrm{e}}
$$

$$
\text{E-Cast-Err}\ \frac{T_1 \not\sim T_2}{\langle T_2 \Leftarrow T_1 \rangle v \to \mathsf{Error}}
\qquad\qquad
\text{E-Cast-Id}\ \frac{}{\langle T \Leftarrow T \rangle v \to v}
$$

$$
\text{E-Cast-Merge}\ \frac{}{\langle T_3 \Leftarrow T_2 \rangle \langle T_2 \Leftarrow T_1 \rangle v \to \langle T_3 \Leftarrow T_1 \rangle v}
$$

$$
\text{E-Cast-Higher-Order}\ \frac{x \notin \mathit{freevars}(v)}{\langle T_1' \longrightarrow T_2' \Leftarrow T_1 \longrightarrow T_2 \rangle v \to \lambda x\colon T_1'\ .\ \langle T_2' \Leftarrow T_2 \rangle (v\ \langle T_1 \Leftarrow T_1' \rangle x)}
$$

Figure 2.2: A Gradual Simply-Typed Lambda Calculus.

2.2.4 Subtyping and gradual typing

What is subtyping

Subtyping [4, 20] extends our notion of types for situations where more than one type satisfies the restrictions we require for a program. To express which types can be used whenever a particular type is expected, we introduce the subtyping relation ($<: \subseteq T \times T$). If a type T_1 is a subtype of type T_2, $T_1 <: T_2$, then whenever T_2 is expected, type T_1 may also be used. This property of the subtyping relation is called substitutability. Subtyping may be introduced in a type system through a new inference rule that expresses the substitutability property, called the subsumption rule:

$$\text{T-Subsumption}\;\frac{\Gamma;\Sigma \vdash \mathrm{e}_1 : T_1 \qquad T_1 <: T_2}{\Gamma;\Sigma \vdash \mathrm{e}_1 : T_2}$$

Subtyping is a reflexive relation, meaning that every type is a subtype of itself. This means that rule [T-Subsumption] can be used anywhere and any amount of times in a derivation, making the type system non deterministic. The standard solution for this problem is to avoid introducing a [T-Subsumption] rule, but instead making explicit use of subtyping in each rule that requires the flexibility of subtyping.

Let's consider as an example a language with different kinds of numbers. For example, integers and floating point numbers (typed `Integer` and `Float`, respectively). We can also define a type `Number` that represents situations when any kind of number may be used.

$$\frac{}{\texttt{Integer} <: \texttt{Number}} \qquad \frac{}{\texttt{Float} <: \texttt{Number}} \qquad \frac{}{T <: T}$$

If then we define a function argument as a Number, we want to use either a Float or an Integer indistinguishably. We may introduce addition as a primitive operation on numbers, using the following inference rule:

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash \mathrm{e}_1 : \texttt{Number}\\ \Gamma;\Sigma \vdash \mathrm{e}_2 : \texttt{Number}\end{array}}{\Gamma;\Sigma \vdash \mathrm{e}_1 \;+\; \mathrm{e}_2 : \texttt{Number}}$$

If we do not have a subsumption rule, addition will need e_1 and e_2 to explicitly have type `Number`. If they only have type `Integer` or `Float`, they won't be accepted. As we already mentioned, the solution is to make explicit use of subtyping in the rule for addition:

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash \mathrm{e}_1 : T_1\\ \Gamma;\Sigma \vdash \mathrm{e}_2 : T_2\\ T_1 <: \texttt{Number} \qquad T_2 <: \texttt{Number}\end{array}}{\Gamma;\Sigma \vdash \mathrm{e}_1 \;+\; \mathrm{e}_2 : \texttt{Number}}$$

Interesting uses for subtyping arise also for function application. To allow functions to receive arguments with more specific types than the type declared for their parameter, we need to modify the inference rule for application:

$$\text{T-App}\frac{\begin{array}{c}\Gamma ; \Sigma \vdash \mathrm{e}_1 : T_1 \longrightarrow T_3 \\ \Gamma ; \Sigma \vdash \mathrm{e}_2 : T_2 \qquad T_2 <: T_1\end{array}}{\Gamma ; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 : T_3}$$

Since arguments may hold any valid type, we also need to define subtyping for function types, $T_1 \longrightarrow T_2 <: T'_1 \longrightarrow T'_2$. A naive approach to define function subtyping would be to directly restrict subtyping both for the arguments and results. Though we need to force that the result can always be subsumed ($T_2 <: T'_2$), we also need to ensure that the argument given to the new function is always usable by the original function ($T'_1 <: T_1$), not the other way around. The subtyping restriction for the results is called *covariant*, while the restriction for the arguments is called *contravariant*. With these restrictions, subtyping for function types is defined as:

$$\frac{T'_1 <: T_1 \qquad T_2 <: T'_2}{T_1 \longrightarrow T_2 <: T'_1 \longrightarrow T'_2}$$

This requirement for function arguments also arises intuitively from the cast evaluation rule for higher-order types, by looking at how the higher order casts are broken into two separated casts in the same way as subtyping is defined:

$$\text{Higher-Order}\frac{x \notin \mathit{freevars}(v)}{\langle T'_1 \longrightarrow T'_2 \Leftarrow T_1 \longrightarrow T_2 \rangle v \rightarrow \lambda x : T'_1\ .\ \langle T'_2 \Leftarrow T_2 \rangle (v\ \langle T_1 \Leftarrow T'_1 \rangle x)}$$

Though the languages we have presented so far do not require any notion of subtyping, we will use subtyping for effect systems. As our final goal is to reconcile gradual typing with effect systems, we need also to introduce the relation between Subtyping and Gradual Typing.

Combining subtyping with gradual typing

Gradual Typing, introduced in [24], is based on the consistency relation ($\sim$).The consistency relation was introduced in opposition to contemporary work in which uncertainty was represented using subtyping [4]. In their original paper [24], Siek and Taha left open the issue of how consistency relates to subtyping. This relation was later addressed when Gradual Typing for Objects [23] was introduced.

Consistency and Subtyping can be combined in a relation called "consistent subtyping" ($\lesssim$). Siek and Taha propose that type a is a consistent subtype of b if there exists a type τ

[4]In "Quasi-Static Typing", the contemporary work mentioned, the dynamic type was represented as a top type. A top type in subtyping is a type with whom every other type is related. (For a top type T, $T' <: T$ holds for every other type T'). Siek and Taha found issues in this approach, which are described in [24].

that is consistent with either a or b and relates by subtyping with the other type. Then an unknown type can be used in any place where a subtype was expected, and any type can be used as a subtype when a type $?$ was expected.

$$a \lesssim b \iff \exists \tau \sim a \;.\; \tau <: b \text{ or } \exists \tau \sim b \;.\; a <: \tau$$

The idea behind this specification is that types a and b can be related by the subtyping relation either directly or *through* consistent types. In their paper, Siek and Taha also prove that using consistency either with a or with b is equivalent. The definition of Consistent Subtyping introduced by Siek and Taha overcomes issues that arise with a naive definition. Suppose we defined consistent subtyping by simply loosening the subtyping relation to use consistent types as follows:

$$a \lesssim b \iff \exists \alpha, \beta \;.\; a \sim \alpha <: \beta \sim b$$

Since subtyping is reflexive, such definition would allow any two types to be related by $\lesssim$, because $a \sim ? <: ? \sim b$. We then need a more restrictive definition for $\lesssim$, like the definition introduced by Siek and Taha.

2.3 Type-and-Effect systems

In our discussion so far, types only represented information about results of programs. The term *effect system* [11] refers to programming languages with a static type system that also provides guarantees about which side-effects are generated during the evaluation of those programs.

A *value* is a candidate result of evaluation. Values do not produce side effects themselves. The languages we have introduced in this work include λ *abstractions*, values that encapsulate pending computation. This pending computation is evaluated when the abstraction is applied to a particular argument.

During application, the instructions encapsulated in a lambda abstraction might generate side effects. These side effects are analyzed by a type-and-effect system, which then requires to also contain effect information. In type-and-effect systems, function types carry an effect annotation, limiting the side effects that may be generated when the function is applied. This annotation is represented as a set written on top of the function type arrow. Thus an abstraction of type $T_1 \xrightarrow{\Phi} T_2$ takes an argument of type T_1 and produces a result of type T_2, while producing *at most* the side effects declared in the set Φ. By carrying this extra information, the type system can produce a conservative approximation of the side effects that may occur in programs.

With this definition of function types, a different type is assigned to functions that only differ in the side effects generated by applying them. This hides the intuition that the annotations are a restriction but not a guarantee, meaning that whenever a function is allowed to generate a set of side effects, it is also allowed to generate fewer side effects. The subtyping relation between function types introduced in section 2.2.4 can be extended to account for effect subsumption [26].

$$\frac{T_1' <: T_1 \qquad T_2 <: T_2' \qquad \Phi \subseteq \Phi'}{T_1 \xrightarrow{\Phi} T_2 <: T_1' \xrightarrow{\Phi'} T_2'}$$

The effect information in a type derivation can either be inferred bottom-up [21] or can be used top-down to enforce an effect discipline [15]. Both designs are valid options, but we follow the latter approach in this work to later build on top of the work on generic type-and-effect systems presented in section 2.5.1.

Section 2.4 introduces a simple language with a type-and-effect system, called the fluent language[5]. This language has the features presented in the literature that introduced effect systems for the first time [11], presented in a clearer and more contemporary form. In later chapters we extend the fluent language to explore the design of gradual effect checking in a simple context.

2.4 An example: the fluent language

Effect systems were introduced by Gifford and Lucassen in 1986 [11]. They focused on the relation between programs and memory (or state), categorizing programs into "effect classes". Each class is built grouping sets of side effects into equivalence classes that vary according to the goal of the system. An expression's effect class determines the subset of the language that the expression must be restricted to. Therefore, effect classes provide a way to declare and separate both functional and imperative styles of programming on the same program.

A language with these features is called fluent by Gifford and Lucassen. A fluent language allows the programmer to declare a piece of code to be purely functional or to be imperative at different degrees, through the usage of effect ascription, a language feature we write as e :: Φ[6], where e is a program and Φ is a restriction for the side effects that may be generated by evaluating e. The process of verifying that the effect restrictions declared by the programmer are actually fulfilled by the program is named "effect checking".

We introduce a simplified version of the language: change some keywords, remove type polymorphism and isolate state through the usage of references. Having a language with few features, clearer keywords and with a clear distinction between identifiers (for function parameters) and memory locations (whose value may change) produces a simpler declaration

[5]The name fluent is inspired by Gifford and Lucassen, which use the name "fluent languages" for languages with a type-and-effect system in [11].

[6]We have renamed in our analysis the original `the` keyword used by Gifford and Lucassen to ::, to improve readability.

of the effect checking process in the type system. We present the type system in contemporary notation, making the type inference rules more readable.

2.4.1 Classifying programs over their use of state

Gifford and Lucassen propose a particular categorization of programs into a set of effect classes, according to their goal: to identify opportunities for concurrent execution and memoization. Other categorizations may also be constructed.

Gifford and Lucassen separate programs into four categories: those that do not use any state (**Pure**), those that only allocate memory (**Function**), those that also read memory (**Observer**) and those that perform assignments (**Procedure**). We can also describe these categories through the sets of state-related privileges that entitle an expression to produce a side effect. We name privileges alloc for allocation, read and write. Effect class **Pure** maps to the empty set of privileges, $\emptyset$. **Function** to the singleton $\{\mathsf{alloc}\}$, **Observer** to sets $\{\mathsf{alloc}, \mathsf{read}\}$ and $\{\mathsf{read}\}$, and **Procedure** to all sets that contain write, like the universe set $\{\mathsf{alloc}, \mathsf{read}, \mathsf{write}\}$.

A model based in sets of privileges has also the benefit of giving the programmer more granularity than Gifford and Lucassen's effect classes, allowing to differentiate privilege sets that were considered "equivalent" by Gifford and Lucassen, like $\{\mathsf{read}, \mathsf{write}\}$ and $\{\mathsf{write}\}$.

To build a simple language, we start from the simply-typed lambda calculus extended with `unit` and references, as introduced in section 2.1.5. The full syntax of the fluent language is defined as follows:

$$l \in \mathbf{Labels}$$
$$\mathbf{Privileges} = \{\texttt{alloc}, \texttt{read}, \texttt{write}\}$$
$$\Phi \in \mathcal{P}\left(\mathbf{Privileges}\right)$$

$$\begin{array}{lcll}
v & ::= & \texttt{unit} \mid l \mid \lambda x\colon T\ .\ \mathrm{e} & \text{Values} \\
\mathrm{e} & ::= & v \mid \mathrm{e}\ \mathrm{e} \mid \texttt{ref}\ \mathrm{e} \mid !\mathrm{e} \mid \mathrm{e} := \mathrm{e} \mid \mathrm{e} :: \Phi & \text{Expressions} \\
T & ::= & \texttt{Unit} \mid T \xrightarrow{\Phi} T & \text{Types}
\end{array}$$

2.4.2 Generated side-effects or privileges required

Typing rules for fluent languages can handle effect information in two different ways, top-down or bottom-up. We call a system bottom-up when effect information is conceived as a result from type inference, and call top-down when effect information is conceived as contextual information to be enforced.Choosing a style depends on the meaning the designer wants to focus on.

Side effects can be considered as another result of evaluation. This notion implies that the static analysis of side effects should *infer* not only the type of the result, but also the side effects that their evaluation will produce. An effectful expression therefore will introduce new side-effects into the resulting side effect set.

Another approach is to consider side effects as privileges. In this case, an expression can only trigger a side effect if it is allowed to do it. In this approach, the type system would type an effectful expression only if it can *check* that every side-effect related privilege is available in the program context.

As a concrete example, we now present typing rule candidates for expressions of the form `ref` e, which is a notation to allocate memory. Allocation can be considered as a side effect of these expressions. As mentioned in the previous section, we will notate this side effect as `alloc`.

In an inference system, we can define a typing rule like

$$\text{T-Ref}\frac{\Gamma; \Sigma \vdash \mathrm{e} \colon T!\Phi}{\Gamma; \Sigma \vdash \texttt{ref}\ \mathrm{e} \colon \texttt{Ref}\ T!\Phi \cup \{\texttt{alloc}\}}$$

where $\mathrm{e} \colon T!\Phi$ means that e has type T and generates the side effects in Φ.

Following this approach, typing produces a pair of results: the type `Ref` T for the expression `ref` e, and also an effect set Φ' which will always contain `alloc` and the side-effects produced by e.

In a top-down system, the set of effects (or privileges) is given as part of the context, and a rule for references shall only verify if the privilege of allocation is available:

$$\text{T-Ref}\frac{\begin{array}{c}\Phi; \Gamma; \Sigma \vdash \mathrm{e} \colon T \\ \texttt{alloc} \in \Phi\end{array}}{\Phi; \Gamma; \Sigma \vdash \texttt{ref}\ \mathrm{e} \colon \texttt{Ref}\ T} \tag{2.3}$$

The first strategy (inference) processes effect information bottom-up, while the second strategy (privilege enforcement) processes effect information top-down.

Throughout this work, we follow the top-down approach where effect sets are considered as privilege sets, and use both names as synonyms. We follow this approach to be able to reuse the generic framework presented in section 2.5.1 that follows this strategy. We provide a proof that the fluent language can be expressed in that generic system in appendix A. A full type system for the fluent language is presented in fig. 2.3, and follows this approach.

In the systems we present, typing might not necessarily produce the most general type for an expression, because of the definition for rule [T-Fn]. The premise of the rule has a free privilege set Φ_1, which implies that any superset of the minimal privilege set required by e would also produce a correct type for the expression. This only means that the type system could also infer a supertype of the principal type of an abstraction. A subyping relation for the fluent language is presented in fig. 2.4. It formalizes the intuitive notion that a function that requires a set of privileges can be used in a context where more privileges are available. The subtyping relation is used in the rule for typing applications [T-App] and in

$$\text{T-Fn}\frac{\Phi_1; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}\colon T_2}{\Phi; \Gamma; \Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})\colon T_1 \xrightarrow{\Phi_1} T_2} \qquad \text{T-Unit}\frac{}{\Phi; \Gamma; \Sigma; \vdash \texttt{unit}\colon \texttt{Unit}}$$

$$\text{T-Loc}\frac{\Sigma(l) = T}{\Phi; \Gamma; \Sigma \vdash l\colon \texttt{Ref}\ T} \qquad \text{T-Var}\frac{\Gamma(x) = T}{\Phi; \Gamma; \Sigma \vdash x\colon T} \qquad \text{T-App}\frac{\begin{array}{c}\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1\colon T_1 \xrightarrow{\Phi_1} T_3 \\ \Phi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2 \\ T_2 <: T_1 \qquad \Phi_1 \subseteq \Phi\end{array}}{\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon \tau_3}$$

$$\text{T-Ref}\frac{\begin{array}{c}\Phi; \Gamma; \Sigma \vdash \mathrm{e}\colon T \\ \{\texttt{alloc}\} \subseteq \Phi\end{array}}{\Phi; \Gamma; \Sigma \vdash \texttt{ref}\ \ \mathrm{e}\colon \texttt{Ref}\ T} \qquad \text{T-Deref}\frac{\begin{array}{c}\Phi; \Gamma; \Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ T \\ \{\texttt{read}\} \subseteq \Phi\end{array}}{\Phi; \Gamma; \Sigma \vdash !\mathrm{e}\colon T}$$

$$\text{T-Assign}\frac{\begin{array}{c}\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ T_1 \\ \Phi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2 \\ \{\texttt{write}\} \subseteq \Phi \qquad T_2 <: T_1\end{array}}{\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \texttt{Unit}} \qquad \text{T-Ascription}\frac{\begin{array}{c}\Phi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T \\ \Phi_1 \subseteq \Phi\end{array}}{\Phi; \Gamma; \Sigma; \vdash \mathrm{e} :: \Phi_1\colon T}$$

Figure 2.3: A type system for the fluent language that follows the top-down approach.

$$\frac{}{T <: T} \qquad \frac{\begin{array}{c}T_1' <: T_1 \\ T_2 <: T_2' \\ \Phi \subseteq \Phi'\end{array}}{T_1 \xrightarrow{\Phi} T_2 <: T_1' \xrightarrow{\Phi'} T_2'}$$

Figure 2.4: Subtyping relation for fluent languages

rule [T-Assign].

2.4.3 Restricting side effects with ascription

To take advantage of their classification of programs into effect classes, Gifford and Lucassen propose a language construct that allows the programmer to restrict the effect class (or set of privileges) that a particular expression in the language may have. The restriction acts as a contract declared by the programmer regarding the usage of state: it limits the effects an expression should produce, rejecting the program otherwise, and also ensures an upper bound of the side effects for consumers of an ascribed program.

We use $\mathrm{e} :: \Phi$ as notation for effect ascription. An expression $(\mathrm{e} :: \Phi)$ verifies that the *required* privilege set for e is a subset of Φ, the *declared* set of privileges. This forces the typechecker to discard programs that are not allowed to use certain features of state. As an example, the program

$$(\texttt{ref unit}) :: \emptyset$$

fails to typecheck because (`ref unit`) requires an `alloc` privilege, which is not available in

$\emptyset$.

An effect ascription also declares itself to require a privilege set Φ, therefore the program $(\texttt{unit} :: \{\texttt{read}\}) :: \emptyset$ also fails, because the contracted expression $(\texttt{unit} :: \{\texttt{read}\})$ is declared to require privileges not available in the $\emptyset$ context.

2.4.4 Typing function application

Even though Gifford and Lucassen propose typing and effect checking as separated processes, they are related as types contain information required by static effect checking for function applications, allowing to accept or discard a particular program. Suppose we could typecheck the following abstraction:

$$(\lambda f\colon \texttt{Unit} \longrightarrow \texttt{Unit}\ .\ (f\ \texttt{unit}) :: \emptyset) \tag{2.4}$$

Statically, the type system cannot identify if this program is correct or not without having some notion of the privileges required by the body of f. It should fail whenever f requires any particular privilege, but not if f is `pure` (requires no privilege). An easy way to distinguish these situations is to extend the notion of function type to include effect information, resulting in the following program:

$$(\lambda f\colon \texttt{Unit} \xrightarrow{\emptyset} \texttt{Unit}\ .\ (f\ \texttt{unit}) :: \emptyset) \tag{2.5}$$

Addition of this information permits the type system to check the integrity of effect information for function applications. If we apply this function with the typed identity function as an argument $(\lambda x\colon \texttt{Unit}\ .\ x)$, it type checks. If the argument is instead $(\lambda x\colon \texttt{Unit}\ .\ (!(\texttt{ref}\ x)))$, which is an identity that uses state, the program fails to typecheck, because the function requires more permissions than those allowed.

2.5 Generic type-and-effect systems

In this section we describe the related work on Generic Type-and-Effect Systems. Marino and Millstein [15] introduce a framework to define type-and-effect systems, which we denote M&M. The framework provides type safety guarantees for the systems it derives, requiring only to prove that some monotonicity restrictions hold. Besides introducing the framework, we show in appendix A that the fluent language introduced in section 2.4 can be considered as an instance of this generic framework.

2.5.1 A generic type-and-effect system

This description of generic type-and-effect systems has been edited from section 2.4 of our paper accepted at the International Conference on Functional Programming 2014[2].

To avoid re-inventing gradual effects for each possible effect discipline, we reuse the generic effect framework introduced by Marino and Millstein (M&M) [15]. The M&M effect framework defines a parametrized typing judgment $\Phi; \Gamma; \Sigma \vdash \mathrm{e} : T$. It checks an expression under a set of privileges Φ, representing the effects that are allowed during the evaluation of the expression e. For instance, here is the generic typing rule for functions:

$$\text{T-Abs}\ \frac{\Phi_1; \Gamma, x : T_1; \Sigma \vdash \mathrm{e} \colon T_2}{\Phi; \Gamma; \Sigma \vdash (\lambda x : T_1\ .\ \mathrm{e})_\varepsilon : \{\varepsilon\} \left(T_1 \xrightarrow{\Phi_1} T_2\right)}$$

Since a function needs no specific permissions, any privilege set Φ will do. The function body itself may require privileges Φ_1 and these are used to annotate the function type. We explain the tag ε shortly.

A given privilege discipline (mutable state, exceptions, etc.) is instantiated by defining two operations, a *check* predicate and an *adjust* function. The check predicate is used to determine whether the current privileges are sufficient to evaluate non-value expression forms. To achieve genericity, the check predicate $\textbf{check}_C$ is indexed by *check contexts* C, which represent the non-value expression forms. The adjust function is used to evolve the available privileges while evaluating the subexpressions of a given expression form. This function takes the current privileges and returns the privileges used to check the considered subexpression. To achieve genericity, the adjust function $\textbf{adjust}_A$ is indexed by *adjust contexts* A, which represent the immediate context around a given subexpression.

Marino and Millstein [15] point out that the definition of check contexts maps to the structure of the reducible expressions in the language (redexes): reducing expressions may produce side effects, and check contexts are used to enforce that this reduction only occurs when the privileges required to produce the associated side effects are available. On the other hand, the definition of adjust contexts maps to the structure of evaluation frames (introduced in section 2.1.1): adjust contexts are used to modify the set of privileges available for evaluating subexpressions, and the process of evaluating subexpressions is directed by evaluation frames.

To increase its overall expressiveness, the framework also incorporates a notion of *tags* ε[7], which represent auxiliary static information for an effect discipline (e.g. abstract locations). Expressions that create new values, like constants and lambdas, are indexed with tags (for example, $\mathtt{unit}_\varepsilon$). The check and adjust contexts contain *tag sets* π so that $\textbf{check}_C$ and $\textbf{adjust}_A$ can leverage static information about the values of subexpressions. To facilitate abstract value-tracking, type constructors are annotated with tagsets, so types take the form $T \equiv \pi\rho$, with ρ a *PreType* [8]. For more precise control, effect disciplines can associate tags to privileges e.g., $\mathtt{read}(\varepsilon_1)$, $\mathtt{read}(\varepsilon_2)$, etc.

[7] The key concerns for developing gradual effects are captured in the simpler tagless framework, which we use in chapter 3.

[8] A pretype represents the part of a type that is not a tagset. In our framework, a pretype ρ may be $\mathtt{Unit}$, $T \xrightarrow{\Phi} T$ (function pretype) or $\mathtt{Ref}\ T$(reference pretype).

For example, a check predicate for controlling mutable state is defined as follows:

$$\begin{aligned}
\mathbf{check}_{!\pi}(\Phi) &\iff \texttt{read} \in \Phi \\
\mathbf{check}_{\texttt{ref}\pi}(\Phi) &\iff \texttt{alloc} \in \Phi \\
\mathbf{check}_{\pi_1 := \pi_2}(\Phi) &\iff \texttt{write} \in \Phi \\
\mathbf{check}_{C}(\Phi) &\text{ holds for all other } C
\end{aligned}$$

In this case, only state-manipulating expression forms have interesting check predicates, which simply require the corresponding privilege; the rest always hold.

Since the assignment expression involves evaluating two subexpressions (the reference and the new value), there are two adjust contexts. The $\downarrow := \uparrow$ context, which corresponds to evaluating the reference to be assigned, and the $\pi := \downarrow$ context, which corresponds to evaluating the assigned value. The $\downarrow$ denotes the subexpression for which privileges should be adjusted. The tagset π represents statically known information about any subexpressions that would be evaluated before the current expression. The $\uparrow$ denotes a subexpression that would be evaluated after the current expression.

For certain disciplines, like mutable state, the adjust function is simply the identity for every context. But one could, for example, require that all subexpressions assigned to references must be effect-free by defining adjust as follows:

$$\begin{aligned}
\mathbf{adjust}_{\pi := \downarrow}(\Phi) &= \emptyset \\
\mathbf{adjust}_{A}(\Phi) &= \Phi \text{ otherwise}
\end{aligned}$$

All typing rules in the generic system use check and adjust to enforce the intended effect discipline. For instance, here is the typing rule for assignment:

$$\text{T-Asgn}\ \frac{\begin{array}{c}\mathbf{adjust}_{\downarrow := \uparrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 : \pi_1\texttt{Ref}\ T_1 \\ \mathbf{adjust}_{\pi_1 := \downarrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 : \pi_2\rho_2 \\ \mathbf{check}_{\pi_1 := \pi_2}(\Phi) \qquad \pi_2\rho_2 <: T_1\end{array}}{\Phi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_{\varepsilon} : \{\varepsilon\}\texttt{Unit}}$$

The subexpressions e_1 and e_2 are typed using adjusted privilege sets. Their corresponding types have associated tagsets π_i that are used to adjust and check privileges. Note that in accord with left-to-right evaluation, $\mathbf{adjust}_{\pi_1 := \downarrow}$ knows which tags are associated with typing e_1. Finally, $\mathbf{check}_{\pi_1 := \pi_2}$ verifies that assignment is allowed with the given permissions and the subexpression tag sets. Subtyping is used here only to account for inclusion of privilege sets between function types.

For maximum flexibility, the framework imposes only two constraints on the definitions of **check** and **adjust**:
Property 1 (Privilege Monotonicity)**.**

- *If* $\Phi_1 \subseteq \Phi_2$ *then* $\mathbf{check}_C(\Phi_1) \Longrightarrow \mathbf{check}_C(\Phi_2)$*;*
- *If* $\Phi_1 \subseteq \Phi_2$ *then* $\mathbf{adjust}_A(\Phi_1) \subseteq \mathbf{adjust}_A(\Phi_2)$*.*

$$\text{T-Fn}\;\frac{\Phi_1; \Gamma, x\colon \tau_1; \Sigma \vdash \mathrm{e}\colon \tau_2}{\Phi; \Gamma; \Sigma \vdash (\lambda x.\mathrm{e})_\varepsilon\colon \{\varepsilon\}(\tau_1 \xrightarrow{\Phi_1} \tau_2)} \qquad \text{T-Unit}\;\frac{}{\Phi; \Gamma; \Sigma \vdash \mathtt{unit}_\varepsilon\colon \{\varepsilon\}\mathtt{Unit}}$$

$$\text{T-Loc}\;\frac{\Sigma(l) = \tau}{\Phi; \Gamma; \Sigma \vdash l_\varepsilon\colon \{\varepsilon\}\mathrm{Ref}\ \tau} \qquad \text{T-Var}\;\frac{\Gamma(x) = \tau}{\Phi; \Gamma; \Sigma \vdash x\colon \tau}$$

$$\text{T-App}\;\frac{\begin{array}{c}\mathbf{adjust}_{\downarrow\uparrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \pi_1(\tau_2 \xrightarrow{\Phi_1} \tau) \\ \mathbf{adjust}_{\pi_1\ \downarrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2\colon \pi_2\rho_2 \\ \mathbf{check}_{\pi_1\ \pi_2}(\Phi) \quad \pi_2\rho_2 <\colon \tau_2 \quad \Phi_1 \subseteq \Phi\end{array}}{\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon \tau} \qquad \text{T-Ref}\;\frac{\begin{array}{c}\mathbf{adjust}_{\mathtt{ref}\ \downarrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}\colon \tau \\ \tau = \pi\rho \quad \mathbf{check}_{\mathtt{ref}\ \pi}(\Phi)\end{array}}{\Phi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_\varepsilon\colon \{\varepsilon\}\mathtt{Ref}\ \tau}$$

$$\text{T-Deref}\;\frac{\begin{array}{c}\mathbf{adjust}_{!\uparrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}\colon \pi\mathtt{Ref}\ \ \tau \\ \mathbf{check}_{!\pi}(\Phi)\end{array}}{\Phi; \Gamma; \Sigma \vdash (!\mathrm{e})_\varepsilon\colon \tau} \qquad \text{T-Assign}\;\frac{\begin{array}{c}\mathbf{adjust}_{\downarrow:=\uparrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \pi_1\mathrm{Ref}\ \tau_1 \\ \mathbf{adjust}_{\pi_1:=\downarrow}(\Phi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2\colon \pi_2\rho_2 \\ \mathbf{check}_{\pi_2:=\pi_2}(\Phi) \quad \pi_2\rho_2 <\colon \tau_1\end{array}}{\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \{\varepsilon\}\mathtt{Unit}}$$

Figure 2.5: Generic Type-and-Effect system introduced by Marino and Millstein

Property 2 (Tag Monotonicity)**.**

- *If* $C_1 \sqsubseteq C_2$ *then* $\mathbf{check}_{C_2}(\Phi) \implies \mathbf{check}_{C_1}(\Phi)$*;*
- *If* $A_1 \sqsubseteq A_2$ *then* $\mathbf{adjust}_{A_2}(\Phi) \subseteq \mathbf{adjust}_{A_1}(\Phi)$*.*

Privilege monotonicity captures the idea that once an expression has sufficient privileges to run, one can always safely add more. This corresponds to effect subsumption in many particular effect systems. In contrast, tag monotonicity captures the idea that more tags implies more uncertainty about the source of a runtime value. The $\sqsubseteq$ relation holds when contexts have the same structure and the tagsets of the first context are subsets of the corresponding tagsets of the second context. For example, $\mathtt{ref}\pi_1 \sqsubseteq \mathtt{ref}\pi_2$ if and only if $\pi_1 \subseteq \pi_2$. In summary, **check** and **adjust** are order-preserving with respect to privileges and order-reversing with respect to tags.

The framework can be instantiated with any pair of check and adjust functions that satisfy both privilege and tag monotonicity. The resulting type system is safe with respect to the corresponding runtime semantics: no runtime privilege check fails, so no program gets stuck.

2.6 Summary

We have introduced several concepts required to understand the work presented in the following chapters. Section 2.1 introduced type systems and language semantics with an example formalization for a language, also defining what makes a formalization "safe" through a property called type safety. The formalized language is the Simply Typed Lambda Calculus

(STLC), a programming language standard in the literature and used as basis for our research. We have also shown the formalization of mutable references, a standard extension of the STLC.

Section 2.2 introduced gradual typing, a type discipline more flexible than the static type system introduced in section 2.1. Gradual typing extends static type systems, accepting programs for which static type information is missing, and introduces runtime checks (casts) to ensure that the optimistic static assumptions made by a gradual typing system are fulfilled at runtime. After introducing gradual typing for the STLC, we extended the language with subtyping and how gradual typing interacts with subtyping.

Section 2.3 introduced type-and-effect systems, a form of type systems which not only focuses on program results as abstracted by types, but also focuses on the side effects that occur during evaluation, like memory usage or exceptions. An example of a language with a type-and-effect system, the fluent language, is introduced in section 2.4.

Section 2.5.1 ends the chapter introducing a generic type-and-effect framework, which can be used to define safe type-and-effect systems. This framework abstracts the similarities between different type-and-effect systems, and allows general reasoning about type-and-effect systems. The section ends presenting the fluent language introduced in section 2.4 as an instance of the generic framework.

We assume familiarity with the ideas here introduced in the following chapter, where concepts from language semantics, type systems, gradual typing and type-and-effect systems are combined to introduce a new concept, gradual effect checking, using the fluent language as a running example.

Chapter 3

Design of Gradual Effect Checking

As we have discussed in section 2.4, the fluent language is a simple example of a type-and-effect system. In the fluent language, programs are evaluated with a particular set of privileges, set that represents the side effects allowed for the program. Programmers may restrict the privileges available through effect ascriptions. The fluent type system statically tracks privileges and verifies that all restrictions are consistent with the program requirements, signaling an error otherwise.

When first proposing effect systems, Gifford and Lucassen claimed that effect checking could be performed either statically or dynamically. Unfortunately, they did not provide a definition of what "dynamic checking" meant, perhaps because the type system they proposed made runtime verifications redundant.

In this chapter we introduce a statically typed language with *gradual effect checking*. This language automatically verifies privilege restrictions in expressions, just as the fluent language, but without forcing the language user to declare complete privilege information. Whenever the system cannot infer statically if the available privileges are sufficient or clearly insufficient, it will introduce a runtime verification. In this language, effect ascription can not only enforce privilege restrictions, but also hide privilege information, enforcing runtime verification.

For simplicity, we use the fluent language as a playground to design gradual effect checking. After we settle on a particular design, in the following chapters we will introduce gradual effect checking in more complex languages. The rest of this chapter presents our design goals, different approaches to introduce the notion of unknown effect information in a fluent language, and our proposal of the gradual effect checking system itself.

3.1 What is gradual effect checking?

In the fluent language, programmers can use the effect ascription construct ($e :: \Phi$) to restrict privileges available for an expression e. Privileges can also be restricted through function type declarations. For example, the function shown as program 3.1, which is defined in a dynamic language without type annotations, should not be applied to an argument that requires any privileges.

$$(\lambda f\ .\ (f\ \texttt{unit}) :: \emptyset) \tag{3.1}$$

With this restriction, the identity function is a valid argument for function 3.1, but $(\lambda x\ .\ (!(\texttt{ref}\ x)))$ is not, because in the fluent language this function requires both the `alloc` and `read` privileges. Statically, this restriction can be enforced annotating the f argument with a function type that includes effect restrictions. An annotated version of 3.1 valid in the fluent language would be

$$(\lambda f\colon \texttt{Unit}\xrightarrow{\emptyset}\texttt{Unit}\ .\ (f\ \texttt{unit}) :: \emptyset) \tag{3.2}$$

The type-and-effect system uses argument f's declared privilege set to check if privilege restrictions in the body of 3.2 are valid. However, this forces the programmer to declare effect annotations early in the development process. A gradual effect checking system is a type-and-effect system in which programs both with and without privilege annotations can interact. The programmer decides when and where to introduce ascriptions and effect annotations in function types. The type system uses the available static information to catch inconsistencies, makes optimistic assumptions when lacking information and verifies at runtime that these assumptions hold.

Therefore, in a gradual effect checking system, the privilege set required by f can be missing. We want to allow the programmer to declare annotated function types, but also to be able to declare function 3.1 using plain type annotations like those in the simply-typed lambda calculus:

$$(\lambda f\colon \texttt{Unit} \longrightarrow \texttt{Unit}\ .\ (f\ \texttt{unit}) :: \emptyset) \tag{3.3}$$

A gradual effect system specifies how the type system behaves when privilege information is missing, like in program 3.3. The design of a gradual effect system shall decide whether this program is rejected or accepted. A program may be accepted either because every privilege restriction is fullfilled, like in the fluent language, or because restrictions *may* be fullfilled, like for function 3.1, in which case some checks must be performed at runtime. The decision of which programs to accept or to reject is guided by the design goals described in the following section.

3.2 Design goals for gradual effect checking

Several goals guide our design of a gradual effect system. A gradual effect checking type system should at least accept more programs than a static effect system. In particular, it should accept all the programs for which static information is not sufficient to make a clear claim (either a "yes" or a "no") about the fullfilment of the privilege restrictions.

A first design goal is to reject inconsistent programs. A gradual effect system must identify inconsistencies between the privilege sets declared in effect ascriptions and the required privileges identified for the ascripted expressions. If a program requires more privileges than those optimistically assumed, runtime privilege checks will always fail, and thus the program inconsistency should generate a static error.

A second design goal is to provide the same guarantees given by a static effect system for annotated programs. This means that programs with complete privilege information should only be accepted if every side effect produced by an expression in the program is allowed in the expression's context. We consider gradual effect checking as an extension to effect checking in static effect systems to achieve this goal, because then the process of gradual effect checking a fully annotated program is equivalent to the original effect checking process.

A third design goal is to ensure that every privilege required at execution time is in fact available. When a gradual effect system does not have enough static effect information, it must make an optimistic estimation. This estimate can include more privileges than those actually available at runtime. Thus every privilege requirement that cannot be statically verified must be verified at runtime.

A fourth design goal is to minimize the runtime cost of privilege checking. Every runtime check is an overhead, so we want to introduce as few runtime checks as possible. If the type system identified statically that a privilege is available, it should not check its availability again at runtime.

3.3 Representing effect uncertainty

To introduce gradual effect checking, we follow a similar approach to how Siek and Taha introduced Gradual Typing[24]. To handle missing type information, Siek and Taha identified that types in the STLC are compared by equality, and introduced a new relation, called consistency ($\sim$), to compare types with missing information. They defined $\sim$ to work as an extension of equality ($a = b \Rightarrow a \sim b$). We also want gradual effect checking to work as an extension of the original fluent type-and-effect system. To do this, we first need to identify how the fluent language handles privileges, and abstract that behavior as a pattern.

The fluent language represents privileges as sets, and verifies privileges by comparing sets through the set containment ($\subseteq$) operation, which defines a partial order among sets. We can consider the static fluent effect system as an instance of a generic system depending on a particular binary relation on sets $\sqsubseteq$, and define our gradual effect system by proposing a new binary relation to replace containment and handle missing information (which we will call unknown privileges). To draw an explicit connection with Gradual Typing, we call this relation *consistent containment* and use the symbol $\underset{\sim}{\sqsubset}$ to denote it.

The type system of the fluent language also uses subtyping. Subtyping expresses the notion that if a context allows to apply a function requiring certain privileges, any other function that requires less privileges is also allowed. To achieve this goal, the subtyping relation in the fluent language compared privilege sets by containment. We then introduce a new notion of subtyping for gradual effect checking, using consistent containment instead

$$\text{T-Fn}\,\frac{\Xi_1; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}\colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})\colon T_1 \xrightarrow{\Xi_1} T_2} \qquad \text{T-Unit}\,\frac{}{\Xi; \Gamma; \Sigma; \vdash \texttt{unit}\colon \texttt{Unit}}$$

$$\text{T-Loc}\,\frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l\colon \texttt{Ref}\ T} \qquad \text{T-Var}\,\frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x\colon T} \qquad \text{T-App}\,\frac{\begin{array}{c}\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\colon T_1 \xrightarrow{\Xi_1} T_3 \\ \Xi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2 \\ T_2 \lesssim T_1 \qquad \Xi_1 \underset{\sim}{\sqsubset} \Xi\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon \tau_3}$$

$$\text{T-Ref}\,\frac{\begin{array}{c}\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T \\ \{\texttt{alloc}\} \underset{\sim}{\sqsubset} \Xi\end{array}}{\Xi; \Gamma; \Sigma \vdash \texttt{ref}\ \ \mathrm{e}\colon \texttt{Ref}\ T} \qquad \text{T-Deref}\,\frac{\begin{array}{c}\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ T \\ \{\texttt{read}\} \underset{\sim}{\sqsubset} \Xi\end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e}\colon T}$$

$$\text{T-Assign}\,\frac{\begin{array}{c}\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ T_1 \\ \Xi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2 \\ \{\texttt{write}\} \underset{\sim}{\sqsubset} \Xi \qquad T_2 \lesssim T_1\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \texttt{Unit}} \qquad \text{T-Ascription}\,\frac{\begin{array}{c}\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T \\ \Xi_1 \underset{\sim}{\sqsubset} \Xi\end{array}}{\Xi; \Gamma; \Sigma; \vdash \mathrm{e} :: \Xi_1\colon T}$$

$$\text{ST-Id}\,\frac{}{T \lesssim T} \qquad \text{ST-Fn}\,\frac{\begin{array}{c}T'_1 \lesssim T_1 \qquad T_2 \lesssim T'_2 \\ \Xi \underset{\sim}{\sqsubset} \Xi'\end{array}}{T_1 \xrightarrow{\Xi} T_2 \lesssim T'_1 \xrightarrow{\Xi'} T'_2}$$

Figure 3.1: A type system for the fluent language with gradual effect checking.

of set containment to compare privilege annotations in function types. We call this notion *consistent subtyping* ($\lesssim$), just like Siek and Taha [23].

We use almost the same inference rules of the type system in the fluent language for gradual effect checking, the only differences being the use of consistent containment $\underset{\sim}{\sqsubset}$ instead of set containment $\subseteq$ and consistent subtyping $\lesssim$ instead of subtyping $<:$. To achieve the design goals presented in section 3.2, we need the $\underset{\sim}{\sqsubset}$ relation to always work like set containment for fully known privilege sets. If this condition holds, the gradual system is equivalent to the static fluent type system when no information is missing.

For a gradual effect system to accept programs that do not have privilege annotations, like $(\lambda f\colon \texttt{Unit} \longrightarrow \texttt{Unit}\ .\ f)$, we need a way to declare missing privilege information in the language. We chose to use the symbol "¿" for any notion of unknown information related to effects, much as Siek and Taha used "?" to declare unknown types in their Gradual Typing framework.

With these definitions, we present in fig. 3.1 a parameterized type system for gradual effect checking, which only depends on the meaning of privilege uncertainty and the corresponding definition of consistent containment ($\underset{\sim}{\sqsubset}$).

We now proceed to introduce different candidate meanings for ¿ and their corresponding definitions for consistent containment. First, we introduce ¿ as an "unknown privilege set". Though this design mirrors the "unknown type" introduced by Siek and Taha, we explain why this approach is insufficient for our design goals. We then present how the notion of "unknown privileges" can be represented as a privilege, which holds a special meaning when is present in the set. Having the ¿ privilege in a set signals that the system cannot statically tell if any more privileges are available or are required.

Different meanings for ¿ affect how the domain of privilege sets is defined. To remind the reader that the domain of privilege sets that can handle missing information is different (and bigger) than the domain of standard privilege sets, we will use different notation for each. **PSet** will represent the domain of standard privilege sets, and $\Phi \in \mathbf{PSet}$, while **CPSet** will represent the domain of privilege sets that can also handle missing information, and $\Xi \in \mathbf{CPSet}$.

3.3.1 First approach: unknown information as a privilege set

In Gradual Typing, unknown type annotations are declared with ?, which is a type like any other type in the system. Thus we attempt a similar approach to define gradual effect checking, representing missing effect information with a distinct set, the unknown privilege set ¿. The ¿ set becomes a black box that may contain any privileges. With this definition, the domain of privilege sets **CPSet** becomes $\mathbf{PSet} \cup \{¿\}$.

To use this set, we need to define a consistent containment relation. This definition requires the following properties:
Property 3. *The unknown set can always be contained in another set, since in the best case, ¿ is empty.*
Property 4. *Any set can be contained in the unknown set, since in the worst case, every privilege is contained in ¿.*

Property 3 allows to statically type programs like

$$(\lambda f\colon \texttt{Unit} \xrightarrow{¿} \texttt{Unit}\ .\ (f\ \texttt{unit}) :: \emptyset)$$

The effect ascription in the body of this abstraction forces a runtime check that $(f\ \texttt{unit})$ does not require any privilege, as forced by the ascription to the empty privilege set $\emptyset$. If a program applied the previous function to $(\lambda x\colon \texttt{Unit}\ .\ !(\texttt{ref}\ x))$, it should identify at runtime that $(f\ \texttt{unit})$ requires a missing privilege set $\{\mathsf{read}, \mathsf{write}\} \not\subseteq \emptyset$, and therefore it should signal an error.

Property 4 makes the type system accept programs that hide effect information, like the following:

$$(\texttt{ref unit}) :: ¿$$

A simple definition for consistent containment forces both properties, and falls back to set containment when there is no unknown information:

$$a \mathrel{\widetilde{\sqsubseteq}} b = \begin{cases} \text{true} & \text{if } a = ¿ \text{ or } b = ¿ \\ a \subseteq b & \text{otherwise} \end{cases} \tag{3.4}$$

3.3.2 Limitations of the first approach

The simplicity of the fluent language makes the unknown privilege set sufficient to define gradual effect checking. The unknown privilege set is sufficient because all the privilege operations in fluent can be represented in terms of consistent containment. If we considered richer operations, the language would start losing static information.

Sets may compose through many operations, like union and intersection. Though the unknown privilege set allows a simple definition for consistent containment, defining consistent union and intersection operations loses information that could be used by the type system.

Union between a set a and a set b ($a \cup b$) shall contain every element in a and every element in b. Statically, the set ¿ is unknown, so it may range from the empty set $\emptyset$ to every possible privilege. Though we can identify that union with set ¿ should contain at least the elements in the set being united to ¿, we have no way to distinguish this partially known set from the set ¿. If the system has an unknown privilege set, we can define consistent union $\sqcup$ as follows:

$$\begin{array}{rcll} \Xi_1 \sqcup \Xi_2 & = & ¿ & \text{if either } \Xi_1 = ¿ \text{ or } \Xi_2 = ¿ \\ \Xi_1 \sqcup \Xi_2 & = & \Xi_1 \cup \Xi_2 & \text{otherwise} \end{array}$$

As an example, consider extending fluent with exceptions in a simple form, having only one kind of exception: `throw`. A `throw` expression would require the privilege of throwing exceptions:

$$\frac{\{\texttt{throw}\} \sqsubseteq\!\!\!\!\sim \Xi}{\Xi; \Gamma; \Sigma \vdash \texttt{throw} \colon T}$$

A `try` e_1 `catch` e_2 expression would handle exceptions, making the `throw` privilege available for e_1:

$$\frac{\begin{array}{c}\Xi \sqcup \{\texttt{throw}\}; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon T \\ \Xi; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T\end{array}}{\Xi; \Gamma; \Sigma \vdash \texttt{try}\ \mathrm{e}_1\ \texttt{catch}\ \mathrm{e}_2 \colon T}$$

Though it is clear that e_1 should assume that `throw` is available, our definition of consistent union cannot make this assumption under the unknown privilege set restriction:

$$\frac{\begin{array}{c}¿ \sqcup \{\texttt{throw}\} = ¿; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon T \\ ¿; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T\end{array}}{¿; \Gamma; \Sigma \vdash \texttt{try}\ \mathrm{e}_1\ \texttt{catch}\ \mathrm{e}_2 \colon T} \tag{3.5}$$

And therefore, whenever an exception is raised in e_1, presence of `throw` privilege needs to be explicitly checked, even though the system could statically identify that the privilege is always available.

3.3.3 Second approach: unknown information as a privilege

One of the design goals of our system is to minimize the runtime cost of effect checking. In the previous section we showed an example where using the unknown privilege set introduces redundant runtime cheks, so we look for a different representation for privilege uncertainty. Instead of presenting uncertainty as an unknown privilege set, we represent uncertainty with a special privilege, named ¿. By using standard set operations like $\cup$ and $\cap$, ¿ can be used directly, and programs like 3.5 do not need to introduce redundant runtime checks.

However, there is still a special semantics we want to give to the element ¿, affecting the behavior of consistent containment. Consistent containment ($\mathrel{\underset{\sim}{\sqsubseteq}}$) differs from standard set containment, because the $\mathrel{\underset{\sim}{\sqsubseteq}}$ operator has to optimistically consider the unknown part of a set. At runtime, ¿ might range from no privileges at all to every privilege in the universe, and both cases have to be accepted statically by $a \mathrel{\underset{\sim}{\sqsubseteq}} b$. Whenever $¿ \in b$, *any* privilege might be part of the set b at runtime, thus any set a might be contained in set b. If $¿ \in a$, the optimistic assumption is that at runtime there is no new privilege introduced, but still the rest of the set a should be contained in b. We define the "rest of the set" as the static part.
Definition 1 (Static Part)**.** *For a set a, we define the static part $|a|$ as $a \setminus \{¿\}$.*

We can also define a notion of containment for static parts.
Definition 2 (Static Containment)**.** *We define the static containment of two sets, $a \leq b$, as $|a| \subseteq |b|$.*

This definition of static containment encapsulates the second assumption we presented, so we can introduce consistent containment as follows:
Definition 3 (Consistent Containment)**.** *We define Consistent Containment, $a \mathrel{\underset{\sim}{\sqsubseteq}} b$, as*

$$a \mathrel{\underset{\sim}{\sqsubseteq}} b \iff ¿ \in b \text{ or } a \leq b$$

3.4 The intermediate language: checking inconsistencies at runtime

The type system we have proposed for the gradual effect fluent language also accepts programs that should fail at runtime. Therefore, to ensure the right runtime behavior we need to introduce runtime checks for the cases where the static information was insufficient. We achieve this goal by translating programs in the gradual effect fluent language to an intermediate language with explicit checks. The translation is performed by a type-directed relation that introduces checks while assigning a type to the program.

In our operational semantics, expressions that generate a side effect include an explicit privilege check, which is required to pass in order to let the expression reduce. If the required privilege is not available, evaluation gets stuck. We mentioned in section 2.1.3 that type safety guarantees that programs do not get stuck. We include the explicit privilege checks in reduction rules so that if we prove that the language is type safe, the proof guarantees that privileges will always be available when required. Type safety for this language is proven in section 3.5.1.

The intermediate language we propose does not include effect ascription, but introduces two dynamic features related to privileges, to which we map effect ascription: `restrict`, which constrains the set of available privileges, and `has`, which verifies that a set of privileges is available at runtime.

The language syntax limits which privilege sets are allowed for `restrict` and `has`. Semantically, `has` represent a check: whenever a `has` verification passes, the system can be sure that the privileges it declares are available. To do so, we limit `has` to only manage concrete privileges, without uncertainty. However, effect ascription may hide privilege information by introducing ¿. This behavior is isolated into the `restrict` construct, which accepts any kind of privilege set.

The syntax of our intermediate language is defined as follows:

$$\begin{array}{lcl}
\Phi & \in & \mathbf{PSet} \\
\Xi & \in & \mathbf{CPSet} \\
T & ::= & \texttt{Unit} \mid \texttt{Ref}\ \ T \mid T \xrightarrow{\Xi} T \\
\mathrm{e} & ::= & \texttt{unit} \mid \texttt{ref}\ \ \mathrm{e} \mid !\ \mathrm{e} \mid \mathrm{e} := \mathrm{e} \mid \\
 & & \lambda x\colon T\ .\ \mathrm{e} \mid x \mid \mathrm{e}\ \mathrm{e} \mid \\
 & & \texttt{restrict}\ \Xi\ \mathrm{e} \mid \texttt{has}\ \Phi\ \mathrm{e} \mid \mathsf{Error} \\
v & ::= & \texttt{unit} \mid l \mid (\lambda x\colon T\ .\ \mathrm{e}) \mid \langle T \Leftarrow T \rangle v
\end{array}$$

3.4.1 A type system for the intermediate language

The type system for the intermediate language we present in fig. 3.2 is very similar to the one presented in fig. 3.1, but removing the [T-Ascription] rule (since the construct does not exist) and adding a [T-Restrict] and a [T-Has] for the new features and, for completeness, a rule to type the runtime effect errors [T-Error].

A program in the intermediate language is the result of a translation. The translation algorithm must insert explicit checks for every case where static information was insufficient in the original program, so that the type system for the intermediate language can be more restrictive and assume every required privilege to be previously checked. The type system isolates checks into the `has` construct, and the type system assumes statically that runtime checks will pass. Rule [T-Has] types the checked expression under the assumption that the verified privileges are in fact available.

In effect ascription, consistent containment ($\mathrel{\widetilde{\sqsubseteq}}$) plays two different roles. On one side, it verifies that the declared set of privileges for the ascripted expression is available. As we already mentioned, this behavior is managed by inserting `has` expressions in the intermediate

$$\text{T-Unit}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \texttt{unit}\colon \texttt{Unit}} \qquad \text{T-Fn}\ \frac{\Xi';\Gamma, x\colon T_1;\Sigma \vdash \mathrm{e}\colon T_2}{\Xi;\Gamma;\Sigma \vdash \lambda x\colon T_1\ .\ \mathrm{e}\colon T_1 \xrightarrow{\Xi'} T_2}$$

$$\text{T-Loc}\ \frac{\Sigma(l) = T}{\Xi;\Gamma;\Sigma \vdash l\colon \texttt{Ref}\ \ T} \qquad \text{T-Var}\ \frac{\Gamma(x) = T}{\Xi;\Gamma;\Sigma \vdash x\colon T}$$

$$\text{T-Ref}\ \frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T \qquad \{\texttt{alloc}\} \subseteq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T} \qquad \text{T-Deref}\ \frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ \ T \qquad \{\texttt{read}\} \subseteq \Xi}{\Xi;\Gamma;\Sigma \vdash !\ \mathrm{e}\colon T}$$

$$\text{T-Assign}\ \frac{\begin{array}{c}\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ \ T_1 \\ \Xi;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2 \\ T_2 <\colon T_1 \qquad \{\texttt{write}\} \subseteq \Xi\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \texttt{Unit}} \qquad \text{T-App}\ \frac{\begin{array}{c}\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\colon T_1 \xrightarrow{\Xi'} T_3 \\ \Xi;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2 \\ T_2 <\colon T_1 \qquad \Xi' \subseteq \Xi\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon T_3}$$

$$\text{T-Restrict}\ \frac{\begin{array}{c}\Xi';\Gamma;\Sigma \vdash \mathrm{e}\colon T \\ \Xi' \lesssim \Xi\end{array}}{\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi'\ \mathrm{e}\colon T} \qquad \text{T-Has}\ \frac{\Phi \cup \Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T}{\Xi;\Gamma;\Sigma \vdash \texttt{has}\ \Phi\ \mathrm{e}\colon T}$$

$$\text{T-Cast}\ \frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T_0 \qquad T_0 <\colon T_1}{\Xi;\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e}\colon T_2} \qquad \text{T-Error}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error}\colon T}$$

Figure 3.2: Type system for the intermediate language

language. But on the other side, it also has to constrain the set of privileges available for the ascripted expression, constraints that also need to be enforced at runtime. The privilege constraining behavior of ascription is encapsulated in `restrict`, as seen in rule [T-Restrict]

3.4.2 Runtime semantics for the intermediate language

In this section, we present the runtime semantics for the intermediate language. We introduce first a notion to relate the privilege sets used for typing and the privilege sets used for evaluation, and we then introduce the reduction rules for the operational semantics.

Modelling concrete privilege sets

In the fluent language, the type system and the runtime semantics used the same kind of privilege sets. In gradual effect checking, the type system uses consistent privilege sets, which model missing information. However, we restrict the operational semantics to use only privilege sets without uncertainty (Φ) as contexts, so evaluation has full knowledge of the available privileges. We then require to restrict the privilege sets that may be used for evaluation when a set with uncertainty is used by the type system.

We call the privilege sets Φ that may be used in the context of a consistent privilege set Ξ as being *modeled* by the consistent set (notated $\Xi \vdash \Phi$).

Definition 4 (Modelling privilege sets)**.** *A privilege set Φ is modeled by a consistent privilege set Ξ, denoted $\Xi \vdash \Phi$, if at least Φ contains all the privileges that are statically known to be part of Ξ. Formally*

$$\Xi \vdash \Phi \iff \Xi \leq \Phi$$

We use this notion of set modelling to relate the type system to the semantics in our definition of type safety, in section 3.5.1.

The operational semantics

We now introduce the operational semantics for the intermediate language. Given that the language is an extension of the simply typed lambda calculus with references and unit, already introduced in section 2.1.5, we only describe the effect-related features `has`, `restrict`, and our extension to handle higher order casts[1] with effect information. The full runtime semantics is presented in fig. 3.4.

As we have previously described, `has` expressions verify that a set of explicit privileges is available at runtime. To do this, it compares the set given as an argument with the set in the context. Rule [E-Has-F] triggers an error when the required privileges are not available. Rule [E-Has-T] evaluates the subexpression, but only if the privilege set given as an argument is contained in the context. Though this verification could be performed only once and then the `has` expresion could be removed, we keep the expression around and reduce the contained expression in it for our type soundness proof. The `has` wrapping is discarded by rule [E-Has-V] when a value is reached, and is discarded by rule [E-Error] when reducing the expression produces an error.

A `restrict` expression limits privileges available in the context. However, the set given as an argument that contains the limit might contain uncertainty. How do we limit to a set which is uncertain? We say that a set that is uncertain does not give any extra runtime information, so we keep the original context instead for rule [E-Rst-1]. When the set has no uncertainty, as in the case for rule [E-Rst-2], we can use it as a new context to evaluate the contained expression.

A higher order cast does not only need to verify the structural subtyping rules, but also verify that the effect restrictions among the sets declared in the types are actually enforced. We achieve this by introducing a `restrict` clause for the set in the target type of the cast, and verify that the privileges declared for the known static part of the origin cast type are also available. This restrictions ensure that required privileges are available for nested higher-order casts.

[1]We describe how to handle higher order casts when introducing an operational semantics for casts in Gradual Typing in section 2.2.3.

3.4.3 Translating programs to the intermediate language

The consistent containment operation ensures that our type system accepts both the programs that can be proved correct statically and those that need verification at runtime. However, we would like our translation to add only a minimal number of verifications, thus separating the cases when privileges can be found statically in Ξ from those where the presence of ¿ in Ξ forces a runtime verification. To achieve this goal we propose the translation algorithm in fig. 3.3.

Our translation algorithm uses two auxiliary functions to reduce the number of rules required to describe the system and to make rules more readable. The *insert-has?* function inserts runtime verifications whenever the statically identified privilege set is not sufficient to satisfy the privilege restrictions established by the type system.

$$\textit{insert-has?}\ \Xi_1\ \Xi_2\ \mathrm{e} = \begin{cases} \mathrm{e} & \text{if } \Xi_1 \subseteq \Xi_2 \\ \texttt{has}\ \ \Xi_1\ \mathrm{e} & \text{if } \Xi_1 \not\subseteq \Xi_2 \end{cases} \tag{3.6}$$

We also use a cast insertion function $\langle\!\langle T_2 \Leftarrow T_1 \rangle\!\rangle$. This function inserts casts whenever they are explicitly required: that is, when two types are related by the consistent subtyping relation of the original language, but not by the subtyping relation of the intermediate language, which is more restrictive.

$$\langle\!\langle T_2 \Leftarrow T_1 \rangle\!\rangle \mathrm{e} = \begin{cases} \mathrm{e} & \text{if } T_1 <: T_2 \\ \langle T_2 \Leftarrow T_1 \rangle \mathrm{e} & \text{otherwise} \end{cases} \tag{3.7}$$

The subtyping relation is defined exactly as in the fluent language, but changing the domain of privilege sets to consistent privilege sets.

3.5 Theorems for gradual effect checking

In this section we introduce formal results about the language. We first introduce restrictions between the original language type system and the translation relation. Lemma 4 guarantees that if a program holds a type in the original type system, the program will hold the same type during translation to the intermediate language. We also formalize type safety for the intermediate language in section 3.5.1.

Lemma 4. *[Translation preserves well-typing] If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' : \tau$*, then* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}' : \tau$ *in the intermediate language.*

Proof. Straightforward induction on the last step of the translation $\Rightarrow$. □

3.5.1 Type safety of the language

We now introduce the type safety statements for the language presented in this chapter. Detailed proofs are provided in appendix B.

Theorem 5 (Progress)**.** *Suppose* e *is a closed, well typed expression (*$\exists\, T, \Sigma, \Xi\ .\ \Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$*). Then either* e *is a value, an* Error*, or else, for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$ *and for any privilege set* Φ *such that* $\Xi \vdash \Phi$*, there is some* e' *and* μ' *with* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$*.*

Proof. By Structural Induction on the typing derivation using the Inversion of Typing for the Intermediate Language Lemmas and the Inversion of $\precsim$ lemma. □

Theorem 6. *(Preservation)*

If $\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T$*,* $\Gamma \mid \Sigma \vDash \mu$ *and* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *with* $\Xi \vdash \Phi$*, then* $\exists \Sigma' \supseteq \Sigma$ *such that* $\Phi; \Gamma; \Sigma' \vdash \mathrm{e}'\colon T$ *and* $\Gamma \mid \Sigma' \vDash \mu'$*.*

Proof. By structural induction over the type derivation, and then by cases on the rules in relation $\rightarrow$ that may apply for terms of the form e accepted by the typing rule. □

$$\text{C-Unit}\;\frac{}{\Xi;\Gamma;\Sigma\vdash \texttt{unit}\Rightarrow \texttt{unit}\colon \texttt{Unit}}\qquad\qquad \text{C-Loc}\;\frac{\Sigma(l)=T}{\Xi;\Gamma;\Sigma\vdash l\Rightarrow l\colon \texttt{Ref}\ T}$$

$$\text{C-Var}\;\frac{\Gamma(x)=T}{\Xi;\Gamma;\Sigma\vdash x\Rightarrow x\colon T}\qquad \text{C-Fn}\;\frac{\Xi';\Gamma,x\colon T_1;\Sigma\vdash \mathrm{e}_1\Rightarrow \mathrm{e}_2\colon T_2}{\Xi;\Gamma;\Sigma\vdash (\lambda x\colon T_1\ .\ \mathrm{e}_1)\Rightarrow (\lambda x\colon T_1\ .\ \mathrm{e}_2)\colon T_1\xrightarrow{\Xi'}T_2}$$

$$\text{C-App}\;\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash \mathrm{e}_1\Rightarrow \mathrm{e}_1'\colon T_1\xrightarrow{\Xi'}T_3\\ \Xi;\Gamma;\Sigma\vdash \mathrm{e}_2\Rightarrow \mathrm{e}_2'\colon T_2\\ T_2<\colon T_1\qquad\qquad \Xi'\underset{\sim}{\sqsubset}\Xi\end{array}}{\Xi;\Gamma;\Sigma\vdash \mathrm{e}_1\ \mathrm{e}_2\Rightarrow \textit{insert-has?}\ \Xi'\ \Xi\ (\langle\!\langle T_2\xrightarrow{\Xi}T_3\Leftarrow T_1\xrightarrow{\Xi'}T_3\rangle\!\rangle\ \mathrm{e}_1')\ \mathrm{e}_2'\colon T_3}$$

$$\text{C-Ref}\;\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash \mathrm{e}\Rightarrow \mathrm{e}'\colon T\\ \{\texttt{alloc}\}\underset{\sim}{\sqsubset}\Xi\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{ref}\ \mathrm{e}\Rightarrow \textit{insert-has?}\ \{\texttt{alloc}\}\ \Xi\ \texttt{ref}\ \mathrm{e}'\colon \texttt{Ref}\ T}$$

$$\text{C-Deref}\;\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash \mathrm{e}\Rightarrow \mathrm{e}'\colon \texttt{Ref}\ T\\ \{\texttt{read}\}\underset{\sim}{\sqsubset}\Xi\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{deref}\ \mathrm{e}\Rightarrow \textit{insert-has?}\ \{\texttt{read}\}\ \Xi\ !\mathrm{e}'\colon T}$$

$$\text{C-Assign}\;\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash \mathrm{e}_1\Rightarrow \mathrm{e}_1'\colon \texttt{Ref}\ T_1\\ \Xi;\Gamma;\Sigma\vdash \mathrm{e}_2\Rightarrow \mathrm{e}_2'\colon T_2\\ \{\texttt{write}\}\underset{\sim}{\sqsubset}\Xi\qquad T_2<\colon T_1\end{array}}{\Xi;\Gamma;\Sigma\vdash \mathrm{e}_1:=\mathrm{e}_2\Rightarrow \textit{insert-has?}\ \{\texttt{write}\}\ \Xi\ \mathrm{e}_1':=\mathrm{e}_2'\colon \texttt{Unit}}$$

$$\text{C-Ascription}\;\frac{\begin{array}{c}\Xi_1;\Gamma;\Sigma\vdash \mathrm{e}\Rightarrow \mathrm{e}'\colon T\\ \Xi_1\underset{\sim}{\sqsubset}\Xi\end{array}}{\Xi;\Gamma;\Sigma;\Phi\vdash \Phi_1::\mathrm{e}\Rightarrow \textit{insert-has?}\ \Xi_1\ \Xi\ \texttt{restrict}\ \Xi_1\ \mathrm{e}'\colon T}$$

Figure 3.3: The translation algorithm for Gradual Effect Fluent

$$\begin{array}{lll} f & ::= & \Box\ \mathrm{e} \mid v\ \Box \mid \texttt{ref}\ \Box \mid !\Box \mid \Box := \mathrm{e} \mid v := \Box \mid \langle T_2 \Leftarrow T_1 \rangle\ \Box \\ g & ::= & f \mid \texttt{has}\ \Phi\ \Box \mid \texttt{restrict}\ \Phi\ \Box \end{array}$$

$$\text{E-Frame}\ \frac{\Phi' \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu' \qquad \Phi' = \Phi}{\Phi \vdash \mathit{plug}(f, \mathrm{e}) \mid \mu \to \mathit{plug}(f, \mathrm{e}') \mid \mu'}$$

$$\text{E-Error}\ \frac{}{\Phi \vdash \mathit{plug}(g, \mathrm{e}) \mid \mu \to \mathsf{Error} \mid \mu'} \qquad \text{E-App}\ \frac{}{\Phi \vdash (\lambda x\colon T\ .\ \mathrm{e})\ v \mid \mu \to [{}^{v}/_{x}]\,\mathrm{e} \mid \mu}$$

$$\text{E-Ref}\ \frac{l \notin \mathrm{dom}\,(\mu) \qquad \{\texttt{alloc}\} \subseteq \Phi}{\Phi \vdash \texttt{ref}\ v \mid \mu \to l \mid \mu[l \mapsto v]} \qquad \text{E-Deref}\ \frac{\mu\ l = \lfloor v \rfloor \qquad \{\texttt{read}\} \subseteq \Phi}{\Phi \vdash !l \mid \mu \to v \mid \mu}$$

$$\text{E-Assign}\ \frac{\{\texttt{write}\} \subseteq \Phi}{\Phi \vdash l := v \mid \mu \to \texttt{unit} \mid \mu[l \mapsto v]} \qquad \text{E-Has-T}\ \frac{\Phi' \subseteq \Phi \qquad \Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \texttt{has}\ \Phi'\ \mathrm{e}' \mid \mu'}$$

$$\text{E-Has-V}\ \frac{}{\Phi \vdash \texttt{has}\ \Phi'\ v \mid \mu \to v \mid \mu} \qquad \text{E-Has-F}\ \frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \mathsf{Error} \mid \mu}$$

$$\text{E-Rst-V}\ \frac{}{\Phi \vdash \texttt{restrict}\ \Xi\ v \mid \mu \to v \mid \mu}$$

$$\text{E-Rst-1}\ \frac{¿ \in \Xi \qquad \Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \to \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'}$$

$$\text{E-Rst-2}\ \frac{¿ \notin \Xi \qquad |\Xi| \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \to \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'} \qquad \begin{array}{l}\text{E-Cast-Id} \\ \dfrac{}{\Phi \vdash \langle T \Leftarrow T \rangle v \mid \mu \to v \mid \mu}\end{array}$$

$$\begin{array}{l}\text{E-Cast-Fn} \\ \dfrac{}{\Phi \vdash \langle T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow T_{11} \xrightarrow{\Xi_1} T_{12} \rangle\,(\lambda x\colon T_{11}\ .\ \mathrm{e}) \mid \mu \to (\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12} \rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ \ (|\Xi_1| \setminus |\Xi_2|)\ \ [{}^{(\langle T_{11} \Leftarrow T_{21} \rangle x)}/_{x}]\,\mathrm{e}) \mid \mu}\end{array}$$

Figure 3.4: Operational semantics for the Intermediate Language

3.6 Summary

In this chapter we have introduced a language with gradual effect checking, an extension for type-and-effect systems that does not force the language user to declare complete privilege information, combining static and dynamic checking of an effect discipline. Section 3.2 introduced the four design goals that were considered to provide a definition of gradual effect checking. Section 3.1 presented the limitations of type-and-effect systems and an example of a program valid with gradual effect checking that is rejected in a static type-and-effect system because of its lack of static effect information.

The lack of information in a program must be formally represented. Section 3.3 shows two different representations for unknown information, first as a privilege set and then as a privilege. Though both representations are sound for the example of the fluent language, we decided against representing unknown information as a privilege set. When set operations like union are part of a type discipline, using a privilege set for unknown information must discard the available information, and that information could be used to reduce dynamic checking.

In this chapter we follow a strategy similar to the introduction of gradual typing in [24] to introduce gradual effect checking. Section 3.4 introduces an intermediate language for gradual effect checking which can perform explicit runtime checks of effect restrictions, and a translation algorithm that inserts checks whenever static information was not sufficient to ensure the availability of a required privilege. Section 3.5 introduced the theorems required to ensure type safety in the introduced language and to verify that the translation algorithm preserves the typing semantics of the original language.

We have introduced gradual effect checking only for a particular type-and-effect system, the fluent language. However, there are many other type-and-effect systems that we'd like to extend with gradual typing. To avoid having to reintroduce gradual effect checking for every type-and-effect system and prove type safety for it, in the next chapter we define gradual effect checking for the generic type-and-effect framework of Marino & Millstein introduced in section 2.5.1.

Chapter 4

Generic Gradual Effect Checking

This chapter introduces Gradual Effect Checking for the Generic Type-and-Effect framework introduced by Marino and Millstein (M&M) [15], which was briefly presented in section 2.5.1.

In chapter 3, we defined a new operation, consistent containment ($\mathrel{\underset{\sim}{\sqsubseteq}}$), and used it to introduce a gradual verion of the fluent language. The consistent containment operation represents our intuitions for gradual effect checking. We also mentioned that for more complex languages, the design of gradual effect checking would require the introducion of new special consistent operations, like consistent set union ($\sqcup$) and consistent set intersection ($\sqcap$). This strategy of building new consistent definitions for the operations used among privilege sets is ad-hoc and does not scale to build a generic framework.

When building a generic framework, we want the system to abstract from particular privilege disciplines and provide guarantees for a broad set of disciplines. The M&M framework accepts any effect discipline that complies with their monotonicity restrictions. We could define a set of consistent operations and limit the framework only to accept effect disciplines that use the operations we have defined. However, that would impose extra restrictions to the original framework, and some disciplines that worked with M&M would not work in the gradual generic framework. We explored a different approach, based on abstract interpretation [8]. This new approach accepts any privilege discipline accepted by M&M. With abstract interpretation, we can provide a system that extends the original generic effect framework instead of imposing further restrictions.

In section 4.1 we model gradual effects in the context of abstract interpretation. This sections assumes no previous knowledge of abstract interpretation, introducing the required concepts as needed. section 4.2 introduces the generic gradual effect checking framework in detail, much in the spirit of the introduction of gradual typing: it defines a language and a type system that uses consistency to increase flexibility, an intermediate language with checks that make the consistency assumptions explicit, triggering an error if static assumptions do not hold at runtime, and a translation algorithm that accepts programs from the original language and inserts all the required checks. We also prove type safety for the intermediate language and that the translation algorithm preserves typing.

The two sections of this chapter have been already published as Sections 3 and 4 of our paper accepted at The ACM SIGPLAN International Conference on Functional Programming 2014 [2].

4.1 Gradual effects as an abstract interpretation

In this section we present a formal analysis of gradual effects, guided by the design principles presented in section 3.2. We use abstract interpretation [8] to define our notion of unknown effects, and find that as a result the formal definitions capture our stated design intentions, and that the resulting framework for gradual effects is quite generic and highly reusable.

4.1.1 The challenge of gradual effects

The central concept underlying gradual effects is the idea of *unknown privileges*, ¿. This concept was inspired by the notion of unknown type ? introduced by Siek and Taha [24], but this concept is not as straightforward to understand and formalize.

First, gradual types reflect the tree structure of type names. Siek and Taha treat gradual types as trees with unknown leaves. Two types are deemed consistent whenever their known parts match up exactly. For instance, the types $? \rightarrow$ `Int` and `Bool` $\rightarrow ?$ are consistent because their $\rightarrow$ constructors line up: ? is consistent with any type structure. In contrast, privilege sets are unordered collections of individual effects, so a structure-based definition of consistency is not as immediately apparent.

Second, under gradual typing, the unknown type always stands for one type, so casts always associate an unknown type with one other concrete type. On the contrary, the unknown privileges annotation ¿ stands for any number of privileges: zero, one, or many.

Third, simple types are related to the final value of a computation. In contrast, privileges are related to the dynamic extent of an expression as it produces a final value. As such, defining what it means to gradually check privileges involves tracking steps of computation, rather than wrapping a final value with type information.

Finally, as we have seen in section 2.3, effect systems naturally induce a notion of subtyping, which must be accounted for in a gradual effect system. In general, subtyping characterizes *substitutability*: which expressions or values can be substituted for others, based on static properties. In prior work, Siek and Taha demonstrate how structural subtyping and gradual typing can be combined [23], but the criteria for substitutability differ substantially between structural types and effects, so it is not straightforward to adapt Siek and Taha's design to suit gradual effects.

Our initial attempts to adapt gradual typing to gradual effects met with these challenges. We found abstract interpretation to be an informative and effective framework in which to specify and develop gradual effects. The rest of this section develops the notion of unknown effect privileges and consistent privilege sets. The rest of the chapter then uses the framework as needed to introduce concepts and formalize gradual effect checking.

4.1.2 Fundamental concepts

This subsection conceives gradual effects as an instance of abstract interpretation. We do not assume any prior familiarity with abstract interpretation: we build up the relevant concepts as needed.

For purpose of discussion, consider again the effect privileges for mutable state from section 2.4:

$$\begin{aligned}\mathbf{Priv} &= \{\texttt{read}, \texttt{write}, \texttt{alloc}\}\\ \mathbf{CPriv} &= \{\texttt{read}, \texttt{write}, \texttt{alloc}, ¿\}\\ \Phi \in \mathbf{PrivSet} &= \mathcal{P}(\mathbf{Priv})\\ \Xi \in \mathbf{CPrivSet} &= \mathcal{P}(\mathbf{CPriv})\end{aligned}$$

We already understand privilege sets Φ, but we want a clear understanding of what consistent privilege sets Ξ—privilege sets that may have unknown effects—really mean. Consider the following two consistent privilege sets:

$$\Xi_1 = \{\texttt{read}\} \qquad \Xi_2 = \{\texttt{read}, ¿\}$$

The set Ξ_1 is completely static: it refers exactly to the set of privileges $\{\texttt{read}\}$. The set Ξ_2 on the other hand is gradual: it refers to the `read` privilege, but leaves open the possibility of other privileges. In this case, the ¿ stands for several possibilities: no additional privileges, the `write` privilege alone, the `alloc` privilege alone, or both `write` and `alloc`.

Thus, each consistent privilege set stands for some set of possible privilege sets. To formalize this interpretation, we introduce a *concretization* function γ, which maps a consistent privilege set Ξ to the concrete set of privilege sets that it stands for.[1]

Definition 5 (Concretization). *Let* $\gamma : \mathbf{CPrivSet} \to \mathcal{P}(\mathbf{PrivSet})$ *be defined as follows:*

$$\gamma(\Xi) = \begin{cases}\{\Xi\} & ¿ \notin \Xi\\ \{(\Xi \setminus \{¿\}) \cup \Phi \mid \Phi \in \mathbf{PrivSet}\} & \textit{otherwise}\end{cases} .$$

Reconsidering our two example consistent privilege sets, we find that

$$\begin{aligned}\gamma(\Xi_1) &= \{\{\texttt{read}\}\}\\ \gamma(\Xi_2) &= \left\{\begin{matrix}\{\texttt{read}, \texttt{write}\}, \{\texttt{read}, \texttt{alloc}\},\\ \{\texttt{read}\}, \{\texttt{read}, \texttt{alloc}, \texttt{write}\}\end{matrix}\right\}\end{aligned}$$

[1]We introduce an *abstraction* function α in section 4.1.4

Since each consistent privilege set stands for a number of possible concrete privilege sets, we say that a particular privilege set Φ is *represented* by a consistent privilege set Ξ if $\Phi \in \gamma(\Xi)$.

If we consider these two resulting sets of privilege sets, it is immediately clear that Ξ_1 is more restrictive about what privilege sets it represents (only one), while Ξ_2 subsumes Ξ_1 in that it also represents $\{\texttt{read}\}$, as well as some others. Thus, Ξ_1 is strictly more *precise* than Ξ_2, and so γ induces a *precision relation* between different consistent privilege sets.

Definition 6 (Precision)**.** Ξ_1 *is less imprecise (i.e. more precise) than* Ξ_2*, notation* $\Xi_1 \sqsubseteq \Xi_2$*, if and only if* $\gamma(\Xi_1) \subseteq \gamma(\Xi_2)$

Precision formalizes the idea that some consistent privilege sets imply more information about the privilege sets that they represent than others. For instance, $\{\texttt{read}\}$ is strictly more precise than $\{\texttt{read}, ¿\}$ because $\{\texttt{read}\} \sqsubseteq \{\texttt{read}, ¿\}$ but not vice-versa.

4.1.3 Lifting predicates to consistent privilege sets

Now that we have established a formal correspondence between consistent privilege sets and concrete privilege sets, we can systematically adapt our understanding of the latter to the former.

Recall the $\mathbf{check}_C$ predicates of the generic effect framework (section 2.5.1), which determine if a particular effect set fulfills the requirements of some effectful operator. Gradual checking implies that checking a consistent privilege set succeeds so long as checking its runtime representative could *plausibly* succeed. We formalize this as a notion of *consistent checking.*

Definition 7 (Consistent Checking)**.** *Let* $\mathbf{check}_C$ *be a predicate on privilege sets. Then we define a corresponding* consistent check *predicate* $\widetilde{\mathbf{check}}_C$ *on consistent privilege sets as follows:*

$$\widetilde{\mathbf{check}}_C(\Xi) \iff \mathbf{check}_C(\Phi) \text{ for some } \Phi \in \gamma(\Xi).$$

Under some circumstances, however, we must be sure that a consistent privilege set *definitely* has the necessary privileges to pass a check. For this purpose we introduce a notion of *strict checking.*

Definition 8 (Strict Checking)**.** *Let* $\mathbf{check}_C$ *be a predicate on privilege sets. Then we define a corresponding* strict check *predicate* $\textbf{\textit{strict-check}}_C$ *on consistent privilege sets as follows:*

$$\textbf{\textit{strict-check}}_C(\Xi) \iff \mathbf{check}_C(\Phi) \text{ for all } \Phi \in \gamma(\Xi).$$

By defining both consistent checking and strict checking in terms of representative sets, our formalizations are both intuitive and independent of the underlying $\mathbf{check}_C$ predicate. Furthermore, these definitions can be recast directly over consistent privilege sets once we settle on a particular $\mathbf{check}_C$ predicate (As an example, we may use the representation of the fluent language under the M&M framework introduced in appendix A).

4.1.4 Lifting functions to consistent privilege sets

In addition to predicates on consistent privilege sets, we must also define functions on them. For instance, the M&M framework is parameterized over a family of adjust functions $\mathbf{adjust}_A : \mathbf{PrivSet} \to \mathbf{PrivSet}$, which alter the set of available effect privileges (section 2.5.1). Using abstract interpretation, we lift these to *consistent* adjust functions $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \to \mathbf{CPrivSet}$. To do so we must first complete the abstract interpretation framework.

Consider our two example consistent privilege sets. Each represents some set of privilege sets, so we expect that adjusting a consistent privilege set should be related to adjusting the corresponding concrete privilege sets. The key insight is that adjusting a consistent privilege set should correspond somehow to adjusting each individual privilege set in its represented collection. For example $\widetilde{\mathbf{adjust}}_A(\{\texttt{read}, \texttt{alloc}\})$ should be related to the set $\{\mathbf{adjust}_A(\{\texttt{read}, \texttt{alloc}\})\}$, and $\widetilde{\mathbf{adjust}}_A(\{\texttt{read}, ¿\})$ should be related to the following set:

$$\left\{\begin{matrix} \mathbf{adjust}_A(\{\texttt{read}, \texttt{write}\}), \mathbf{adjust}_A(\{\texttt{read}, \texttt{alloc}\}), \\ \mathbf{adjust}_A(\{\texttt{read}\}), \mathbf{adjust}_A(\{\texttt{read}, \texttt{alloc}, \texttt{write}\}) \end{matrix}\right\}$$

To formalize these relationships, we need an *abstraction* function $\alpha : \mathcal{P}(\mathbf{PrivSet}) \to \mathbf{CPrivSet}$ that maps collections of privilege sets back to corresponding consistent privilege sets. For such a function to make sense, it must at least be *sound.*

Proposition 7 (Soundness)**.** $\Upsilon \subseteq \gamma(\alpha(\Upsilon))$ *for all* $\Upsilon \in \mathcal{P}(\mathbf{PrivSet})$.

Soundness implies that the corresponding consistent privilege set $\alpha(\Upsilon)$ represents at least as many privilege sets as the original collection Υ. A simple and sound definition of α is $\alpha(\Upsilon) = \{¿\}$. This definition is terrible, though, because it needlessly loses information. For instance, $\alpha(\gamma(\Xi_1)) = \{¿\}$, and since $\{¿\}$ represents every possible privilege set, that mapping loses all the information in the original set. At the least, we would like $\alpha(\gamma(\Xi_1)) = \Xi_1$.

Our actual definition of α is far better than the one proposed above:

Definition 9 (Abstraction)**.** *Let* $\alpha : \mathcal{P}(\mathbf{PrivSet}) \to \mathbf{CPrivSet}$ *be defined as follows*[2]*:*

$$\alpha(\Upsilon) = \begin{cases} \Phi & \Upsilon = \{\Phi\} \\ (\bigcap \Upsilon) \cup \{¿\} & \textit{otherwise.} \end{cases}$$

In words, abstraction preserves the common concrete privileges, and adds unknown privileges to the resulting consistent set if needed. As required, this abstraction function α is sound.

Even better though, given our interpretation of consistent privilege sets, this α is the best possible one.

Proposition 8 (Optimality)**.** *Suppose* $\Upsilon \subseteq \gamma(\Xi)$*. Then* $\alpha(\Upsilon) \sqsubseteq \Xi$*.*

Optimality ensures that α gives us not only a sound consistent privilege set, but also the

[2]For simplicity, we assume Υ is not empty, since $\alpha(\emptyset) = \bot$ plays no role in our development.

$$\phi \in \mathbf{Priv}, \quad \xi \in \mathbf{CPriv} = \mathbf{Priv} \cup \{¿\}$$
$$\Phi \in \mathbf{PrivSet} = \mathcal{P}(\mathbf{Priv}), \quad \Xi \in \mathbf{CPrivSet} = \mathcal{P}(\mathbf{CPriv})$$
$$\varepsilon \in \mathsf{Tags}\ .\ \pi \in \mathcal{P}(\mathsf{Tags})$$

$$\begin{array}{lcll}
w & ::= & \texttt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} \mid l & \text{Prevalues} \\
v & ::= & w_{\varepsilon} & \text{Values} \\
\mathrm{e} & ::= & x \mid v \mid \mathrm{e}\ \mathrm{e} \mid \mathrm{e} :: \Xi & \text{Terms} \\
 & & \mid (\texttt{ref}\ \mathrm{e})_{\varepsilon} \mid !\mathrm{e} \mid (\mathrm{e} := \mathrm{e})_{\varepsilon} & \\
T & ::= & \pi\ \rho & \text{Types} \\
\rho & ::= & \texttt{Unit} \mid T \xrightarrow{\Xi} T \mid \texttt{Ref}\ T & \text{PreTypes} \\
A & ::= & \downarrow\uparrow \mid \pi \downarrow \mid \texttt{ref}\ \downarrow \mid !\downarrow & \text{Adjust Contexts} \\
 & & \mid \downarrow := \uparrow \mid \pi := \downarrow & \\
C & ::= & \pi\ \pi \mid \texttt{ref}\ \pi \mid !\pi \mid \pi := \pi & \text{Check Contexts}
\end{array}$$

Figure 4.1: Syntax of the source language

most precise one[3]. In our particular case, optimality implies that $\alpha(\gamma(\Xi)) = \Xi$ for all Ξ but one: $\alpha(\gamma(\{\texttt{read}, \texttt{write}, \texttt{alloc}, ¿\})) = \{\texttt{read}, \texttt{write}, \texttt{alloc}\}$. Both consistent privilege sets represent the same thing.

Using α and γ, we can lift any function f on privilege sets to a function on consistent privilege sets. In particular, we lift the generic adjust functions:
Definition 10 (Consistent Adjust)**.**
Let $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \to \mathbf{CPrivSet}$ *be defined as follows:*

$$\widetilde{\mathbf{adjust}}_A(\Xi) = \alpha\left(\{\mathbf{adjust}_A(\Phi) \mid \Phi \in \gamma(\Xi)\}\right).$$

The $\widetilde{\mathbf{adjust}}$ function reflects all of the information that can be retained when conceptually adjusting all the sets represented by some consistent privilege set.

The $\widetilde{\mathbf{check}}$ and $\widetilde{\mathbf{adjust}}$ operators are critical to our generic presentation of gradual effects. Both definitions are independent of the underlying concrete definitions of **check** and **adjust**. As we show through the rest of the paper, in fact, the abstract interpretation framework presented here time and again provides a clear and effective way to conceive and formalize concepts that we need for gradual effect checking.

4.2 A generic framework for gradual effects

In this section we present a generic framework for gradual effect systems. As is standard for gradual checking, the framework includes a source language that supports unknown annota-

[3]Abstract interpretation literature expresses this in part by saying that α and γ form a *Galois connection*[9].

tions, an internal language that introduces runtime checks, and a type-directed translation from the former to the latter.

4.2.1 The source language

The core language (fig. 4.1) is a simply-typed functional language with a unit value, mutable state, and effect ascriptions $\mathrm{e} :: \Xi$. The language is parameterized on some finite set of effect privileges **Priv**, as well as a set of tags **Tag**. The **Priv** set is the basis for consistent privileges **CPriv**, privilege sets **PrivSet**, and consistent privilege sets **CPrivSet**. The **Tag** set is the basis for tag sets **TagSet**. Each type constructor is annotated with a tag set, so types are annotated deeply. Each value-creating expression is annotated with a tag so that effect systems can abstractly track values. The type of a function carries a consistent privilege set Ξ that characterizes the privileges required to execute the function body.

The source language also specifies a set of adjust contexts A and check contexts C, as presented in section 2.5.1. Each adjust context is determined by an evaluation context frame f (section 2.1.1). They index $\widetilde{\mathbf{adjust}}_A$ to determine how privileges are altered when evaluating in a particular context. Similarly, the check contexts correspond to program operations like function application. They index $\widetilde{\mathbf{check}}_C$ to determine which privileges are needed to perform the operation.

Most of the concepts here presented are inherited from the generic M&M framework [15], already introduced in section 2.5.1.

Figure 4.2 presents the type system. The judgment $\Xi; \Gamma; \Sigma \vdash \mathrm{e} : T$ means that the expression e has type T in the lexical environment Γ and store typing Σ, when provided with the privileges Ξ. Based on the judgment, e is free to perform any of the effectful operations denoted by the privileges in Ξ. If the consistent privilege set contains the unknown privileges ¿, then e might also try any other effectful operation, but at runtime a check for the necessary privileges is performed.

Each type rule extends the standard formulation with operations to account for effects. All notions of gradual checking are encapsulated in consistent effect sets Ξ and operations on them. The [T-Fn] rule associates some sufficient set of privileges with the body of the function. In practice we can deduce a minimal set to avoid spurious checks.

The [T-App] rule illustrates the structure of the non-value typing rules. It enhances the M&M typing rule for function application (introduced in section 2.5.1) to support gradual effects. In particular, each privilege check from the original rule is replaced with a *consistent* counterpart: consistent predicates succeed as long as the consistent privilege sets represent some plausible concrete privilege set, and consistent functions represent information about what is possible in their resulting consistent set. $\widetilde{\mathbf{adjust}}$ and $\widetilde{\mathbf{check}}$ are defined in section 4.1, and we use the same techniques introduced there to lift effect subtyping to a notion of *consistent subtyping*. To do so, we first lift traditional privilege set containment to *consistent containment*:

Definition 11 (Consistent Containment). Ξ_1 *is* consistently contained *in* Ξ_2, *notation* $\Xi_1 \,\widetilde{\sqsubseteq}$

$$\boxed{\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T} \qquad \text{T-Fn}\;\frac{\Xi_1; \Gamma, x \colon T_1; \Sigma \vdash \mathrm{e} \colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x \colon T_1 \;.\; \mathrm{e})_\varepsilon \colon \{\varepsilon\} T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{T-Unit}\;\frac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit}_\varepsilon \colon \{\varepsilon\}\mathtt{Unit}} \qquad \text{T-Loc}\;\frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l_\varepsilon \colon \{\varepsilon\}\mathtt{Ref}\ T}$$

$$\text{T-Var}\;\frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \colon T} \qquad \text{T-App}\;\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon \pi_2\rho_2 \\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \qquad \widetilde{\mathbf{check}}_{\pi_1\pi_2}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3}$$

$$\text{T-Eff}\;\frac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \colon T \qquad \Xi_1 \underset{\sim}{\sqsubseteq} \Xi}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e} :: \Xi_1) \colon T} \qquad \text{T-Ref}\;\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \colon \pi\rho \\ \widetilde{\mathbf{check}}_{\mathtt{ref}\ \pi}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_\varepsilon \colon \{\varepsilon\}\mathtt{Ref}\ \pi\rho}$$

$$\text{T-Deref}\;\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \colon \pi\mathtt{Ref}\ T \\ \widetilde{\mathbf{check}}_{!\pi}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \colon T} \qquad \text{T-Asgn}\;\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon \pi_2\rho_2 \\ \widetilde{\mathbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1\end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \colon \{\varepsilon\}\mathtt{Unit}}$$

Figure 4.2: Type system for the source language

Ξ_2 *if and only if* $\Phi_1 \subseteq \Phi_2$ *for* some $\Phi_1 \in \gamma(\Xi_1)$ *and* $\Phi_2 \in \gamma(\Xi_2)$[4].

Consistent containment means that privilege set containment may hold unless we guarantee that it cannot. Of course, this claim must sometimes be protected with a runtime check in the internal language, as discussed further in the next section. Consistent subtyping $\lesssim$ is defined by replacing the privilege subset premise of traditional effect subtyping with consistent containment.

$$\frac{\pi_1 \subseteq \pi_2}{\pi_1\rho \lesssim \pi_2\rho} \qquad \frac{\begin{array}{cc} T_3 \lesssim T_1 & T_2 \lesssim T_4 \\ \pi_1 \subseteq \pi_2 & \Xi_1 \underset{\sim}{\sqsubseteq} \Xi_2 \end{array}}{\pi_1 T_1 \xrightarrow{\Xi_1} T_2 \lesssim \pi_2 T_3 \xrightarrow{\Xi_2} T_4}$$

This relation expresses plausible substitutability. Consistent containment is not transitive, and as a result neither is consistent subtyping. This property is directly analogous to consistent subtyping for gradual object systems [23].

All other rules in the type system can be characterized as consistent liftings of the corresponding M&M rules. Each uses $\mathbf{adjust}_A$ to type subexpressions, and $\mathbf{check}_C$ to check privileges.

Finally, [T-Eff] reflects the consistent counterpart of static effect ascriptions, which do not appear in the M&M system. The rule requires that the ascribed consistent privileges be

[4]We give $\underset{\sim}{\sqsubseteq}$ a simple direct characterization in section 4.2.2.

$$\begin{array}{llll}
\mathrm{e} & ::= & \ldots \mid \mathsf{Error} \mid \langle T \Leftarrow T\rangle \mathrm{e} & \text{Terms} \\
 & & \mid \texttt{has}\ \Phi\ \mathrm{e} \mid \texttt{restrict}\ \Xi\ \mathrm{e} & \\
f & ::= & \square\ \mathrm{e} \mid v\ \square \mid (\texttt{ref}\ \square)_\varepsilon & \text{Frames} \\
 & & \mid !\square \mid (\square := \mathrm{e})_\varepsilon \mid (w_\varepsilon := \square)_\varepsilon & \\
g & ::= & f \mid \langle T_2 \Leftarrow T_1\rangle \square \mid \texttt{has}\ \Phi\ \square & \text{Error Frames} \\
 & & \mid \texttt{restrict}\ \Xi\ \square &
\end{array}$$

Figure 4.3: Syntax of the internal language

consistently contained in the current consistent privileges. Ascribing ¿ delays some privilege checks to runtime, as discussed next.

4.2.2 The internal language

The semantics of the source language is given by a type-directed translation to an internal language that makes runtime checks explicit. This section presents the internal language. The translation is presented in section 4.2.3.

fig. 4.3 presents the syntax of the internal language. It extends the source language with explicit features for managing runtime effect checks. The Error construct indicates that a runtime effect check failed, and aborts the rest of the computation. Casts $\langle T \Leftarrow T\rangle$e express type coercions between consistent types. The `has` operation checks for the availability of particular effect privileges at runtime. The `restrict` operation restricts the privileges available while evaluating its subexpression.

Frames represent evaluation contexts in our small-step semantics. By using frames, we present a system with structural semantics like the M&M framework while defining fewer evaluation rules as in a reduction semantics.

$\boxed{\Xi;\Gamma;\Sigma \vdash \mathrm{e} \colon T}$

$$\text{IT-App}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \colon \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \colon \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi) \qquad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <\colon \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3 \end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3}$$

$$\text{IT-Cast}\ \frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e} \colon T_0 \qquad T_0 <\colon T_1 \qquad T_1 \lesssim T_2}{\Xi;\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle \mathrm{e} \colon T_2} \qquad \text{IT-Has}\ \frac{(\Phi \cup \Xi);\Gamma;\Sigma \vdash \mathrm{e} \colon T}{\Xi;\Gamma;\Sigma \vdash \texttt{has}\ \Phi\ \mathrm{e} \colon T}$$

$$\text{IT-Error}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error} \colon T} \qquad \text{IT-Rst}\ \frac{\Xi_1;\Gamma;\Sigma \vdash \mathrm{e} \colon T \qquad \Xi_1 \leq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e} \colon T}$$

$$\text{IT-Ref}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \colon \pi\rho \\ \textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ \mathrm{e})_{\varepsilon} \colon \{\varepsilon\}\texttt{Ref}\ \pi\rho} \qquad \text{IT-Deref}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \colon \pi\texttt{Ref}\ T \\ \textbf{\textit{strict-check}}_{!\pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !\mathrm{e} \colon T}$$

$$\text{IT-Asgn}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \colon \pi_1\texttt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \colon \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <\colon T_1 \end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_{\varepsilon} \colon \{\varepsilon\}\texttt{Unit}}$$

Figure 4.4: Typing rules for the internal language

$$\text{E-Ref}\ \frac{\mathbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi) \qquad l \notin \operatorname{dom}(\mu)}{\Phi \vdash (\texttt{ref}\ w_{\varepsilon_1})_{\varepsilon_2} \mid \mu \to l_{\varepsilon_2} \mid \mu[l \mapsto w_{\varepsilon_1}]}$$

$$\text{E-Asgn}\ \frac{\mathbf{check}_{\{\varepsilon_1\}:=\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (l_{\varepsilon_1} := w_{\varepsilon_2})_{\varepsilon} \mid \mu \to \texttt{unit}_{\varepsilon} \mid \mu[l \mapsto w_{\varepsilon_2}]}$$

$$\text{E-Deref}\ \frac{\mathbf{check}_{!\{\varepsilon\}}(\Phi) \qquad \mu(l) = v}{\Phi \vdash !l_{\varepsilon} \mid \mu \to v \mid \mu}$$

$$\text{E-Frame}\ \frac{\mathbf{adjust}_{A(f)}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash f[\mathrm{e}] \mid \mu \to f[\mathrm{e}'] \mid \mu'}$$

$$\text{E-Error}\ \frac{}{\Phi \vdash g[\mathsf{Error}] \mid \mu \to \mathsf{Error} \mid \mu}$$

E-Has-T
$$\frac{\Phi' \subseteq \Phi \qquad \Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \texttt{has}\ \Phi'\ \mathrm{e}' \mid \mu'}$$

E-Has-V
$$\frac{}{\Phi \vdash \texttt{has}\ \Phi'\ v \mid \mu \to v \mid \mu}$$

E-Has-F
$$\frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \mathsf{Error} \mid \mu}$$

$$\text{E-Rst-V}\ \frac{}{\Phi \vdash \texttt{restrict}\ \Xi\ v \mid \mu \to v \mid \mu}$$

$$\text{E-Rst}\ \frac{\Phi'' = \max\{\Phi' \in \gamma(\Xi) \mid \Phi' \subseteq \Phi\} \qquad \Phi'' \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \to \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'}$$

E-App
$$\frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (\lambda x\colon T_1\ .\ \mathrm{e})_{\varepsilon_1}\ w_{\varepsilon_2} \mid \mu \to [w_{\varepsilon_2}/x]\,\mathrm{e} \mid \mu}$$

E-Cast-Frame
$$\frac{\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \mid \mu \to \langle T_2 \Leftarrow T_1\rangle \mathrm{e}' \mid \mu'}$$

E-Cast-Id
$$\frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2\rho \Leftarrow \pi_1\rho\rangle w_{\varepsilon} \mid \mu \to w_{\varepsilon} \mid \mu}$$

E-Cast-Fn
$$\frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2 T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow \pi_1 T_{11} \xrightarrow{\Xi_1} T_{12}\rangle\,(\lambda x\colon T_{11}\ .\ \mathrm{e})_{\varepsilon} \mid \mu \to (\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ \ (|\Xi_1| \setminus |\Xi_2|)\ \ [(\langle T_{11} \Leftarrow T_{21}\rangle x)/x]\,\mathrm{e})_{\varepsilon} \mid \mu}$$

Figure 4.5: Small-step semantics of the internal language

Static semantics The type system of the internal language (fig. 4.4) mostly extends the surface language type system, with a few critical differences. First, recall that type rules for source language operators, like function application [T-App], verify effects based on *consistent* checking: so long as some representative privilege set is checkable, the expression is accepted. In contrast, the internal language introduces new typing rules for these operators, like [IT-App] (changes highlighted in gray).

In the internal language, effectful operations *must* have enough privileges to be performed: plausibility is not sufficient anymore. As we see in the next section, consistent checks from source programs are either resolved statically or rely on runtime privilege $\widetilde{\text{checks}}$ to guarantee satisfaction before reaching an effectful operation. For this reason, uses of **check** are replaced with ***strict-check*** (section 4.1.3, definition 8). Consistent subtyping $\lesssim$ is replaced with a notion of subtyping $<:$ that is based on ordinary set containment for consistent privilege sets and tags:

$$\frac{\pi_1 \subseteq \pi_2}{\pi_1\rho <: \pi_2\rho} \qquad\qquad \frac{\begin{array}{cc} T_3 <: T_1 & T_2 <: T_4 \\ \pi_1 \subseteq \pi_2 & \Xi_1 \subseteq \Xi_2 \end{array}}{\pi_1 T_1 \xrightarrow{\Xi_1} T_2 <: \pi_2 T_3 \xrightarrow{\Xi_2} T_4}$$

The intuition is that an expression that can be typed with a given set of consistent permissions should still be typable if additional permissions become available. We formalize this intuition below.

In addition to ordinary set containment, the internal language depends on a stronger notion of containment that focuses on statically known permissions. A consistent privilege set represents some number of concrete privilege sets, each containing some different privileges, but most consistent privilege sets have some reliable information. For instance, any set represented by $\Xi = \{\texttt{read}, ?\}$ may have a variety of privileges, but any such set will surely contain the `read` privilege. We formalize this idea in terms of concretization as the *static part* of a consistent privilege set.
Definition 12 (Static Part). *The* static part *of a consistent privilege set,* $|\cdot| :$ **CPrivSet** $\rightarrow$ **PrivSet** *is defined as*

$$|\Xi| = \bigcap \gamma(\Xi).$$

The definition directly embodies the intuition of "all reliable information," but this operation also has a simple direct characterization: $|\Xi| = \Xi \setminus \{¿\}$.[5]

Using the notion of static part, we define the concept of *static containment* for consistent privilege sets.
Definition 13 (Static Containment). Ξ_1 *is* statically contained *in* Ξ_2*, notation* $\Xi_1 \leq \Xi_2$*, if and only if* $|\Xi_1| \subseteq |\Xi_2|$.

The intuition behind static containment is that an expression can be safely used in any context that is guaranteed to provide at least its statically-known privilege requirements.

We need static containment to help us characterize effect subsumption in the internal language. Privilege subsumption says that if Φ is sufficient to type e, then so can any larger

[5]The γ-based definition is useful for proving Strong Effect Subsumption (proposition 11 below).

set Φ' [26]. To establish this, we must consider properties of both ***strict-check*** and $\widetilde{\textbf{adjust}}$. Conveniently, ***strict-check*** is monotonic with respect to consistent privilege set containment.

Lemma 9.
If $\textit{\textbf{strict-check}}_C(\Xi_1)$ *and* $\Xi_1 \subseteq \Xi_2$ *then* $\textit{\textbf{strict-check}}_C(\Xi_2)$*.*

To the contrary, though, $\widetilde{\textbf{adjust}}$ is not monotonic with respect to set containment on consistent privilege sets. Instead, it *is* monotonic with respect to static containment.
Lemma 10. *If* $\Xi_1 \leq \Xi_2$ *then* $\widetilde{\textbf{adjust}}_C(\Xi_1) \leq \widetilde{\textbf{adjust}}_C(\Xi_2)$

We exploit this to establish effect subsumption.
Proposition 11 (Strong Effect Subsumption)**.**
If $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T$ *and* $\Xi_1 \leq \Xi_2$*, then* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T$*.*

Proof. By induction over the typing derivations $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T$. □

Corollary 12 (Effect Subsumption)**.**
If $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T$ *and* $\Xi_1 \subseteq \Xi_2$*, then* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T$*.*

Proof. Set containment implies static containment. □

We now turn to the new syntactic forms of the internal language. Casts represent explicit dynamic checks for consistent subtyping relationships. The `has` operator checks if the privileges in Φ are currently available. Its subexpression e is typed using the consistent set that is extended statically with Φ.[6]

The `restrict` operator constrains its subexpression to be typable in a consistent privilege set that is statically-contained in the current set. Since ¿ does not play a role in static containment, the set Ξ_1 can introduce dynamism that was not present in Ξ. As we will see when we translate source programs, this is key to how ascription can introduce more dynamism into a program.

As it happens, we can use notions from this section to simply characterize notions that we, for reasons of conceptual clarity, defined using the concretization function and collections of plausible privilege sets. The concretization-based definitions clearly formalize our intentions, but these new extensionally equivalent characterizations are well suited to efficient implementation.

First, we can characterize consistent containment as an extension of static containment, and strict checking as simply checking the statically known part of a consistent privilege set.

Proposition 13.

1. $\Xi_1 \mathrel{\widetilde{\sqsubseteq}} \Xi_2$ *if and only if* $\Xi_1 \leq \Xi_2$ *or* ¿ $\in \Xi_2$*.*

[6]Note that $\Phi \cup \Xi$ is the same as lifting the function $f(\Phi') = \Phi \cup \Phi'$, and $\Phi \mathrel{\widetilde{\sqsubseteq}} \Xi$ is the same as lifting the predicate $P(\Phi') = \Phi \subseteq \Phi'$.

2. *$\textbf{\textit{strict-check}}_C(\Xi)$ if and only if $\mathbf{check}_C(|\Xi|)$.*

Furthermore, we can characterize consistent checking based on whether the consistent privilege set in question contains unknown privileges.
Proposition 14.

1. *If $¿ \in \Xi$ then $\widetilde{\mathbf{check}}_C(\Xi)$ if and only if $\mathbf{check}_C(\mathbf{PrivSet})$.*
2. *If $¿ \notin \Xi$ then $\widetilde{\mathbf{check}}_C(\Xi)$ if and only if $\mathbf{check}_C(\Xi)$.*

Dynamic semantics Figure 4.5 presents the evaluation rules of the internal language. The judgment $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ means that under the privilege set Φ and store μ, the expression e takes a step to e' and μ'. Effectful constructs consult Φ to determine whether they have sufficient privileges to proceed.

The `has` expression checks dynamically for privileges. If the privileges in Φ are available, then execution may proceed: if not, then an Error is thrown. Note that in a real implementation, `has` only needs to check for privileges once: the semantics keeps `has` around only to support our type safety proof.

The `restrict` expression restricts the privileges available in the dynamic extent of the current subexpression. The intuition is as follows. Ξ represents any number of privilege sets. At least one of those sets must be contained in Φ or the program gets stuck: `restrict` cannot add new privileges. So `restrict` limits its subexpression to the largest subset of currently available privileges that Ξ can represent. In practice, this means that if Ξ is fully static, then Ξ represents only one subset Φ' of Φ and the subexpression can only use those privileges. If $¿ \in \Xi$, then Ξ can represent *all* of Φ, so the privilege set is not restricted at all. This property of `restrict` enables ascription to support dynamic privileges.

Since function application is controlled under some effect disciplines, the [E-App] rule is guarded by the $\mathbf{check}_{\mathsf{app}}$ predicate inherited from the M&M framework. If this check fails, then the program is stuck. More generally, any effectful operation added to the framework is guarded by such a check. These checks are needed to give intensional meaning to our type safety theorem: if programs never get stuck, then any effectful operation that is encountered must have the proper privileges to run. This implies that either the permissions were statically inferred by the type checker, or the operation is guarded by a `has` expression, which throws an Error if needed privileges are not available. It also means that thanks to type safety, an actual implementation would not need *any* of the $\mathbf{check}_C$ checks: the `has` checks suffice. This supports the pay-as-you-go principle of gradual checking.

Higher-order casts incrementally verify at runtime that consistent subtyping really implies privilege set containment. In particular they guard function calls. First, they restrict the set of available privileges to detect privilege inconsistencies in the function body. Then, they check the resulting privilege set for the minimal privileges needed to validate the containment relationship. Intuitively, we only need to check for the statically determined permissions that are not already accounted for.

To illustrate, consider the following example:$\{\mathtt{read}, \mathtt{alloc}\} \mathrel{\widetilde{\sqsubseteq}} \{\mathtt{read}, ¿\}$ because `alloc`

could be in a representative of $\{\texttt{read}, ¿\}$, but $\{\texttt{read}, \texttt{alloc}\} \not\sqsubseteq \{\texttt{read}, ¿\}$ since that is not definitely true. Thus, to be sure at runtime, we must check for
$|\{\texttt{read}, \texttt{alloc}\}| \setminus \ |\{\texttt{read}, ¿\}| = \{\texttt{alloc}\}$. Note that the rule [E-Cast-Fn] uses the standard approach to higher-order casts due to Findler and Felleisen [10]. As a formalization convenience, the rule uses substitution directly rather than function application so as to protect the implementation internals from effect checks and adjustments. In practice the internal language would simply use function application without checking or adjusting privileges.

Type safety We prove type safety in the style of Wright and Felleisen [28]. Program execution begins with a closed term e as well as an initial privilege set Φ. The initial program must be well typed and the privilege set must be represented by the consistent privilege set Ξ used to type the program. Under these conditions, the program will not get stuck.

Our statements of Progress and Preservation introduce the representation restrictions between consistent privilege sets and the privilege sets used as contexts for evaluation. These restrictions can be summarized in that typing ensures that evaluation does not get stuck in any particular context represented statically.[7]

Theorem 15 (Progress)**.** *Suppose* $\Xi; \emptyset; \Sigma \vdash \mathrm{e} \colon T$*. Then either* e *is a value* v*, an* Error*, or* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for all privilege sets* Φ *such that* $\exists \Phi' \in \gamma(\Xi)$ *such that* $\Phi' \subseteq \Phi$ *and for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$*.*

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash \mathrm{e} \colon T$. □

Theorem 16 (Preservation)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T$*, and* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for* $\Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$*, then* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}' \colon T'$ *for some* $T' <: T$ *and* $\exists \Sigma' \supseteq \Sigma$*.*

Proof. By structural induction over the typing derivation. Preservation of types under substitution for values (required for [E-App]) and for identifiers (required for [E-Cast-Fn]) follows as a standard proof since neither performs effects. □

4.2.3 Translating source programs to the internal language

Figure 4.6 presents the type-directed translation of source programs to the internal language (the interesting parts have been highlighted). The translation uses static type and effect information from the source program to determine where runtime checks are needed in the corresponding internal language program. In particular, any consistent check, containment, or subtyping that is not also a strict check, static containment, or static subtyping, respectively, must be guarded by a `has` expression (for checks and containments) or a cast (for subtypings).

Recall from section 4.2.2 that the `has` expression checks if some particular privileges are available at runtime. The translation system determines for each program point which privileges (if any) must be checked. Since the generic framework imposes only privilege and

[7]We also proved soundness for a minimal system with neither tags nor state.

$\boxed{\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e} \colon T}$

$$\text{C-Fn}\;\dfrac{\Xi_1; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x\colon T_1 \;.\; \mathrm{e})_\varepsilon \Rightarrow (\lambda x\colon T_1 \;.\; \mathrm{e}')_\varepsilon \colon \{\varepsilon\} T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{C-Unit}\;\dfrac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit}_\varepsilon \Rightarrow \mathtt{unit}_\varepsilon \colon \{\varepsilon\}\mathtt{Unit}} \qquad \text{C-Var}\;\dfrac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \Rightarrow x\colon T}$$

$$\text{C-Loc}\;\dfrac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l_\varepsilon \Rightarrow l_\varepsilon \colon \{\varepsilon\}\mathtt{Ref}\ T} \qquad \text{C-App}\;\dfrac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon \pi_2\rho_2 \\ \mathrm{e}''_1 = (\,\langle\!\langle \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \rangle\!\rangle\ \mathrm{e}'_1) \\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \\ \widetilde{\mathbf{check}}_{\pi_1\pi_2}(\Xi) \qquad\qquad \Phi = \Delta_{\pi_1\pi_2}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) \colon T_3}$$

$$\text{C-Eff}\;\dfrac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \qquad \Xi_1 \underset{\sim}{\sqsubset} \Xi \qquad \Phi = (|\Xi_1| \setminus |\Xi|)}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e} :: \Xi_1) \Rightarrow \textit{insert-has?}(\Phi, \mathtt{restrict}\ \Xi_1\ \mathrm{e}') \colon T}$$

$$\text{C-Ref}\;\dfrac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \pi\rho \qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\ \pi}(\Xi) \qquad \Phi = \Delta_{\mathtt{ref}\ \pi}(\Xi)}{\Xi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, \left(\mathtt{ref}\ \mathrm{e}'\right)_\varepsilon) \colon \{\varepsilon\}\mathtt{Ref}\ \pi\rho}$$

$$\text{C-Deref}\;\dfrac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \pi\mathtt{Ref}\ T \\ \widetilde{\mathbf{check}}_{!\pi}(\Xi) \qquad\qquad \Phi = \Delta_{!\pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \Rightarrow \textit{insert-has?}(\Phi, !\mathrm{e}') \colon T}$$

$$\text{C-Asgn}\;\dfrac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon \pi_2\rho_2 \qquad \widetilde{\mathbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1 \qquad \Phi = \Delta_{\pi_1:=\pi_2}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, \left(\mathrm{e}'_1 := \mathrm{e}'_2\right)_\varepsilon) \colon \{\varepsilon\}\mathtt{Unit}}$$

Figure 4.6: Translation of source programs to the internal language

tag monotonicity restrictions on the **check** and **adjust** functions, deducing these checks can be subtle.

Consider a hypothetical check predicate for a mutable state effect discipline:

$$\mathbf{check}_C(\Phi) \iff \texttt{read} \in \Phi \text{ or } \texttt{write} \in \Phi.$$

Though strange here, an effect discipline that is satisfied by one of two possible privileges is generally plausible, and in fact satisfies the monotonicity restrictions. When, say, the consistent check $\widetilde{\mathbf{check}}_C(\{¿\})$ succeeds in some program, which privileges should be checked at runtime?

The key insight is that the internal language program must check for all privileges that can produce a minimal satisfying privilege set. In the case of the above example, we must conservatively check for *both* `read` and `write`. However, we do not need to check for any privileges that are already known to be statically available.

We formalize this general idea as follows. First, since we do not want to require and check for any more permissions than needed, we only consider all possible *minimal* privilege sets that satisfy the check. We isolate the minimal privilege sets using the *mins* function:

$$\mathit{mins}(\Upsilon) = \{\Phi \in \Upsilon \mid \forall \Phi' \in \Upsilon . \Phi' \not\subset \Phi\}.$$

Given some consistent privilege set Ξ, we identify all of its plausible privilege sets that satisfy a particular check, and select only the minimal ones. In many cases there is a unique minimal set, but as above, there may not.[8] To finish, we coalesce this collection of minimal privileges, and remove any that are already statically known to be available based on Ξ. These steps are combined in the following function.

Definition 14 (Minimal Privilege Check). *Let C be some checking context. Then define $\Delta_C : \mathbf{CPrivSet} \to \mathbf{PrivSet}$ as follows:*

$$\Delta_C(\Xi) = \left(\bigcup \mathit{mins}(\{\Phi \in \gamma(\Xi) \mid \mathbf{check}_C(\Phi)\})\right) \setminus |\Xi|$$

The Δ_C function transforms a given consistent privilege set into the minimal conservative set of additional privileges needed to safely pass the $\mathbf{check}_C$ function. For instance, the [C-App] translation rule uses it to guard a function application, if need be, with a runtime privilege check. These checks are introduced by the *insert-has?* metafunction.

$$\mathit{insert\text{-}has?}(\Phi, \mathrm{e}) = \begin{cases} \mathrm{e} & \text{if } \Phi = \emptyset \\ \texttt{has}\ \Phi\ \mathrm{e} & \text{otherwise} \end{cases}$$

Note that the metafunction only inserts a check if needed. This supports the pay-as-you-go principle of gradual checking.

[8]One could retain precision by extending our abstraction to support *disjunctions* of consistent effect sets, at the cost of increased complexity in the translation and type system.

Since [C-App] also appeals to consistent subtyping, a cast may be introduced in the translation as well. For this, we appeal to a cast insertion metafunction:

$$\langle\!\langle T_2 \Leftarrow T_1 \rangle\!\rangle \mathrm{e} = \begin{cases} \mathrm{e} & \text{if } T_1 <: T_2 \\ \langle T_2 \Leftarrow T_1 \rangle \mathrm{e} & \text{otherwise.} \end{cases}$$

Once again, casts are only inserted when static subtyping does not already hold.

The [C-Eff] rule translates effect ascription in the source language to the `restrict` form in the internal language. If more privileges are needed to ensure static containment between Ξ_1 and Ξ, then translation inserts a runtime `has` check to bridge the gap.[9]

Crucially, the translation system preserves typing.

Theorem 17 (Translation preserves typing). *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' : T$ *in the source language then* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}' : T$ *in the internal language.*

Proof. By structural induction over the translation derivation rules. The proof relies on the fact that $\Delta_C(\Xi)$ introduces enough runtime checks (via *insert-has?*) that any related ***strict-check***$_C(\Xi)$ predicate is sure to succeed at runtime, so those rules do not get stuck. The instance of *insert-has?* in the [C-Eff] rule plays the same role there. □

4.3 Summary

The ideas presented in this chapter have already been published in the International Conference on Functional Programming [2]. In section 4.1, we modelled the ideas of gradual effect checking already introduced in the previous chapter in terms of abstract interpretation. Using this model, we introduced in section 4.2 an extension to the generic type-and-effect framework of Marino and Millstein [15] that provides gradual effect checking without imposing any further restrictions to the framework expressivity.

The original framework introduced by Marino and Millstein required some runtime information only to prove type safety, and that information could be safely avoided in an implementation. Implementing the semantics presented in this chapter cannot avoid the runtime effect information, because it is required to provide dynamic effect checking. In the next chapter we detail which information is now not redundant, and present an alternative runtime semantics that requires reduced runtime information. Both semantics are type safe, and we describe the tradeoffs between them.

[9]The formula for Φ is analogous to the Δ_C operation for $\mathbf{check}_C$.

Chapter 5

A Conservative Semantics with Reduced Runtime Information

In the generic M&M framework [15], both tags and all the privilege-related information are redundant at execution time. The static type system ensures that every runtime check will pass, thus runtime tag and privilege information is only included in the operational semantics to ensure type safety. An implementation of the runtime semantics does not need to carry any tag or privilege information.

In the generic gradual effect framework, static information might not be sufficient to ensure that all runtime verifications succeed. When the type system does not have enough information, the translation algorithm wraps a `has` expression around the original expression. This `has` expression checks for sufficient privileges, and triggers an error if the check fails. The `has` construct depends on the set of privileges Φ used as a context, thus for `has` to work the privilege context cannot be discarded and has to be available at runtime.

To calculate the appropriate privilege sets useds as contexts, the semantics we have introduced so far requires to hold tag information at runtime. Tag information imposes a space overhead for evaluation. In this chapter, we explore a way to give implementors the option of not requiring runtime tags, while understanding the expressiveness impact of the technique. We propose an alternative semantics for the generic gradual effect framework that lifts the dependency on runtime tag information, only depending on privilege set Φ context information at runtime. We call this new semantics a *conservative semantics*.

We also formalize the relation between the conservative semantics and the semantics proposed in chapter 4. Programs that reduce to a value in the conservative semantics reduce to the very same value (modulo tag information) in the semantics for the generic gradual effect checking system introduced in fig. 4.5. Unfortunately, the price to pay for having less information is precision: some programs that produced a result in the generic semantics will instead trigger an error in the conservative semantics.

$$
\begin{array}{lcll}
w & ::= & \texttt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} \mid l & \text{Prevalues} \\
v & ::= & w_{\varepsilon} & \text{Values} \\
\mathrm{e} & ::= & x \mid v \mid (\mathrm{e}\ \mathrm{e})_{\pi} \mid \mathsf{Error} \mid \langle T \Leftarrow T \rangle \mathrm{e} \mid (\texttt{ref}\ \mathrm{e})_{\varepsilon} \mid !\mathrm{e} \mid (\mathrm{e} := \mathrm{e})_{(\varepsilon,\pi)} & \text{Terms} \\
 & & \mid \texttt{has}\ \Phi\ \mathrm{e} \mid \texttt{restrict}\ \Xi\ \mathrm{e} & \\
T & ::= & \pi\rho & \text{Types} \\
\rho & ::= & \texttt{Unit} \mid T \xrightarrow{\Xi} T \mid \texttt{Ref}\ T & \text{PreTypes} \\
A & ::= & \downarrow\uparrow \mid \pi \downarrow \mid \texttt{ref}\ \downarrow \mid !\downarrow & \text{Adjust Contexts} \\
 & & \mid \downarrow:=\uparrow \mid \ \pi :=\downarrow & \\
C & ::= & \pi\ \pi \mid \texttt{ref}\pi \ \mid !\pi \ \mid \ \pi := \pi & \text{Check Contexts} \\
f & ::= & (\square\ \mathrm{e})_{\pi} \mid (v\ \square)_{\pi} & \text{Frames} \\
 & & (\texttt{ref}\ \square)_{\varepsilon} \mid !\square \mid (\square := \mathrm{e})_{(\varepsilon,\pi)} \mid (v := \square)_{(\varepsilon,\pi)} & \\
g & ::= & f \mid \langle T_2 \Leftarrow T_1 \rangle \square \mid \texttt{has}\ \Phi\ \square \mid \texttt{restrict}\ \Xi\ \square & \text{Error Frames}
\end{array}
$$

Figure 5.1: Conservative Language Syntax

5.1 Making tag information redundant at runtime

In the generic operational semantics introduced in chapter 4, tag information is used only to compute privilege sets through **adjust** functions and to verify **check** conditions. We will first clarify why **check** predicates are redundant, so we can later focus on the usages of **adjust** and study how to avoid requiring tag information.

check predicates in the operational semantics are made redundant by the type system. In the intermediate language presented for the generic gradual effect framework, every call to **check** in the operational semantics uses a check context limited to the tags of the values in the expression to be reduced. For example, to reduce a $(\texttt{ref unit}_{\varepsilon_1})_{\varepsilon_2}$ expression under a privilege context Φ, rule [E-Ref] verifies that $\mathbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi)$ holds. At the same time, typing an expression that reduces to $(\texttt{ref unit}_{\varepsilon_1})_{\varepsilon_2}$ requires a matching $\boldsymbol{\mathit{strict\text{-}check}}_{\texttt{ref}\ \pi}(\Xi)$ predicate to hold, which may use a check context with either more tags or the same tags required in the reduction rules. By the tag monotonicity lemma (property 2), $\mathbf{check}_{C}(\Phi)$ implies $\mathbf{check}_{C'}(\Phi)$ if C' contains less tags than C ($C' \sqsubseteq C$). At the same time, a call to $\boldsymbol{\mathit{strict\text{-}check}}_{C}(\Xi)$ always ensures that for every feasible privilege set $\Phi \in \gamma(\Xi)$, $\mathbf{check}_{C}(\Phi)$ holds. Thus by statically requiring a $\boldsymbol{\mathit{strict\text{-}check}}_{\texttt{ref}\ \pi}(\Xi)$ predicate in the type system with $\varepsilon_1 \in \pi$, type safety (theorem 20 and theorem 21) ensures that every **check** predicate in the operational semantics always holds, and therefore checking $\mathbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi)$ at runtime is redundant.

We can now analyze in which cases **adjust** requires tag information. The only interesting case is for evaluation rule [E-Frame], which alters the set of available privileges by using **adjust** functions. **adjust** only use tag information through adjust contexts. An adjust context holds tag information only for expressions composed of multiple subexpressions, where tag information represents previously reduced subexpressions. Tag information available in the adjust context may be then used to alter the privileges available to evaluate the next subexpression. This situation only arises for two forms of adjust contexts in our semantics: $\pi\downarrow$ and $\pi :=\downarrow$in rule [E-Frame], corresponding to expressions that use rules [T-App] and [T-Asgn] in the typing relation. The generic gradual effect system introduced in chapter 4 uses a tag approximation in the type system, using the tagset π obtained when typing e_1 in expressions of the form $\mathrm{e}_1\ \mathrm{e}_2$ and $\mathrm{e}_1 := \mathrm{e}_2$ to generate the respective adjust contexts $\pi\downarrow$ and $\pi :=\downarrow$ used to type e_2 in rules [T-App] and [T-Asgn], but uses exact tags provided by tag annotations of values in the operational semantics. By type safety, the exact tag ε used in the operational semantics is guaranteed to be a member of the set π used in the type derivation. With this restriction, the tag monotonicity lemmas of the generic framework ensure that the privilege information available at runtime will always be equal or greater than the privilege information used by the type system.

In section 5.1.1 we propose a language transformation that does not require runtime tags on values, using the available static information instead. This comes at the price that some programs that the generic semantics in fig. 4.5 accepted will now be rejected, because of the tag monotonicity lemmas. An example of a program rejected by the conservative semantics is shown in section 5.1.2.

5.1.1 The conservative semantics

Syntax for the conservative language is introduced in fig. 5.1, which highlights the interesting differences with the language from chapter 4. The conservative language carries syntactic tag information for the only cases where tag information was needed: evaluation of frames $(v\ \square)$ and $\pi := \square$, which induced the adjust contexts $\pi\downarrow$ and $\pi :=\downarrow$, respectively. Thus function applications $(\mathrm{e}_1\ \mathrm{e}_2)$ and assignments $((\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon)$ now also carry a tagset π that will be used by **adjust** instead of the tag obtained from values. This tagset will be inserted by a translation algorithm (in fig. 5.3) that annotates these expressions with the tagset used when typing the expression.

The operational semantics is introduced in fig. 5.4. Only 3 rules are different: rules [E-App], [E-Asgn] and [E-Frame]. Rules [E-App] and [E-Asgn] do not change their semantics, but they now operate on expressions with extra syntactic information. This extra tagset information is no longer required after rules [E-App] or [E-Asgn] are applied, so those rules discard the information.

Rule [E-Frame] changes in how the adjust context for a particular frame is obtained. In the generic system introduced in fig. 4.5, rule [E-Frame] uses an A function to infer the proper adjust context for an evaluation frame. In the case for frames $v\ \square$ and $(v := \square)_{\varepsilon'}$, it used the tag in v to create a singleton tagset and produce the corresponding $\{\varepsilon\}\downarrow$ or $\{\varepsilon\} :=\downarrow$ context. In the conservative semantics, we introduce a new function A' shown in fig. 5.2. In A', frames of the form $(v\ \square)_\pi$ and $(v := \square)_{(\varepsilon,\pi)}$ carry an extra tagset π, which is used to produce the corresponding $\pi\ \downarrow$ or $\pi :=\downarrow$ adjust context. Therefore, no tag information is used for infering the adjust context, making tag information redundant at runtime. We formally prove this

$$\begin{array}{rcl}
A'\left((\square\ \mathrm{e})_{\pi}\right) & = & \downarrow\uparrow \\
A'\left((v\ \square)_{\colorbox{lightgray}{π}}\right) & = & \colorbox{lightgray}{π}\downarrow \\
A'\left((\texttt{ref}\ \square)_{\varepsilon}\right) & = & \texttt{ref}\ \downarrow \\
A'\left(!\square\right) & = & !\downarrow \\
A'\left((\square := \mathrm{e})_{(\varepsilon,\pi)}\right) & = & \downarrow:=\uparrow \\
A'\left((v := \square)_{(\varepsilon,\colorbox{lightgray}{π})}\right) & = & \colorbox{lightgray}{π} :=\downarrow
\end{array}$$

Figure 5.2: New frame translation function A', mapping annotated evaluation frames to adjust contexts.

statement in section 5.5.

5.1.2 Example of a rejected program

Which programs that are accepted and don't produce an error in the generic gradual effect checking semantics result in an error in the conservative semantics? In this section, we try to answer this question by constructing an example program that produces different results when evaluated in each semantics. In short, which programs are rejected depends solely on the concrete definition of the **adjust** function.

Since programs without `has` constructs never go to Error, programs with full effect annotations never go to Error either. The difference between both semantics only affects programs in which some runtime verification needs to take place.

Tag monotonicity lemmas ensure that when tags are *removed* from a set π, **adjust** can only *add* new privileges to the resulting set. Therefore, for programs to have different behavior between both semantics, **adjust** functions have to make strict use of this condition, and for the same privilege set, produce different resulting privilege sets depending on the adjust context used.

$$\mathbf{adjust}_{\pi\downarrow}(\Phi) \subset \mathbf{adjust}_{\{\varepsilon\}\downarrow}(\Phi)$$

This behavior may only happen with the adjust contexts where there is explicit usage of the tagsets, $\pi\ \downarrow$ and $\pi :=\downarrow$. We therefore propose, as an example, the following definition for **adjust**:

$$\mathbf{adjust}_{\pi\downarrow}(\Phi) = \begin{cases} \Phi \cup \{\phi\} & \text{if } \Phi \subseteq \{\varepsilon\} \\ \Phi & \text{otherwise} \end{cases}$$

and use the identity function for any other kind of adjust context.

To build an interesting example, we need different behavior for different privilege information at runtime. The only language constructs that change behavior with different Φ's are those of the form `has` Φ e. We also require the `has` construct to be in the argument position

$$\text{IT-Fn}\ \frac{\Xi_1; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})_{\varepsilon} \Rrightarrow (\lambda x\colon T_1\ .\ \mathrm{e}')_{\varepsilon}\ \colon \{\varepsilon\}T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{IT-Unit}\ \frac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit}_{\varepsilon} \Rrightarrow \mathtt{unit}_{\varepsilon}\colon \{\varepsilon\}\mathtt{Unit}} \qquad \text{IT-Loc}\ \frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l_{\varepsilon} \Rrightarrow l_{\varepsilon}\colon \{\varepsilon\}\mathtt{Ref}\ T}$$

$$\text{IT-Var}\ \frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \Rrightarrow x\colon T}$$

$$\text{IT-App}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rrightarrow \mathrm{e}_1'\colon \pi_1 \left(T_1 \xrightarrow{\Xi_1} T_3\right) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rrightarrow \mathrm{e}_2'\colon \pi_2\rho_2 \\ \boldsymbol{\mathit{strict\text{-}check}}_{\pi_1\pi_2}(\Xi) \qquad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <\colon \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3 \end{array}}{\Xi; \Gamma; \Sigma \vdash\ \mathrm{e}_1\ \mathrm{e}_2 \Rrightarrow (\mathrm{e}_1'\ \mathrm{e}_2')_{\pi_1}\ \colon T_3}$$

$$\text{IT-Cast}\ \frac{\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon T_0 \qquad T_0 <\colon T_1 \qquad T_1 \lesssim T_2}{\Xi; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \Rrightarrow \langle T_2 \Leftarrow T_1\rangle \mathrm{e}'\colon T_2}$$

$$\text{IT-Has}\ \frac{(\Phi \cup \Xi); \Gamma; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon T}{\Xi; \Gamma; \Sigma \vdash \mathtt{has}\ \Phi\ \mathrm{e} \Rrightarrow \mathtt{has}\ \Phi\ \mathrm{e}'\colon T}$$

$$\text{IT-Rst}\ \frac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon T \qquad \Xi_1 \leq \Xi}{\Xi; \Gamma; \Sigma \vdash \mathtt{restrict}\ \Xi_1\ \mathrm{e} \Rrightarrow \mathtt{restrict}\ \Xi_1\ \mathrm{e}'\colon T}$$

$$\text{IT-Error}\ \frac{}{\Xi; \Gamma; \Sigma \vdash \mathsf{Error} \Rrightarrow \mathsf{Error}\colon T}$$

$$\text{IT-Ref}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon \pi\rho \\ \boldsymbol{\mathit{strict\text{-}check}}_{\mathtt{ref}\ \pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_{\varepsilon} \Rrightarrow (\mathtt{ref}\ \mathrm{e}')_{\varepsilon}\ \colon \{\varepsilon\}\mathtt{Ref}\ \pi\rho}$$

$$\text{IT-Deref}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rrightarrow \mathrm{e}'\colon \pi\mathtt{Ref}\ T \\ \boldsymbol{\mathit{strict\text{-}check}}_{!\pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \Rrightarrow !\mathrm{e}'\colon T}$$

$$\text{IT-Asgn}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rrightarrow \mathrm{e}_1'\colon \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rrightarrow \mathrm{e}_2'\colon \pi_2\rho_2 \\ \boldsymbol{\mathit{strict\text{-}check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <\colon T_1 \end{array}}{\Xi; \Gamma; \Sigma \vdash\ (\mathrm{e}_1 := \mathrm{e}_2)_{\varepsilon} \Rrightarrow (\mathrm{e}_1' := \mathrm{e}_2')_{(\varepsilon,\pi_1)}\ \colon \{\varepsilon\}\mathtt{Unit}}$$

Figure 5.3: Type-directed tag addition. It introduces the tag approximation of the generic language explicitly, to be used on adjust contexts for evaluation.

of an application: a program of the form $(\mathrm{e}_1\ \texttt{has}\ \Phi'\mathrm{e}_2)_{\{\varepsilon_1,\varepsilon_2\}}$ in the conservative semantics. We introduce an annotation $\{\varepsilon_1, \varepsilon_2\}$ as a set of the minimal size required to have different behavior based only on tags. To have a set $\{\varepsilon_1, \varepsilon_2\}$ frame annotation, expression e_1 needs to be typed as $\{\varepsilon_1, \varepsilon_2\}T_1 \xrightarrow{\Xi} T_2$, and therefore can only reduce to an abstraction, which may have either tag ε_1 or ε_2.

To define a concrete program that follows these restrictions, we just write e_1 as a variable f with the appropiate type, bound in a λ-abstraction.

$$\mathrm{e}_3 = (\lambda f\colon \{\varepsilon_1, \varepsilon_2\}T_1 \xrightarrow{\emptyset} T_2\ .\ f\ \texttt{has}\ \{\phi\}!l_\varepsilon)_\varepsilon$$

We define function $\mathbf{check}_{!\pi}(\Phi) \iff \phi \in \Phi$, and $\mathbf{check}_C(\Phi)$ to always hold for any other check context. Given our definition of **adjust**, the following example is a valid program:

$$(\mathrm{e}_3\ (\lambda x\colon T_1\ .\ x)_{\varepsilon_1})_{\{\varepsilon\}}$$

This program can be typed with $\Xi = \{¿\}$ and evaluates to different results in each semantics. To focus on the interesting step of evaluation , we first apply substitution of the argument in the body of the function. Then we get the following cases for evaluation with $\Phi = \emptyset$. In the conservative semantics:

$$\frac{\mathbf{adjust}_{\{\varepsilon_1,\varepsilon_2\}\downarrow}(\Phi) \vdash \texttt{has}\ \{\phi\}!l \mid \mu \rightsquigarrow \mathsf{Error} \mid \mu}{\Phi \vdash ((\lambda x\colon T_1\ .\ x)_{\varepsilon_1}\ \texttt{has}\ \{\phi\}!l)_{\{\varepsilon_1,\varepsilon_2\}} \mid \mu \rightsquigarrow^* \mathsf{Error} \mid \mu}$$

and in the generic gradual effect checking semantics:

$$\frac{\mathbf{adjust}_{\{\varepsilon_1,\varepsilon_2\}\downarrow}(\Phi) \vdash \texttt{has}\ \{\phi\}!l \mid \mu \rightarrow !l \mid \mu}{\Phi \vdash (\lambda x\colon T_1\ .\ x)_{\varepsilon_1}\ \texttt{has}\ \{\phi\}!l \mid \mu \rightsquigarrow^* (\lambda x\colon T_1\ .\ x)_{\varepsilon_1}\ !l \mid \mu}$$

Which will produce a value depending on $\mu(l)$.

$$\textsc{E-Ref}\ \frac{l \notin \mathrm{dom}\,(\mu) \qquad \mathbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi)}{\Phi \vdash (\texttt{ref}\ w_{\varepsilon_1})_{\varepsilon_2} \mid \mu \rightsquigarrow l \mid \mu[l \mapsto w]} \qquad \textsc{E-Deref}\ \frac{\mu(l) = v \qquad \mathbf{check}_{!l_\varepsilon}(\Phi)}{\Phi \vdash !l_\varepsilon \mid \mu \rightsquigarrow v \mid \mu}$$

$$\textsc{E-Asgn}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (l_{\varepsilon_1} := w_{\varepsilon_2})_{(\varepsilon,\pi)} \mid \mu \rightsquigarrow \texttt{unit}_\varepsilon \mid \mu[l \mapsto w_{\varepsilon_2}]} \qquad \textsc{E-Frame}\ \frac{\mathbf{adjust}_{A'(f)}(\Phi) \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'}{\Phi \vdash f[\mathrm{e}] \mid \mu \rightsquigarrow f[\mathrm{e}'] \mid \mu'}$$

$$\textsc{E-Error}\ \frac{}{\Phi \vdash g[\mathsf{Error}] \mid \mu \rightsquigarrow \mathsf{Error} \mid \mu} \qquad \textsc{E-App}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash ((\lambda x\colon T_1\ .\ \mathrm{e})_{\varepsilon_1}\ w_{\varepsilon_2})_\pi \mid \mu \rightsquigarrow [^{w_{\varepsilon_2}}/_x]\,\mathrm{e} \mid \mu}$$

$$\textsc{E-Cast-Frame}\ \frac{\Phi \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \mid \mu \rightsquigarrow \langle T_2 \Leftarrow T_1\rangle \mathrm{e}' \mid \mu'} \qquad \textsc{E-Has-T}\ \frac{\Phi' \subseteq \Phi \qquad \Phi \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \rightsquigarrow \texttt{has}\ \Phi'\ \mathrm{e}' \mid \mu'}$$

$$\textsc{E-Has-V}\ \frac{}{\Phi \vdash \texttt{has}\ \Phi'\ w \mid \mu \rightsquigarrow w \mid \mu} \qquad \textsc{E-Has-F}\ \frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \rightsquigarrow \mathsf{Error} \mid \mu}$$

$$\textsc{E-Rst}\ \frac{\Phi'' = \max\{\Phi' \in \gamma(\Xi) \mid \Phi' \subseteq \Phi\} \qquad \Phi'' \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \rightsquigarrow \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'} \qquad \textsc{E-Rst-V}\ \frac{}{\Phi \vdash \texttt{restrict}\ \Xi\ w \mid \mu \rightsquigarrow w \mid \mu}$$

$$\textsc{E-Cast-Id}\ \frac{\pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2 \rho \Leftarrow \pi_1 \rho\rangle w \mid \mu \rightsquigarrow w \mid \mu}$$

E-Cast-Fn

$$\frac{\pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2 T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow \pi_1 T_{11} \xrightarrow{\Xi_1} T_{12}\rangle\ (\lambda x\colon T_{11}\ .\ \mathrm{e}) \mid \mu \rightsquigarrow (\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ \ (|\Xi_1| \setminus |\Xi_2|)\ \ [^{(\langle T_{11} \Leftarrow T_{21}\rangle x)}/_x]\,\mathrm{e}) \mid \mu}$$

Figure 5.4: Full Conservative Semantics, with special frame translation function A', that maps annotated evaluation frames to adjust contexts.

5.2 New semantics is a conservative approximation

We call the semantics proposed in section 5.1 a *conservative approximation.* By conservative approximation we mean that if a program reduces to a value and a store in the modified semantics, it reduces to the same value and store in the generic gradual effect checking language (modulo tag annotations). If a program reduces to a runtime error in the conservative semantics, the program either reduces to an error or to a value in the generic gradual effect checking semantics. Thus the relation between both semantics is not bijective.

We define the relation between both semantics formally in the Conservative Approximation Theorem (theorem 18). To define the theorem, we first define two auxiliary notions: A tagset erasure function ($[\![]\!]^\pi$) that maps programs from the conservative language syntax to the generic gradual effect checking language syntax by removing the extra annotations, and a *simulation relation* that captures the relation between both languages. This simulation relation must be preserved throughout evaluation by theorem 18.

Definition 15 ($[\![]\!]^\pi$ function)**.** *We define function* $[\![]\!]^\pi : \mathbf{Expr}_{Conservative} \to \mathbf{Expr}_{Generic}$ *as follows:*

$$\begin{array}{rcl}
[\![\texttt{unit}_\varepsilon]\!]^\pi &=& \texttt{unit}_\varepsilon \\
[\![l_\varepsilon]\!]^\pi &=& l_\varepsilon \\
[\![(\lambda x\colon T\ .\ \mathrm{e})_\varepsilon]\!]^\pi &=& (\lambda x\colon T\ .\ [\![\mathrm{e}]\!]^\pi)_\varepsilon \\
[\![(\mathrm{e}_1\ \mathrm{e}_2)_\pi]\!]^\pi &=& [\![\mathrm{e}_1]\!]^\pi\ [\![\mathrm{e}_2]\!]^\pi \\
[\![\langle T_1 \Leftarrow T_0\rangle \mathrm{e}]\!]^\pi &=& \langle T_1 \Leftarrow T_0\rangle [\![\mathrm{e}]\!]^\pi \\
[\![\texttt{has}\ \Phi\ \mathrm{e}]\!]^\pi &=& \texttt{has}\ \Phi\ [\![\mathrm{e}]\!]^\pi \\
[\![\texttt{restrict}\ \Xi\ \mathrm{e}]\!]^\pi &=& \texttt{restrict}\ \Xi\ [\![\mathrm{e}]\!]^\pi \\
[\![\mathsf{Error}]\!]^\pi &=& \mathsf{Error} \\
[\![(\texttt{ref}\ \mathrm{e})_\varepsilon]\!]^\pi &=& (\texttt{ref}\ [\![\mathrm{e}]\!]^\pi)_\varepsilon \\
[\![!\mathrm{e}]\!]^\pi &=& ![\![\mathrm{e}]\!]^\pi \\
[\![(\mathrm{e}_1 := \mathrm{e}_2)_{(\varepsilon,\pi)}]\!]^\pi &=& ([\![\mathrm{e}_1]\!]^\pi := [\![\mathrm{e}_2]\!]^\pi)_\varepsilon
\end{array}$$

Simulation Relation. The simulation relation encapsulates how we want to relate programs from the conservative semantics with programs in the generic gradual effect checking semantics. To avoid confusion, we will underscore with a $_C$ relations that should hold in the conservative semantics, and with a $_O$ relations that should hold in the generic gradual semantics introduced in chapter 4.

We define this simulation relation formally as follows:
Definition 16 (Simulation Relation)**.**

$$\frac{\begin{array}{c}\Xi; \Gamma; \Sigma \vdash_C \mathrm{e}_2 : T_2 \\ \mathrm{e}_1 = [\![\mathrm{e}_2]\!]^\pi \\ \Gamma; \Sigma \vDash_O \mu_1 \qquad \Gamma; \Sigma \vDash_C \mu_2 \\ \mu_1 = [\![]\!]^\pi \circ \mu_2\end{array}}{\Xi; \Gamma; \Sigma\ \Vdash\ (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)}$$

A pair (e_2, μ_2) from the conservative language is related to a pair (e_1, μ_1) in the generic gradual effect checking language with a context $\Xi;\Gamma;\Sigma$ by the simulation relation if:

1. e_2 can be typed in the conservative type system using the context (Exists a type T such that $\Xi;\Gamma;\Sigma \vdash_C \mathrm{e}_2 \colon T$).
2. e_1 and e_2 correspond to the same expression modulo tag information ($\mathrm{e}_1 = [\![\mathrm{e}_2]\!]^\pi$)
3. Both stores μ_1 and μ_2 are consistent with the context, and μ_1 is equivalent to μ_2 without the extra tagset information.

Definition 17 (Valid simulation privilege sets)**.**

$$\Phi \sim \Xi \iff \exists \Phi' \subseteq \Phi \ . \ \Phi' \in \gamma(\Xi)$$

We say that a privilege set Φ validly simulates a consistent privilege set Ξ if and only if there exists a set in the concretization of Ξ that is contained in Φ.

This definition of valid simulation privilege sets is an extension of the concretization function. Unlike concretization, the set of valid simulation privilege sets always contains all the sets that contain at least the privileges in the static part of Ξ. We require this flexibility to prove intermediate results in our path towards proving the conservative approximation theorem, which we now define.

Theorem 18 (Conservative Approximation)**.** . *Let $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rrightarrow \mathrm{e}_2 \colon T$, μ_1 and μ_2 such that $\Xi;\Gamma;\Sigma \ \Vdash \ (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)$, and $\Phi \sim \Xi$. If $\Phi \vdash \mathrm{e}_2 \mid \mu_2 \rightsquigarrow^* v_2 \mid \mu_2'$, then $\exists v_1$ and μ_1' such that $\Phi \vdash \mathrm{e}_1 \mid \mu_1 \rightarrow^* v_1 \mid \mu_1'$ and $\exists \Sigma' \supseteq \Sigma$ such that $\Xi;\Gamma;\Sigma' \ \Vdash \ (v_1, \mu_1') \sim (v_2, \mu_2')$.*

Proof. To prove this theorem, we establish an intermediate strong conservative approximation lemma (theorem 19). Then this theorem reduces to the reflexive-transitive closure of theorem 19. □

Theorem 19 (Strong Conservative Approximation)**.** .

Let $\Xi;\Gamma;\Sigma \ \Vdash \ (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)$ and $\Phi \sim \Xi$. If $\Phi \vdash \mathrm{e}_2 \mid \mu_2 \rightsquigarrow \mathrm{e}_2' \mid \mu_2'$, then for any $\Phi' \sim \Xi$, either:

- $\Phi' \vdash \mathrm{e}_2' \mid \mu_2' \rightsquigarrow^* \mathsf{Error} \mid \mu_2'$
- $\exists \mathrm{e}_1'$ *and* μ_1' *such that* $\Phi' \vdash \mathrm{e}_1 \mid \mu_1 \rightarrow \mathrm{e}_1' \mid \mu_1'$ *and* $\exists \Sigma' \supseteq \Sigma$ *such that* $\Xi;\Gamma;\Sigma' \ \Vdash \ (\mathrm{e}_1', \mu_1') \sim (\mathrm{e}_2', \mu_2')$.

Proof. We provide a proof sketch here. The reader interested in more details should also look at the appendix D.

The proof goes by structural induction over $\rightsquigarrow$. Since both semantics are mostly equivalent modulo differences in rules [E-Frame] in each semantics, that rule is the only interesting case for the proof.

The key step in the proof is allowing usage of the induction hypothesis for rule [E-Frame], since there are different adjust contexts used in each semantics. To do so we first establish a principle of well-formedness for frames, in which the tags that are used to generate adjust contexts are always contained in the syntactic annotations introduced in the conservative

$$\text{IT-Fn}\;\dfrac{\Xi_1; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}\colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})_{\varepsilon}\colon \{\varepsilon\} T_1 \xrightarrow{\Xi_1} T_2} \qquad \text{IT-Unit}\;\dfrac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit}_{\varepsilon}\colon \{\varepsilon\}\mathtt{Unit}}$$

$$\text{IT-Loc}\;\dfrac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l_{\varepsilon}\colon \{\varepsilon\}\mathtt{Ref}\ T} \qquad \text{IT-Var}\;\dfrac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x\colon T}$$

$$\text{IT-App}\;\dfrac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \colorbox{lightgray}{π_0} \left(T_1 \xrightarrow{\Xi_1} T_3\right) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2\colon \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi) \qquad \colorbox{lightgray}{π_0}\, T_1 \xrightarrow{\Xi_1} T_3 <:\ \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3\end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1\ \mathrm{e}_2)_{\pi_1}\colon T_3}$$

$$\text{IT-Cast}\;\dfrac{\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T_0 \qquad T_0 <:\ T_1 \qquad T_1 \lesssim T_2}{\Xi; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e}\colon T_2} \qquad \text{IT-Has}\;\dfrac{(\Phi \cup \Xi); \Gamma; \Sigma \vdash \mathrm{e}\colon T}{\Xi; \Gamma; \Sigma \vdash \mathtt{has}\ \Phi\ \mathrm{e}\colon T}$$

$$\text{IT-Rst}\;\dfrac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T \qquad \Xi_1 \leq \Xi}{\Xi; \Gamma; \Sigma \vdash \mathtt{restrict}\ \Xi_1\ \mathrm{e}\colon T} \qquad \text{IT-Error}\;\dfrac{}{\Xi; \Gamma; \Sigma \vdash \mathsf{Error}\colon T}$$

$$\text{IT-Ref}\;\dfrac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}\colon \pi\rho \\ \textbf{\textit{strict-check}}_{\mathtt{ref}\ \pi}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_{\varepsilon}\colon \{\varepsilon\}\mathtt{Ref}\ \pi\rho} \qquad \text{IT-Deref}\;\dfrac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}\colon \pi\mathtt{Ref}\ T \\ \textbf{\textit{strict-check}}_{!\pi}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e}\colon T}$$

$$\text{IT-Asgn}\;\dfrac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \colorbox{lightgray}{π_0}\,\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2\colon \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <:\ T_1 \qquad \colorbox{lightgray}{$\pi_0 \subseteq \pi_1$}\end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_{(\varepsilon,\pi_1)}\colon \{\varepsilon\}\mathtt{Unit}}$$

Figure 5.5: Type system for the language with extra tagset information

language. We then prove that simulation ensures well-formed frames, and that there is a partial ordering between the adjust context used in the conservative semantics and those in the original semantics ($A(f_O) \sqsubseteq A'(f_C)$).

This partial oredering can be used to ensure that if Φ is a valid simulation privilege set for Ξ ($\Phi \sim \Xi$), then also $\mathbf{adjust}_{A(f_O)}(\Phi) \sim \widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)$. This relation enables usage of the induction hypothesis. □

5.3 Type safety of the conservative semantics

We have presented in this chapter a conservative semantics for the generic gradual effect framework, ensuring that if a program reduces to a value in the conservative semantics, it will reduce to the same value (without the extra annotations required) in the generic gradual effect checking semantics. We have not yet proved type safety for this new semantics, and we proceed to do so in this section. The key difference with the type system for the intermediate language is the explicit subsumption of tagsets (which is highlighted in boxes in fig. 5.5. We now proceed to prove type safety.

Theorem 20 (Progress)**.** *Suppose* $\Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$*. Then either* e *is a value* v*, an* Error*, or* $\Phi \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'$ *for all privilege sets* $\Phi \in \gamma(\Xi)$ *and for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$*.*

Proof. By Structural Induction on type derivation. Since most of the typing rules are equal to the generic gradual effect checking language, for which we have already proven Progress, we fall back to the generic gradual effect checking language proof for the uninteresting cases. For [IT-App] and [IT-Asgn], the proof is almost analogous, except that the new language does not recur to an argument based on tag monotonicity for adjust. This means that we do not need the condition $\forall \Phi' \supseteq \Phi \in \gamma(\Xi)$ present in the generic gradual effect checking language. □

Theorem 21 (Preservation)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T$*, and* $\Phi \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'$ *for* $\Phi \in \gamma(\Xi)$ *with* $\Gamma \mid \Sigma \vDash \mu$*, then* $\exists \Sigma' \supseteq \Sigma$ *such that* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}'\colon T'$ *with* $T' <: T$*.*

Proof. By Structural Induction on type derivation. As in the case for progress, the proof is analogous to the generc gradual effect checking language, except for the cases for rules [IT-App] with rule [E-Frame] and $f = (v\square)_\pi$, and rule [IT-Asgn] with rule [E-Frame] and $f = (v := \square)_{(\varepsilon,\pi)}$, which are simpler in the sense that they do not require the superset restriction that was required in the generic gradual effect checking language to account for the difference between the tagset used at evaluation and the one used for typing. □

5.4 Redundancy of tags in the conservative semantics

We have repeatedly stated that we introduced the conservative semantics to avoid carrying tag information at runtime. To formalize this idea, we introduce a simulation argument. We can easily define an operational semantics $\hookrightarrow$ based on the $\rightsquigarrow$ conservative semantics, but removing any calls to **check**, and also define a tag-removal function $[\![]\!]^\varepsilon$ as follows

$$\begin{array}{rcl}
[\![\mathtt{unit}_\varepsilon]\!]^\varepsilon & = & \mathtt{unit} \\
[\![l_\varepsilon]\!]^\varepsilon & = & l \\
[\![(\lambda x\colon T\ .\ \mathrm{e})_\varepsilon]\!]^\varepsilon & = & (\lambda x\colon T\ .\ [\![\mathrm{e}]\!]^\varepsilon) \\
[\![(\mathrm{e}_1\ \mathrm{e}_2)_\pi]\!]^\varepsilon & = & ([\![\mathrm{e}_1]\!]^\varepsilon\ [\![\mathrm{e}_2]\!]^\varepsilon)_\pi \\
[\![\langle T_1 \Leftarrow T_0\rangle \mathrm{e}]\!]^\varepsilon & = & \langle T_1 \Leftarrow T_0\rangle [\![\mathrm{e}]\!]^\varepsilon \\
[\![\mathtt{has}\ \Phi\ \mathrm{e}]\!]^\varepsilon & = & \mathtt{has}\ \Phi\ [\![\mathrm{e}]\!]^\varepsilon \\
[\![\mathtt{restrict}\ \Xi\ \mathrm{e}]\!]^\varepsilon & = & \mathtt{restrict}\ \Xi\ [\![\mathrm{e}]\!]^\varepsilon \\
[\![\mathsf{Error}]\!]^\varepsilon & = & \mathsf{Error} \\
[\![(\mathtt{ref}\ \mathrm{e})_\varepsilon]\!]^\varepsilon & = & \mathtt{ref}\ [\![\mathrm{e}]\!]^\varepsilon \\
[\![!\mathrm{e}]\!]^\varepsilon & = & ![\![\mathrm{e}]\!]^\varepsilon \\
[\![(\mathrm{e}_1 := \mathrm{e}_2)_{(\varepsilon,\pi)}]\!]^\varepsilon & = & ([\![\mathrm{e}_1]\!]^\varepsilon := [\![]\!]^\varepsilon \mathrm{e}_2)_\pi
\end{array}$$

We can then state the following theorem relating both semantics:
Theorem 22 (**check** and tags are redundant in $\rightsquigarrow$)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T$ *and* $\Phi \vdash \mathrm{e} \mid \mu \rightsquigarrow \mathrm{e}' \mid \mu'$ *for* $\Phi \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$*, then also* $\Phi \vdash \mathcal{E}'(\mathrm{e}) \mid \mathcal{E}'(\mu) \hookrightarrow \mathcal{E}'(\mathrm{e}') \mid \mathcal{E}'(\mu')$*.*

Proof. **check** predicates are made redundant by the typing hypothesis, since we know in any case that ***strict-check***$_A(\Xi)$ implies $\mathbf{check}_A(\Phi)\ \forall\Phi \in \gamma(\Xi)$. If we remove the **check** predicates, ε tag annotations can be considered redundant because they are not used in the runtime semantics at any interesting spot. □

5.5 Summary

Section 5.1 introduced an alternative semantics for the generic gradual effect checking framework. Unlike the semantics introduced in the previous chapter, this semantics does not depend on tag annotations for values, which therefore can be safely removed in an implementation. Section 5.4 presents a proof of this property, while section 5.5 proves that the new semantics is type safe. We call the introduced semantics a conservative semantics since whenever a program reduces to a value in the conservative semantics, it reduces to the same value in the semantics of the previous chapter, modulo tag annotations. This relation is stated as a theorem and proven in section 5.2.

This chapter closes our exposition about generic gradual effect checking. In the next chapter, we take a step further to introduce flexibility not only for effect annotations, but also for type annotations by combining gradual effect checking with gradual typing.

Chapter 6

Gradual Type-and-Effect Systems

The generic gradual effect checking framework introduced in chapter 4 empowers the programmer to decide when and where to introduce effect annotations, introducing the necessary checks when static information is not sufficient to enforce an effect discipline. However, the flexibility provided by gradual effect checking only applies to effect annotations: programs still require type annotations. In this chapter, we take a step further and give the programmer full flexibility over both effect and type annotations.

We combine the generic gradual effect checking framework introduced in chapter 4 with gradual typing to provide complete gradual typing for type-and-effect systems. We do so in two steps: we first combine gradual typing with a simplified version of the generic gradual effect checking framework that lacks tags, and then we introduce gradual typing for the full generic gradual effect checking system introduced in chapter 4. Tag annotations interact with the type system in non trivial ways, so we avoid them at first to show explicitly that gradual typing is practically orthogonal with gradual effect checking. After introducing gradual typing without tags, we focus on the interactions between tag annotations and gradual typing.

6.1 Gradual typing for type-and-effect systems without tags

In this section we introduce gradual typing for a simplified version of the gradual effect checking framework without tags. We follow a similar strategy to the structure followed by Siek and Taha to introduce Gradual Typing [24]: After explaining the simplified framework, we propose an alternative type system that is more flexible than the original language type-system. The new type system uses type consistency to model which types may produce a valid program. We then introduce an intermediate language and a translation algorithm that inserts explicit runtime checks when static information is not sufficient to ensure that a program is valid according to the type and effect restrictions of the language. Finally, we introduce a runtime semantics for the language and establish type safety.

Source Language

$$\phi \in \textbf{Priv}, \quad \xi \in \textbf{CPriv} = \textbf{Priv} \cup \{¿\}$$
$$\Phi \in \textbf{PrivSet} = \mathcal{P}(\textbf{Priv}), \quad \Xi \in \textbf{CPrivSet} = \mathcal{P}(\textbf{CPriv})$$

$$\begin{array}{lcll}
v & ::= & \mathtt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} \mid l & \text{Values} \\
\mathrm{e} & ::= & x \mid v \mid \mathrm{e}\ \mathrm{e} \mid \mathrm{e} :: \Xi & \text{Terms} \\
 & & \mid \mathtt{ref}\ \mathrm{e} \mid !\mathrm{e} \mid \mathrm{e} := \mathrm{e} & \\
T & ::= & \mathtt{Unit} \mid T \xrightarrow{\Xi} T \mid \mathtt{Ref}\ T \mid \mathtt{Dyn} & \text{Types} \\
A & ::= & \downarrow\uparrow \mid \bullet \downarrow \mid \mathtt{ref}\ \downarrow \mid !\downarrow & \text{Adjust Contexts} \\
 & & \mid \downarrow := \uparrow \mid \bullet := \downarrow & \\
C & ::= & \bullet\ \bullet \mid \mathtt{ref}\bullet \mid !\bullet \mid \bullet := \bullet & \text{Check Contexts}
\end{array}$$

Figure 6.1: Syntax of the source language

6.1.1 Simplifying the framework

The generic gradual effect checking framework introduced in chapter 4 provides tag annotations for values. These annotations are inherited from the generic M&M framework and provide increased expressivity. To focus our analysis on the interactions between gradual typing and gradual effect checking, we consider a system without tag annotations. With this restriction, we can simplify our exposition and explain the subtleties introduced by tag annotations later.

We define a system without tags by imposing restrictions to the generic framework. We restrict the universe of available tag annotations to be a singleton, which we denote $\{\bullet\}$ ($\bullet$ is simply a placeholder). Since there is only one tag annotation available ($\bullet$), every value in the language is annotated with the same tag, thus the tag annotation becomes redundant and we can remove it. For example, value `unit` is equivalent to the tagged value $\mathtt{unit}_\bullet$.

In the generic framework, tag annotations leak from values into types, adjust contexts and check contexts. By restricting the system to one tag available, leaked tag annotations also become redundant. We then define types without tag annotations, and replace the tag set restrictions in adjust and check contexts with $\bullet$ placeholders.

We use notation `Dyn` instead of ? for the dynamic type used in gradual typing to avoid confusion with the unknown effect annotation ¿ of gradual effect checking. The full syntax of the language is presented in fig. 6.1.

$$\boxed{\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T} \qquad \text{T-Fn}\frac{\Xi_1; \Gamma, x \colon T_1; \Sigma \vdash \mathrm{e} \colon T_2}{\Xi; \Gamma; \Sigma \vdash \lambda x \colon T_1 \,.\, \mathrm{e} \colon T_1 \xrightarrow{\Xi_1} T_2} \qquad \text{T-Unit}\frac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit} \colon \mathtt{Unit}}$$

$$\text{T-Loc}\frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l \colon \mathtt{Ref}\ T} \qquad \text{T-Var}\frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \colon T} \qquad \text{T-App}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon T \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T_2 \\ T \lesssim T_2 \xrightarrow{\Xi} T_3 \qquad \widetilde{\mathbf{check}}_{\bullet\bullet}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3}$$

$$\text{T-Eff}\frac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \colon T \qquad \Xi_1 \sqsubseteq\!\!\!\!\sim\ \Xi}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e} :: \Xi_1) \colon T} \qquad \text{T-Ref}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \colon T \\ \widetilde{\mathbf{check}}_{\mathtt{ref}\ \bullet}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathtt{ref}\ \mathrm{e} \colon \mathtt{Ref}\ T}$$

$$\text{T-Deref}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \colon T_1 \\ T_1 \sim \mathtt{Ref}\ T \qquad \widetilde{\mathbf{check}}_{!\bullet}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \colon T} \qquad \text{T-Asgn}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon T \qquad T \sim \mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\bullet:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T_2 \\ \widetilde{\mathbf{check}}_{\bullet:=\bullet}(\Xi) \qquad T_2 \lesssim T_1\end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2 \colon \mathtt{Unit}}$$

Figure 6.2: Typing rules for the source language

6.1.2 A type system based on consistency

We automatically annotate programs with a `Dyn` type wherever static type annotations are missing. Type restrictions must be able to handle both the `Dyn` type and types with partial missing information (like `Ref Dyn`). Like in the case of Gradual Typing as introduced in section 2.2.1, we replace any important type equality restriction with type consistency. We define type consistency for the language in section 6.1.3. The type system obtained is very similar to the typesystem used for gradual effect checking and introduced in fig. 4.2. The only rules that differ are rules [T-App], [T-Deref] and [T-Asgn], rules on which we add a type consistency restriction. The full type system is introduced in fig. 6.2.

6.1.3 Extending type consistency for effects

The type system uses type consistency to statically accept expressions with type `Dyn` (or whose type is partially unknown) whenever expressions with particular type are required, like in the function position of a function application. Type consistency acts as a relaxed form of equivalence: whenever a certain type T_1 is required for a program to be valid, a program with an unknown type must also be statically accepted, because the type system is not able to distinguish between them. The definition of type consistency introduced by Siek and Taha [24] does not consider effect annotations, so we need to provide a type consistency relation that provides this flexibility for type-and-effect systems.

Our analysis of gradual effect checking did not require a notion of effect consistency. Instead, we based gradual effect checking on consistent containment, which acted as a relaxed form of set containment. We can reuse consistent containment to define a notion of effect consistency, building on the same strategy that reduces set equality to a pair of set containment operations ($a = b \iff a \subseteq b$ and $b \subseteq a$)[1]. We use this property to define effect consistency as follows:

Definition 18 (Effect consistency)**.**

We say that two consistent privilege sets Ξ_1 *and* Ξ_2 *are consistent , denoted* $\Xi_1 \simeq \Xi_2$*, if* $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi_2$ and $\Xi_2 \sqsubseteq\!\!\!\!\sim \Xi_1$.

We use this definition of effect consistency to provide a definition of type consistency that handles effects. As in Siek and Taha [24], reference cell types are only consistent with themselves (since type consistency is reflexive). We define type consistency as follows:

$$\text{C-Refl}\frac{}{\tau \sim \tau} \qquad \text{C-UnR}\frac{}{\tau \sim \texttt{Dyn}} \qquad \text{C-UnL}\frac{}{\texttt{Dyn} \sim \tau}$$

$$\text{C-Fun}\frac{\sigma_1 \sim \tau_1 \quad \sigma_2 \sim \tau_2 \quad \Xi_\sigma \simeq \Xi_\tau}{\left(\sigma_1 \xrightarrow{\Xi_\sigma} \sigma_2\right) \sim \left(\tau_1 \xrightarrow{\Xi_\tau} \tau_2\right)}$$

6.1.4 Consistent subtyping

Effect annotations on type-and-effect systems lead to a natural notion of subtyping, so subtyping must considered from the start to define an interesting notion of gradual typing for type-and-effect systems. As introduced in section 2.2.4, we can combine subtyping and gradual typing by introducing consistent subtyping, which we defined as follows[2]:

Definition 19 (Consistent Subtyping)**.** *Consistent subtyping* ($\lesssim$) *is defined as*

$$a \lesssim b \iff \exists \alpha \sim a \ . \ \alpha <: b \text{ or } \exists \beta \sim b \ . \ a <: \beta$$

This definition of consistent subtyping makes use of subtyping, which we already defined for gradual effect checking in section 4.2.2 and we now transcribe:

$$\text{ST-Id}\frac{}{T <: T} \qquad \text{ST-Abs}\frac{T_3 <: T_1 \quad T_2 <: T_4 \quad \Xi_1 \subseteq \Xi_2}{T_1 \xrightarrow{\Xi_1} T_2 <: T_3 \xrightarrow{\Xi_2} T_4}$$

This definition of subtyping plus the type consistency relation we have just introduced sustain our definition for consistent subtyping.

[1]This theorem can be proved by the axiom of extensionality in ZFC set theory.

[2]When Siek and Taha introduce consistent subtyping in [23], they provide first an algorithmic definition of the relation and then prove that the definition we provide in theorem 23 is equivalent. We do not provide an algorithmic definition of consistent subtyping, we attempt to focus on the specification of the problem instead of the implementation of the solution.

$$
\begin{array}{lcll}
v & ::= & \ldots \mid \langle T \Leftarrow T\rangle v & \text{Values} \\
\mathrm{e} & ::= & \ldots \mid \mathsf{Error} \mid \langle T \Leftarrow T\rangle \mathrm{e} & \text{Terms} \\
 & & \mid \texttt{has}\ \Phi\ \mathrm{e} \mid \texttt{restrict}\ \Xi\ \mathrm{e} & \\
f & ::= & \square\ \mathrm{e} \mid v\ \square \mid \texttt{ref}\ \square & \text{Frames} \\
 & & \mid !\square \mid \square := \mathrm{e} \mid v := \square & \\
g & ::= & f \mid \langle T_2 \Leftarrow T_1\rangle \square \mid \texttt{has}\ \Phi\ \square & \text{Error Frames} \\
 & & \mid \texttt{restrict}\ \Xi\ \square &
\end{array}
$$

Figure 6.3: Syntax of the internal language

Properties of consistent subtyping

When defining consistent containment, Siek and Taha [23] went a step further than our definition, proving that if there exist an α that fullfills the $\alpha <: b$, then *also* exists a β that fullfills $a <: \beta$, meaning that both definitions are equivalent. We also want to state this property for our definition of consistent subtyping.

Theorem 23 (Consistent subtyping equivalence)**.**

$$\exists \alpha \sim a\ .\ \alpha <: b \iff \exists \beta \sim b\ .\ a <: \beta$$

Proof. By structural induction over the type consistency definition $\sim$. Detailed proof in appendix E. □

This is not the only interesting property our definition of consistent subtyping has. When defining gradual effect checking in chapter 3, we provided a definition of consistent subtyping that took into account only effect annotations. That definition is clearly different from what we have just proposed, but we can prove that in absence of the `Dyn` type, both definitions are equivalent. If both definitions are equivalent in presence of gradual effect annotations, we can consider our new definition of consistent subtyping as an extension of consistent subtyping for gradual effect checking.

Theorem 24. *If $T_1 \lesssim_{GE} T_2$, then also $T_1 \lesssim_{GT} T_2$*[3]

Proof. By structural induction on the definition of subtyping in gradual effect checking ($\lesssim_{GE}$), using lemma 69 for effect annotations in function types. Both the lemma and a proof for it are provided in appendix E. □

6.1.5 Intermediate language

Introducing gradual typing requires extending the language with type casts, but type casts were already introduced for gradual effect checking. Thus the intermediate language intro-

[3]We use $\lesssim_{GE}$ to denote the consistent subtyping relation used in chapter 3 and chapter 4. We use $\lesssim_{GT}$ to denote the consistent subtyping relation defined in this chapter.

duced in fig. 6.3 does not introduce any new language constructs. The typing relation for the intermediate language introduced in fig. 6.4 is almost the same as the one used for generic gradual effect checking without tag annotations.

The only difference between both typing relations is that we have removed the consistent subtyping restriction used in rule [IT-Cast]. In gradual effect checking, we could include that restriction because type casts would never fail on themselves, but would instead be reduced to a combination of effect-related `restrict` and `has` constructs that may or may not fail. When we introduce type `Dyn`, we require more flexibility to preserve safety, in particular for type preservation: A program $\langle \texttt{Nat} \Leftarrow \texttt{Dyn}\rangle\langle \texttt{Dyn} \Leftarrow \texttt{Unit}\rangle\texttt{unit}$ should reduce to a cast $\langle \texttt{Nat} \Leftarrow \texttt{Unit}\rangle\texttt{unit}$ that would later fail, but a cast $\langle \texttt{Nat} \Leftarrow \texttt{Unit}\rangle$ cannot be typed if rule [IT-Cast] considers a consistent subtyping restriction between `Unit` and `Nat`.

The operational semantics introduced in fig. 6.5 extends the semantics defined for gradual effect checking with the extra behavior required to handle casts with `Dyn` types. Unlike gradual effect checking, type casts in this semantics may fail for reasons other than effect restrictions, because type consistency is not transitive. Consider the program $((\lambda f : \texttt{Dyn} \,.\, (f\ \texttt{unit}))\ \texttt{unit})$. After translation and substitution, this program translates to

$$((\langle \texttt{Unit} \xrightarrow{\Xi} \texttt{Dyn} \Leftarrow \texttt{Dyn}\rangle\langle \texttt{Dyn} \Leftarrow \texttt{Unit}\rangle\texttt{unit})\ \texttt{unit})$$

First, both casts should be merged into a $\langle \texttt{Unit} \xrightarrow{\Xi} \texttt{Dyn} \Leftarrow \texttt{Unit}\rangle$ cast, behavior provided by rule [E-Cast-Merge]. This new cast should then fail because both types are not consistent subtypes, behavior provided by rule [E-Cast-Bad].

Minimal interactions between gradual typing and gradual effect checking in the translation algorithm

The translation algorithm, presented in fig. 6.6, is a combination between the translation rules defined for generic gradual effect checking and gradual typing. The only interesting overlap arises in rule [C-App-2], where a $\langle T_2 \xrightarrow{\Xi} \texttt{Dyn} \Leftarrow \texttt{Dyn}\rangle$ is introduced. In gradual typing, an application where e_1 has type `Dyn` requires a cast to ensure that e_1 is actually a function. That restriction would be fulfilled by inserting a cast with *any* privilege set, in particular a set with unknown privileges like $\{¿\}$. Unfortulately, this set is not sufficient. Using a privilege set $\{¿\}$ does not take into account the restriction arising from type-and-effect systems that a function that is applied cannot generate more side effects than those allowed in the context of application. To make this assumption explicit, the cast must restrict the privileges to the context of privileges available Ξ.

The following program shows an example of improper behavior if we introduce a cast to $\{¿\}$ instead of a cast to the effect context Ξ, using the generic gradual effect version of the fluent language as the effect discipline:

$$((\lambda f : \texttt{Dyn} \,.\, (f\ \texttt{unit}) :: \emptyset)\ \textit{effectful-argument})$$

In this program, *effectful-argument* represent a properly typed function in scope that generates a `write` effect ($\Gamma(\textit{effectful-argument}) = \texttt{Unit} \xrightarrow{\{\texttt{write}\}} \texttt{Unit}$, for example). The translation should introduce enough runtime checks to make this program fail, since the context where f is applied does not allow side effects ($(f\ \texttt{unit}) :: \emptyset$). If rule [C-App-2] used $\{¿\}$ instead of Ξ, then this program would not produce the required runtime error for the use of *effectful-argument*, because the $\{\texttt{write}\}$ effects of *effectful-argument* can safely be hidden into a set with unknown effects like $\{¿\}$. Since rule [C-App-2] uses the privilege context instead, which in this case is the empty set $\emptyset$, the runtime semantics produces an effect error as expected.

6.1.6 Type safety

In this section we provide proof sketches for type safety of the intermediate language we have introduced. Detailed versions of these proofs are provided in appendix E.

Theorem 25 (Progress)**.** *Suppose* $\Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$. *Then either* e *is a value* v, *an* Error*, or* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for all privilege sets* Φ *such that* $\exists \Phi' \in \gamma(\Xi)$ *such that* $\Phi' \subseteq \Phi$ *and for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$.

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$. Compared with the proof of progress for theorem 15, the only interesting cases arise for rule [IT-Cast]. For rule [IT-Cast], the proof proceeds by cases over the presence (or lack) of consistent subtyping between T_1 and T_2. □

Theorem 26 (Preservation)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T$, *and* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *for* $\Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$, *then* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}'\colon T'$ *for some* $T' <: T$ *and* $\exists \Sigma' \supseteq \Sigma$.

Proof. By structural induction over the typing derivation and the applicable evaluation rules. All the new evaluation rules apply to casts, so the only interesting case again is for typing with [IT-Cast], where for most cases the conclusion follows directly from the typing derivation of the premise, since the rules do not modify terms but extract a subexpression instead. The case for failing casts is also trivial thanks to rule [IT-Error]. □

Theorem 27 (Translation preserves typing)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T$ *in the source language then* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}'\colon T$ *in the internal language.*

Proof. By structural induction over the translation rules. There is no interesting details for the proof beyond the details presented in the proof of theorem 17. □

$\boxed{\Xi; \Gamma \vdash \mathrm{e} \colon T}$

$$\textsc{IT-App}\;\frac{\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1\colon \left(T_1 \xrightarrow{\Xi_1} T_3\right) \quad \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2 \quad \textbf{\textit{strict-check}}_{\bullet\bullet}(\Xi) \quad T_1 \xrightarrow{\Xi_1} T_3 <\colon T_2 \xrightarrow{\Xi} T_3}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2\colon T_3}$$

$$\textsc{IT-Loc}\;\frac{\Sigma(l) = T}{\Xi;\Gamma;\Sigma \vdash l\colon \texttt{Ref}\ T}$$

$$\textsc{IT-Var}\;\frac{\Gamma(x) = T}{\Xi;\Gamma;\Sigma \vdash x\colon T}$$

$$\textsc{IT-Cast}\;\frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T_0 \quad T_0 <\colon T_1}{\Xi;\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e}\colon T_2}$$

$$\textsc{IT-Has}\;\frac{(\Phi \cup \Xi);\Gamma;\Sigma \vdash \mathrm{e}\colon T}{\Xi;\Gamma;\Sigma \vdash \texttt{has}\ \Phi\ \mathrm{e}\colon T}$$

$$\textsc{IT-Error}\;\frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error}\colon T}$$

$$\textsc{IT-Rst}\;\frac{\Xi_1;\Gamma;\Sigma \vdash \mathrm{e}\colon T \quad \Xi_1 \leq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e}\colon T}$$

$$\textsc{IT-Ref}\;\frac{\widetilde{\mathbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}\colon T \quad \textbf{\textit{strict-check}}_{\texttt{ref}\ \bullet}(\Xi)}{\Xi;\Gamma;\Sigma \vdash \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T}$$

$$\textsc{IT-Deref}\;\frac{\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ T \quad \textbf{\textit{strict-check}}_{!\bullet}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !\mathrm{e}\colon T}$$

$$\textsc{IT-Asgn}\;\frac{\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ T_1 \quad \widetilde{\mathbf{adjust}}_{\bullet:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2 \quad \textbf{\textit{strict-check}}_{\bullet:=\bullet}(\Xi) \quad T_2 <\colon T_1}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon \texttt{Unit}}$$

Figure 6.4: Typing rules for the internal language

$$\text{E-Ref}\ \dfrac{\mathbf{check}_{\texttt{ref}\ \bullet}(\Phi) \qquad l \notin \operatorname{dom}(\mu)}{\Phi \vdash (\texttt{ref}\ v) \mid \mu \to l \mid \mu[l \mapsto v]} \qquad \text{E-Asgn}\ \dfrac{\mathbf{check}_{\bullet := \bullet}(\Phi)}{\Phi \vdash (l := v) \mid \mu \to \texttt{unit} \mid \mu[l \mapsto v]} \qquad \text{E-Deref}\ \dfrac{\mathbf{check}_{!\bullet}(\Phi) \qquad \mu(l) = v}{\Phi \vdash !l \mid \mu \to v \mid \mu}$$

$$\text{E-Frame}\ \dfrac{\mathbf{adjust}_{A(f)}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash f[\mathrm{e}] \mid \mu \to f[\mathrm{e}'] \mid \mu'} \qquad \text{E-Error}\ \dfrac{}{\Phi \vdash g[\mathsf{Error}] \mid \mu \to \mathsf{Error} \mid \mu} \qquad \begin{array}{l}\text{E-Has-T}\\ \dfrac{\Phi' \subseteq \Phi \qquad \Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \texttt{has}\ \Phi'\ \mathrm{e}' \mid \mu'}\end{array}$$

$$\begin{array}{l}\text{E-Has-V}\\ \dfrac{}{\Phi \vdash \texttt{has}\ \Phi'\ v \mid \mu \to v \mid \mu}\end{array} \qquad \begin{array}{l}\text{E-Has-F}\\ \dfrac{\Phi' \not\subseteq \Phi}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \mathsf{Error} \mid \mu}\end{array} \qquad \text{E-Rst-V}\ \dfrac{}{\Phi \vdash \texttt{restrict}\ \Xi\ v \mid \mu \to v \mid \mu}$$

$$\text{E-Rst}\ \dfrac{\Phi'' = \max\{\Phi' \in \gamma(\Xi) \mid \Phi' \subseteq \Phi\} \qquad \Phi'' \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \to \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'} \qquad \begin{array}{l}\text{E-App}\\ \dfrac{\mathbf{check}_{\bullet\bullet}(\Phi)}{\Phi \vdash (\lambda x\colon T_1\ .\ \mathrm{e})\ v \mid \mu \to [{}^{v_{\varepsilon_2}}/_{x}]\,\mathrm{e} \mid \mu}\end{array}$$

$$\begin{array}{l}\text{E-Cast-Frame}\\ \dfrac{\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \mid \mu \to \langle T_2 \Leftarrow T_1\rangle \mathrm{e}' \mid \mu'}\end{array} \qquad \begin{array}{l}\text{E-Cast-Id}\\ \dfrac{}{\Phi \vdash \langle T \Leftarrow T\rangle v \mid \mu \to v \mid \mu}\end{array} \qquad \begin{array}{l}\text{E-Cast-Merge}\\ \dfrac{}{\Phi \vdash \langle T_2 \Leftarrow \texttt{Dyn}\rangle\langle \texttt{Dyn} \Leftarrow T_1\rangle v \mid \mu \to \langle T_2 \Leftarrow T_1\rangle v \mid \mu}\end{array}$$

$$\begin{array}{l}\text{E-Cast-Bad}\\ \dfrac{T_1 \not\lesssim T_2}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle v \mid \mu \to \mathsf{Error} \mid \mu}\end{array}$$

$$\begin{array}{l}\text{E-Cast-Fn}\\ \dfrac{}{\Phi \vdash \langle T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow T_{11} \xrightarrow{\Xi_1} T_{12}\rangle\ (\lambda x\colon T_{01}\ .\ \mathrm{e}) \mid \mu \to (\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\ [{}^{(\langle T_{01} \Leftarrow T_{21}\rangle x)}/_{x}]\,\mathrm{e}) \mid \mu}\end{array}$$

Figure 6.5: Small-step semantics of the internal language

$$\boxed{\Xi; \Gamma \vdash \mathrm{e} \Rightarrow \mathrm{e} \colon T}$$

$$\text{C-Fn}\;\frac{\Xi_1; \Gamma, x \colon T_1; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x \colon T_1 \,.\, \mathrm{e}) \Rightarrow (\lambda x \colon T_1 \,.\, \mathrm{e}') : T_1 \xrightarrow{\Xi_1} T_2} \qquad \text{C-Unit}\;\frac{}{\Xi; \Gamma; \Sigma \vdash \texttt{unit} \Rightarrow \texttt{unit} \colon \texttt{Unit}} \qquad \text{C-Var}\;\frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \Rightarrow x \colon T}$$

$$\text{C-App-1}\;\frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon (T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon T_2 \\ \mathrm{e}''_1 = (\langle\!\langle (T_2 \xrightarrow{\Xi} T_3) \Leftarrow (T_1 \xrightarrow{\Xi_1} T_3) \rangle\!\rangle \mathrm{e}'_1) \\ (T_1 \xrightarrow{\Xi_1} T_3) \lesssim (T_2 \xrightarrow{\Xi} T_3) \\ \widetilde{\mathbf{check}}_{\bullet\bullet}(\Xi) \qquad \Phi = \Delta_{\bullet\bullet}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) : T_3} \qquad \text{C-App-2}\;\frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon \texttt{Dyn} \\ \widetilde{\mathbf{adjust}}_{\bullet\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon T_2 \\ \mathrm{e}''_1 = (\ \langle (T_2 \xrightarrow{\Xi} \texttt{Dyn}) \Leftarrow \texttt{Dyn} \rangle \mathrm{e}'_1\ ) \\ \widetilde{\mathbf{check}}_{\bullet\bullet}(\Xi) \qquad \Phi = \Delta_{\bullet\bullet}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) : \texttt{Dyn}}$$

$$\text{C-Loc}\;\frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l \Rightarrow l \colon \texttt{Ref}\ T}$$

C-Ref

$$\frac{\widetilde{\mathbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \qquad \widetilde{\mathbf{check}}_{\texttt{ref}\ \bullet}(\Xi) \qquad \Phi = \Delta_{\texttt{ref}\ \bullet}(\Xi)}{\Xi; \Gamma; \Sigma \vdash (\texttt{ref}\ \mathrm{e}) \Rightarrow \textit{insert-has?}\big(\Phi, \big(\texttt{ref}\ \mathrm{e}'\big)\big) : \texttt{Ref}\ T}$$

$$\text{C-Deref-1}\;\frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \texttt{Ref}\ T \\ \widetilde{\mathbf{check}}_{!\bullet}(\Xi) \qquad \Phi = \Delta_{!\bullet}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \Rightarrow \textit{insert-has?}(\Phi, !\mathrm{e}') : T}$$

C-Deref-2

$$\frac{\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \texttt{Dyn} \qquad \widetilde{\mathbf{check}}_{!\bullet}(\Xi) \qquad \Phi = \Delta_{!\bullet}(\Xi)}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} \Rightarrow \textit{insert-has?}\Big(\Phi, !\ \langle \texttt{Ref Dyn} \Leftarrow \texttt{Dyn} \rangle \mathrm{e}'\Big) : \texttt{Dyn}}$$

C-Eff

$$\frac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T \qquad \Xi_1 \sqsubseteq \Xi \qquad \Phi = (|\Xi_1| \setminus |\Xi|)}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e} :: \Xi_1) \Rightarrow \textit{insert-has?}\big(\Phi, \texttt{restrict}\ \Xi_1\ \mathrm{e}'\big) : T}$$

$$\text{C-Asgn-1}\;\frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon \texttt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\bullet:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon T_2 \\ \widetilde{\mathbf{check}}_{\bullet:=\bullet}(\Xi) \qquad T_2 \lesssim T_1 \qquad \Phi = \Delta_{\bullet:=\bullet}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2) \Rightarrow \textit{insert-has?}\Big(\Phi, \Big(\mathrm{e}'_1\ :=\ \langle\!\langle T_1 \Leftarrow T_2 \rangle\!\rangle \mathrm{e}'_2\Big)\Big) : \texttt{Unit}}$$

C-Asgn-2

$$\frac{\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1 \colon \texttt{Dyn} \qquad \widetilde{\mathbf{adjust}}_{\bullet:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2 \colon T_2 \qquad \widetilde{\mathbf{check}}_{\bullet:=\bullet}(\Xi) \qquad T_2 \lesssim T_1 \qquad \Phi = \Delta_{\bullet:=\bullet}(\Xi)}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2) \Rightarrow \textit{insert-has?}\Big(\Phi, \Big(\Big(\langle \texttt{Ref}\ T_2 \Leftarrow \texttt{Dyn} \rangle \mathrm{e}'_1\Big) := \mathrm{e}'_2\Big)\Big) : \texttt{Unit}}$$

Figure 6.6: Translation of source programs to the internal language

$$\phi \in \mathbf{Priv}, \quad \xi \in \mathbf{CPriv} = \mathbf{Priv} \cup \{¿\}$$
$$\Phi \in \mathbf{PrivSet} = \mathcal{P}(\mathbf{Priv}), \quad \Xi \in \mathbf{CPrivSet} = \mathcal{P}(\mathbf{CPriv})$$
$$\varepsilon \in \mathbf{Tags}\ .\ \pi \in \mathcal{P}(\mathbf{Tags})$$

$$\begin{array}{lcll}
w & ::= & \texttt{unit} \mid \lambda x\colon T\ .\ \mathrm{e} \mid l & \text{Prevalues} \\
v & ::= & w_\varepsilon & \text{Values} \\
\mathrm{e} & ::= & x \mid v \mid \mathrm{e}\ \mathrm{e} \mid \mathrm{e} :: \Xi & \text{Terms} \\
 & & \mid (\texttt{ref}\ \mathrm{e})_\varepsilon \mid !\mathrm{e} \mid (\mathrm{e} := \mathrm{e})_\varepsilon & \\
T & ::= & \pi\ \rho \mid \texttt{Dyn} & \text{Types} \\
\rho & ::= & \texttt{Unit} \mid T \xrightarrow{\Xi} T \mid \texttt{Ref}\ T & \text{PreTypes} \\
A & ::= & \downarrow\uparrow \mid \pi \downarrow \mid \texttt{ref}\ \downarrow \mid !\downarrow & \text{Adjust Contexts} \\
 & & \mid \downarrow:=\uparrow \mid \pi :=\downarrow & \\
C & ::= & \pi\ \pi \mid \texttt{ref}\pi \mid !\pi \mid \pi := \pi & \text{Check Contexts}
\end{array}$$

Figure 6.7: Syntax of the source language

6.2 System with tags with conservative assumptions

We have already introduced gradual typing for a simplified version of gradual effect checking without tag annotations. In this section we describe the challenges introduced by combining gradual typing and tag annotations. Tag annotations interact with types and the type system. For example, the tag set of the function type in an application is used to generate the context of privileges available for typing the argument. To avoid diving into these details from the start, we have separated the introduction of gradual typing as presented in previous section from the interactions between gradual typing and tag annotations, topic on which we focus in this section.

Tag annotations are part of types. A type contains a set of tags that restrict the valid tag annotations for values in that type. Thus any proposal for gradual typing in a system with tag annotations must be explicit about what tags are assumed valid in the context of a type that does not explicitly declare a set of tags, like type `Dyn`. We take advantage of the tag monotonicity restrictions imposed by the generic M&M framework to conservatively assume the universe of tag annotations as a valid set for these contexts.

6.2.1 Tags and gradual typing

Gradual effect checking inherits a novel definition of types from the generic M&M framework on which it builds [15]. In gradual effect checking, types are the combination of a set of tag annotations and a pretype ρ. The set of tag annotations in a particular type denotes which tag annotations a value may carry to have the declared type: values with other tag annotations cannot have that type. The pretype denotes our previous definition for types:

the system contains a `Unit` pretype, reference pretypes (`Ref` T) and function pretypes $T\xrightarrow{\Xi}T$. Both reference and function pretypes are defined recursively with respect to types, so they contain nested tuples of pretypes and tag annotation sets.

Gradual typing extends a type system to permit programs missing type information, which in the context of gradual effect checking forces `Dyn` to represent a type, not a pretype. If `Dyn` were a pretype instead, programs would require tag annotations in every type, forcing static checking of tag annotations. This requirement defeats the idea of gradual typing, which is to combine both static and dynamic checking in programs.

We introduce the syntax for the gradual type-and-effect language in fig. 6.7. The only difference between the gradual type-and-effect language and the language introduced for gradual effect checking in fig. 4.1 is the introduction of the `Dyn` type.

Since tags affect the definition of types, we need to introduce new definitions for the gradual typing concepts based on types that were presented in the previous section, like type consistency and consistent subtyping.

Definition 20 (Type Consistency)**.** *We define type consistency as a reflexive relation, so we introduce the necessary changes to the relation to be reflexive with respect to tags:*

$$\textit{C-Refl}\,\frac{}{\tau \sim \tau} \qquad \textit{C-UnR}\,\frac{}{\tau \sim \texttt{Dyn}} \qquad \textit{C-UnL}\,\frac{}{\texttt{Dyn} \sim \tau}$$

$$\textit{C-Fun}\,\frac{\sigma_1 \sim \tau_1 \qquad \sigma_2 \sim \tau_2 \qquad \Xi_\sigma \simeq \Xi_\tau}{\pi\left(\sigma_1 \xrightarrow{\Xi_\sigma} \sigma_2\right) \sim \pi\left(\tau_1 \xrightarrow{\Xi_\tau} \tau_2\right)}$$

The consistency relation is defined over *types*, not over pretypes. The function pretype is recursively defined in terms of two types ($T\xrightarrow{\Xi}T$, with T a type), so rule [C-Fun] can ensure that the relation is reflexive for function *types* simply by restricting the tag set on the related types to be the same, and falling back to requiring structural type consistency between the types in the function pretype.

For subtyping, we just preserve the subtyping relation introduced in chapter 4, treating `Dyn` as being neutral to subtyping as Siek and Taha do in [23]:

$$\text{ST-Id}\,\frac{\pi_1 \subseteq \pi_2}{\pi_1\rho <: \pi_2\rho} \qquad \text{ST-Abs}\,\frac{T_3 <: T_1 \qquad T_2 <: T_4 \qquad \Xi_1 \subseteq \Xi_2 \qquad \pi_1 \subseteq \pi_2}{\pi_1(T_1\xrightarrow{\Xi_1}T_2) <: \pi_2(T_3\xrightarrow{\Xi_2}T_4)}$$

$$\text{ST-Dyn}\,\frac{}{\texttt{Dyn} <: \texttt{Dyn}}$$

As in section 6.1, we combine subtyping with type consistency to provide consistent subtyping. Though at first glance both definitions may look the same, the reader must not forget that the difference lays on the definitions of subtyping ($<:$) and type consistency ($\sim$) just introduced, as consistent subtyping is defined in terms of these definitions.

$$\boxed{\Xi; \Gamma; \Sigma \vdash \mathrm{e} : T}$$

$$\text{T-Fn}\ \frac{\Xi_1; \Gamma, x : T_1; \Sigma \vdash \mathrm{e} : T_2}{\Xi; \Gamma; \Sigma \vdash (\lambda x : T_1\ .\ \mathrm{e})_{\varepsilon} : \{\varepsilon\} T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{T-Unit}\ \frac{}{\Xi; \Gamma; \Sigma \vdash \mathtt{unit}_{\varepsilon} : \{\varepsilon\}\mathtt{Unit}} \qquad \text{T-Loc}\ \frac{\Sigma(l) = T}{\Xi; \Gamma; \Sigma \vdash l_{\varepsilon} : \{\varepsilon\}\mathtt{Ref}\ T} \qquad \text{T-Var}\ \frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x : T}$$

$$\text{T-App}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 : T \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 : T_2 \\ T_2 \sim \pi_2\rho_2 \qquad T \lesssim \pi_1(T_2 \xrightarrow{\Xi} T_3) \qquad \widetilde{\mathbf{check}}_{\pi_1\pi_2}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 : T_3} \qquad \text{T-Eff}\ \frac{\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} : T \qquad \Xi_1 \,\widetilde{\sqsubseteq}\, \Xi}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e} :: \Xi_1) : T}$$

$$\text{T-Ref}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} : T \\ T \sim \pi\rho \qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\ \pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathtt{ref}\ \mathrm{e})_{\varepsilon} : \{\varepsilon\}\mathtt{Ref}\ \pi\rho} \qquad \text{T-Deref}\ \frac{\begin{array}{c} \widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e} : T_1 \\ T_1 \sim \pi\mathtt{Ref}\ T \qquad \widetilde{\mathbf{check}}_{!\pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash !\mathrm{e} : T}$$

$$\text{T-Asgn}\ \frac{\begin{array}{cc} \widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 : T & T \sim \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 : T_2 & T_2 \sim \pi_2\rho_2 \\ \widetilde{\mathbf{check}}_{\pi_1:=\pi_2}(\Xi) & T_2 \lesssim T_1 \end{array}}{\Xi; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_{\varepsilon} : \{\varepsilon\}\mathtt{Unit}}$$

Figure 6.8: Typing rules for the source language

Definition 21 (Consistent Subtyping)**.** *Consistent subtyping ($\lesssim$) is defined as*

$$a \lesssim b \iff \exists \alpha \sim a\ .\ \alpha <: b \text{ or } \exists \beta \sim b\ .\ a <: \beta$$

We use type consistency and consistent subtyping in our definition of a type system for the language as presented in fig. 6.8. The intermediate language syntax is introduced in fig. 6.9, and only differs from the syntax for the intermediate language presented in the previous section in the usage of tag annotations. This is only the only difference arising in the type system for the intermediate language, which is introduced in fig. 6.10.

Like in the previous section, the operational semantics introduced in fig. 6.11 is the semantics of the generic gradual effect checking framework (in this case, making full usage of tag annotations) with extensions to manage casts related to type `Dyn`. We introduce rule [E-Cast-Merge] to reduce casts that go through the `Dyn` type, rule [E-Cast-Dyn] to eliminate identity casts for type `Dyn`, a case that is not managed by rule [E-Cast-Id], and rule [E-Cast-Bad] for casts that should fail at runtime and generate an error.

Section 6.2.2 presents the interactions between gradual typing and tag annotations that the translation algorithm must take into consideration. The translation algorithm itself is introduced in section 6.2.3, where we describe the rationale behind the design of the different translation rules.

$$\begin{array}{lcll}
v & ::= & \ldots \mid \langle T \Leftarrow T\rangle v & \text{Values} \\
\mathrm{e} & ::= & \ldots \mid \mathsf{Error} \mid \langle T \Leftarrow T\rangle \mathrm{e} & \text{Terms} \\
 & & \mid \texttt{has}\ \Phi\ \mathrm{e} \mid \texttt{restrict}\ \Xi\ \mathrm{e} & \\
f & ::= & \square\ \mathrm{e} \mid v\ \square \mid (\texttt{ref}\ \square)_\varepsilon & \text{Frames} \\
 & & \mid !\square \mid (\square := \mathrm{e})_\varepsilon \mid (w_\varepsilon := \square)_\varepsilon & \\
g & ::= & f \mid \langle T_2 \Leftarrow T_1\rangle \square \mid \texttt{has}\ \Phi\ \square & \text{Error Frames} \\
 & & \mid \texttt{restrict}\ \Xi\ \square &
\end{array}$$

Figure 6.9: Syntax of the internal language

$$\boxed{\Xi;\Gamma \vdash \mathrm{e} \colon T}$$

$$\text{IT-App}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 : \pi_1\left(T_1 \xrightarrow{\Xi_1} T_3\right) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi) \qquad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <: \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3}$$

$$\text{IT-Loc}\;\frac{\Sigma(l) = T}{\Xi;\Gamma;\Sigma \vdash l \colon \texttt{Ref}\ T} \qquad \text{IT-Var}\;\frac{\Gamma(x) = T}{\Xi;\Gamma;\Sigma \vdash x \colon T} \qquad \text{IT-Cast}\;\frac{\Xi;\Gamma;\Sigma \vdash \mathrm{e} \colon T_0 \qquad T_0 <: T_1}{\Xi;\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \colon T_2}$$

$$\text{IT-Has}\;\frac{(\Phi \cup \Xi);\Gamma;\Sigma \vdash \mathrm{e} \colon T}{\Xi;\Gamma;\Sigma \vdash \texttt{has}\ \Phi\ \mathrm{e} \colon T} \qquad \text{IT-Error}\;\frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error} \colon T}$$

$$\text{IT-Rst}\;\frac{\Xi_1;\Gamma;\Sigma \vdash \mathrm{e} \colon T \qquad \Xi_1 \leq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e} \colon T} \qquad \text{IT-Ref}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \colon \pi\rho \\ \textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ \mathrm{e})_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho}$$

$$\text{IT-Deref}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \colon \pi\texttt{Ref}\ T \\ \textbf{\textit{strict-check}}_{!\pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !\mathrm{e} \colon T} \qquad \text{IT-Asgn}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 : \pi_1\texttt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <: T_1\end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Figure 6.10: Typing rules for the internal language

$$\text{E-Ref}\ \frac{\textbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi) \qquad l \notin \operatorname{dom}(\mu)}{\Phi \vdash (\texttt{ref}\ w_{\varepsilon_1})_{\varepsilon_2} \mid \mu \to l_{\varepsilon_2} \mid \mu[l \mapsto w_{\varepsilon_1}]}
\qquad
\text{E-Asgn}\ \frac{\textbf{check}_{\{\varepsilon_1\}:=\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (l_{\varepsilon_1} := w_{\varepsilon_2})_{\varepsilon} \mid \mu \to \texttt{unit}_{\varepsilon} \mid \mu[l \mapsto w_{\varepsilon_2}]}$$

$$\text{E-Deref}\ \frac{\textbf{check}_{!\{\varepsilon\}}(\Phi) \qquad \mu(l) = v}{\Phi \vdash !l_{\varepsilon} \mid \mu \to v \mid \mu}
\qquad
\text{E-Frame}\ \frac{\textbf{adjust}_{A(f)}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash f[\mathrm{e}] \mid \mu \to f[\mathrm{e}'] \mid \mu'}
\qquad
\text{E-Error}\ \frac{}{\Phi \vdash g[\mathsf{Error}] \mid \mu \to \mathsf{Error} \mid \mu}$$

$$\overset{\text{E-Has-T}}{\frac{\Phi' \subseteq \Phi \qquad \Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \texttt{has}\ \Phi'\ \mathrm{e}' \mid \mu'}}
\qquad
\overset{\text{E-Has-V}}{\frac{}{\Phi \vdash \texttt{has}\ \Phi'\ v \mid \mu \to v \mid \mu}}
\qquad
\overset{\text{E-Has-F}}{\frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \to \mathsf{Error} \mid \mu}}$$

$$\text{E-Rst-V}\ \frac{}{\Phi \vdash \texttt{restrict}\ \Xi\ v \mid \mu \to v \mid \mu}
\qquad
\text{E-Rst}\ \frac{\Phi'' = \max\{\Phi' \in \gamma(\Xi) \mid \Phi' \subseteq \Phi\} \qquad \Phi'' \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \texttt{restrict}\ \Xi\ \mathrm{e} \mid \mu \to \texttt{restrict}\ \Xi\ \mathrm{e}' \mid \mu'}$$

$$\overset{\text{E-App}}{\frac{\textbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (\lambda x\colon T_1\ .\ \mathrm{e})_{\varepsilon_1}\ v_{\varepsilon_2} \mid \mu \to [^{v_{\varepsilon_2}}/_{x}]\,\mathrm{e} \mid \mu}}
\qquad
\overset{\text{E-Cast-Frame}}{\frac{\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e} \mid \mu \to \langle T_2 \Leftarrow T_1\rangle \mathrm{e}' \mid \mu'}}
\qquad
\overset{\text{E-Cast-Id}}{\frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2\rho \Leftarrow \pi_1\rho\rangle w_{\varepsilon} \mid \mu \to w_{\varepsilon} \mid \mu}}$$

$$\overset{\text{E-Cast-Merge}}{\frac{}{\Phi \vdash \langle T_2 \Leftarrow \texttt{Dyn}\rangle\langle \texttt{Dyn} \Leftarrow T_1\rangle v \mid \mu \to \langle T_2 \Leftarrow T_1\rangle v \mid \mu}}
\qquad
\overset{\text{E-Cast-Dyn}}{\frac{}{\Phi \vdash \langle \texttt{Dyn} \Leftarrow \texttt{Dyn}\rangle v \mid \mu \to v \mid \mu}}
\qquad
\overset{\text{E-Cast-Bad}}{\frac{T_1 \not\lesssim T_2}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle v \mid \mu \to \mathsf{Error} \mid \mu}}$$

E-Cast-Fn

$$\frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2 T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow \pi_1 T_{11} \xrightarrow{\Xi_1} T_{12}\rangle\,(\lambda x\colon T_{01}\ .\ \mathrm{e})_{\varepsilon} \mid \mu \to (\lambda x\colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle \texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\ [^{(\langle T_{01} \Leftarrow T_{21}\rangle x)}/_{x}]\,\mathrm{e})_{\varepsilon} \mid \mu}$$

Figure 6.11: Small-step semantics of the internal language

6.2.2 Tags interact with the translation algorithm

The job of the translation algorithm is to introduce the runtime checks necessary to ensure that a program is safe. The translation algorithm makes the optimistic assumptions of the original language explicit by inserting effect checks (`restrict` and `has` constructs) and type casts. In gradual effect checking, a type contains a set of tags that restricts the tag annotations a value with that type may have. But what tags are valid for the case when we do not statically know the type of an expression (the case of type `Dyn`)?

We cannot avoid this question to introduce gradual typing for type-and-effect systems. In the generic type-and-effect framework, the tag sets derived from typing subexpressions are repeatedly used to define check and adjust contexts required to type the expression. If we revisit the type system presented in fig. 6.8, tag sets from subexpression types are used repeatedly, either to define the $\widetilde{\textbf{check}}$ predicates as in rules [T-App], [T-Ref], [T-Deref] and [T-Asgn], or to define adjust contexts that generate consistent privilege sets, sets which are then required as context to type e_2 subexpressions in rules [T-App] and [T-Asgn]. Any assumption for a tag set in the context of a `Dyn` type must ensure safety for these uses, so that $\widetilde{\textbf{check}}$ predicates hold and that the privileges provided by $\widetilde{\textbf{adjust}}$ are always available.

To generate a safe system, we make the conservative assumption that an expression with a type `Dyn` may at runtime have *any* tag annotation. Therefore, we assume that an expression with type `Dyn` must provide the *universe set of tag annotations* (denoted **Tags**). This assumption is also sustained by the tag monotonicity restrictions part of gradual effect checking, which were inherited from the generic M&M framework. If we use the set **Tags**, these monotonicity restrictions ensure that the required restrictions for ***strict-check*** predicates and $\widetilde{\textbf{adjust}}$ functions will always hold in the ways required to provide type safety for the system.

6.2.3 Rules for the translation algorithm

In this section, we introduce the translation algorithm for gradual type-and-effect systems proposed in fig. 6.12 and fig. 6.13. As already mentioned in the previous section, the translation assumes the universe of tag annotations in the situations when types do not provide tag information (the case of `Dyn`).

Whenever possible, we use the tag information available on types instead of recurring to the assumptions introduced in the previous section. To do so, we introduce two separate translation rules for (`ref` e) and !e constructs (when e has type `Dyn` or does not, respectively), and to four separate translation rules for application and assignment expressions (when e_1 and e_2 have type `Dyn` or do not, respectively).

In the case of `ref` e and !e constructs, rules [C-Ref-1], [C-Ref-2], [C-Deref-1] and [C-Deref-2] handle type and tag assumptions. In gradual typing, there was no need to introduce separate rules for `ref` e constructs. We introduce separate rules to limit the case where assumptions for the check context are required. In rule [C-Ref-2], when e has type `Dyn`, the

$\boxed{\Xi;\Gamma \vdash \mathrm{e} \Rightarrow \mathrm{e}\colon T}$

$$\text{C-Fn}\ \frac{\Xi_1;\Gamma, x\colon T_1;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T_2}{\Xi;\Gamma;\Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e})_\varepsilon \Rightarrow (\lambda x\colon T_1\ .\ \mathrm{e}')_\varepsilon \colon \{\varepsilon\}T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{C-Unit}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \mathtt{unit}_\varepsilon \Rightarrow \mathtt{unit}_\varepsilon\colon \{\varepsilon\}\mathtt{Unit}} \qquad \text{C-Var}\ \frac{\Gamma(x) = T}{\Xi;\Gamma;\Sigma \vdash x \Rightarrow x\colon T}$$

$$\text{C-Loc}\ \frac{\Sigma(l) = T}{\Xi;\Gamma;\Sigma \vdash l_\varepsilon \Rightarrow l_\varepsilon\colon \{\varepsilon\}\mathtt{Ref}\ T}$$

$$\text{C-App-1}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1\colon \pi_1(T_1 \xrightarrow{\Xi_1} T_3)\\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2\colon \pi_2\rho_2\\ \mathrm{e}''_1 = (\langle\!\langle \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3)\rangle\!\rangle \mathrm{e}'_1)\\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3)\\ \widetilde{\mathbf{check}}_{\pi_1\pi_2}(\Xi) \qquad \Phi = \Delta_{\pi_1\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) : T_3}$$

$$\text{C-App-2}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1\colon \pi_1(T_1 \xrightarrow{\Xi_1} T_3)\\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2\colon \mathtt{Dyn}\\ \mathrm{e}''_1 = (\langle\!\langle \pi_1(T_1 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3)\rangle\!\rangle \mathrm{e}'_1)\\ \Xi_1 \sqsubseteq\!\!\!\sim \Xi \qquad \widetilde{\mathbf{check}}_{\pi_1\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\pi_1\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}\Big(\Phi, \mathrm{e}''_1\ (\langle\!\langle T_1 \Leftarrow \mathtt{Dyn}\rangle\!\rangle \mathrm{e}'_2)\Big) : T_3}$$

$$\text{C-App-3}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1\colon \mathtt{Dyn}\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2\colon \pi_2\rho_2\\ \mathrm{e}''_1 = (\ \langle\!\langle \mathbf{Tags}(\pi_2\rho_2 \xrightarrow{\Xi} \mathtt{Dyn}) \Leftarrow \mathtt{Dyn}\rangle\!\rangle\ \mathrm{e}'_1)\\ \widetilde{\mathbf{check}}_{\mathbf{Tags}\pi_2}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) : T_3}$$

$$\text{C-App-4}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}'_1\colon \mathtt{Dyn}\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}'_2\colon \mathtt{Dyn}\\ \mathrm{e}''_1 = (\ \langle\!\langle \mathbf{Tags}(\mathtt{Dyn} \xrightarrow{\Xi} T_3) \Leftarrow \mathtt{Dyn}\rangle\!\rangle\ \mathrm{e}'_1)\\ \widetilde{\mathbf{check}}_{\mathbf{TagsTags}}(\Xi) \qquad \Phi = \Delta_{\mathbf{TagsTags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \textit{insert-has?}(\Phi, \mathrm{e}''_1\ \mathrm{e}'_2) : T_3}$$

C-Eff
$$\frac{\Xi_1;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T \qquad \Xi_1 \sqsubseteq\!\!\!\sim \Xi \qquad \Phi = (|\Xi_1| \setminus |\Xi|)}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e} :: \Xi_1) \Rightarrow \textit{insert-has?}(\Phi, \mathtt{restrict}\ \Xi_1\ \mathrm{e}') : T}$$

Figure 6.12: Translation of source programs to the internal language, part I

$\boxed{\Xi; \Gamma \vdash \mathrm{e} \Rightarrow \mathrm{e} \colon T}$

C-Ref-1

$$\frac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\,\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \pi\rho \qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\,\pi}(\Xi) \qquad \Phi = \Delta_{\mathtt{ref}\,\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\mathtt{ref}\ \mathrm{e})_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, (\mathtt{ref}\ \mathrm{e}')_\varepsilon) : \{\varepsilon\}\mathtt{Ref}\ \pi\rho}$$

C-Ref-2

$$\frac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\,\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \mathtt{Dyn} \qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\ \mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\mathtt{ref}\ \mathbf{Tags}}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\mathtt{ref}\ \mathrm{e})_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, (\mathtt{ref}\ \mathrm{e}')_\varepsilon) : \{\varepsilon\}\mathtt{Ref}\ \mathtt{Dyn}}$$

$$\text{C-Deref-1}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \pi\mathtt{Ref}\ T \\ \widetilde{\mathbf{check}}_{!\pi}(\Xi) \qquad \Phi = \Delta_{!\pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !\mathrm{e} \Rightarrow \textit{insert-has?}(\Phi, !\mathrm{e}') : T}$$

$$\text{C-Deref-2}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon \mathtt{Dyn} \\ \widetilde{\mathbf{check}}_{!\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{!\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !\mathrm{e} \Rightarrow \textit{insert-has?}\Big(\Phi,\ !\langle \mathbf{Tags}(\mathtt{Ref\ Dyn}) \Leftarrow \mathtt{Dyn}\rangle \mathrm{e}'\Big) : T}$$

C-Asgn-1

$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon \pi_2\rho_2 \qquad \widetilde{\mathbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1 \qquad \Phi = \Delta_{\pi_1:=\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, (\mathrm{e}_1' := \mathrm{e}_2')_\varepsilon) : \{\varepsilon\}\mathtt{Unit}}$$

C-Asgn-2

$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon \mathtt{Dyn} \qquad \widetilde{\mathbf{check}}_{\pi_1:=\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\pi_1:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \Rightarrow \textit{insert-has?}\Big(\Phi, \big(\mathrm{e}_1' := \langle T_1 \Leftarrow \mathtt{Dyn}\rangle \mathrm{e}_2'\big)_\varepsilon\Big) : \{\varepsilon\}\mathtt{Unit}}$$

C-Asgn-3

$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon \mathtt{Dyn} \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon \pi_2\rho_2 \qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\pi_2}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}:=\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \Rightarrow \textit{insert-has?}\Big(\Phi, \Big(\big(\langle \mathbf{Tags}(\mathtt{Ref\ Dyn}) \Leftarrow \mathtt{Dyn}\rangle \mathrm{e}_1'\big) := \mathrm{e}_2'\Big)_\varepsilon\Big) : \{\varepsilon\}\mathtt{Unit}}$$

C-Asgn-4

$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon \mathtt{Dyn} \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon \mathtt{Dyn} \qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon \Rightarrow \textit{insert-has?}\Big(\Phi, \Big(\big(\langle \mathbf{Tags}(\mathtt{Ref\ Dyn}) \Leftarrow \mathtt{Dyn}\rangle \mathrm{e}_1'\big) := \mathrm{e}_2'\Big)_\varepsilon\Big) : \{\varepsilon\}\mathtt{Unit}}$$

Figure 6.13: Translation of source programs to the internal language, part II

translation rule uses a check context `ref` **Tags** for the $\widetilde{\textbf{check}}$ predicate and for the Δ function that collects the missing privileges (if any) to perform allocation. For !e constructs, like in gradual typing, we have two separate translation rules. Rule [C-Deref-2] makes explicit the assumption that e should be a reference cell (or that e should have type $\pi\texttt{Ref}\ T$ for some π and T) and the assumptions required for the check contexts used in $\widetilde{\textbf{check}}$ and Δ as in rule [C-Ref-2].

There are four translation rules for function applications. Rule [C-App-1] is exactly the same rule [C-App-1] used in the generic gradual effect system, and handles gradual effect checking assuming that types for the function and the argument are known. Rule [C-App-2] handles the case when the argument type is unknown (`Dyn`), and inserts a cast for the argument from `Dyn` to the type of the function parameter. It makes the assumption that e'_2 might hold any particular set of tags, so the **check** and Δ functions use the universe of tags available for the argument side. Rule [C-App-2] needs a cast on the function side for gradual effect checking, cast that performs an effect coercion ensuring that the privileges required for the function type (Ξ_1) are actually available on the context Ξ.

Rule [C-App-3] handles the case where the expression in the function position of an application has type `Dyn`. We assume that the type of the argument is known, leaving the case of both elements having type `Dyn` for rule [C-App-4]. In rule [C-App-3], we do not have tag information about the function type, information that is needed to adjust the set of privileges available to translate e_2. As in rule [C-App-2], we make the assumption of carrying the maximum set of tags, because monotonicity ensures that a maximum set of tags provides the minimum set of privileges. This set is also used for the **check** and Δ functions, and for the cast that ensures that e_1 is a function at all. By our definition of subtyping, a function which has any set of tags π_1 (and the appropriate privilege set and parameter and return types) will pass the cast since always $\pi_1 \subseteq \textbf{Tags}$.

Rule [C-App-4] makes the assumptions from rule [C-App-3] explicit, but also it assumes that the argument has type `Dyn`. This is done separately because if e'_2 has type `Dyn`, we need to make tag assumptions also for the argument to generate the check contexts required by **check** and Δ.

Analogous assumptions are made for translation of assignment expressions in rules [C-Asgn-1], [C-Asgn-2], [C-Asgn-3] and [C-Asgn-4].

6.2.4 Type safety

In this section we provide proof sketches for type safety of the intermediate language we have introduced. Detailed versions of these proofs are provided in appendix E.

Theorem 28 (Progress)**.** *Suppose* $\Xi; \emptyset; \Sigma \vdash e \colon T$. *Then either* e *is a value* v*, an* Error*, or* $\Phi \vdash e \mid \mu \rightarrow e' \mid \mu'$ *for all privilege sets* Φ *such that* $\exists \Phi' \in \gamma(\Xi)$ *such that* $\Phi' \subseteq \Phi$ *and for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$.

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash e \colon T$. The proof strategy is equivalent to the one followed in the case without tags, making use of the tag monotonicity lemmas to ensure restrictions for **check** and **adjust**. □

Theorem 29 (Preservation)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T$*, and* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *for* $\Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$*, then* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}' \colon T'$ *for some* $T' <\colon T$ *and* $\exists \Sigma' \supseteq \Sigma$*.*

Proof. By structural induction over the typing derivation and the applicable rules. The proof follows as in the system without tags, since there is no particularly interesting interaction. □

Theorem 30 (Translation preserves typing)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T$ *in the source language then* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}' \colon T$ *in the internal language.*

Proof. This is the only interesting theorem for tags, since the major differences with the tagless framework arise for the translation algorithm. The extra casts introduced assign tags in a way that typing in the intermediate language is preserved. □

6.3 Summary

This chapter describes the final contribution of our work, a system that integrates gradual typing with gradual effect checking. This system combines static and dynamic checking of effect and type annotations for generic type-and-effect systems.

We have isolated the combination of gradual typing and gradual effect checking in section 6.1 from the analysis of the interactions between tag annotations and gradual typing in section 6.2. This separation highlights the minimal interactions that arise between gradual typing and gradual effect checking when tag annotations are not required for an effect discipline.

At this point, we have introduced all the technical contributions of our work. The next and final chapter describes on the conclusions that arise from our research.

Chapter 7

Conclusions

7.1 Contributions

In this work we have proven that gradual type-and-effect systems are both possible and sound. We addressed the challenges of combining gradual typing with type-and-effect systems, introducing new relations among types and language constructs to provide gradual enforcement of an effect discpline. We also explored and presented how these new features interact with previous ideas to provide complete gradual typing for type-and-effect systems.

Gradual effect checking encapsulates the concepts and restrictions required to provide gradual effect annotations and for allowing the migration from unannotated programs towards a static effect discipline. Introducing gradual typing in a system with gradual effect checking is practically orthogonal, because every interaction between gradual typing and gradual effect checking can be represented in terms of previously defined concepts and does not require new language constructions. Even the challenges introduced by the increased expressivity of the generic framework with tag annotations prove self contained: we provided a simpler explanation of gradual typing for type-and-effect systems in terms of a simplified gradual effect checking system without tag-related features, and later moved on to address these challenges.

Part of our work has already been published in an international peer-reviewed ACM conference. The paper "A theory of Gradual Effect Systems" [2] presents our proposal for generic gradual effect checking. Our work in gradual typing for type-and-effect systems is yet to be published. We believe our work to also be of interest for language designers, implementors and researchers interested in providing more flexible and more expresive type disciplines, as well as for those interested in the theoretical background of novel language features and their design.

This work adapts for type-and-effect systems the core idea behind gradual typing, the introduction of a consistency relation to loosen static type restrictions. Consistency provides flexibility with its lack of transitivity. To ensure type safety, gradual typing requires runtime checks that verify the transitivity of consistent static assumptions. This notion of consistency may be applied to different type disciplines and concepts, as we do in this work for effect annotations. By introducing a definition for consistency in a previously unexplored context (effect annotations), we show that there is still interesting research to be done in the theory of gradual typing, both to analyze the application of gradual ideas on different unexplored type disciplines, and to extract from these different applications a more general theory that encompasses the core concepts of gradual typing and guides their application in a particular type discipline.

We have followed both lines of exploration for the restricted domain of type-and-effect systems, both introducing gradual typing for a particular type-and-effect system and providing guidelines for generic applications of the core concepts of gradual effect checking using the abstract interpretation framework. The abstract interpretation framework provides a clear and sound justification for our design intuitions. It provided a generalization of gradual effects that was sufficient to introduce gradual effect checking as a new feature for a generic type-and-effect framework. With abstract interpretation, gradual effect checking did not require any further restrictions to the generic framework on which we based our work. In future work we want to explore broader applications of abstract interpretation in the context of gradual typing.

Reasoning about side effects introduces further complexity for the design of a type system, and we must take this complexity into consideration to provide developers with the flexibility for migrating unnatotated programs towards a static effect discipline. As has been presented in previous work, effect annotations interact in nontrivial ways with type annotations, specifically for the case of function type annotations. These interactions also arise in our design of gradual effect checking, where we combine language constructions from the literature used to verify type restictions at runtime (type casts) and we introduce new operations to verify effect restrictions at runtime. Interactions between types and effects led us to propose novel semantics for the behavior of higher-order casts (type casts between function types), which must take into consideration the side effect restrictions that must be imposed and verified at runtime.

Previous work on gradual typing has presented type casts as performing a double function, both "run-time type checking and coercing" [22]. Both functions must also be performed for the effect discipline, but we provided different constructs for each action, `has` for effect checking and `restrict` for effect coercing. We provide separate constructs to clarify concepts in our proposed framework.

We provide a specification for gradual effect checking, so that we may both focus in the novel ideas introduced by the language and provide guarantees for the language behavior (primarily type safety). A type safety proof requires carrying some extra information at runtime, information also used by the runtime effect checking structures to provide the appropriate runtime behavior. In a fully annotated program, this extra runtime information is made redundant by the static guarantees provided by the type system, but this information must be available at runtime when a program makes use of gradual effect annotations, so that runtime checks are properly performed. We attempted to provide choices for developers interested in implementing our framework by designing two different operational semantics that handle the tradeoff between the need for runtime tag information and the full preservation of the original effect discipline.

7.2 Future work

We believe that the following research ideas may be explored in future work:

- **Blame for effects**. In the event of a runtime inconsistency related to the effect discipline, the languages we have so far introduced produce an Error expression. This construction makes our language simple, but does not provide the programmer with feedback about the source of the runtime inconsistency or failure. In gradual typing, a similar issue arises for type casts, and solutions like the blame calculus and the coercion calculus have been proposed. These approaches take into consideration the fact that the lazy semantics of higher order casts may produce a late cast failure, which might relate to a cast error in a non local portion of the program. A blame system should point to the original source of failure to provide developers with appropriate feedback. A blame system also characterizes which programs should never be signaled as the cause of a failure (a blame theorem). We believe that gradual effect checking would benefit from a blame system, and the design of such a system is a line of work we certainly hope to pursue in the near future.
- **Gradual tags**. Our exposition of gradual typing for type-and-effect systems made conservative assumptions about tag annotations for types. Though a thorough exploration of gradual typing might benefit of the development of an extended version of our generic gradual effect checking system with gradual tags, it is yet to be determined if type-and-effect systems that make use of tag annotations would benefit at all from a notion of gradual tags that would permit interactions between programs with and without tag annotations. This research question must be addressed before attempting to build a sound system, for which we believe our generic gradual effect framework may serve as a starting point.
- **Abstract interpretation for gradual typing** Throughout this work we benefit from the use of abstract interpretation to present and justify the core concepts of gradual effect checking in a generic way. We believe that abstract interpretation ideas may be introduced to find patterns in other representations of gradual typing and to provide a simpler base to present and define gradual typing in general, from which to derive generic guidelines to introduce gradual typing for any particular type discipline.
- **Mining software repositories for source code that may benefit from a gradual effect systems**. We recognize that our work lacks an empirical validation as a useful tool for programmers. Our hypothesis is that analyzing software repositories would provide concrete examples of the use of type-and-effect systems that may benefit from gradual effect checking, as well as cases where the restrictive nature of type-and-effect systems makes programmers avoid declaring effect annotations and where migration would become feasible with the introduction of gradual effect checking. This analysis would also provide interesting feedback on how programmers reason about side effects throughout the evolution of software projects.
- **Implement our concepts in a practical programming language with an ef-**

fect system. We are interested in making our work available for programmers to use in a practical programming language. An implementation may also aid software engineering researchers in evaluating the impact of the use of our proposal, and aid to empirically evaluate the claim that gradual effect checking improves the use and declaration of type-and-effect disciplines by software developers. Though we did not pursue this line of research yet, there are some programming languages which may work as an interesting starting point to implement gradual effect checking. Scala, an increasingly popular programming language, already provides a form of generic effect systems, which could be extended with gradual effect checking. Koka, a research-oriented programming language with a type-and-effect system developed by Microsoft, may be another particularly interesting candidate to consider.

- **Type-and-effect systems for gradual typing**. In this work we have explored the combination of gradual typing and type-and-effect systems by introducing a type-and-effect system and then presenting gradual typing as a language extension. We conjecture that this approach should be equivalent to having started with a language that has gradual typing, and extending it with a type-and-effect system. Since we have not explored this road yet, the equivalence is not proven. We are not certain if designing a type-and-effect system for a gradual typing system sheds new light on the interactions between gradual typing and type-and-effect systems or if it is trivial and redundant from the work we have already presented.

Bibliography

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*, pages 201–214, Austin, Texas, USA, January 2011. ACM Press.

[2] Felipe Bañados, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *ICFP*, 2014.

[3] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(0):135 – 154, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Information and Computation*, pages 51–67. Springer-Verlag, 1988.

[5] Luca Cardelli. Type systems. *Handbook of Computer Science and Engineering*, 1997.

[6] Alonso Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[7] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):pp. 346–366, April 1932.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[9] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[10] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Int. Conf. on Functional Programming*, October 2002.

[11] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP*, pages 28–38, 1986.

[12] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201 – 206, 1994.

[13] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *HOSC*, volume 23, pages 167–189, 2010.

[14] Daan Leijen. Koka: Programming with row polymorphic effect types. In *Mathematically Structured Functional Programming 2014*. EPTCS, March 2014.

[15] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI*, pages 39–50, 2009.

[16] Robin Milner. A theory of type polymorphism in programming. In *JCSS*, pages 348–375, 1978.

[17] Andrew Myers. Evaluation contexts: Lecture notes. Retrieved from www.cs.cornell.edu/Courses/cs6110/2011sp/lectures/lecture09.pdf, February 2011.

[18] Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification, Version 2.11.*

[19] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[20] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258. Springer Berlin Heidelberg, 1980.

[21] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*, pages 258–282, 2012.

[22] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In *ESOP*, 2009.

[23] Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.

[24] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *SFP*, pages 81–92, 2006.

[25] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, 2010.

[26] Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *PEPM*, pages 45–53, 1995.

[27] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP*, 2009.

[28] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.

Appendix A

The Fluent Language as an Instantiation of Marino and Millstein's Generic Type-and-Effect System

The Fluent Language and the type system in fig. 2.3 can be considered as an instantiation of Marino and Millstein's generic type-and-effect system. Though this claim is discussed in section 4.3 of the original paper [15], Marino and Millstein neither formalize the semantics of fluent languages nor present a proof of instantiation.

We prove the fluent language as an instance of the generic framework to show that fluent is a simple example that can drive our discussion of gradual effect checking and that if the same intuitions are applied to the generic framework, we can get back the original fluent system as an instantiation.

We also aim to show in this section a formal example of how to use Marino and Millstein and the requirements that the framework imposes.

By using Marino and Millstein's framework, a sound type system (and semantics) can be obtained for a type-and-effect system just by defining a set of privileges, the particular extra syntax of the system (in our case, effect ascription) and two particular functions, check and adjust, which will manage privileges for all language constructs and will be part of their type system.

When introducing the fluent language, the privilege domain was already defined ($\Phi \subseteq \{\texttt{alloc}, \texttt{read}, \texttt{write}\}$). We propose the following check and adjust functions for the system:

$$\mathbf{check}_C(\Phi) = \begin{cases} \{\texttt{alloc}\} \subseteq \Phi & \text{if } C = \texttt{ref}\ \ \pi \\ \{\texttt{read}\} \subseteq \Phi & \text{if } C = !\pi \\ \{\texttt{write}\} \subseteq \Phi & \text{if } C = \pi_1 := \pi_2 \\ \Phi_1 \subseteq \Phi & \text{if } C = \pi :: \Phi_1 \\ \mathbf{true} & \text{otherwise} \end{cases}$$

$$\mathbf{adjust}_A(\Phi) = \begin{cases} \Phi_1 & \text{if } A = \downarrow :: \Phi_1 \text{ and } \Phi_1 \subseteq \Phi \\ \Phi & \text{otherwise} \end{cases}$$

To ensure type safety, the framework requires the user to prove the following properties about the proposed **check** and **adjust** functions:
Lemma 31. *If* $\Phi_1 \subseteq \Phi_2$ *and* $\mathbf{check}_C(\Phi_1)$*, then* $\mathbf{check}_C(\Phi_2)$

Proof. By cases on the structural definition of Check Contexts and using transitivity of $\subseteq$. □

Proof. (Detailed) Let's recall how a Check Context is defined.

$$\begin{array}{rcl} C & ::= & \pi_1\ \pi_2 \\ & | & \texttt{ref}\ \pi \\ & | & !\pi \\ & | & \pi_1 := \pi_2 \\ & | & \pi :: \Phi_1 \end{array}$$

Since our definition of **check** does not take into account the tagsets, we can structurally prove by cases on the definition of check contexts and our definition of the check function.

- $C = \texttt{ref}\ \pi$ Then if $\mathbf{check}_C(\Phi_1)$, then $\{\texttt{alloc}\} \subseteq \Phi_1$. Since $\Phi_1 \subseteq \Phi_2$, then by transitivity of $\subseteq$ also $\{\texttt{alloc}\} \subseteq \Phi_2$ and thus $\mathbf{check}_C(\Phi_2)$.
- $C = !\pi$. Analogous to $C = \texttt{ref}\ \pi$.
- $C = \pi_1 := \pi_2$. Analogous to $C = \texttt{ref}\ \pi$.
- $C = \pi :: \Phi$. If $\mathbf{check}_C(\Phi_1)$, then $\Phi \subseteq \Phi_1$. Since $\Phi_1 \subseteq \Phi_2$, then also $\Phi \subseteq \Phi_2$ and thus $\mathbf{check}_C(\Phi_2)$.
- Rest of the cases. $\mathbf{check}_C(\Phi)$ holds for any Φ, thus always $\mathbf{check}_C(\Phi_2)$.

□

Lemma 32. *If* $\Phi_1 \subseteq \Phi_2$*, then adjust*$(\Phi_1, A) \subseteq$ *adjust*(Φ_2, A)

Proof. By cases on the structural definition of adjust contexts.

- $A = \downarrow :: \Phi'$. If $\Phi' \subseteq \Phi_1$, then $\mathbf{adjust}_A(\Phi_1) = \Phi'$. By transitivity of $\subseteq$, then $\Phi' \subseteq \Phi_2$ too and $\mathbf{adjust}_A(\Phi_2) = \mathbf{adjust}_A(\Phi_1)$.

  If $\Phi' \not\subseteq \Phi_1$, then $\Phi_1 \subset \Phi'$ and $\mathbf{adjust}_A(\Phi_1) = \Phi_1$. If $\Phi_1 \subseteq \Phi_2$, then either $\mathbf{adjust}_A(\Phi_2) = \Phi'$ or $\mathbf{adjust}_A(\Phi_2) = \Phi_2$, but both cases are $\supseteq \mathbf{adjust}_A(\Phi_1)$.

- Rest of the cases. Direct from the definition of $\mathbf{adjust}_A(\Phi) = \Phi$

$\square$

Lemma 33. *If* $C_2 \sqsubseteq C_1$ *and* $\mathbf{check}_{C_1}(\Phi)$*, then* $\mathbf{check}_{C_2}(\Phi)$

Proof. Trivial since our definition of **check** depends only on Φ and not in C. $\square$

Lemma 34. *If* $A_2 \sqsubseteq A_1$*, then* $\mathbf{adjust}_{A_1}(\Phi) \subseteq \mathbf{adjust}_{A_2}(\Phi)$

Proof. Like in the case of lemma 33, this case is also trivial since our definition of **adjust** depends only on Φ and not on A. $\square$

To make the instantiation even more clear, in fig. A.1 we map the original rules in Marino and Millstein generic system to the rules in fig. 2.3. It is clear that the rules map one to one after discarding the unused tag annotations.

$$\text{T-Fn}\ \frac{\Phi_1;\Gamma,x\colon\tau_1;\Sigma\vdash \mathrm{e}\colon\tau_2}{\Phi;\Gamma;\Sigma\vdash(\lambda x.\mathrm{e})_\varepsilon\colon\{\varepsilon\}(\tau_1\xrightarrow{\Phi_1}\tau_2)} \quad\Rightarrow\quad \text{T-Fn}\ \frac{\Phi_1;\Gamma,x\colon T_1;\Sigma\vdash \mathrm{e}\colon T_2}{\Phi;\Gamma;\Sigma\vdash(\lambda x\colon T_1\ .\ \mathrm{e})\colon T_1\xrightarrow{\Phi_1}T_2}$$

$$\text{T-Unit}\ \frac{}{\Phi;\Gamma;\Sigma\vdash \texttt{unit}_\varepsilon\colon\{\varepsilon\}\texttt{Unit}} \quad\Rightarrow\quad \text{T-Unit}\ \frac{}{\Phi;\Gamma;\Sigma;\vdash \texttt{unit}\colon\texttt{Unit}}$$

$$\text{T-Loc}\ \frac{\Sigma(l)=\tau}{\Phi;\Gamma;\Sigma\vdash l_\varepsilon\colon\{\varepsilon\}\mathrm{Ref}\ \tau} \quad\Rightarrow\quad \text{T-Loc}\ \frac{\Sigma(l)=T}{\Phi;\Gamma;\Sigma\vdash l\colon\texttt{Ref}\ T}$$

$$\text{T-Var}\ \frac{\Gamma(x)=\tau}{\Phi;\Gamma;\Sigma\vdash x\colon\tau} \quad\Rightarrow\quad \text{T-Var}\ \frac{\Gamma(x)=T}{\Phi;\Gamma;\Sigma\vdash x\colon T}$$

$$\text{T-App}\ \frac{\begin{array}{c}\mathbf{adjust}_{\downarrow\uparrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\colon\pi_1(\tau_2\xrightarrow{\Phi_1}\tau)\\ \mathbf{adjust}_{\pi_1\ \downarrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}_2\colon\pi_2\rho_2\\ \mathbf{check}_{\pi_1\ \pi_2}(\Phi) \qquad \pi_2\rho_2<:\tau_2 \qquad \Phi_1\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\ \mathrm{e}_2\colon\tau} \quad\Rightarrow\quad \text{T-App}\ \frac{\begin{array}{c}\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\colon T_1\xrightarrow{\Phi_1}T_3\\ \Phi;\Gamma;\Sigma\vdash \mathrm{e}_2\colon T_2\\ T_2<:T_1 \qquad \Phi_1\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\ \mathrm{e}_2\colon\tau_3}$$

$$\text{T-Ref}\ \frac{\begin{array}{c}\mathbf{adjust}_{\texttt{ref}\ \downarrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}\colon\tau\\ \tau=\pi\rho \qquad \mathbf{check}_{\texttt{ref}\ \pi}(\Phi)=\{\texttt{alloc}\}\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash(\texttt{ref}\ \mathrm{e})_\varepsilon\colon\{\varepsilon\}\texttt{Ref}\ \tau} \quad\Rightarrow\quad \text{T-Ref}\ \frac{\begin{array}{c}\Phi;\Gamma;\Sigma\vdash \mathrm{e}\colon T\\ \{\texttt{alloc}\}\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash \texttt{ref}\ \ \mathrm{e}\colon\texttt{Ref}\ T}$$

$$\text{T-Deref}\ \frac{\begin{array}{c}\mathbf{adjust}_{!\uparrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}\colon\pi\texttt{Ref}\ \ \tau\\ \mathbf{check}_{!\pi}(\Phi)=\{\texttt{read}\}\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash(!\mathrm{e})_\varepsilon\colon\tau} \quad\Rightarrow\quad \text{T-Deref}\ \frac{\begin{array}{c}\Phi;\Gamma;\Sigma\vdash \mathrm{e}\colon\texttt{Ref}\ T\\ \{\texttt{read}\}\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash !\mathrm{e}\colon T}$$

$$\text{T-Assign}\ \frac{\begin{array}{c}\mathbf{adjust}_{\downarrow:=\uparrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\colon\pi_1\mathrm{Ref}\ \tau_1\\ \mathbf{adjust}_{\pi_1:=\downarrow}(\Phi)=\Phi \qquad \Phi;\Gamma;\Sigma\vdash \mathrm{e}_2\colon\pi_2\rho_2\\ \mathbf{check}_{\pi_2:=\pi_2}(\Phi)=\{\texttt{write}\}\subseteq\Phi \qquad \pi_2\rho_2<:\tau_1\end{array}}{\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1:=\mathrm{e}_2\colon\{\varepsilon\}\texttt{Unit}} \quad\Rightarrow\quad \text{T-Assign}\ \frac{\begin{array}{c}\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1\colon\texttt{Ref}\ T_1\\ \Phi;\Gamma;\Sigma\vdash \mathrm{e}_2\colon T_2\\ \{\texttt{write}\}\subseteq\Phi \qquad T_2<:T_1\end{array}}{\Phi;\Gamma;\Sigma\vdash \mathrm{e}_1:=\mathrm{e}_2\colon\texttt{Unit}}$$

$$\text{T-Ascription}\ \frac{\begin{array}{c}\mathbf{adjust}_{\downarrow::\Phi_1}(\Phi)=\Phi_1 \qquad \Phi_1\vdash \mathrm{e}\colon\tau\\ \tau=\pi\rho \qquad \mathbf{check}_{\pi::\Phi_1}(\Phi)=\Phi_1\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma\vdash \mathrm{e}::\Phi_1\colon\tau} \quad\Rightarrow\quad \text{T-Ascription}\ \frac{\begin{array}{c}\Phi_1;\Gamma;\Sigma\vdash \mathrm{e}\colon T\\ \Phi_1\subseteq\Phi\end{array}}{\Phi;\Gamma;\Sigma;\vdash \mathrm{e}::\Phi_1\colon T}$$

Figure A.1: Simplifying the rules of the Marino and Millstein framework to produce an ad-hoc type system for the Fluent Language

Appendix B

Soundness Proof for Gradual Effect Fluent

In this section we prove some statements and theorems about the Gradual Effect Fluent language. Any program typable in the Fluent language gets the same type in Gradual Effect Fluent, and its translation won't insert any `check` or `adjust` constructs. We also prove type soundness as presented by [28], by introducing and proving theorems for Progress and Preservation in the language.

We have already introduced the statements and theorems for type safety with section 3.5.1. The following section shows proofs in detail.

Whenever there is a risk of confusion, like in theorem 36, we introduce subscripts to distinguish operations defined both for the fluent language (with a $_F$ subscript) and for the gradual effect fluent language (with a $_{GEF}$ subscript).

Lemma 35. *(Preorder)*

If $a \subseteq b$, then $a \precsim b$.

Proof. Since $a \subseteq b$, $|a| \subseteq |b|$, then $a \precsim b$. □

Theorem 36. *(Consistency of Fluent with Gradual Effect Fluent)*

If $\Phi; \Gamma; \Sigma \vdash_F \mathrm{e} \colon T$, then $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e} \colon T$.

Proof. By induction on the structure of $\vdash_F$.

Case (T-Unit)**.** *By* [T-Unit]$_{GEF}$*,* $\Phi; \Gamma; \Sigma \vdash_{GEF}$ `unit` : `Unit`*.*

Case (T-Fn)**.** *By the induction hypothesis,* $\Phi_1; \Gamma, x \colon T_1; \Sigma \vdash_{GEF} \mathrm{e} \colon T_2$*. We can thus use* [T-Fn]$_{GEF}$ *and* $\Phi; \Gamma; \Sigma \vdash_{GEF} (\lambda x \colon T_1 \,.\, \mathrm{e}) \colon T_1 \xrightarrow{\Phi_1} T_2$*.*

Case (T-Loc)**.** *, As* $\Phi; \Gamma; \Sigma \vdash_F l \colon \mathtt{Ref}\ T$*,* $\Sigma(l) = T$*. Then we can use the rule* [T-Loc]$_{GEF}$ *and* $\Phi; \Gamma; \Sigma \vdash_{GEF} l \colon \mathtt{Ref}\ T$*.*

Case (T-Var). *Analogous to [T-Loc].*

Case (T-Ref). *Since* $\Phi; \Gamma; \Sigma \vdash_F \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T$*, then* $\{\texttt{alloc}\} \subseteq \Phi$*. By The Preorder Lemma, then* $\{\texttt{alloc}\} \sqsubseteq \Phi$*.*

By Induction Hypothesis $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e}\colon T$*. Then we can use the rule* $[\text{T-Ref}]_{GEF}$ *and* $\Phi; \Gamma; \Sigma \vdash_{GEF} \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T$*.*

Case (T-Deref). *Analogous to [T-Ref].*

Case (T-Assign). *. As in [T-Ref], by the Preorder Lemma,* $\{\texttt{alloc}\} \sqsubseteq \Phi$*. As* $[\text{T-Assign}]_F$ *applies,* $T_2 <: T_1$*. We can then use the* $[\text{T-Assign}]_{GEF}$ *rule since all other premises are fullfilled by induction hypothesis.*

Case (T-App). *Since* $\Phi; \Gamma; \Sigma \vdash_F \mathrm{e}_1\ \mathrm{e}_2\colon T_3$*, by inversion lemmas we know that* $\exists T_1, T_2, \Phi_1$ *such that* $T_2 <: T_1$ *,* $\Phi_1 \subseteq \Phi$*,* $\Phi; \Gamma; \Sigma \vdash_F \mathrm{e}_1\colon T_1 \xrightarrow{\Phi_1} T_3$ *and* $\Phi; \Gamma; \Sigma \vdash_F \mathrm{e}_2\colon T_2$*.*

Then, by induction hypothesis, $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e}_1\colon T_1 \xrightarrow{\Phi_1} T_3$ *and* $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e}_2\colon T_2$*.*

We can then apply the $[\text{T-App}]_{GEF}$ *rule and then* $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e}_1\ \mathrm{e}_2\colon T_3$*.*

Case (T-Ascription). *Since* $\Phi; \Gamma; \Sigma; \vdash_F \mathrm{e} :: \Phi_1\colon T$*, then* $\Phi_1 \subseteq \Phi$*. By the Preorder Lemma, then* $\Phi_1 \sqsubseteq \Phi$*.*

By Induction Hypothesis $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e}\colon T$*. Then we can use the rule* $[\text{T-With-Effects}]_{GEF}$ *and* $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e} :: \Phi_1\colon T$*.*

□

Theorem 37. *(Static Translation)*

If $\Phi; \Gamma; \Sigma \vdash_F \mathrm{e}\colon T$*, then* $\Phi; \Gamma; \Sigma \vdash_{GEF} \mathrm{e} \Rightarrow \mathrm{e}'\colon T$ *with* $\mathrm{e}' \neq \texttt{has}\ \Phi'\ \mathrm{e}''$*.*

Proof. By induction on the structure of $\vdash_F$.

Case (T-Unit). *By [C-Unit],* $\Phi; \Gamma; \Sigma \vdash \texttt{unit} \Rightarrow \texttt{unit}\colon \texttt{Unit}$*.*

And $\texttt{unit} \neq \texttt{has}\ \Phi'\ \mathrm{e}''$*.*

Case (T-Fn). *By Induction Hypothesis, we know that* $\Phi'; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_2\colon T_2$*. Thus we can apply [C-Fn] to know that* $\Phi; \Gamma; \Sigma \vdash (\lambda x\colon T_1\ .\ \mathrm{e}_1) \Rightarrow (\lambda x\colon T_1\ .\ \mathrm{e}_2) T T_1 \xrightarrow{\Phi'} T_2$*.*

$(\lambda x\colon T_1\ .\ \mathrm{e}_2) \neq \texttt{has}\ \Phi'\ \mathrm{e}''$*.*

Case (T-Loc). *By inversion on the hypothesis, we know that* $\Gamma(l) = T$*, then by [C-Loc],* $\Phi; \Gamma; \Sigma \vdash l \Rightarrow l\colon \texttt{Ref}\ T$*.* $l \neq \texttt{has}\ \Phi'\ \mathrm{e}''$

Case (T-Var). *By inversion on the hypothesis, we know that* $\Sigma(x) = T$*. Then by [C-Var],* $\Phi; \Gamma; \Sigma \vdash x \Rightarrow x\colon T$*.* $x \neq \texttt{has}\ \Phi'\ \mathrm{e}''$

Case (T-Ref). *By inversion on the hypothesis, we know that* $\{\texttt{alloc}\} \subseteq \Phi$ *and by Induction Hypothesis,* $\Phi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T$*. By the preorder lemma,* $\{\texttt{alloc}\} \sqsubseteq \Phi$*.*

Since $\{\mathsf{alloc}\} \subseteq \Phi$*, insert-has?*$\{\mathsf{alloc}\}\ \Phi\ \mathrm{e} = \mathrm{e}$*, so we can infer that* $\Phi; \Gamma; \Sigma \vdash \texttt{ref}\ \mathrm{e} \Rightarrow \texttt{ref}\ \mathrm{e}' \colon \texttt{Ref}\ T$*.*

$\texttt{ref}\ \mathrm{e}' \neq \texttt{has}\ \Phi'\ \mathrm{e}''$

Case (T-Deref)**.** *Analogous to [T-Ref].*

Case (T-Assign)**.** *By inversion on the hypothesis and induction hypothesis, we know that* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon \texttt{Ref}\ T_1$*,* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2$*,* $\{\texttt{write}\} \subseteq \Phi$ *and* $T_2 <: T_1$*.*

We can only use [C-Assign] to transform $\mathrm{e}_1 := \mathrm{e}_2$*, and since insert-has?*$\{\texttt{write}\}\ \Phi\ \mathrm{e} = \mathrm{e}$*,* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2 \Rightarrow \mathrm{e}_1' := \mathrm{e}_2' \colon \texttt{Unit}$*.*

$\mathrm{e}_1' := \mathrm{e}_2' \neq \texttt{has}\ \Phi'\ \mathrm{e}''$

Case (T-App)**.** *By inversion on the hypothesis and induction hypothesis, we know that* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rightarrow \mathrm{e}_1' \colon T_1 \xrightarrow{\Phi_1} T_3$*,* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_2 \Rightarrow \mathrm{e}_2' \colon T_2$*,* $T_2 <: T_1$ *and* $\Phi_1 \subseteq \Phi$*. By preorder lemma, also* $\Phi_1 \precsim \Phi$*.*

Given the definition of privilege sets in the fluent language, $\{¿\} \not\subseteq \Phi$*,* $\{¿\} \not\subseteq \Phi_1$*, thus* $\Phi_1 \subseteq \Phi$ *and insert-has? does not introduce any* ***has*** *constructs. Since rule* $[\text{T-App}]_F$ *uses subtyping, then no casts are inserted either.*

Then by [C-App], $\Phi; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \Rightarrow \mathrm{e}_1'\ \mathrm{e}_2' \colon T_3$*.*

$\mathrm{e}_1'\ \mathrm{e}_2' \neq \texttt{has}\ \Phi'\ \mathrm{e}''$

Case (T-Ascription)**.** *By inversion on the hypothesis and induction hypothesis,we know both that* $\Phi_1; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T$ *and* $\Phi_1 \subseteq \Phi$*. By preorder lemma also* $\Phi_1 \precsim \Phi$*. We can only use [C-With-Effects] to translate and, since* $\Phi_1 \subseteq \Phi$*, insert-has?*$\Phi_1\ \Phi\texttt{restrict}\ \Phi_1\ \mathrm{e} = \texttt{restrict}\ \Phi_1\ \mathrm{e}$ *and then infer that* $\Phi; \Gamma; \Sigma \vdash \mathrm{e} :: \Phi_1 \Rightarrow \texttt{restrict}\ \Phi_1\ \mathrm{e}' \colon T$*.*

$\texttt{restrict}\ \Phi_1\ \mathrm{e}' \neq \texttt{has}\ \ \Phi'\ \mathrm{e}''$.

□

B.0.1 Type safety of the intermediate language

Lemma 38. *(Translation preserves well-typing) If* $\Phi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' \colon T$*, then* $\Phi; \Gamma; \Sigma \vdash \mathrm{e}' \colon T$ *in the intermediate language.*

Proof. Straightforward induction on the last step of the translation $\Rightarrow$. □

Lemma 39. *(Inversion of* $\precsim$*) If* $a \precsim b$*, then either*

1. $a \subseteq b$
2. $a \not\subseteq b$*, and either* $(a \setminus \{¿\}) \subseteq b$ *or* $¿ \in b$*.*

Proof. Immediate from the definition of $\precsim$. □

Lemma 40. *(Inversion of Typing for the Intermediate Language)*

1. *If* $\Xi;\Gamma;\Sigma \vdash l\colon T$*, then* $l \in \mathit{dom}(\Sigma)$ *and* $T = \texttt{Ref}\ (\Sigma(l))$.
2. *If* $\Xi;\Gamma;\Sigma \vdash x\colon T$*, then* $x \in \mathit{dom}(\Gamma)$ *and* $T = \Gamma(x)$.
3. *If* $\Xi;\Gamma;\Sigma \vdash \lambda x\colon T_1\ .\ \mathrm{e}\colon T$*, then* $\exists \Xi', T_2\ .\ \Xi';\Gamma,x\colon T_1;\Sigma \vdash \mathrm{e}\colon T_2$ *and* $T = T_1 \xrightarrow{\Xi} T_2$.
4. *If* $\Xi;\Gamma;\Sigma \vdash \texttt{ref}\ \mathrm{e}\colon T$*, then* $\exists T'\ .\ \Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T'$, $T = \texttt{Ref}\ T'$ *and* $\{\texttt{alloc}\} \subseteq \Xi$.
5. *If* $\Xi;\Gamma;\Sigma \vdash !\mathrm{e}\colon T$*, then* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon \texttt{Ref}\ T$*, and* $\{\texttt{read}\} \subseteq \Xi$.
6. *If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1 := \mathrm{e}_2\colon T$*, then* $\exists T_1, T_2\ .\ \Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\colon \texttt{Ref}\ T_1$ *and* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2$, $\{\texttt{write}\} \subseteq \Xi$ *and* $T_2 <: T_1$ *and* $T = \texttt{Unit}$.
7. *If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\,\mathrm{e}_2\colon T$*, then* $\exists T_1, T_2, T_3, \Xi'$ *such that* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_1\colon T_1 \xrightarrow{\Xi'} T_3$, $\Xi;\Gamma;\Sigma \vdash \mathrm{e}_2\colon T_2$, $T_2 <: T_1$, $\Xi' \sqsubseteq_{\sim} \Xi$ *and* $T = T_3$.
8. *If* $\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi'\ \mathrm{e}\colon T$*, then* $\Xi';\Gamma;\Sigma \vdash \mathrm{e}\colon T$ *and* $\Xi' \leq \Xi$.
9. *If* $\Xi;\Gamma;\Sigma \vdash \texttt{has}\ \Phi'\ \mathrm{e}\colon T$*, then* $\Phi' \cup \Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T$.
10. *If* $\Xi;\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1\rangle \mathrm{e}\colon T_2$*, then* $\exists T_0 . T_0 <: T_1$ *such that* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T_0$.

Proof. Immediate from the definition of the typing relation. □

Lemma 41. *(Canonical Forms)*

1. *If* v *is a value of type* $\texttt{Unit}$*, then* $v = \texttt{unit}$ *or* $v = \langle \texttt{Unit} \Leftarrow T_1\rangle v'$.
2. *If* v *is a value of type* $\texttt{Ref}\ T$*, then either* $\exists\, l\ .\ v = l$ *and* $\Sigma(l) = T$ *or* $v = \langle \texttt{Ref}\ T \Leftarrow T_1\rangle v'$.
3. *If* v *is a value of type* $T_1 \xrightarrow{\Phi} T_2$*, then* $v = \lambda x\colon T_1\ .\ \mathrm{e}$ *or* $v = \langle T_1 \xrightarrow{\Phi} T_2 \Leftarrow T\rangle v'$.

Proof. For each type, there is only one possible derivation rule from values to that type. □

Theorem 42 (Progress). *Suppose* e *is a closed, well typed expression (*$\exists\ T, \Sigma, \Xi\ .\ \Xi;\emptyset;\Sigma \vdash \mathrm{e}\colon T$*). Then either* e *is a value, an* Error*, or else, for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$*, and for any privilege set* Φ *such that* $\Xi \vdash \Phi$*, there is some* e' *and* μ' *with* $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$.

Proof. By Structural Induction on the typing derivation using the Inversion of Typing for the Intermediate Language Lemmas and the Inversion of $\sqsubseteq_{\sim}$ lemma. The only interesting case is rule [T-Has] ($\mathrm{e} = \texttt{has}\ \Phi\ \mathrm{e}'$). By structural induction with e', it is either a value, in which case e can always progress with [E-Has-V], an Error, in which case e can always progress with rule [E-Error], or it progresses. In case it progresses, depending on Φ, the system can always choose between rules [E-Has-T] and [E-Has-F] to progress. □

Lemma 43. *(Permutation) If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T$ *and* Δ *is a permutation of* Γ*, then* $\Xi;\Delta;\Sigma \vdash \mathrm{e}\colon T$*. Moreover, the latter derivation has the same depth as the former.*

Proof. As in Pierce [19], by straightforward induction on typing derivations since our extensions to the lambda calculus do not make any special usage of the environment Γ. □

Lemma 44. *(Weakening) If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}\colon T$ *and* $x \notin \mathit{dom}(\Gamma)$*, then* $\Xi;\Gamma,x\colon T';\Sigma \vdash \mathrm{e}\colon T$*. Moreover, the latter derivation has the same depth as the former.*

Proof. As in Pierce [19], by straightforward induction on typing derivations since our extensions to the lambda calculus do not make any special usage of the environment Γ. $\square$

Lemma 45. *(Preservation of Types Under Substitution) If* $\Xi; \Gamma, x\colon T'; \Sigma \vdash \mathrm{e}\colon T$ *and* $\Xi; \Gamma; \Sigma \vdash s\colon T'$*, then* $\Xi; \Gamma; \Sigma \vdash [{}^{s}\!/\!_{x}]\,\mathrm{e}\colon T$.

Proof. By induction on the last step of derivation for the statement $\Xi; \Gamma, x\colon T'; \Sigma \vdash \mathrm{e}\colon T$, as in Pierce [19]. $\square$

Theorem 46. *(Preservation)*

If $\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T$, $\Gamma \mid \Sigma \vDash \mu$ *and* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *with* $\Xi \vdash \Phi$*, then* $\exists \Sigma' \supseteq \Sigma$ *such that* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}'\colon T$ *and* $\Gamma \mid \Sigma' \vDash \mu'$.

Proof. By structural induction over the type derivation $\Xi; \Gamma; \Sigma \vdash \mathrm{e}\colon T$, and then by cases on the rules in relation $\to$ that may apply for terms of the form e accepted by the typing rule.

Case (T-Unit)**.** *By definition of the semantics,* $\nexists \mathrm{e}' \,.\, \texttt{unit} \to \mathrm{e}'$.

Case (T-Fn)**.** *Analogous to [T-Unit].*

Case (T-Loc)**.** *Analogous to [T-Unit].*

Case (T-Var)**.** *Analogous to [T-Unit].*

Case (T-Ref)**.** *If* $\Gamma \mid \Sigma \vDash \mu$, $\Phi \vdash \texttt{ref}\ \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *only by:*

- *[E-Ref]. Then* e *is a value of type* T *and* $\Phi \vdash \texttt{ref}\ \mathrm{e} \mid \mu \to l \mid \mu[l \mapsto \mathrm{e}]$. *We can build* $\Sigma' = \Sigma[l \mapsto T]$*, with* $l \notin dom(\Sigma)$. *By construction* $\Sigma' \supseteq \Sigma$. *Thanks to this extension and the fact that* $\mathrm{e}\colon T$*, we know that* $\Gamma \mid \Sigma' \vDash \mu[l \mapsto v]$. *With* Σ'*,we can use [T-Loc] to infer that* $\Xi; \Gamma; \Sigma' \vdash l\colon \texttt{Ref}\ T$.
- *[E-Frame] with* $f = \texttt{ref}\ \square$ *By Induction Hypothesis, we know that, as* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}'' \mid \mu'$, $\exists \Sigma' \supseteq \Sigma$ *such that* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}''\colon T$ *and* $\Gamma \mid \Sigma' \vDash \mu'$. *Besides that, we know that* $\{\texttt{alloc}\} \subseteq \Phi$ *and* $\{\texttt{alloc}\} \subseteq \Xi$. *We can then use [T-Ref] to infer that* $\Xi; \Gamma; \Sigma' \vdash \texttt{ref}\ \mathrm{e}''\colon \texttt{Ref}\ T$.
- *[E-Error] Trivial by rule* [T-Error].

Case (T-Deref)**.** *We can also analyze the* $\to$ *cases that apply for* $!\mathrm{e}$.

- *[E-Deref]. Then* $\mathrm{e} = l$ *for some* $l \in dom(\Sigma)$. *Since the rule [E-Deref] preserves* μ*, then we can use* $\Sigma' = \Sigma$. *By hypothesis* $\Gamma \mid \Sigma \vDash \mu$*, and since* $v = \mu(l)$ *then* $\Xi; \Gamma; \Sigma \vdash v\colon T$.
- *[E-Frame] with* $f =!\square$ *Analogous to [E-Frame] in [T-Ref].*
- *[E-Error] Trivial by rule* [T-Error].

Case (T-Assign)**.** *If* $\Gamma \mid \Sigma \vDash \mu$*, then* $\Phi \vdash \mathrm{e}_1 := \mathrm{e}_2 \mid \mu \to \mathrm{e}' \mid \mu'$ *only by:*

- *[E-Assign]. Then* $\mathrm{e}_1 = l$ *for some* $l \in dom(\Sigma)$ *and* $\mathrm{e}_2 = v$. *Since* $\Xi; \Gamma; \Sigma \vdash l := v\colon \texttt{Unit}$ *and* $\mathrm{e}' = \texttt{unit}$*, we can know by [T-Unit] that* $\Xi; \Gamma; \Sigma \vdash \texttt{unit}\colon \texttt{Unit}$. *Finally, since* $l \in dom(\Sigma)$ *and* $\Gamma \mid \Sigma \vDash \mu$, $\Gamma \mid \Sigma \vDash \mu[l \mapsto v] = \mu'$.
- *[E-Frame] with* $f = \square := \mathrm{e}$.*Analogous to [E-Frame] in [T-Ref], but using [T-Assign] instead of [T-Ref] and* `write` *instead of* `alloc`.

- *[E-Frame] with $f = v := \square$.Analogous to [E-Frame] with $f = \square := \mathrm{e}$.*
- *[E-Error] Trivial by rule* [T-Error].

Case (T-App). *If $\Gamma \mid \Sigma \vDash \mu$, then $\Phi \vdash \mathrm{e}_1\ \mathrm{e}_2 \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ only by:*

- *[E-App]. By preservation of types under substitution lemma.*
- *[E-Frame] with $f = \square\ \mathrm{e}$. Analogous to [T-Assign] and [E-Frame] with $f = \square := \mathrm{e}$.*
- *[E-Frame] with $f = v\ \square$. Analogous to [T-Assign] and [E-Frame] with $f = v := \square$.*
- *[E-Error] Trivial by rule* [T-Error].

Case (T-Restrict). $\Phi \vdash \texttt{restrict}\ \Xi'\ \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *only by:*

- *[E-Rst-V]. Direct by induction hypothesis.*
- *[E-Rst-1]. Then we can use the induction hypothesis and* [T-Restrict] *again.*
- *[E-Rst-2]. Always $\Xi \vdash |\Xi|$, thus we can use the induction hypothesis and* [T-Restrict] *again.*
- *[E-Frame] with $f = \texttt{restrict}\ \Xi'\ \square$. Analogous to [T-Ref] with [E-Frame].*
- *[E-Error] Trivial by rule* [T-Error].

Case (T-Has). $\Phi \vdash \texttt{has}\ \Phi'\ \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *only by:*

- *[E-Has-T] By Induction hypothesis and using rule* [T-Has] *again.*
- *[E-Has-V] Direct by induction hypothesis.*
- *[E-Has-F] We can always use rule* [T-Error] *to type* $\mathrm{e}' = \mathsf{Error}$.
- *[E-Error] Trivial by rule* [T-Error].

Case (T-Error). *Cannor occur since there is no evaluation rule just for* $\mathrm{e} = \mathsf{Error}$.

Case (T-Cast). $\Phi \vdash \langle T_2 \Leftarrow T_1 \rangle \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ *only by:*

- *[E-Cast-Id] Then is direct by the induction hypothesis.*
- *[E-Frame] with $f = \langle T_2 \Leftarrow T_1 \rangle \square$ Also direct by induction hypothesis and reusing rule* [T-Cast].
- *[E-Cast-Fn]. By applying rules* [T-Has], [T-Restrict] *and* [T-Cast] *and rebuilding the function with* [T-Fn].

□

Appendix C

Detailed Proofs for Generic Gradual Effect Checking

In this appendix we present detailed proofs for the theorems presented in chapter 4. If the reader is not interested in the detailed proofs and considers the higher level descriptions of the proofs already described as sufficient, then the reader may skip this chapter.

Lemma 47. *$\forall \Phi \in \gamma(\Xi)$, $|\Xi| \subseteq \Phi$.*

Proof. By definition of $|\bullet|$,

$$|\Xi| = \bigcap_{\Phi \in \gamma(\Xi)} \Phi$$

and then the lemma follows by definition of intersection. $\square$

Proposition 48. $|\Xi| = \Xi \setminus \{¿\}$

Proof. By cases on the definition of γ.

Case ($¿ \notin \Xi$)**.** *Then* $|\Xi| = \bigcap \{\Xi\} = \Xi = \Xi \setminus \{¿\}$.

Case ($¿ \in \Xi$)**.** *Then* $|\Xi| = \bigcap \{(\Xi \setminus \{¿\}) \cup \Phi \mid \Phi \in \mathcal{P}(\mathbf{PrivSet})\} = \Xi \setminus \{¿\}$.

$\square$

Lemma 49. $|\Xi| \in \gamma(\Xi)$.

Proof. By cases on the definition of γ:

Case ($¿ \notin \Xi$)**.** *Since γ produces a singleton with Ξ, intersection over the singleton retrieves Ξ.*

Case ($¿ \in \Xi$)**.** *Since $\emptyset \in \mathcal{P}(\mathbf{CPrivSet})$, $\Xi \setminus \{¿\} \in \gamma(\Xi)$, which also is the intersection of every possible set in $\gamma(\Xi)$.*

$\square$

Lemma 50. $\Xi_1 \subseteq \Xi_2 \Rightarrow \Xi_1 \leq \Xi_2$

Proof. By proposition 48 and definition of $\subseteq$, $|\Xi_1| \subseteq |\Xi_2|$, which is the definition of $\leq$. ☐

Lemma 51. $\Xi_1 \leq \Xi_2$ *and* $\boldsymbol{strict\text{-}check}_A(\Xi_1) \Rightarrow \boldsymbol{strict\text{-}check}_A(\Xi_2)$

Proof. Since $\boldsymbol{strict\text{-}check}_C(\Xi_1)$, then $\forall\Phi \in \gamma(\Xi_1)$, $\mathbf{check}_C(\Phi)$. In particular, by lemma 49, $\mathbf{check}_C(|\Xi_1|)$. By Privilege Monotonicity property 1 for **check**, therefore, $\mathbf{check}_C(|\Xi_2|)$. Then by property 1 for **check** and by lemma 47, $\mathbf{check}_C(\Phi)\, \forall\Phi \in \Xi_2$ and thus $\boldsymbol{strict\text{-}check}_C(\Xi_2)$. ☐

Lemma 9.
If $\boldsymbol{strict\text{-}check}_C(\Xi_1)$ *and* $\Xi_1 \subseteq \Xi_2$ *then* $\boldsymbol{strict\text{-}check}_C(\Xi_2)$.

Proof. By lemma 50, $\Xi_1 \leq \Xi_2$. Therefore, the lemma follows from lemma 51. ☐

Lemma 52. $|\alpha(\Upsilon)| = \bigcap \Upsilon$, *for* $\Upsilon \neq \emptyset$.

Proof. By cases on the definition of $\alpha(\Upsilon)$.

Case ($\Upsilon = \{\Phi\}$ branch)**.** *then* $\Phi = \alpha(\Upsilon)$, *and since dom*$(\alpha) = \mathcal{P}(\mathbf{PrivSet})$, ¿ $\notin \Phi$. *Therefore* $\gamma(\Phi) = \Upsilon$, *and therefore by definition of* $|\bullet|$, $|\alpha(\Upsilon)| = \bigcap \Upsilon$.

Case (otherwise branch)**.** *Then* $\alpha(\Upsilon) = (\bigcap \Upsilon) \cup \{$¿$\}$.
Thus $|\alpha(\Upsilon)| = \bigcap \{(\bigcap \Upsilon) \cup \Phi \mid \Phi \in \mathcal{P}(\mathbf{PrivSet})\}$ *and thus* $|\alpha(\Upsilon)| = \bigcap \Upsilon$.

☐

Lemma 53. *If* $\bigcap(\Upsilon_1) \in \Upsilon_1$ *and* $\bigcap(\Upsilon_1) \subseteq \bigcap(\Upsilon_2)$, *then* $\bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \Upsilon_1\} \subseteq \bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \Upsilon_2\}$.

Proof. Suppose $\bigcap(\Upsilon_1) \in \Upsilon_1$ and $\bigcap(\Upsilon_1) \subseteq \bigcap(\Upsilon_2)$. Now suppose $\phi \in \bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \Upsilon_1\}$. Then since $\bigcap(\Upsilon_1) \in \Upsilon_1$, in particular $\phi \in \mathbf{adjust}_A(\bigcap(\Upsilon_1))$ too.

Now let $\Phi \in \Upsilon_2$. Since $\bigcap(\Upsilon_1) \subseteq \bigcap(\Upsilon_2)$, it follows that $\bigcap(\Upsilon_1) \subseteq \Phi$. So by monotonicity, $\phi \in \mathbf{adjust}_A(\Phi)$.

Thus, since Φ is arbitrary, $\phi \in \mathbf{adjust}_A(\Phi)$ for all $\Phi \in \Upsilon_2$ and thus $\phi \in \bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \Upsilon_2\}$. ☐

Lemma 10. *If* $\Xi_1 \leq \Xi_2$ *then* $\widetilde{\mathbf{adjust}}_C(\Xi_1) \leq \widetilde{\mathbf{adjust}}_C(\Xi_2)$

Proof. By definition of $\leq$ and $|\bullet|$, $\bigcap(\gamma(\Xi_1)) \subseteq \bigcap(\gamma(\Xi_2))$. Also, by lemma 49, $\bigcap(\gamma(\Xi_1)) \in \gamma(\Xi_1)$. Thus, by lemma 53, $\bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \gamma(\Xi_1)\} \subseteq \bigcap \{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \gamma(\Xi_2)\}$.

Given that by definition of γ, for any Ξ $\gamma(\Xi) \neq \emptyset$, we can infer by lemma 52 that $|\alpha(\{\mathbf{adjust}_A(\Phi) \mid \forall\Phi \in \gamma(\Xi_1)\})| \subseteq |\alpha(\{\mathbf{adjust}_A(\Phi \mid \forall\Phi \in \gamma(\Xi_2))\})|$. By definition of $\widetilde{\mathbf{adjust}}$, this is equivalent to $|\widetilde{\mathbf{adjust}}_A(\Xi_1)| \subseteq |\widetilde{\mathbf{adjust}}_A(\Xi_2)|$, which at the same time is the definition of $\widetilde{\mathbf{adjust}}_A(\Xi_1) \leq \widetilde{\mathbf{adjust}}_A(\Xi_2)$. ☐

Proposition 13.

1. *$\Xi_1 \underset{\sim}{\sqsubseteq} \Xi_2$ if and only if $\Xi_1 \leq \Xi_2$ or ¿ $\in \Xi_2$.*

2. *$\boldsymbol{strict\text{-}check}_C(\Xi)$ if and only if $\mathbf{check}_C(|\Xi|)$.*

Proof. 1.

Case ($\Leftarrow$)**.** *By definition of $\underset{\sim}{\sqsubseteq}$, ¿ $\in \Xi_2 \Rightarrow \mathbf{Priv} \in \gamma(\Xi_2)$, thus for any $\Phi_1 \in \gamma(\Xi_1$, there exists $\mathbf{Priv} \in \gamma(\Xi_2)$ such that $\Phi_1 \subseteq \mathbf{Priv}$.*

Case ($\Rightarrow$)**.** *By definition of $\underset{\sim}{\sqsubseteq}$, there exists $\Phi_1 \in \gamma(\Xi_1)$ and $\Phi_2 \in \gamma(\Xi_2)$ such that $\Phi_1 \subseteq \Phi_2$. By lemma 49, we know that $|\Xi_1| \in \gamma(\Xi_1)$ and $|\Xi_2| \in \gamma(\Xi_2)$. We thus proceed by cases over Φ_2.*

- *$\Phi_2 = |\Xi_2|$. Then $\Phi_1 \subseteq |\Xi_2|$. By lemma 47, $|\Xi_1| \subseteq |\Xi_2|$ and thus $\Xi_1 \leq \Xi_2$.*
- *$\Phi_2 \neq |\Xi_2|$. Then by lemma 49, $\text{card}\,(\gamma(\Xi_2)) > 1$ and by the definition of γ, we can infer that ¿ $\in \Xi_2$.*

2.

Case ($\Rightarrow$)**.** *By definition of $\boldsymbol{strict\text{-}check}$, $\mathbf{check}_C(\Phi)\,\forall\Phi \in \gamma(\Xi)$. By lemma 49, then $\mathbf{check}_C(|\Xi|)$.*

Case ($\Leftarrow$)**.** *By lemma 47 and privilege monotonicity property 1 for $\mathbf{check}$, then $\mathbf{check}_C(\Phi)\,\forall\Phi \in \gamma(\Xi)$, which is the definition of $\boldsymbol{strict\text{-}check}$.*

□

Proposition 14.

1. *If ¿ $\in \Xi$ then $\widetilde{\mathbf{check}}_C(\Xi)$ if and only if $\mathbf{check}_C(\mathbf{PrivSet})$.*
2. *If ¿ $\notin \Xi$ then $\widetilde{\mathbf{check}}_C(\Xi)$ if and only if $\mathbf{check}_C(\Xi)$.*

Proof. 1.

Case ($\Leftarrow$)**.** *Since ¿ $\in \Xi$, then $\mathbf{PrivSet} \in \gamma(\Xi)$. Thus since $\mathbf{check}_C(\mathbf{PrivSet})$, $\exists\Phi = \mathbf{PrivSet} \in \gamma(\Xi) \mid \mathbf{check}_C(\Phi)$, which is the definition of $\widetilde{\mathbf{check}}_C(\Xi)$.*

Case ($\Rightarrow$)**.** *By definition of $\widetilde{\mathbf{check}}$, $\exists\Phi \in \gamma(\Xi) \mid \mathbf{check}_C(\Phi)$. Since $\Phi \in \mathcal{P}\,(\mathbf{PrivSet})$, then $\Phi \subseteq \mathbf{PrivSet}$, and thus by privilege monotonicity property 1 for $\mathbf{check}$, then $\mathbf{check}_C(\mathbf{PrivSet})$.*

2. Since ¿ $\notin \Xi$, $\gamma(\Xi) = \{\Xi\}$. This means that, by definition of $\widetilde{\mathbf{check}}$, $\mathbf{check}_C(\Xi)$.
   Also, by definition of $|\bullet|$, $|\Xi| = \Xi$ and thus $\mathbf{check}_C(|\Xi|)$.

□

Theorem 54. $\Phi \in \gamma(\Xi) \Rightarrow \mathbf{adjust}_A(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_A(\Xi))$.

Proof. Let $\Phi \in \gamma(\Xi)$. Then $\mathbf{adjust}_A(\Phi) \in \{\mathbf{adjust}_A(\Phi') \mid \Phi' \in \gamma(\Xi)\}$.

By proposition 7, $\{\mathbf{adjust}_A(\Phi') \mid \Phi' \in \gamma(\Xi)\} \subseteq \gamma(\alpha(\{\mathbf{adjust}_A(\Phi') \mid \Phi' \in \gamma(\Xi)\}))$, which by definition 10 is equivalent to $\gamma(\widetilde{\mathbf{adjust}}_A(\Xi))$. □

Lemma 55. $\widetilde{\mathbf{check}}_C(\Xi) \Rightarrow \textbf{\textit{strict-check}}_C(\Delta_C(\Xi) \cup \Xi)$

i.e. If $\mathbf{check}_C(\Phi)$ *for some* $\Phi \in \gamma(\Xi)$*, then* $\mathbf{check}_C(\Phi)$ *for every* $\Phi \in \gamma(\Delta_C(\Xi) \cup \Xi)$*.*

Proof. Suppose $\mathbf{check}_C(\Phi)$ for some $\Phi \in \gamma(\Xi)]$

Then $\Upsilon = \{\Phi \in \gamma(\Xi) \mid \mathbf{check}_C(\Phi)\} \neq \emptyset$ so $\Phi = \bigcup \mathit{mins}(\Upsilon)$ exists.

Furthermore, by M & M monotonicity, $\mathbf{check}_C(\Phi)$.

Note that $\Phi \subseteq \Phi \setminus |\Xi| \cup \Xi = \Delta_C(\Xi) \cup \Xi$, so if $\Phi_2 \in \gamma(\Delta_C(\Xi) \cup \Xi)$ then $\Phi \subseteq \Phi_2$ and by M & M monotonicity, $\mathbf{check}_C(\Phi_2)$.

□

Lemma 56. $\Xi_1 \leq (|\Xi_1| \setminus |\Xi_2|) \cup \Xi_2$

Proof. By definition of set complement, $\Phi \cup \Phi^C = \mathbb{U}$. thus $|\Xi_1| = |\Xi_1| \cap (|\Xi_2|^C \cup |\Xi_2|)$. By the law of distributivity among sets, it is equal to $(|\Xi_1| \cap |\Xi_2|^C) \cup (|\Xi_1| \cap |\Xi_2|)$. By definition of set difference, it is equivalent to $(|\Xi_1| \setminus |\Xi_2|) \cup (|\Xi_1| \cap |\Xi_2|) \subseteq (|\Xi_1| \setminus |\Xi_2|) \cup |\Xi_2|)$, by properties of set intersection. By proposition 48, this is equal to $|(|\Xi_1| \setminus |\Xi_2|) \cup \Xi_2|$, and thus $\Xi_1 \leq (|\Xi_1| \setminus \Xi_2) \cup \Xi_2$. □

C.1 Progress and preservation proofs for the gradual effect framework presented in the paper

In this section we introduce proofs for progress and preservation in the Gradual Effect Framework. It also introduces items that will be used at some point in one of the proofs. Note that some required items have already been proven in previous sections.
Lemma 57 (Canonical Values)**.**

1. *If* $\Xi; \Gamma; \Sigma \vdash v \colon \pi\texttt{Unit}$*, then* $\exists \varepsilon \in \pi$ *and* $v = \texttt{unit}_\varepsilon$*.*
2. *If* $\Xi; \Gamma; \Sigma \vdash v \colon \pi T_1 \xrightarrow{\Xi} T_2$*, then* $\exists \varepsilon \in \pi$ *and* $v = (\lambda x \colon T_1 \,.\, \mathrm{e})_\varepsilon$*.*
3. *If* $\Xi; \Gamma; \Sigma \vdash v \colon \pi\texttt{Ref}\ T$*, then* $\exists \varepsilon \in \pi$ *and* $v = l_\varepsilon$ *and* $\Sigma(l) = T$*.*

Proof. The only rules for typing values in our type system are [IT-Loc], [IT-Fn] and [IT-Unit], respectively. They associate the types in the premises with the expressions in the conclusions. □

Theorem 15 (Progress)**.** *Suppose* $\Xi; \emptyset; \Sigma \vdash \mathrm{e} \colon T$*. Then either* e *is a value* v*, an* Error*, or* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *for all privilege sets* Φ *such that* $\exists \Phi' \in \gamma(\Xi)$ *such that* $\Phi' \subseteq \Phi$ *and for any store* μ *such that* $\emptyset \mid \Sigma \vDash \mu$*.*

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash \mathrm{e} \colon T$.

Case ([IT-Unit], [IT-Loc] and [IT-Fn]). $\texttt{unit}_\varepsilon$*,* l_ε *and* $\lambda x\colon T\ .\ \mathrm{e}_\varepsilon$ *are values.*

Case ([IT-Var]). *This case cannot happen by hypothesis.*

Case ([IT-Error]). Error *is an* Error.

Case ([IT-App]). *By Induction Hypothesis, we know that* e_1 *either:*

- $\forall \Phi' \supseteq \Phi'' \in \gamma(\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi))$ *and* $\forall \mu.\emptyset \mid \Sigma \vDash \mu$*,* $\Phi' \vdash \mathrm{e}_1 \mid \mu \to \mathrm{e}_1' \mid \mu'$. *By theorem 54, since* $\Phi \in \gamma(\Xi)$*,* $\mathbf{adjust}_{\downarrow\uparrow}(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi))$. *Then* $\mathbf{adjust}_{\downarrow\uparrow}(\Phi) \vdash \mathrm{e}_1 \mid \mu \to \mathrm{e}_1' \mid \mu'$ *and we can use rule* [E-Frame] *with* $f = \Box \mathrm{e}_2$ *and* $\Phi \vdash \mathrm{e}_1\ \mathrm{e}_2 \mid \mu \to \mathrm{e}_1'\ \mathrm{e}_2 \mid \mu'$.
- $\mathrm{e}_1 = \mathsf{Error}$. *Then rule* [E-Error] *applies with* $g = \Box\ \mathrm{e}$.
- e_1 *is a value. By lemma 57,* $\Xi;\emptyset;\Sigma \vdash \mathrm{e}_1 \colon \pi_1 T_1 \xrightarrow{\Xi_1} T_3$ *means that* $\exists \varepsilon_1 \in \pi_1$ *such that* $\mathrm{e}_1 = (\lambda x\colon T_1\ .\ \mathrm{e})_{\pi_1}$. *Now by Induction Hypothesis, we also know that* e_2 *either:*
  - $\forall \Phi' \supseteq \Phi'' \in \gamma(\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi))$ *and* $\forall \mu.\emptyset \mid \Sigma \vDash \mu$*,* $\Phi' \vdash \mathrm{e}_2 \mid \mu \to \mathrm{e}_2' \mid \mu'$. *In particular, by monotonicity we know that* $\mathbf{adjust}_{\pi_1\downarrow}(\Phi) \subseteq \mathbf{adjust}_{\{\varepsilon\}\downarrow}(\Phi)$ *for* $\varepsilon \in \pi_1$. *Then by arguments analogous to the case for* e_1 *we can use rule* [E-Frame] *with frame* $f = v\Box$ *and thus* $\Phi \vdash \mathrm{e}_1\ \mathrm{e}_2 \mid \mu \to \mathrm{e}_1\ \mathrm{e}_2' \mid \mu'$.
  - $\mathrm{e}_2 = \mathsf{Error}$. *Then rule* [E-Error] *applies with* $g = v\ \Box$.
  - *Is a value. By lemma 57, we only know that* $\exists \varepsilon_2 \in \pi_2$ *such that* $v = w_{\varepsilon_2}$. *This means that* $\{\varepsilon_1\}\{\varepsilon_2\} \sqsubseteq_C \pi_1\pi_2$[1]

    By typing premises, $\boldsymbol{strict\text{-}check}_{\pi_1\pi_2}(\Xi)$. *By definion of* $\boldsymbol{strict\text{-}check}$*, then* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{\pi_1\pi_2}(\Phi)$. *By M & M tag monotonicity lemma, then* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)$ *and thus rule* [E-App] *can be applied.*

Case ([IT-Cast]). *By induction hypothesis, either*

- $\forall \Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\forall \mu.\emptyset \mid \Sigma \vDash \mu$*,* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$. *Then rule* [E-Cast-Frame] *can always be applied.*
- $\mathrm{e} = \mathsf{Error}$. *Then rule* [E-Error] *always applies.*
- e *is a value. Then by lemma 57, either:*
  - $T_0 = \pi_0\texttt{Unit}$. *By lemma 57,* $\mathrm{e} = \texttt{unit}_\varepsilon$ *for some* $\varepsilon \in \pi_0$. *Since* $T_1 \lesssim T_2$*, then* $T_1 = \pi_1\rho_1$*,* $T_2 = \pi_2\rho_2$ *and* $\pi_1 \subseteq \pi_2$. *Thus* [E-Cast-Id] *applies.*
  - $T_0 = \pi_0\texttt{Ref}\ T$. *Analogous to the case for* $\texttt{Unit}$.
  - $T_0 = \pi_0 T_1' \xrightarrow{\Xi_1'} T_3'$. *By analogous arguments,* $\exists \varepsilon \in \pi_0\ .\ \mathrm{e} = (\lambda x\colon T\ .\ \mathrm{e}')_\varepsilon$. *Then rule* [E-Cast-Fn] *applies.*

Case ([IT-Has]). *By induction hypothesis, either*

- $\forall \Phi' \supseteq \Phi''' \in \gamma(\Phi \cup \Xi)$ *and* $\forall \mu.\emptyset \mid \Sigma \vDash \mu$*,* $\Phi' \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$. *Our original expression is of the form* $\texttt{has}\ \Phi''\mathrm{e}$. *We proceed by cases on* Φ''.
  - $\Phi'' \subseteq \Phi'$. *Then rule* [E-Has-T] *can be applied.*
  - $\Phi'' \not\subseteq \Phi'$. *Then rule* [E-Has-F] *can be applied.*
- $\mathrm{e} = \mathsf{Error}$. *Then rule* [E-Error] *can be applied with* $g = \texttt{has}\ \Phi\ \Box$.

[1] $\sqsubseteq_C$ means the partial order on check contexts defined by Marino and Millstein.

- e *is a value. Then rule* [E-Has-V] *can be applied.*

Case ([IT-Rst])**.** *By induction hypothesis, either*

- $\forall \Phi_1 \supseteq \Phi'_1 \in \gamma(\Xi_1)$ *and* $\forall \mu. \emptyset \mid \Sigma \vDash \mu$, $\Phi_1 \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$. *By the typing hypothesis, we know that* $\Xi_1 \leq \Xi$. *Thus* $\exists \Phi_1 \in \gamma(\Xi_1).\Phi_1 \subseteq \Phi, \forall \Phi \in \gamma(\Xi)$, *and therefore also* $\forall \Phi' \supseteq \Phi \in \gamma(\Xi)$ *Thus rule* [E-Rst] *can alwaysbe applied.*
- $\mathrm{e} = \mathsf{Error}$. *Then rule* [E-Error] *can be applied.*
- e *is a value. Then rule* [E-Rst-V] *can be applied.*

Case ([IT-Ref])**.** *By induction hypothesis, either*

- $\forall \Phi \supseteq \Phi' \in \gamma(\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi))$ *and* $\forall \mu. \emptyset \mid \Sigma \vDash \mu$, $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$. *By theorem 54, for any* $\Phi \in \gamma(\Xi)$, $\mathbf{adjust}_{\mathtt{ref}\ \downarrow}(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_{\mathtt{ref}\ \downarrow}(\Xi))$. *Thus* $\mathbf{adjust}_{\mathtt{ref}\ \downarrow}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *and rule* [E-Frame] *can be applied with* $f = (\mathtt{ref}\ \square)_\varepsilon$.
- $\mathrm{e} = \mathsf{Error}$. *Thus rule* [E-Error] *can be applied.*
- e *is a value.*

  By Hypothesis, ***strict-check***$_{\mathtt{ref}\ \pi}(\Xi)$. *By definition of* ***strict-check****, then* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{\mathtt{ref}\ \pi}(\Xi)$. *By lemma 57, we know that* $\mathrm{e} = w_\varepsilon$ *for some* $\varepsilon \in \pi$. *Thus* $\mathtt{ref}\ \{\varepsilon\} \sqsubseteq_C \mathtt{ref}\ \pi$ *and by M & M tag monotonicity lemma,* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{\mathtt{ref}\ \{\varepsilon\}}(\Phi)$. *Therefore, rule* [E-Ref] *can be applied.*

Case ([IT-Deref])**.** *By induction hypothesis, either*

- $\forall \Phi' \supseteq \Phi \in \gamma(\Xi)$ *and* $\forall \mu. \emptyset \mid \Sigma \vDash \mu$, $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$. *By theorem 54, for any* $\Phi \in \gamma(\Xi)$, $\mathbf{adjust}_{!\downarrow}(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi))$. *Thus* $\mathbf{adjust}_{!\downarrow}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *and rule* [E-Frame] *can be applied with frame* $f = \mathtt{ref}\ \square$.
- $\mathrm{e} = \mathsf{Error}$. *Thus rule* [E-Error] *can be applied.*
- e *is a value. By lemma 57, then* $\mathrm{e} = l_\varepsilon$ *for some* $\varepsilon \mathrm{in} \pi$.

  By Hypothesis, ***strict-check***$_{!\pi}(\Xi)$. *By definition of* ***strict-check****, then* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{!\pi}(\Xi)$. *Since* $\varepsilon \in \pi$, $!\{\varepsilon\} \sqsubseteq_C !\pi$. *Thus by M & M tag monotonicity lemma,* $\forall \Phi \in \gamma(\Xi).\mathbf{check}_{!\{\varepsilon\}}(\Phi)$. *Since* l_ε *is typed, then* $l \in dom(\Sigma)$, *and since* $\emptyset \mid \Sigma \vDash \mu$, *then* $l \in dom(\mu)$ *and then rule* [E-Deref] *can always be applied.*

Case ([IT-Asgn])**.** *By induction hypothesis, either*

- $\forall \Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\forall \mu. \emptyset \mid \Sigma \vDash \mu$, $\Phi \vdash \mathrm{e}_1 \mid \mu \to \mathrm{e}'_1 \mid \mu'$. *By theorem 54, for any* $\Phi \in \gamma(\Xi)$, $\mathbf{adjust}_{\downarrow:=\uparrow}(\Xi) \in \gamma(\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi))$. *Thus* $\mathbf{adjust}_{\downarrow:=\uparrow}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *and rule* [E-Frame] *can be applied with frame* $\square := \mathrm{e}$.
- $\mathrm{e}_1 = \mathsf{Error}$. *Thus rule* [E-Error] *can be applied.*
- e_1 *is a value. By lemma 57 (Canonical Values),* $\mathrm{e}_1 = l_{\varepsilon_1}$ *for some* $\varepsilon_1 \in \pi_1$. *Now, also by induction hypothesis, either*
  - $\forall \Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\forall \mu. \emptyset \mid \Sigma \vDash \mu$, $\Phi \vdash \mathrm{e}_2 \mid \mu \to \mathrm{e}'_2 \mid \mu'$. *By theorem 54, for any* $\Phi \in \gamma(\Xi)$, $\mathbf{adjust}_{\pi_1:=\downarrow}(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi))$. *Thus* $\mathbf{adjust}_{\pi_1:=\downarrow}(\Phi) \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *and rule* [E-Frame] *can be applied with frame* $l_{\varepsilon_1} := \square$.
  - $\mathrm{e} = \mathsf{Error}$. *Thus rule* [E-Error] *can be applied.*

– e *is a value.*

By lemma 57 we only know that $\exists \varepsilon_2 \in \pi_2$ *such that* $v = w_{\varepsilon_2}$. *This means that* $\{\varepsilon_1\} := \{\varepsilon_2\} \sqsubseteq_C \pi_1 := \pi_2$.

By hypothesis we know that $\textbf{\textit{strict-check}}_{\pi_1 := \pi_2}(\Xi)$. *By definition of* ***strict-check****, then* $\forall \Phi \in \gamma(\Xi).\textbf{check}_{\pi_1 := \pi_2}(\Phi)$. *By M & M tag monotonicity lemma, then* $\forall \Phi \in \gamma(\Xi).\textbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)$ *and thus rule* [E-Asgn] *can be applied.*

$\square$

Lemma 58.

1. $\Xi; \Gamma; \Sigma \vdash v \colon T \Rightarrow \Xi'; \Gamma; \Sigma \vdash v \colon T$
2. $\Xi; \Gamma; \Sigma \vdash x \colon T \Rightarrow \Xi'; \Gamma; \Sigma \vdash x \colon T$
3. $\Xi; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle x \colon T_2 \Rightarrow \Xi'; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle x \colon T_2$

Proof. 1. We proceed by cases on v.
Case ($\texttt{unit}_\varepsilon$)**.** *Then we can use rule* [IT-Unit] *for any other* Ξ'.
Case ($(\lambda x \colon T\ .\ \mathrm{e})_\varepsilon$)**.** *There is only one typing rule for functions. We can reuse the same* [IT-Fn] *To type the function to the same type in a context* Ξ' *by reusing the original premise.*
Case (l_ε)**.** *Since* Σ *is preserved, we can reuse rule* [IT-Loc] *to type* l_ε *with any other* Ξ'.

2. There is only one rule for typing variable identifiers, [IT-Var]. Since the lemma preserves the environment Γ, we can use rule [IT-Var] to type the identifier in any Ξ' context.
3. The typing rule for casts reduces this case to the previous one.

$\square$

Proposition 11 (Strong Effect Subsumption)**.**
If $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \colon T$ *and* $\Xi_1 \leq \Xi_2$, *then* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e} \colon T$.

Proof. By structural induction over the typing derivations for $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e} \colon T$

Case (Rules [IT-Fn], [IT-Unit], [IT-Loc], [IT-Var] and [IT-Error])**.** *All of these rules do not enfore a restriction between the* Ξ_2 *in the conclusions and any* Ξ *(if existent) in the premises, so the same rule can be directly re-used to infer* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e} \colon T$.

Case (Rule [IT-App])**.** *By lemma 10, since* $\Xi_1 \leq \Xi_2$, $\widetilde{\textbf{adjust}}_A(\Xi_1) \leq \widetilde{\textbf{adjust}}_A(\Xi_2)$ *for any* A, *in particular both for* $A = \downarrow\uparrow$ *and* $A = \pi_1\downarrow$.

Thus by Induction Hypothesis, we can infer both that $\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi_2)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon \pi_1 T_1 \xrightarrow{\Xi'} T_3$ *and that* $\widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi_2)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon \pi_2 T_2$.

By lemma 51, we also know that $\textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi_2)$.

By hypothesis we also know that $\pi_1 T_1 \xrightarrow{\Xi'} T_2 <\colon \pi_1 T_2 \xrightarrow{\Xi} T_3$ *and then we can use rule* [IT-App] *to infer that* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}_1\ \mathrm{e}_2 \colon T_3$.

Case (Rule [IT-Cast]). *By Induction Hypothesis,* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T_1$ *and thus we can directly use* [IT-Cast] *to infer* $\Xi_2; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle \mathrm{e}\colon T_2$.

Case (Rule [IT-Has]). *Since by hypothesis* $|\Xi_1| \subseteq |\Xi_2|$, *in particular we know that* $\Phi \cup |\Xi_1| \subseteq \Phi \cup \Xi_2$.

We know that $|\Phi \cup \Xi| = \Phi \cup |\Xi|$, *then* $|\Phi \cup \Xi_1| \subseteq |\Phi \cup \Xi_2|$ *and thus* $\Phi \cup \Xi_1 \leq \Phi \cup \Xi_2$.

By Induction Hypothesis, $\Phi \cup \Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T$. *Then we can use rule* [IT-Has] *to infer that* $\Xi_2; \Gamma; \Sigma \vdash \texttt{has}\ \Phi\ \mathrm{e}\colon T$.

Case (Rule [IT-Rst]). *(*$\Xi_1; \Gamma; \Sigma \vdash \texttt{restrict}\ \Xi'\ \mathrm{e}\colon T$*)*

By hypothesis we know that $\Xi' \leq \Xi_1$ *and thus by transitivity of* $\subseteq$, $\Xi' \leq \Xi_2$.

Therefore, we can use rule [IT-Rst] *with the premises of the hypothesis to infer that* $\Xi_2; \Gamma; \Sigma \vdash \texttt{restrict}\ \Xi'\ \mathrm{e}\colon T$.

Case (Rule [IT-Ref]). *By lemma 10,* $\widetilde{\mathbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi_1) \leq \widetilde{\mathbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi_2)$, *and thus by Induction Hypothesis,* $\widetilde{\mathbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi_2)\,; \Gamma; \Sigma \vdash \mathrm{e}\colon \pi\rho$. *By lemma 51,* $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi_2)$. *Thus we can reuse rule* [IT-Ref] *to infer that* $\Xi_2; \Gamma; \Sigma \vdash \texttt{ref}\ \mathrm{e}\colon \texttt{Ref}\ T$.

Case (Rule [IT-Deref]). *Analogous to rule* [IT-Ref].

Case (Rule [IT-Asgn]). *By lemma 10 and Induction Hypothesis,* $\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi_2)\,; \Gamma; \Sigma \vdash \mathrm{e}_1\colon \pi_1 \texttt{Ref}\ T_1$ *and* $\widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi_2)\,; \Gamma; \Sigma \vdash \mathrm{e}_2\colon \pi_2\rho_2$.

By lemma 51, $\textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi_2)$. *Since also* $\pi_2\rho_2 <: T_1$, *we can reuse rule* [IT-Asgn] *to infer that* $\Xi_2; \Gamma; \Sigma \vdash (\mathrm{e}_1 := \mathrm{e}_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}$.

□

Corollary 12 (Effect Subsumption).
If $\Xi_1; \Gamma; \Sigma \vdash \mathrm{e}\colon T$ *and* $\Xi_1 \subseteq \Xi_2$, *then* $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T$.

Proof. By lemma 50, $\Xi_1 \leq \Xi_2$. Thus, by Strong Subsumption proposition 11, $\Xi_2; \Gamma; \Sigma \vdash \mathrm{e}\colon T$. □

Lemma 59 (Tag monotonicity for ***strict-check***). *If* $C_2 \sqsubseteq_C C_1$ *and* $\textbf{\textit{strict-check}}_{C_1}(\Xi) \Rightarrow \textbf{\textit{strict-check}}_{C_2}(\Xi)$

Proof. By definition of ***strict-check***, $\forall \Phi \in \gamma(\Xi), \mathbf{check}_{C_1}(\Phi)$. By M & M tag monotonicity lemma, since $C_2 \sqsubseteq_C C_1$, then $\mathbf{check}_{C_2}(\Phi)$ too and therefore $\textbf{\textit{strict-check}}_{C_2}(\Xi)$. □

Lemma 60. *If* $\bigcap \Upsilon \in \Upsilon$ *and* $A_2 \sqsubseteq_A A_1$, *then*
$\bigcap \left\{ \mathbf{adjust}_{A_1}(\Phi)\, \forall \Phi \in \Upsilon \right\} \subseteq \bigcap \left\{ \mathbf{adjust}_{A_2}(\Phi)\, \forall \{\in\} \Upsilon \right\}$.

Proof. Suppose $\phi \in \bigcap \left\{ \mathbf{adjust}_{A_1}(\Phi)\, \forall \Phi \in \Upsilon \right\}$. Since $\bigcap \Upsilon \in \Upsilon$, then also $\phi \in \mathbf{adjust}_{A_1}(\bigcap \Upsilon)$. By M & M tag monotonicity for **adjust**, then also $\phi \in \mathbf{adjust}_{A_2}(\bigcap \Upsilon)$. Since $\bigcap \Upsilon \subseteq$

$\Phi \forall \Phi \in \Upsilon$, then $\forall \Phi \in \Upsilon$, by privilege monotonicity, $\phi \in \mathbf{adjust}_{A_2}(\Phi)$ and therefore $\phi \in \bigcap \{\mathbf{adjust}_{A_2}(\Phi) \, \forall \Phi \in \Upsilon\}$. □

Lemma 61 (Tag Monotonicity for $\widetilde{\mathbf{adjust}}$)**.** *If* $A_2 \sqsubseteq_A A_1$, *then* $\widetilde{\mathbf{adjust}}_{A_1}(\Xi) \leq \widetilde{\mathbf{adjust}}_{A_2}(\Xi)$.

Proof. By lemma 49, $\bigcap \gamma(\Xi) \in \gamma(\Xi)$. Thus, by lemma 60, then $\bigcap \{\mathbf{adjust}_{A_1}(\Phi) \, \forall \Phi \in \Upsilon\} \subseteq \bigcap \{\mathbf{adjust}_{A_2}(\Phi) \, \forall \Phi \in \Upsilon\}$.

Given that by definition of γ, for any $\Xi, \gamma(\Xi) \neq \emptyset$, we can infer by lemma 52 that $|\alpha(\{\mathbf{adjust}_{A_1}(\Phi) \, \forall \Phi \in \gamma(\Xi)\})| \subseteq |\alpha(\{\mathbf{adjust}_{A_2}(\Phi) \, \forall \Phi \in \gamma(\Xi)\})|$. By definition of $\widetilde{\mathbf{adjust}}$, this is equivalent to $|\widetilde{\mathbf{adjust}}_{A_1}(\Xi)| \subseteq |\widetilde{\mathbf{adjust}}_{A_2}(\Xi)|$, which at the same time is the definition of $\widetilde{\mathbf{adjust}}_{A_1}(\Xi) \leq \widetilde{\mathbf{adjust}}_{A_2}(\Xi)$. □

Theorem 62 (Preservation of types under substitution)**.** *If* $\Xi; \Gamma, x\colon T_1; \Sigma \vdash \mathrm{e}_3\colon T_3$ *and* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2$ *with* $T_2 <: T_1$ *and either* $\mathrm{e}_2 = v$ *or* $\mathrm{e}_2 = \langle T_1 \Leftarrow T_0 \rangle x$, *then* $\Xi; \Gamma; \Sigma \vdash [{}^{\mathrm{e}_2}/_x]\, \mathrm{e}_3\colon T'$ *and* $T' <: T_3$.

Proof. By structural induction over the typing derivation for e_2.

Case ([IT-Unit], [IT-Loc] and [IT-Error])**.** *Trivial since substitution does not change the expression.*

Case ([IT-Var])**.** *By definition of substitution, the interesting cases are:*

- $\mathrm{e}_3 = y \neq x$ $([{}^{\mathrm{e}_2}/_x]\, y = y)$. *Then by assumption we know that* $\Gamma(y) = T_3$ *and thus we can infer that* $\Xi; \Gamma; \Sigma \vdash y\colon T_3$.
- $\mathrm{e}_3 = x$ $([{}^{\mathrm{e}_2}/_x]\, x = \mathrm{e}_2)$. *Then by the theorem hypothesis we know that* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2$. *We also know that* $\Xi; \Gamma, x\colon T_1; \Sigma \vdash x\colon T_3$, *which means that* $T_3 = T_1$ *and thus* $T' = T_2 <: T_1 = T_3$.

Case ([IT-Fn])**.**

- $\lambda x\colon T \,.\, \mathrm{e}$. *Then substitution does not affect the body and thus we reuse the original type derivation.*
- $\lambda y\colon T \,.\, \mathrm{e}$ *Then by induction hypothesis, substitution of the body preserves typing and thus rule* [IT-Fn] *can be used to reconstruct the type for the modified expression.*

Case ([IT-Ref], [IT-Deref], [IT-Cast], [IT-Has] and [IT-Rst])**.** *Analogous to the case for* [T-Fn]*, since substitution for these expression is defined just as recursive calls to substitution for the premises in the typing rules.*

Case ([IT-App])**.** *(*$\mathrm{e}_3 = \mathrm{e}'_1\ \mathrm{e}'_2$*)*

By lemma 58, we can can infer that $\Xi'; \Gamma; \Sigma \vdash \mathrm{e}_2\colon T_2$, *in particular for* $\Xi' = \widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)$ *and for* $\Xi' = \widetilde{\mathbf{adjust}}_{\pi'_1\downarrow}(\Xi)$. *Thus we can use the induction hypotheses in both subexpressions of* $\mathrm{e}_3 = \mathrm{e}'_1\ \mathrm{e}'_2$.

Therefore, while $\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}'_1 \colon \pi'_1 T'_1 \xrightarrow{\Xi'} T'_3$ *, by induction hypothesis also* $\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_1 \colon \pi''_1 T''_1 \xrightarrow{\Xi''} T''_3$ *with* $\pi''_1 T''_1 \xrightarrow{\Xi''} T''_3 <: \pi'_1 T'_1 \xrightarrow{\Xi'} T'_3$ *which means that* $\pi''_1 \subseteq \pi'_1$ *and* $T'_1 <: T''_1$.

At the same time, since $\widetilde{\textbf{adjust}}_{\pi'_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}'_2 \colon T'_2$, *by induction hypothesis also* $\widetilde{\textbf{adjust}}_{\pi'_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_2 \colon T''_2$ *with and* $T''_2 <: T'_2$, *thus* $T''_2 = \pi''_2\rho''_2$, $T'_2 = \pi'_2\rho'_2$ *and* $\pi''_2 \subseteq \pi'_2$.

Snce $\pi''_1 \downarrow \sqsubseteq_A \pi'_1 \downarrow$, *then by lemma 61* $\widetilde{\textbf{adjust}}_{\pi''_1\downarrow}(\Xi) \leq \widetilde{\textbf{adjust}}_{\pi'_1\downarrow}(\Xi)$. *Then by Strong Subsumption proposition 11,* $\widetilde{\textbf{adjust}}_{\pi''_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_2 \colon T''_2$.

Since $\pi''_1\pi''_2 \sqsubseteq_C \pi'_1\pi'_2$ *and* $\textbf{\textit{strict-check}}_{\pi'_1\pi'_2}(\Xi)$, *by lemma 59,* $\textbf{\textit{strict-check}}_{\pi''_1\pi''_2}(\Xi)$.

Also, by hypothesis $T'_2 <: T'_1$, *which by transitivity of subtyping means that* $T''_2 <: T''_1$. *Therefore, we know that* $\pi''_1 T''_1 \xrightarrow{\Xi''} T''_3 <: \pi''_1 T''_2 \xrightarrow{\Xi} T''_3$, *and we can use rule* [IT-App] *to infer back that* $\Xi;\Gamma;Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_1\ \ [\mathrm{e}_2/x]\,\mathrm{e}'_2 \colon T''_3$, *and by transitivity of subtyping,* $T''_3 <: T_3$.

Case ([IT-Asgn])**.** *By Induction Hypothesis, if* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}'_1 \colon \pi'_1 \mathtt{Ref}\ T_1$ *and* $\Xi;\Gamma;\Sigma \vdash \mathrm{e}'_2 \colon \pi'_2\rho'_2$, *also* $\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_1 \colon \pi''_1\mathtt{Ref}\ T_1$ *with* $\pi''_1 \subseteq \pi'_1$*(by the definition of subtyping for* `Ref` T*) and* $\widetilde{\textbf{adjust}}_{\pi'_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_2 \colon \pi''_2\rho''_2$, *with* $\pi''_2\rho''_2 <: \pi'_2\rho'_2$. *By transitivity of subtyping, since* $\pi'_1\rho'_2 <: T_1$, *then* $\pi''_2\rho''_2 <: T_1$.

Given that $\pi''_1 :=\downarrow \sqsubseteq_A \pi'_1 :=\downarrow$, *we can infer by lemma 61 that* $\widetilde{\textbf{adjust}}_{\pi'_1:=\downarrow}(\Xi) \leq \widetilde{\textbf{adjust}}_{\pi''_1:=\downarrow}(\Xi)$. *Then, by proposition 11, also* $\widetilde{\textbf{adjust}}_{\pi''_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e}'_2 \colon \pi''_2\rho''_2$.

Finally, since $\pi''_1 := \pi''_2 \sqsubseteq_C \pi'_1 := \pi'_2$ *and* $\textbf{\textit{strict-check}}_{\pi'_1:=\pi'_2}(\Xi)$, *then also* $\textbf{\textit{strict-check}}_{\pi''_1:=\pi''_2}(\Xi)$ *and we then can use rule* [IT-Asgn] *to infer back that* $\Xi;\Gamma;\Sigma \vdash ([\mathrm{e}_2/x]\,\mathrm{e}'_1 := [\mathrm{e}_2/x]\,\mathrm{e}'_2)_\varepsilon \colon \{\varepsilon\}\mathtt{Unit}$.

□

Lemma 63. *If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e} \colon T$, *then* $\Xi;\Gamma;\Sigma' \vdash \mathrm{e} \colon T$ *for* $\Sigma' \supseteq \Sigma$.

Proof. By structural induction over the typing derivation. The only interesting case is for [IT-Loc]. The rest follow by applying the induction hypothesis and reusing the typing rules.

Case (Rule [IT-Loc])**.** *Since* $\Sigma \subseteq \Sigma'$, *then* $\Sigma'(l) = \Sigma(l) \forall l \in dom(\Sigma)$. *Thus we can reuse* [IT-Loc] *to infer that* $\Xi;\Gamma;\Sigma' \vdash l \colon \mathtt{Ref}\ T$ *with* $T = \Sigma'(l) = \Sigma(l)$.

□

Theorem 16 (Preservation)**.** *If* $\Xi;\Gamma;\Sigma \vdash \mathrm{e} \colon T$, *and* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *for* $\Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$, *then* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi;\Gamma;\Sigma' \vdash \mathrm{e}' \colon T'$ *for some* $T' <: T$ *and* $\exists \Sigma' \supseteq \Sigma$.

Proof. By structural induction over the typing derivation and the applicable evaluation rules.

Case (Rules [IT-Fn], [IT-Unit], [IT-Loc] [IT-Var] and [IT-Error])**.** *These rules are trivial since there is no rule in the operational semantics that takes these expressions as premises to step.*

Case ([IT-App] and [E-Frame] with $f = \Box\ t$)**.** *By theorem 54, we can use the Induction Hypothesis to infer that* $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}_1{}' \colon \pi_1' T_1' \xrightarrow{\Xi''} T_3'$ *and* $\pi_1' T_1' \xrightarrow{\Xi''} T_3' <: \pi_1 T_1 \xrightarrow{\Xi'} T_3$*. By definition of subtyping,* $T_1 <: T_1'$ *and therefore* $T_2 <: T_1'$*.*

Since $\pi_1' \downarrow \sqsubseteq_A \pi_1 \downarrow$*,* $\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi) \leq \widetilde{\mathbf{adjust}}_{\pi_1'\downarrow}(\Xi)$ *and therefore by proposition 11, since* $\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T_2$*, also* $\widetilde{\mathbf{adjust}}_{\pi_1'\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2 \colon T_2$*.*

Also, by lemma 63, we know that $\widetilde{\mathbf{adjust}}_{\pi_1'\downarrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}_2 \colon T_2$*.*

Since $T_2 = \pi_2 \rho_2$ *and* $\pi_1' \pi_2 \sqsubseteq_C \pi_1 \pi_2$*, we can use lemma 59 to infer that* $\boldsymbol{strict\text{-}check}_{\pi_1'\pi_2}(\Xi)$*.*

Finally, since we already know that $\pi_1' T_1' \xrightarrow{\Xi''} T_3' <: \pi_1 T_1 \xrightarrow{\Xi'} T_3$*, also we know that* $\pi_1' T_1' \xrightarrow{\Xi''} T_3' <: \pi_1' T_2 \xrightarrow{\Xi} T_3$ *. Thus we can reuse rule* [IT-App] *to infer that* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}_1'\ \mathrm{e}_2 \colon T_3'$ *and we know that* $T_3' <: T_3$*.*

Case ([IT-App] and [E-Frame] with $f = w_\varepsilon\ \Box$)**.** *By hypothesis,* $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_1 \colon \pi_1 T_1 \xrightarrow{\Xi_1} T_3$*. By lemma 63, we know that* $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}_1 \colon T_1 \xrightarrow{\Xi_1} T_3$*.*

Since $\mathbf{adjust}_{\pi_1\downarrow}(\Phi) \in \gamma(\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi))$ *and* $\mathbf{adjust}_{\pi_1\downarrow}(\Phi) \subseteq \mathbf{adjust}_{\{\varepsilon\}\downarrow}(\Phi)$ *for* $\varepsilon \in \pi_1$*, we can also use the induction hypothesis to infer that* $\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}_2' \colon T_2'$ *and* $T_2' <: T_2$*.*

Since by hypothesis and definition of subtyping, $T_2 <: T_1$*, then also* $T_2' <: T_1$*, and therefore we can infer that* $\pi_1 T_1 \xrightarrow{\Xi_1} T_3 <: \pi_1 T_2' \xrightarrow{\Xi} T_3$ *and we can reuse rule* [IT-App] *to infer that* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}_1\ \mathrm{e}_2' \colon T_3$*.*

Case ([IT-App] and [E-App])**.** *In this case,* $\mathrm{e}_1 = \lambda x \colon T_1\ .\ \mathrm{e}$*, and* $\Xi'; \Gamma, x \colon T_1; \Sigma \vdash \mathrm{e} \colon T_3$*.*

Thus by theorem 62, $\Xi'; \Gamma; \Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e} \colon T_3'$*, with* $T_3' <: T_3$*. Then by corollary 12,* $\Xi; \Gamma; \Sigma \vdash [\mathrm{e}_2/x]\,\mathrm{e} \colon T_3'$*,* $T_3' <: T_3$*.*

Case ([IT-Has] and [E-Has-T])**.** $\mathrm{e} = \mathtt{has}\ \Phi\ \mathrm{e}'$*. Therefore, application of* [E-Has-T] *takes the form* $\dfrac{\Phi \subseteq \Phi' \qquad \Phi' \vdash \mathrm{e}' \to \mathrm{e}''}{\Phi' \vdash \mathtt{has}\ \Phi\ \mathrm{e}' \to \mathtt{has}\ \Phi\ \mathrm{e}''}$ *with* $\Phi' \in \gamma(\Xi)$*.*

Since $\Phi \subseteq \Phi'$*, then also* $\Phi' \in \gamma(\Phi \cup \Xi)$ *and then by induction hypothesis* $\Phi \cup \Xi; \Gamma; \Sigma' \vdash \mathrm{e}'' \colon T'$*,* $T' <: T$*. We can then use rule* [IT-Has] *to infer that* $\Xi; \Gamma; \Sigma' \vdash \mathtt{has}\ \Phi \mathrm{e}'' \colon T'$ *too.*

Case ([IT-Has] and [E-Has-V])**.** *By induction hypothesis and lemma 58, in particular* Ξ *instead of* $\Phi \cup \Xi$*.*

Case ([IT-Has] and [E-Has-F])**.** *Trivial by using rule* [T-Error].

Case ([IT-Rst] and [E-Rst]). *Since by rule* [E-Rst] $\Phi'' \in \gamma(\Xi_1)$*, we can use induction hypothesis to infer that* $\Xi_1; \Gamma; \Sigma' \vdash \mathrm{e}' \colon T'$*,* $T' <: T$*. Then we reuse rule* [IT-Rst] *to infer that* $\Xi; \Gamma; \Sigma' \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e}' \colon T'$*.*

Case ([IT-Rst] and [E-Rst-V]). *By induction hypothesis and using lemma 58, in particular* Ξ *instead of* Ξ_1 *(analogous to* [IT-Has] *and* [E-Has-V]*).*

Case ([IT-Cast] and [E-Cast-Id]). *Follows trivially by induction hypothesis.*

Case ([IT-Cast] and [E-Cast-Fn]). *By definition of* [IT-Cast] *and static subtyping,* $\Xi_1 \leq \Xi_2$*. By Induction Hypothesis,* $\Xi; \Gamma; \Sigma' \vdash \lambda x \colon T_{11}\ .\ \mathrm{e} \colon T_{11} \xrightarrow{\Xi_1'} T_{12}'$*,* $T_{11} \xrightarrow{\Xi_1'} T_{12}' <: T_{11} \xrightarrow{\Xi_1} T_{12}$*. By construction this means that* $\Xi_1; \Gamma, x \colon T_{11}; \Sigma' \vdash \mathrm{e} \colon T_{12}'$*.*

We can use then the following inference for the type of the resulting expression of [E-Cast-Fn]*:*

$$
\textit{T-Fn}\ \dfrac{\textit{IT-Cast}\ \dfrac{\textit{IT-Rst}\ \dfrac{\textit{IT-Has}\ \dfrac{\textit{Proposition 11}\ \dfrac{\textit{Theorem 62}\ \dfrac{\textit{Hypothesis}\ \dfrac{}{\Xi_1; \Gamma, x \colon T_{11}; \Sigma' \vdash \mathrm{e} \colon T_{12}'}}{\Xi_1; \Gamma, x \colon T_{21}; \Sigma' \vdash [{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{12}'}}{|\Xi_1| \cup \Xi_2; \Gamma, x \colon T_{21}; \Sigma' \vdash [{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{12}'}}{\Xi_2; \Gamma, x \colon T_{21}; \Sigma' \vdash \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\,[{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{12}'}}{\Xi_2; \Gamma, x \colon T_{21}; \Sigma' \vdash \texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\,[{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{12}'}}{\Xi_2; \Gamma, x \colon T_{21}; \Sigma' \vdash \langle T_{22} \Leftarrow T_{12}\rangle\texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\,[{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{22}}}{\Xi; \Gamma; \Sigma' \vdash \lambda x \colon T_{21}\ .\ \langle T_{22} \Leftarrow T_{12}\rangle\texttt{restrict}\ \Xi_2\ \texttt{has}\ (|\Xi_1| \setminus |\Xi_2|)\,[{}^{\langle T_{11} \Leftarrow T_{21}\rangle}\!/\!_{x}]\,\mathrm{e} \colon T_{21} \xrightarrow{\Xi_2} T_{22}}
$$

Case ([IT-Cast] and [E-Cast-Frame]). *By Induction Hypothesis,* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}' \colon T_1'$*,* $T_1' <: T_1$*. Then we can reuse rule* [IT-Cast] *to infer* $\Xi; \Gamma; \Sigma' \vdash \langle T_2 \Leftarrow T_1\rangle\mathrm{e}' \colon T_2$*.*

Case ([IT-Ref] and [E-Frame] with $f = \texttt{ref}\ \square$). *By theorem 54, since* $\Phi \in \gamma(\Xi)$*,* $\textbf{adjust}_{\downarrow\uparrow}(\Phi) \in \gamma(\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi))$ *and thus by Induction Hypothesis,* $\widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}' \colon \pi'\rho'\ ,\ \pi'\rho' <: \pi\rho$*.*

Since $\texttt{ref}\ \pi' \sqsubseteq_C \texttt{ref}\ \pi$*, by lemma 59 then* $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi'}(\Xi)$ *.Thus we can use rule* [IT-Ref] *to infer that* $\Xi; \Gamma; \Sigma' \vdash (\texttt{ref}\ \mathrm{e})_\varepsilon \colon \{\varepsilon\}\texttt{Ref}\ \pi'\rho'$*.*

Case ([IT-Ref] and [E-Ref]). *(*$\Phi \vdash \texttt{ref}\ v \mid \mu \to l \mid \mu' = \mu[l \mapsto v]$*)*

By Hypothesis we know that $\widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash v \colon T$*. By lemma 58, then also* $\Xi; \Gamma; \Sigma \vdash v \colon T$*. Since* $l \notin dom(\Sigma)$*, we can introduce* $\Sigma' = \Sigma[l \mapsto T]$*. By construction,* $\Sigma \subseteq \Sigma'$ *and* $\Gamma \mid \Sigma' \vDash \mu'$*. Thus we can use rule* [IT-Loc] *to infer that* $\Xi; \Gamma; \Sigma' \vdash l \colon \texttt{Ref}\ T$*.*

Case ([IT-Deref] and [E-Frame] with $f = !\square$). *By Induction Hypothesis,* $\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}' \colon \pi'\texttt{Ref}\ T$*,* $\pi'\texttt{Ref}\ T <: \pi\texttt{Ref}\ T$*.*

Since $!\pi' \sqsubseteq_C !\pi$*, by lemma 59 we know that* $\textbf{\textit{strict-check}}_{!\pi'}(\Xi)$*. Then we can reuse rule* [IT-Deref] *to infer that* $\Xi; \Gamma; \Sigma' \vdash !\mathrm{e}' \colon T$*.*

Case ([IT-Deref] and [E-Deref]). *Direct from* $\Gamma \mid \Sigma \vDash \mu$*. This implies that if* $\Sigma(l) = T_1$*,* $\Xi; \Gamma; \Sigma \vdash v \colon T_0$*,* $T_0 <: T_1$*.*

Case ([IT-Asgn] and [E-Frame] with $f = (\square := \mathrm{e})_\varepsilon$). *By theorem 54 since* $\Phi \in \gamma(\Xi)$*, we know that* $\textbf{adjust}_{\downarrow:=\uparrow}(\Phi) \in \gamma(\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi))$*. Therefore, by Induction Hypothesis we know that* $\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma' \vdash \mathrm{e}_1' \colon \pi_1'\texttt{Ref}\ T_1$ *with* $\pi_1'\texttt{Ref}\ T_1 <: \pi_1\texttt{Ref}\ T_1$*.*

Since $\Sigma' \supseteq \Sigma$, by lemma 63 we know that $\widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma' \vdash \mathrm{e}_2\colon T_2$. Since $\pi'_1 :=\downarrow \sqsubseteq_A \pi_1 :=\downarrow$, by lemma 61 we know that $\widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi) \leq \widetilde{\mathbf{adjust}}_{\pi'_1:=\downarrow}(\Xi)$. Therefore by Strong Subsumption proposition 11, we also know that $\widetilde{\mathbf{adjust}}_{\pi'_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma' \vdash \mathrm{e}_2\colon T_2$.

*$T_2 = \pi_2\rho_2$. Since $\pi'_1\pi_2 \subseteq_C \pi_1\pi_2$, then also **strict-check**$_{\pi'_1\pi_2}(\Xi)$ by lemma 59.*

Since by hypothesis $T_2 <: T_1$, we can use rule [IT-Asgn] *to infer that $\Xi;\Gamma;\Sigma' \vdash (\mathrm{e}'_1 := \mathrm{e}_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}$.*

Case ([IT-Asgn] and [E-Frame] with $f = (w_{\varepsilon'} := \Box)_\varepsilon$)**.** *By Hypothesis, $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}_1\colon \pi_1\mathtt{Ref}\ T_1$.*

By theorem 54, since $\Phi \in \gamma(\Xi)$, $\mathbf{adjust}_{\pi_1:=\downarrow}(\Phi) \in \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)$. By Monotonicity, also $\mathbf{adjust}_{\pi_1:=\downarrow}(\Phi) \subseteq \mathbf{adjust}_{\{\varepsilon'\}:=\downarrow}(\Phi)$ for $\varepsilon' \in \pi_1$. Thus we can use the Induction Hypothesis to infer that $\widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,;\Gamma;\Sigma' \vdash \mathrm{e}'_2\colon T'_2$ with $T'_2 <: T_2$. By transitivity of subtyping, $T'_2 <: T_1$.

By lemma 63, also $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma' \vdash \mathrm{e}_1\colon \pi_1\mathtt{Ref}\ T_1$.

Since we have all the premises required, we can reuse rule [IT-Asgn] *to infer that $\Xi;\Gamma;\Sigma' \vdash (\mathrm{e}_1 := \mathrm{e}'_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}$.*

Case ([IT-Asgn] and [E-Asgn])**.** *Since rule* [E-Asgn] *preserves the tag from the assignment into the $\mathtt{unit}_\varepsilon$ value, we can always use rule* [IT-Unit] *to infer that $\Xi;\Gamma;\Sigma \vdash \mathtt{unit}_\varepsilon\colon \{\varepsilon\}\mathtt{Unit}$, and since by hypothesis we know that $\Xi;\Gamma;\Sigma \vdash v\colon T_2$ and $\Xi;\Gamma;\Sigma \vdash l\colon \mathtt{Ref}\ T_1$ with $T_2 <: T_1$, we also know that $\Gamma \mid \Sigma \vDash \mu[l \mapsto v]$.*

□

Theorem 17 (Translation preserves typing)**.** *If $\Xi;\Gamma;\Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}'\colon T$ in the source language then $\Xi;\Gamma;\Sigma \vdash \mathrm{e}'\colon T$ in the internal language.*

Proof. By structural induction over the translation derivation rules.

Case ([C-Unit], [C-Loc] and [C-Var])**.** *Using the rule premises we can trivially apply rules* [IT-Unit] *and* [IT-Var]*, respectively.*

Case ([C-Fn])**.** *By Induction Hypothesis, $\Xi_1;\Gamma, x\colon T_1;\Sigma \vdash \mathrm{e}'\colon T_2$. Thus we can use rule* [IT-Fn] *to infer that $\Xi;\Gamma;\Sigma \vdash \lambda x\colon T_1\ .\ \mathrm{e}'\colon T_1 \xrightarrow{\Xi_1} T_2$.*

Case ([C-App])**.** *By Induction Hypothesis, we know that $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}'_1\colon \pi_1\left(T_1 \xrightarrow{\Xi'} T_3\right)$. And therefore also $\widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi)\,;\Gamma;\Sigma \vdash \mathrm{e}'_2\colon \pi_2\rho_2$.*

We also know that $\pi_1\left(T_1 \xrightarrow{\Xi'} T_3\right) \lesssim \pi_1\left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right)$. Thus we can use rule [IT-Cast] *to infer that $\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi)\,;\Gamma;\Sigma \vdash \langle\!\langle \pi_1\left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right) \Leftarrow \pi_1\left(T_1 \xrightarrow{\Xi'} T_3\right)\rangle\!\rangle \mathrm{e}'_1\colon \pi_1\left(T'_2 \xrightarrow{\Xi''} T_3\right)$ with $T'_2 = T_1$ or $T'_2 = \pi_2\rho_2$, and $\Xi'' = \Xi'$ or $\Xi'' = \Xi$ (depending on the insertions of casts). In any case, $T'_2 \xrightarrow{\Xi''} T_3 <: \pi_2 T_2 \xrightarrow{\Xi} T_3$*

Since $\widetilde{\textbf{check}}_{\pi_1\pi_2}(\Xi)$*, by lemma 55, we know that* $\textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Delta_{\texttt{app}}(\Xi) \cup \Xi)$.

Finally, we proceed on the cases for insert-has?.

1. $\Phi = \emptyset$. *In this case, we also know that* $\textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi)$ *because* $\emptyset \cup \Xi = \Xi$*. Then we can apply rule* [IT-App] *to infer that* $\Xi; \Gamma; \Sigma \vdash \left(\langle\!\langle \pi_1 \left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right) \Leftarrow \pi_1 \left(T_1 \xrightarrow{\Xi'} T_3\right)\rangle\!\rangle \mathrm{e}_1'\right) \mathrm{e}_2' \colon T_3$.
2. $\Phi \neq \emptyset$. *By privilege monotonicity and subsumption corollary 12, we know both that* $\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Delta_{\pi_1\pi_2}(\Xi) \cup \Xi)\,; \Gamma; \Sigma \vdash \langle\!\langle \pi_1 \left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right) \Leftarrow \pi_1 \left(T_1 \xrightarrow{\Xi'} T_3\right)\rangle\!\rangle \mathrm{e}_1' \colon T_2' \xrightarrow{\Xi''} T_3$ *and* $\widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Delta_{\pi_1\pi_2}(\Xi) \cup \Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}_2' \colon \pi_2\rho_2$*. We can then use rule* [IT-App] *to infer that* $\Delta_{\pi_1\pi_2}(\Xi) \cup \Xi; \Gamma; \Sigma \vdash \left(\langle\!\langle \pi_1 \left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right) \Leftarrow \pi_1 \left(T_1 \xrightarrow{\Xi'} T_3\right)\rangle\!\rangle \mathrm{e}_1'\right) \mathrm{e}_2' \colon T_3$*. Therefore, we can use rule* [IT-Has] *to infer that* $\Xi; \Gamma; \Sigma \vdash \texttt{has}\ \Delta_{\pi_1\pi_2}(\Xi)\ \left(\left(\langle\!\langle \pi_1 \left(\pi_2\rho_2 \xrightarrow{\Xi} T_3\right) \Leftarrow \pi_1 \left(T_1 \xrightarrow{\Xi'} T_3\right)\rangle\!\rangle \mathrm{e}_1'\right) \mathrm{e}_2'\right) \colon T_3$.

Case ([C-Eff])**.** *By Induction Hypothesis,* $\Xi_1; \Gamma \vdash \mathrm{e}' \colon T$*. We proceed on the cases for insert-has?.*

1. $\Phi = \emptyset$. *Therefore* $\nexists \phi \in |\Xi_1|$ *such that* $\phi \notin |\Xi|$*, thus* $|\Xi_1| \subseteq |\Xi|$ *and* $\Xi_1 \leq \Xi$*. We can therefore use rule* [IT-Rst] *to infer that* $\Xi; \Gamma; \Sigma \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e}' \colon T$.
2. $\Phi \neq \emptyset$. *By lemma 56,* $\Xi_1 \leq (|\Xi_1| \setminus |\Xi|) \cup \Xi$*. We can then use rule* [IT-Rst] *to infer that* $\Phi \cup \Xi_1; \Gamma; \Sigma \vdash \texttt{restrict}\ \Xi_1\ \mathrm{e}' \colon T$ *and thus use rule* [IT-Has] *to infer that* $\Xi; \Gamma; \Sigma \vdash \texttt{has}\ (|\Xi_1| \setminus |\Xi|)\ \texttt{restrict}\ \Xi_1\ \mathrm{e}' \colon T$.

Case ([C-Ref])**.** *By Induction Hypothesis,* $\widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}' \colon \pi\rho$*. We proceed on the cases for insert-has?. Since* $\widetilde{\textbf{check}}_{\texttt{ref}\ \pi}(\Xi)$*, by lemma 55, we know that* $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Delta_{\texttt{ref}\ \pi}(\Xi) \cup \Xi)$.

1. $\Phi = \emptyset$. *Then* $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)$ *and we can use rule* [IT-Ref] *to infer that* $\Xi; \Gamma; \Sigma \vdash (\texttt{ref}\ \mathrm{e}')_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho$.
2. $\Phi \neq \emptyset$. *By privilege monotonicity,* $\Delta_{\texttt{ref}\ \pi}(\Xi) \cup \widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash \mathrm{e}' \colon \pi\rho$*. Since* $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Delta_{\texttt{ref}\ \pi}(\Xi) \cup \Xi)$*, we therefore can use rule* [IT-Ref] *to infer that* $\Delta_{\texttt{ref}\ \pi}(\Xi) \cup \widetilde{\textbf{adjust}}_{\texttt{ref}\ \downarrow}(\Xi)\,; \Gamma; \Sigma \vdash (\texttt{ref}\ \mathrm{e}')_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho$*. We then can use rule* [IT-Has] *to infer that* $\Xi; \Gamma; \Sigma \vdash \texttt{has}\ \Delta_{\texttt{ref}\ \pi}(\Xi)\ (\texttt{ref}\ \mathrm{e}')_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho$.

Case ([C-Deref])**.** *Analogous to case* [C-Ref]

Case ([C-Asgn])**.** *Analogous to case* [C-App].

□

Appendix D

Proof of the Conservative Approximation Theorem

To prove the conservative approximation theorem we define a series of lemmas, which we use to prove a strong conservative approximation lemma, from which the original conservative approximation theorem follows as a corollary. We now proceed to provide the details of the proof.

Definition 22 (Well-formed conservative frames)**.** *We say that a frame f_C is well-formed when the frame is of the form $(\square\ \mathrm{e})_\pi$, $(\texttt{ref}\ \square)_\varepsilon$, $(\square := \mathrm{e})_{(\varepsilon,\pi)}$ or $!\square$.*

If a frame is $(w_\varepsilon\ \square)_\pi$, then it is well-formed if $\varepsilon \in \pi$.

If a frame is of the form $(w_{\varepsilon_1} := \square)_{(\varepsilon,\pi)}$, then the frame is well-formed if $\varepsilon_1 \in \pi$.

.

Lemma 64 (Typing in the conservative semantics ensures well-formed frames)**.** *If $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T$, and $\mathrm{e} = f_C[\mathrm{e}']$, then f_C is a well-formed frame.*

Proof. By cases on the final rule of the type derivation.

Case (Rules [IT-Fn], [IT-Unit], [IT-Loc], [IT-Var], [IT-Cast], [IT-Has], [IT-Rst], and [IT-Error])**.** *Trivial sinced the typed expresion is not of the form $f_C[\mathrm{e}']$.*

Case (Rules [IT-Ref] and [IT-Deref])**.** *Trivial since the candidate frames are always well-formed.*

Case (Rule [IT-App])**.** *There are two candiate structures for frames f_C with rule* [IT-App], *and we analize each case:*

- $f_C = (\square\ \mathrm{e})_\pi$. *This kind of frame is always well-formed.*
- $f_C = (w_\varepsilon\ \square)_\pi$. *Then* $\mathrm{e} = f_C[\mathrm{e}']$. *By inversion lemmas, we know that* $\widetilde{\mathbf{adjust}}_{\downarrow\ \uparrow}(\Xi)\,; \Gamma; \Sigma \vdash w_\varepsilon \colon \{\varepsilon\}T_1 \xrightarrow{\Xi_1} T_3$ *and also that* $\{\varepsilon\}T_1 \xrightarrow{\Xi_1} T_3 <: \pi T_2 \xrightarrow{\Xi} T_3$, *with* $\widetilde{\mathbf{adjust}}_{\pi\downarrow}(\Xi)\,; \Gamma; \Sigma\ \vdash$

$e' \colon T_2$. By inversion of the subtyping relation we know that $\{\varepsilon\} \subseteq \pi$, and therefore frame f_C is well-formed.

Case (Rule [IT-Asgn])**.** *Analogous to rule* [IT-App]*.*

$\square$

Lemma 65 (Partial order for adjust context's tagset erasure)**.** *Let f_C be a well-formed frame such that $[\![f_C[e]]\!]^\pi = f_O[[\![e]\!]^\pi]$. Then $A(f_O) \sqsubseteq A'(f_C)$.*

Proof. By cases on frames f_C.

Case. $(\square\ e)_\pi$, $(\texttt{ref}\ \square)_\varepsilon$, $!\square$, *and* $(\square := e)_{(\varepsilon,\pi)}$ *Trivial since if $[\![f_C[e]]\!]^\pi = f_O[[\![e]\!]^\pi]$, $A'(f_C) = A(f_O)$.*

Case $(f_C = (w_\varepsilon\ \square)_\pi)$**.** *By definition, $A'(f_C) = \pi\ \downarrow$. Also, since $[\![f_C[e]]\!]^\pi = f_O[[\![e]\!]^\pi]$, then $f_O = w_\varepsilon\ \square$, and thus $A(f_O) = \{\varepsilon\}\ \downarrow$.*

Since f_C is well formed, $\varepsilon \in \pi$, and thus $\{\varepsilon\} \subseteq \pi$, which implies that $A(f_O) \sqsubseteq A'(f_C)$.

Case $(f_C = (w_{\varepsilon_1} := \square)_{(\varepsilon,\pi)})$**.** *Analogous to case $f_C = (w_\varepsilon\ \square)_\pi$.*

$\square$

Lemma 66 (Adjust context simulation)**.** *If $\Phi \sim \Xi$ and $A(f_O) \sqsubseteq A'(f_C)$, then $\mathbf{adjust}_{A(f_O)}(\Phi) \sim \widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)$.*

Proof. By definition of the simulation relation, $\Phi \sim \Xi \Rightarrow \exists \Phi' \subseteq \Phi$ such that $\Phi' \in \gamma(\Xi)$.
By Soundness of the abstract interpretation (proposition 7),
$\mathbf{adjust}_{A'(f_C)}(\Phi') = \Phi'' \in \gamma(\widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi))$.

By monotonicity lemmas, since $\Phi' \subseteq \Phi$, then also $\Phi'' \subseteq \mathbf{adjust}_{A'(f_C)}(\Phi)$. Since $A(f_O) \sqsubseteq A'(f_C)$, then by tag monotonicity lemma, $\mathbf{adjust}_{A'(f_C)}(\Phi) \subseteq \mathbf{adjust}_{A(f_O)}(\Phi)$. Thus by transitivity of $\subseteq$, $\Phi'' \subseteq \mathbf{adjust}_{A(f_O)}(\Phi)$.

Thus we know that $\exists \Phi'' \subseteq \mathbf{adjust}_{A(f_O)}(\Phi)$, such that $\Phi'' \in \gamma(\widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)$, which is the definition of $\mathbf{adjust}_{A(f_O)}(\Phi) \sim \widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)$.

$\square$

Lemma 67 (Inversion Lemmas for the Simulation Relation)**.**

1. *If $\Xi; \Gamma; \Sigma \Vdash (e_1, \mu_1) \sim (e_2, \mu_2)$, then:*
   - *There exists T_2 such that $\Xi; \Gamma; \Sigma \vdash_C e_2 \colon T_2$ in the conservative language type system.*
   - $e_1 = [\![e_2]\!]^\pi$.
   - *$\Gamma; \Sigma \vDash_O \mu_1$ (for the generic gradual effect checking language), $\Gamma; \Sigma \vDash_C \mu_2$ (for the conservative language), and $\mu_1 = [\![ ]\!]^\pi \circ \mu_2$.*

2. If $\Phi \sim \Xi$, then there exists a set $\Phi' \subseteq \Phi$ such that $\Phi' \in \gamma(\Xi)$.

Theorem 19 (Strong Conservative Approximation)**.** .

Let $\Xi; \Gamma; \Sigma \ \Vdash \ (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)$ and $\Phi \sim \Xi$. If $\Phi \vdash \mathrm{e}_2 \mid \mu_2 \rightsquigarrow \mathrm{e}_2' \mid \mu_2'$, then for any $\Phi' \sim \Xi$, either:

- $\Phi' \vdash \mathrm{e}_2' \mid \mu_2' \rightsquigarrow^* \mathsf{Error} \mid \mu_2'$
- *$\exists \mathrm{e}_1'$ and μ_1' such that $\Phi' \vdash \mathrm{e}_1 \mid \mu_1 \rightarrow \mathrm{e}_1' \mid \mu_1'$ and $\exists \Sigma' \supseteq \Sigma$ such that $\Xi; \Gamma; \Sigma' \ \Vdash \ (\mathrm{e}_1', \mu_1') \sim (\mathrm{e}_2', \mu_2')$.*

Proof. By structural induction over $\rightsquigarrow$. To avoid ambiguity, we use f_C for frames used in evaluation with $\rightsquigarrow$.

Case ([E-Frame] with $f_C = (w_\varepsilon\ \Box)_\pi$)**.** *($\mathrm{e}_2 = f_C[\mathrm{e}_2']$) We know by inversion lemmas for the simulation relation that $\mathrm{e}_1 = [\![\mathrm{e}_2]\!]^\pi$. By definition of $[\![]\!]^\pi$, we then know that $\mathrm{e}_1 = f_O[\mathrm{e}_1']$ with $f_O = ([\![w_\varepsilon]\!]^\pi\ \Box)$ and $\mathrm{e}_1' = [\![\mathrm{e}_2']\!]^\pi$.*

Since by inversion lemmas for the simulation relation, we know that e_2 types in the conservative semantics, we then also know by the "typing ensures well-formed conservative frames" lemma that f_C is well-formed, so that the tag $\varepsilon \in \pi$.

We can now rewrite $\mathrm{e}_1 = [\![\mathrm{e}_2]\!]^\pi$ as $f_O[[\![\mathrm{e}_2']\!]^\pi] = [\![f_C[\mathrm{e}_2']]\!]^\pi$. Since f_c is also well-formed, we can use the partial order for adjust context's tagset erasure lemma 65 to infer that $A(f_O) \sqsubseteq A'(f_C)$. This assumption will be useful to define a Φ' to usage with the structural induction hypothesis.

Since evaluation follows by [E-Frame], we use the induction hypothesis with $\mathbf{adjust}_{A'(f_C)}(\Phi) \vdash \mathrm{e}_2' \mid \mu_2 \rightsquigarrow \mathrm{e}_2'' \mid \mu_2'$, as we know that $\widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)\,; \Gamma\ \Sigma \ \Vdash \ (\mathrm{e}_1', \mu_1) \sim (\mathrm{e}_2', \mu_2)$.

Let Φ' be any $\Phi' \mathrel{\widetilde{\sim}} \Xi$. Since $A(f_O) \sqsubseteq A'(f_C)$, by the adjust context simulation lemma, $\mathbf{adjust}_{A(f_O)}(\Phi') \sim \widetilde{\mathbf{adjust}}_{A'(f_C)}(\Xi)$. Thus we then know that either:

- $\mathbf{adjust}_{A(f_O)}(\Phi') \vdash \mathrm{e}_2'' \mid \mu_2' \rightsquigarrow^* \mathsf{Error} \mid \mu_2'$. *Then* [E-Error] *applies and therefore $\Phi' \vdash f_C[\mathrm{e}_2''] \mid \mu_2' \rightsquigarrow^* \mathsf{Error} \mid \mu_2'$.*
- $\mathbf{adjust}_{A(f_O)}(\Phi') \vdash \mathrm{e}_1' \mid \mu_1 \rightarrow \mathrm{e}_1'' \mid \mu_1'$, *and $\Xi; \Gamma; \Sigma' \ \Vdash \ (\mathrm{e}_1'', \mu_1') \sim (\mathrm{e}_2'', \mu_2')$. We can therefore apply rule* [E-Frame] *to infer that $\Phi' \vdash f_O[\mathrm{e}_1'] \mid \mu_1 \rightarrow f_O[\mathrm{e}_1''] \mid \mu_1'$.*

  By inversion lemmas of the simulation relation, we know that $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}_2'' \colon T$. We can use typing rule [IT-App] *to infer that $\Xi; \Gamma; \Sigma' \vdash f_C[\mathrm{e}_2''] \colon T'$. Therefore, also $\Xi; \Gamma; \Sigma' \ \Vdash \ (f_O[\mathrm{e}_1'']\ \mu_1') \sim (f_C[\mathrm{e}_2''], \mu_2')$.*

Case ([E-Frame] with $f = v := \Box$)**.** *Analogous to* [E-Frame] *with $f = v\ \Box$.*

Case ([E-Frame] with other frames)**.** *In any other case, $f_C = f_O$, so it is analogous to* [E-Frame] *with $f = v\ \Box$, but does not require a call to the adjust context simulation lemma.*

Case (Other rules)**.** *The rest of the rules follow by structural induction. Any other rule is structurally equivalent between $\rightarrow$ and $\rightsquigarrow$.*

$\square$

Lemma 68 (Translation implies simulation)**.** *Let* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rrightarrow \mathrm{e}_2 \colon T$. *For any* μ_2 *such that* $\Gamma; \Sigma \vDash \mu_2$, *then exists* μ_1 *such that* $\Xi; \Gamma; \Sigma \Vdash (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)$.

Proof. By structural induction over the translation relation. We use $\mu_1 = [\![]\!]^{\pi} \circ \mu_2$, and inductively we collect all the preconditions to infer simulation. There is no particularly interesting case, since the translation relation ensures that always $\mathrm{e}_1 = [\![\mathrm{e}_2]\!]^{\pi}$. $\square$

Theorem 18 (Conservative Approximation)**.** . *Let* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}_1 \Rrightarrow \mathrm{e}_2 \colon T$, μ_1 *and* μ_2 *such that* $\Xi; \Gamma; \Sigma \Vdash (\mathrm{e}_1, \mu_1) \sim (\mathrm{e}_2, \mu_2)$, *and* $\Phi \sim \Xi$. *If* $\Phi \vdash \mathrm{e}_2 \mid \mu_2 \rightsquigarrow^* v_2 \mid \mu_2'$, *then* $\exists v_1$ *and* μ_1' *such that* $\Phi \vdash \mathrm{e}_1 \mid \mu_1 \rightarrow^* v_1 \mid \mu_1'$ *and* $\exists \Sigma' \supseteq \Sigma$ *such that* $\Xi; \Gamma; \Sigma' \Vdash (v_1, \mu_1') \sim (v_2, \mu_2')$.

Proof. This theorem reduces to the reflexive-transitive closure of the Strong Conservative Approximation Lemma, using $\Phi' = \Phi$. $\square$

Appendix E

Proofs Related to Gradual Type-and-Effect Systems

E.1 Properties of consistent subtyping

Theorem 23 (Consistent subtyping equivalence).

$$\exists \alpha \sim a \;.\; \alpha <: b \iff \exists \beta \sim b \;.\; a <: \beta$$

Proof. ($\Rightarrow$)

By structural induction over the type consistency definition $\sim$.

Case ([C-Refl]). *Then $a = \alpha$ and $a <: b$, so $\beta = b$ suffices.*

Case ([C-UnR]). *Then $a = \texttt{Dyn}$. By rule* [C-UnL]*, we know that $\texttt{Dyn} \sim b$ for any b and $\texttt{Dyn} <: \texttt{Dyn}$, so $\beta = \texttt{Dyn}$ suffices.*

Case ([C-UnL]). *Then $\alpha = \texttt{Dyn}$ and therefore $b = \texttt{Dyn}$. By rule* [C-UnR]*, any type $\beta \sim \texttt{Dyn}$, so in particular $\beta = a$ suffices since $a <: a$.*

Case ([C-Fun]). *Then α, a and b are function types ($a = a_1 \xrightarrow{\Xi_a} a_2$ and $b = b_1 \xrightarrow{\Xi_b} b_2$). By hypothesis we know that $\exists \alpha_1 \sim b_1$ such that $\alpha_1 <: a_1$ and by induction hypothesis we know that $\exists \beta_2 \sim b_2$ such that $a_2 <: \beta_2$. We can then build $\beta = \alpha_1 \xrightarrow{\Xi_1} \beta_2$ and $a <: \beta$.*

($\Leftarrow$) Analogous. $\square$

Lemma 69. *$\Xi_1 \sqsubseteq_{\sim} \Xi_2$ if and only if there exists a Ξ' such that $\Xi_1 \simeq \Xi'$ and $\Xi' \subseteq \Xi_2$*

Proof. $\Rightarrow$ On cases for the definition of $\sqsubseteq_{\sim}$.

Case ($¿ \in \Xi_2$). *Trivially, $\Xi' \subseteq \Xi_2$ for any Ξ'. Let $\Xi' = \{¿\} \cup (\Xi_1 \setminus |\Xi_2|)$. The question is whether $\Xi_1 \simeq \Xi'$.*

Since $¿ \in \Xi'$, $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi'$. At the same time, by construction $|\Xi'| \subseteq |\Xi_1|$ and thus in any case $\Xi' \sqsubseteq\!\!\!\!\sim \Xi_1$, meaning that $\Xi_1 \simeq \Xi'$.

Case ($|\Xi_1| \subseteq |\Xi_2|$)**.** *Trivial with $\Xi' = |\Xi_1|$.*

$\Leftarrow$ On cases for $\simeq$. Since $\Xi_1 \simeq \Xi'$, there are two possible cases:

Case ($¿ \in \Xi_1$ and $¿ \in \Xi'$)**.** *Since $\Xi' \subseteq \Xi_2$, then $¿ \in \Xi_2$ and thus $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi_2$.*

Case ($|\Xi_1| \subseteq |\Xi'|$ or $|\Xi'| \subseteq |\Xi_1|$)**.**

- *If $|\Xi_1| \subseteq |\Xi'|$, then since $\Xi' \subseteq \Xi_2$, then also $|\Xi_1| \subseteq |\Xi_2|$ and thus $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi_2$.*
- *If $|\Xi'| \subseteq |\Xi_1|$, for $\Xi' \simeq \Xi_1$ to hold, either $¿ \in \Xi'$ or $|\Xi_1| \subseteq |\Xi'|$ must be true. If the first is true, then also $¿ \in \Xi_2$ and thus $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi_2$. In the second case, by transitivity, also $|\Xi_1| \subseteq |\Xi_2|$ and thus $\Xi_1 \sqsubseteq\!\!\!\!\sim \Xi_2$.*

□

E.2 Gradual typing for type-and-effect systems without tags

Lemma 70 (Canonical Values)**.**

- *If $\Xi; \Gamma; \Sigma \vdash v\colon \texttt{Unit}$, then $v = \texttt{unit}$.*
- *If $\Xi; \Gamma; \Sigma \vdash l\colon \texttt{Ref}\ T$, then $v = l$, with $\Sigma(l) = T$.*
- *If $\Xi; \Gamma; \Sigma \vdash v\colon \texttt{Unit}$, then $v = \texttt{unit}$.*
- *If $\Xi; \Gamma; \Sigma \vdash v\colon \texttt{Dyn}$, then $v = \langle \texttt{Dyn} \Leftarrow T \rangle v'$.*

Proof. By structural induction over the typing derivation. There is only one typing rule that applies in each case and inversion of the rule provides the conclusions in the lemmas. □

Theorem 25 (Progress)**.** *Suppose $\Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$. Then either* e *is a value v, an* Error*, or $\Phi \vdash \mathrm{e} \mid \mu \rightarrow \mathrm{e}' \mid \mu'$ for all privilege sets Φ such that $\exists \Phi' \in \gamma(\Xi)$ such that $\Phi' \subseteq \Phi$ and for any store μ such that $\emptyset \mid \Sigma \vDash \mu$.*

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash \mathrm{e}\colon T$.

Case ([IT-Cast])**.** ***Since the only difference with Gradual Effect Checking arises on casts, this is the only interesting rule for this proof.*** *By induction hypothesis, either:*

- e$'$ *is a value. In this case, types T_1 and T_2 may or may not be a consistent subtype. If they are not,* [E-Cast-Bad] *always applies. If they are, we can verify cases on the structural definition of consistent subtyping. We follow the structure of*

$$a \lesssim b \iff \exists \beta \sim b\ .\ a <: \beta$$

and analyze for all the cases by definition of type consistency for T_2:

  - $\texttt{Dyn} <: T_1$, *and* T_2 *is any type. Thus by definition of subtyping* $T_1 = \texttt{Dyn}$ *and also* $T_0 = \texttt{Dyn}$. *By Canonical Values lemma 70, then* $\mathrm{e}' = \langle \texttt{Dyn} \Leftarrow T' \rangle v$. *then* $\mathrm{e} = \langle T_2 \Leftarrow \texttt{Dyn} \rangle \langle \texttt{Dyn} \Leftarrow T' \rangle v$ *and rule* [E-Cast-Merge] *applies. If* $T_2 = \texttt{Dyn}$, *then rule* [E-Cast-Id] *also applies.*
  - $T_2 = T_{21} \xrightarrow{\Xi} T_{22}$ and $\exists T'_{21}, T'_{22}$, and Ξ' *such that* $T_1 <: T'_{21} \xrightarrow{\Xi'} T'_{22}$. *By definition of subtyping, this means that* $T_1 = T_{11} \xrightarrow{\Xi''} T_{12}$. *Since* $\mathrm{e}' \colon T_0$ *with* $T_0 <: T_1$, *we also know that* T_0 *is a function type and by canonical values lemma 70 we know that* $\mathrm{e}' = \lambda x \colon T_{01}\ .\ \mathrm{e}''$ *and rule* [E-Cast-Fn] *can always be applied.*
  - $T_1 <: T_2$. *We analyze all the cases for the last step of the derivation of subtyping. If rule* [ST-Id] *is used, then rule* [E-Cast-Id] *may always be applied. If rule* [ST-Abs] *is used, we know that both* T_1 *and* T_2 *are function types and we can apply rule* [E-Cast-Fn].
- e' *is an* Error, *in which case rule* [E-Error] *applies.*
- *For any* Φ *such that* $\exists \Phi' \in \gamma(\Xi')$ *such that* $\Phi' \subseteq \Phi$, *then* $\Phi \vdash \mathrm{e}' \mid \mu \to \mathrm{e}'' \mid \mu'$ *for any* μ. *We can then always apply rule* [E-Cast-Frame].

Case (Other rules)**.** *Since we have not modified the progress theorem, we can reuse the proof strategy followed for generic gradual effect checking.*

□

Theorem 26 (Preservation)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \colon T$, *and* $\Phi \vdash \mathrm{e} \mid \mu \to \mathrm{e}' \mid \mu'$ *for* $\Phi \supseteq \Phi' \in \gamma(\Xi)$ *and* $\Gamma \mid \Sigma \vDash \mu$, *then* $\Gamma \mid \Sigma' \vDash \mu'$ *and* $\Xi; \Gamma; \Sigma' \vdash \mathrm{e}' \colon T'$ *for some* $T' <: T$ *and* $\exists \Sigma' \supseteq \Sigma$.

Proof. By structural induction over the typing derivation and the applicable evaluation rules. Since the only typing rule that changes is [IT-Cast], we focus on that case and rely on the previously introduced proofs for preservation for the interesting reader. The proof strategy is equivalent for all of these rules.

Case ([IT-Cast] and rule [E-Cast-Frame])**.** *By induction hypothesis, there exists* T' *such that* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}' \colon T'$ *and* $T' <: T_0$. *By transitivity of subtyping, we can reuse rule* [IT-Cast] *to infer that* $\Xi; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle \mathrm{e}' \colon T_2$.

Case ([IT-Cast] and rule [E-Cast-Id])**.** *Trivial from the typing derivation, since* $\Xi; \Gamma; \Sigma \vdash v \colon T_0$, $T_0 <: T_1$ *and* $T_1 = T_2$.

Case ([IT-Cast] and rule [E-Cast-Merge])**.** *From the typing derivation we know that* $\Xi; \Gamma; \Sigma \vdash v \colon T_0$, *with* $T_0 <: T_1$. *We then may reuse rule* [IT-Cast] *to prove that* $\Xi; \Gamma; \Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle v \colon T_2$.

Case ([IT-Cast] and rule [E-Cast-Bad])**.** *We can always use* [IT-Error] *to reconstruct the required type* T.

Case ([IT-Cast] and rule [E-Cast-Fn])**.** *From the typing derivation we know that* $\mathrm{e} \colon T_{01} \xrightarrow{\Xi_0} T_{02}$. *We can follow the same derivation done in theorem 16 for this very case and prove that types are preserved.*

□

Theorem 27 (Translation preserves typing)**.** *If* $\Xi; \Gamma; \Sigma \vdash \mathrm{e} \Rightarrow \mathrm{e}' : T$ *in the source language then* $\Xi; \Gamma; \Sigma \vdash \mathrm{e}' : T$ *in the internal language.*

Proof. By structural induction over the translation derivation rules. The casts introduced for the translation algorithm preserve typing in the intermediate language. There is no interesting details for the proof, besides of what is already presented in the proof of theorem 17. □