A Theory of Gradual Effect Systems

Felipe Bañados Schwerter [*]
PLEIAD Lab
Computer Science Department (DCC)
University of Chile
fbanados@dcc.uchile.cl

Ronald Garcia [†]
Software Practices Lab
Department of Computer Science
University of British Columbia
rxg@cs.ubc.ca

Éric Tanter [‡]
PLEIAD Lab
Computer Science Department (DCC)
University of Chile
etanter@dcc.uchile.cl

Abstract

Effect systems have the potential to help software developers, but their practical adoption has been very limited. We conjecture that this limited adoption is due in part to the difficulty of transitioning from a system where effects are implicit and unrestricted to a system with a static effect discipline, which must settle for conservative checking in order to be decidable. To address this hindrance, we develop a theory of gradual effect checking, which makes it possible to incrementally annotate and statically check effects, while still rejecting statically inconsistent programs. We extend the generic type-and-effect framework of Marino and Millstein with a notion of unknown effects, which turns out to be significantly more subtle than unknown types in traditional gradual typing. We appeal to abstract interpretation to develop and validate the concepts of gradual effect checking. We also demonstrate how an effect system formulated in Marino and Millstein's framework can be automatically extended to support gradual checking.

Categories and Subject Descriptors D.3.1 [*Software*]: Programming Languages—Formal Definitions and Theory

Keywords Type-and-effect systems; gradual typing; abstract interpretation

1. Introduction

Type-and-effect systems allow static reasoning about the computational effects of programs. Effect systems were originally introduced to safely support mutable variables in functional languages [11], but more recently, effect systems have been developed for a variety of effect domains, e.g., I/O, exceptions, locking, atomicity, confinement, and purity [1–3, 12, 13, 17, 18].

To abstract from specific effect domains and account for effect systems in general, Marino and Millstein (M&M) developed a generic effect system [15]. In their framework, effect systems are seen as granting and checking privileges. Genericity is obtained by parameterizing the type system and runtime semantics of a language with two operations, called *adjust* and *check*, which respectively specify how the set of held privileges is adjusted and checked during type checking. A particular effect system is instantiated by providing a syntax for effects and a definition of the check and adjust operations. They demonstrate that several effect systems from the literature can be formulated as instantiations of the generic framework.

The generic effect system underlies the design of the Scala effect checker plugin, which extends the M&M framework with a form of effect polymorphism [17]. Several specific effect systems for this plugin include IO effects, exceptions, and more recently, state effects [18].

Despite their obvious advantages in terms of static reasoning, the adoption of effect systems has been rather limited in practice. While effect polymorphism supports the definition of higher-order functions that are polymorphic in the effects of their arguments (e.g., map), writing fully-annotated effectful programs is complex, and is hardly ever done.[1]

We conjecture that an important reason for the limited adoption of effect systems is the difficulty of transitioning from a system where 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. We follow the line of work on *gradual* verification of program properties (e.g., gradual typing [22, 23], gradual ownership types [21], gradual typestate [10, 26]), and develop a theory of gradual effect systems. Our contributions are as follows:

- We shed light on the meaning of gradual effect checking, and its fundamental differences from traditional gradual typing, by formulating it in the framework of abstract interpretation [4]. Abstract interpretation allows us to clearly and precisely specify otherwise informal design intentions about gradual effect systems. Key notions like the meaning of unknown effects, consistent privilege sets, and consistent containment between them, are defined in terms of abstraction and concretization operations.
- We extend the generic effect system of Marino and Millstein into a generic framework for gradual effects. As with gradual typing, our approach relies on a translation to an internal language with explicit checks and casts. The nature of these checks and casts is, however, quite different. We prove the type safety of the internal language and the preservation of typability by the translation.
- We demonstrate how an effect system formulated in the M&M framework can be immediately extended to support gradual

[*] Funded by CONICYT-PCHA/Magíster Nacional/2013-22130167

[†] Partially funded by an NSERC discovery grant

[‡] Partially funded by FONDECYT project 1110051

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628149

[1] Pure functional languages like Haskell and Clean are notable exceptions.

checking by lifting existing adjust and check functions to the gradual setting.

- We present a concrete instantiation of the generic framework to gradually check exceptions. The resulting system is compact and provides a tangible and self-contained example of gradual effect checking.

We believe this work can help effect system developers extend their designs with support for gradual checking, thereby facilitating their adoption.

2. Background and Motivation

In this section, we introduce the idea of static effect checking, and give an intuition for how gradual effect checking is related. We finish with a brief introduction to the M&M generic framework for specifying type-and-effect systems.

2.1 Effect Systems

Effect systems classify the computational effects that an expression performs when evaluated. To illustrate this idea, consider a simple functional language with integers, booleans, and references. We focus on three mutable state effects: `alloc`, `read` and `write`.

A value such as 7 or $(\lambda x : \texttt{Int} \,.\, x)$ has no effect; neither does an arithmetic expression whose sub-expressions have no effect, such as $7 + 12$. Conversely, creating a reference such as `ref` 6 has type `Ref Int` and effect `alloc`. Similarly, an assignment expression such as $x := 2$ has type `Unit` and effect `write`, and dereferencing a reference $!x$ has the type of the reference content, and effect `read`.

Since functions are values they have no effects, but they may perform effects when applied. To modularly check effects, then, function types are annotated with the effects of the function body. For instance, the function f:

$$f = \lambda x : \texttt{Ref Int} \,.\, !\, x$$

has type $(\texttt{Ref Int}) \xrightarrow{\{\texttt{read}\}} \texttt{Int}$ because a `read` effect happens during the application of the function. Note that the effect may not happen during some applications of a function, for instance (assuming $y : \texttt{Bool}$ is in scope):

$$g = \lambda x : \texttt{Ref Int} \,.\, \texttt{if } y \texttt{ then } x := 3; 0 \texttt{ else } 1$$

has type $(\texttt{Ref Int}) \xrightarrow{\{\texttt{write}\}} \texttt{Int}$ because its applications *may* perform a `write` effect.

Of course, an expression can induce more than one effect, hence the use of *effect sets* in the annotations. Though the language does not define any notion of subtyping on types themselves, effect sets leads to a natural notion of subtyping [25]. Consider the following higher-order function:

$$h : ((\texttt{Ref Int}) \xrightarrow{\{\texttt{read},\texttt{alloc}\}} \texttt{Int}) \xrightarrow{\cdots} \texttt{Int}$$

This function restricts the effects of its function argument to $\{\texttt{read}, \texttt{alloc}\}$. Intuitively, it is valid to apply h to f, whose effect set is $\{\texttt{read}\}$, because that would not violate the expectations of h. In other words:

$$(\texttt{Ref Int}) \xrightarrow{\{\texttt{read}\}} \texttt{Int} <: (\texttt{Ref Int}) \xrightarrow{\{\texttt{read},\texttt{alloc}\}} \texttt{Int}$$

because the effects of the former are a subset of the latter. Conversely, it is invalid to apply h to g.

From effects to privileges. Following Marino and Millstein [15], we interpret effect systems in terms of *privilege checking*: to each effectful operation corresponds a privilege required to perform it. For instance, we can view `alloc`, `read` and `write` as the privileges required to respectively allocate, dereference and assign a reference. In this framework, the function type $(\texttt{Ref Int}) \xrightarrow{\{\texttt{read}\}} \texttt{Int}$ is interpreted as the type of a function that *requires* the `read` privilege in order to be applied. Effect checking ensures that sufficient privileges have been granted to perform effectful operations.

2.2 Towards Gradual Effect Checking

Programming in the presence of a statically checked discipline brings stronger guarantees about the behavior of programs, but doing so is demanding. In addition, one is limited by the fact that the checker is conservative. Recently, several practical effect systems have been applied to existing libraries, and the empirical findings highlight the need to occasionally bypass static effect checking [12, 17].

For instance, the JavaUI effect system [12], which prevents non-UI threads from accessing UI objects or invoking UI-thread-only methods, cannot be used to verify libraries that dynamically check which thread they are running on and adapt their behavior accordingly. As explained by the authors, the patterns of dynamic checks they found in existing code go beyond simple if-then-else statements and so cannot be handled simply by specializing the static type system. While JavaUI lives with this limitation, the Scala effect plugin [17] has recently been updated with an `@unchecked` annotation to simply turn off effect checking locally. The use of this annotation however breaks the guarantees offered by the effect system, since there are no associated runtime checks.

In the realm of standard type systems, gradual typing [23] is a promising approach that alleviates the complexity and conservativeness issues by integrating static and dynamic checking seamlessly and safely. The appeal of gradual typing has inspired the development of gradual approaches to a variety of type disciplines, including objects [14, 22, 24], ownership types [21], typestates [10, 26], and information flow typing [6].

This paper develops gradual effect checking, following the core design principles that are common to all gradual checking approaches: *(a)* The same language can support both fully static and fully dynamic checking of program properties. *(b)* The programmer has fine grained control over the static-to-dynamic spectrum. *(c)* The gradual checker statically rejects programs on the basis that they surely go wrong; programs that *may* go right are accepted statically, but subject to dynamic checking. *(d)* Runtime checks are minimized based on static information. *(e)* Violations of properties are detected as runtime errors—there are no stuck programs.

2.3 Gradual Effects in Action

Recall the function g defined in Sec. 2.1, which requires $\{\texttt{write}\}$ privileges. The program $h\ g$ is rejected because h only accepts functions that require $\{\texttt{read}, \texttt{alloc}\}$ privileges. Even if the programmer knows that for a particular use of g, the `if` condition y is false—and thus needs no `write` privilege after all—the program is rejected.

In direct analogy to the unknown type ? introduced by Siek and Taha [23] for gradual typing, we introduce *statically unknown privileges*, denoted ¿, to our language. One can ascribe unknown privileges to any expression e, using the notation $e :: ¿$. For instance, if g is defined as:

$$g = \lambda x : \texttt{Ref Int} \,.\, \texttt{if } y \texttt{ then } (x := 3; 0) :: ¿ \texttt{ else } 1$$

then it is given the type $(\texttt{Ref Int}) \xrightarrow{\{¿\}} \texttt{Int}$. The application $h\ g$ is now statically accepted by the gradual effect system. At runtime, if only the `else` branch is ever executed, then no error occurs. If, on the other hand, the programmer wrongly assumed that g would not require the `write` privilege and the `then` branch is executed, an effect error is raised, preventing the assignment to x.

The ascription expression $e :: ¿$ introduces dynamic checking semantics. Statically, it *hides* the privileges required by e from the surrounding context, and allows the subexpressions of e to attempt effectful operations. At runtime, checks occur to ensure that the static privileges that e requires are available as needed.

One can partially expose (and hence dynamically check) required privileges by ascribing specific privileges in addition to $¿$. For instance, $e :: \{\texttt{read}, ¿\}$ statically reveals that e requires the `read` privilege, but hides other potential requirements.[2]

The static-to-dynamic spectrum We have illustrated the use of gradual effect checking from the point of view of "softening static checking"—introducing islands of dynamicity in an otherwise static verification process. Gradual verification is about supporting both ends of the static-to-dynamic spectrum as well as any middle ground. We now discuss gradual effect checking from the point of view of "hardening dynamic checking"—introducing static checks in an otherwise dynamic verification process.

A fully-dynamic effectful program corresponds to a gradually-typed program without any effect-related annotations in which all effectful operations are wrapped by a $:: ¿$ ascription.[3] Static checking trivially succeeds because all expressions hide their required privileges. Forbidden effects will only be detected at runtime. Then, the programmer can progressively introduce static privilege annotations (function argument types, ascriptions) and remove $:: ¿$ ascriptions, statically revealing required privileges. The static checker may reject the program if inconsistencies are detected, or it may accept the program and runtime errors may occur. As more static information is revealed, fewer dynamic checks are required. The effect discipline is hardened.

2.4 Generic Effect Systems

To avoid re-inventing gradual effects for each possible effect discipline, we build on the generic effect framework Marino and Millstein (M&M) [15], which we briefly describe in this section.

The M&M effect framework defines a parameterized typing judgment $\Phi; \Gamma; \Sigma \vdash e : T$. 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:

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

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.

To increase its overall expressiveness, the framework also incorporates a notion of *tags* ε, 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. 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 $T \equiv \pi\rho$. For more precise control, effect disciplines can associate tags to privileges e.g., $\texttt{read}(\varepsilon_1)$, $\texttt{read}(\varepsilon_2)$, etc. [4]

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

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

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:

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

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:

$$\text{T-Asgn}\ \frac{\begin{array}{c}\textbf{adjust}_{\downarrow := \uparrow}(\Phi)\,; \Gamma; \Sigma \vdash e_1 : \pi_1 \texttt{Ref}\ T_1 \\ \textbf{adjust}_{\pi_1 := \downarrow}(\Phi)\,; \Gamma; \Sigma \vdash e_2 : \pi_2\rho_2 \\ \textbf{check}_{\pi_1 := \pi_2}(\Phi) \qquad \pi_2\rho_2 <: T_1\end{array}}{\Phi; \Gamma; \Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

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, $\textbf{adjust}_{\pi_1 := \downarrow}$ knows which tags are associated with typing e_1. Finally, $\textbf{check}_{\pi_1 := \pi_2}$ 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.

[2] In a static effect system, an effect ascription $e :: \{\texttt{read}\}$ is directly analogous to a type ascription [16]. Static effect ascriptions were introduced by Gifford and Lucassen [11].

[3] This corresponds to the translation of terms from the untyped λ-calculus to the gradually-typed λ-calculus, which lifts all functions to the $? \to ?$ type to introduce runtime checks [23].

[4] Gradual effects are compatible with effect systems that do not need tags. See Sec. 5.

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

Property 2 (Tag Monotonicity)**.**

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

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, $\texttt{ref}\pi_1 \sqsubseteq \texttt{ref}\pi_2$ 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.

3. Gradual Effects as an Abstract Interpretation

In this section we present a formal analysis of gradual effects, guided by the design principles presented in Sec. 2.2. We use abstract interpretation [4] 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.

3.1 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 [23], 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 leafs. Two types are deemed consistent whenever their known parts match up exactly. For instance, the types $? \to \texttt{Int}$ and $\texttt{Bool} \to ?$ are consistent because their $\to$ 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 Sec. 2.1, 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 [22], 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 paper then uses the framework as needed to introduce concepts and formalize gradual effect checking.

3.2 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 Sec. 2.1:

$$\Phi \in \textbf{PrivSet} = \mathcal{P}(\{\texttt{read}, \texttt{write}, \texttt{alloc}\})$$
$$\Xi \in \textbf{CPrivSet} = \mathcal{P}(\{\texttt{read}, \texttt{write}, \texttt{alloc}, ¿\})$$

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.[5]

Definition 1 (Concretization)**.** *Let* $\gamma : \textbf{CPrivSet} \to \mathcal{P}(\textbf{PrivSet})$ *be defined as follows:*

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

Reconsidering our two example consistent privilege sets, we find that

$$\gamma(\Xi_1) = \{\{\texttt{read}\}\}$$
$$\gamma(\Xi_2) = \left\{ \begin{array}{c} \{\texttt{read}, \texttt{write}\}, \{\texttt{read}, \texttt{alloc}\}, \\ \{\texttt{read}\}, \{\texttt{read}, \texttt{alloc}, \texttt{write}\} \end{array} \right\}$$

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 2 (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

[5] We introduce an *abstraction* function α in Sec. 3.4

than others. For instance, $\{\texttt{read}\}$ is strictly more precise than $\{\texttt{read}, ¿\}$ because $\{\texttt{read}\} \sqsubseteq \{\texttt{read}, ¿\}$ but not vice-versa.

3.3 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 (Sec. 2.4), 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 3 (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:*

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

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

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

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 (cf. Sec. 5).

3.4 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 $\mathbf{adjust}_A : \mathbf{PrivSet} \rightarrow \mathbf{PrivSet}$, which alter the set of available effect privileges (Sec. 2.4). Using abstract interpretation, we lift these to *consistent* adjust functions $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \rightarrow \mathbf{CPrivSet}$. 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 $\alpha : \mathcal{P}(\mathbf{PrivSet}) \rightarrow \mathbf{CPrivSet}$ 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 1 (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 5 (Abstraction). *Let* $\alpha : \mathcal{P}(\mathbf{PrivSet}) \rightarrow \mathbf{CPrivSet}$ *be defined as follows*[6]*:*

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

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 2 (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 most precise one[7]. 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 6 (Consistent Adjust).
Let $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \rightarrow \mathbf{CPrivSet}$ *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. 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 annotations, an internal language that introduces runtime checks, and a type-directed translation from the former to the latter.

4.1 The Source Language

The core language (Fig. 1) is a simply-typed functional language with a unit value, mutable state, and effect ascriptions $e :: \Xi$. 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

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

[7] Abstract interpretation literature expresses this in part by saying that α and γ form a *Galois connection*[5].

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

$$\varepsilon \in \mathsf{Tags} \,.\, \pi \in \mathcal{P}(\mathsf{Tags})$$

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

Figure 1. Syntax of the source language

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. Each adjust context is determined by an evaluation context frame f (Sec. 4.2). They index $\widetilde{\textbf{adjust}}_A$ 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 $\widetilde{\textbf{check}}_C$ to determine which privileges are needed to perform the operation.

Fig. 2 presents the type system. The judgment $\Xi; \Gamma; \Sigma \vdash e : T$ 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 (similar to [T-Asgn] in Sec. 2.4) 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. $\widetilde{\textbf{adjust}}$ and $\widetilde{\textbf{check}}$ are defined in Sec. 3, 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 7 (Consistent Containment)**.** Ξ_1 *is* consistently contained *in* Ξ_2*, notation* $\Xi_1 \mathrel{\underset{\sim}{\sqsubset}} \Xi_2$ *if and only if* $\Phi_1 \subseteq \Phi_2$ *for* some $\Phi_1 \in \gamma(\Phi_1)$ *and* $\Phi_2 \in \gamma(\Xi_2)$[8].

[8] We give $\mathrel{\underset{\sim}{\sqsubset}}$ a simple direct characterization in Sec. 4.2.

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

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

$$\text{T-Unit}\;\frac{}{\Xi; \Gamma; \Sigma \vdash \texttt{unit}_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

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

$$\text{T-App}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 : \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 : \pi_2\rho_2 \\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \qquad \widetilde{\textbf{check}}_{\pi_1\pi_2}(\Xi)\end{array}}{\Xi; \Gamma; \Sigma \vdash e_1\; e_2 : T_3}$$

$$\text{T-Eff}\;\frac{\Xi_1; \Gamma; \Sigma \vdash e\colon T \qquad \Xi_1 \mathrel{\underset{\sim}{\sqsubset}} \Xi}{\Xi; \Gamma; \Sigma \vdash (e :: \Xi_1) : T}$$

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

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

$$\text{T-Asgn}\;\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 : \pi_1\texttt{Ref}\; T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 : \pi_2\rho_2 \\ \widetilde{\textbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1\end{array}}{\Xi; \Gamma; \Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Figure 2. Type system for the source language

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.

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

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 [22].

All other rules in the type system can be characterized as consistent liftings of the corresponding M&M rules. Each uses $\widetilde{\textbf{adjust}}_A$ to type subexpressions, and $\widetilde{\textbf{check}}_C$ 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 consistently contained in the current consistent privileges. Ascribing ¿ delays some privilege checks to runtime, as discussed next.

4.2 The Internal Language

The semantics of the source language is given by a type-directed translation to an internal language that makes runtime checks ex-

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

Figure 3. Syntax of the internal language

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

$$
\text{IT-App}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 : \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1\pi_2}(\Xi) \qquad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <: \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3\end{array}}{\Xi; \Gamma; \Sigma \vdash e_1\ e_2 : T_3}
$$

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

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

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

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

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

$$
\text{IT-Asgn}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 : \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <: T_1\end{array}}{\Xi; \Gamma; \Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}}
$$

Figure 4. Typing rules for the internal language

plicit. This section presents the internal language. The translation is presented in Sec. 4.3.

Fig. 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.

Static semantics The type system of the internal language (Fig. 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 checks to guarantee satisfaction before reaching an effectful operation. For this reason, uses of $\widetilde{\textbf{check}}$ are replaced with ***strict-check*** (Sec. 3.3, Def. 4). Consistent subtyping $\lesssim$ is replaced with a notion of subtyping $<:$ that is based on ordinary set containment for consistent privilege sets and tags:

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

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 $\Xi = \{\mathtt{read}, ?\}$ 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 8 (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: $|\Xi| = \Xi \setminus \{¿\}$.[9]

Using the notion of static part, we define the concept of *static containment* for consistent privilege sets.

Definition 9 (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 set Φ' [25]. 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 3.
If $\textbf{\textit{strict-check}}_C(\Xi_1)$ *and* $\Xi_1 \subseteq \Xi_2$ *then* $\textbf{\textit{strict-check}}_C(\Xi_2)$.

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 4. *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.

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

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

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

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

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

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

E-Cast-Fn

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

Figure 5. Small-step semantics of the internal language

Proposition 5 (Strong Effect Subsumption).
If $\Xi_1; \Gamma; \Sigma \vdash e\colon T$ *and* $\Xi_1 \le \Xi_2$, *then* $\Xi_2; \Gamma; \Sigma \vdash e\colon T$.

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

Corollary 6 (Effect Subsumption).
If $\Xi_1; \Gamma; \Sigma \vdash e\colon T$ *and* $\Xi_1 \subseteq \Xi_2$, *then* $\Xi_2; \Gamma; \Sigma \vdash e\colon T$.

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.[10]

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 7.

1. $\Xi_1 \underset{\sim}{\sqsubset} \Xi_2$ *if and only if* $\Xi_1 \subseteq \Xi_2$ *or* $¿ \in \Xi_2$.
2. ***strict-check***$_C(\Xi)$ *if and only if* $\mathbf{check}_C(|\Xi|)$.

[10] Note that $\Phi \cup \Xi$ is the same as lifting the function $f(\Phi') = \Phi \cup \Phi'$, and $\Phi \underset{\sim}{\sqsubset} \Xi$ is the same as lifting the predicate $P(\Phi') = \Phi \subseteq \Phi'$.

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

Proposition 8.

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 Fig. 5 presents the evaluation rules of the internal language. The judgment $\Phi \vdash e \mid \mu \to e' \mid \mu'$ 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 **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 $\textbf{check}_C$ 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: $\{\texttt{read}, \texttt{alloc}\} \mathrel{\underset{\sim}{\sqsubseteq}} \{\texttt{read}, ¿\}$ because `alloc` *could* be in a representative of $\{\texttt{read}, ¿\}$, but $\{\texttt{read}, \texttt{alloc}\} \not\mathrel{\underset{\sim}{\sqsubseteq}} \{\texttt{read}, ¿\}$ 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 [8]. 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 [27]. 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.[11]

Theorem 9 (Progress)**.** *Suppose $\Xi; \emptyset; \Sigma \vdash e : T$. Then either e is a value v, an* Error*, or $\Phi \vdash e \mid \mu \to e' \mid \mu'$ for all privilege sets $\Phi \in \gamma(\Xi)$ and for any store μ such that $\emptyset \mid \Sigma \vdash \mu$.*

Proof. By structural induction over derivations of $\Xi; \emptyset; \Sigma \vdash e : T$. ☐

Theorem 10 (Preservation)**.** *If $\Xi; \Gamma; \Sigma \vdash e : T$, and $\Phi \vdash e \mid \mu \to e' \mid \mu'$ for $\Phi \in \gamma(\Xi)$ and $\Gamma \mid \Sigma \vdash \mu$, then $\Gamma \mid \Sigma' \vdash \mu'$ and $\Xi; \Gamma; \Sigma' \vdash e' : T'$ for some $T' <: T$ and $\Sigma' \supseteq \Sigma$.*

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. ☐

4.3 Translating Source Programs to the Internal Language

Fig. 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 Sec. 4.2 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)

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

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

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

$$\text{C-Unit}\ \frac{}{\Xi; \Gamma; \Sigma \vdash \texttt{unit}_\varepsilon \Rightarrow \texttt{unit}_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

$$\text{C-Var}\ \frac{\Gamma(x) = T}{\Xi; \Gamma; \Sigma \vdash x \Rightarrow x : T}$$

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

$$\text{C-App}\ \frac{\begin{array}{c} \widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \\ e''_1 = (\ \langle\langle \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \rangle\rangle\ e_1') \\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \\ \widetilde{\textbf{check}}_{\pi_1\pi_2}(\Xi) \qquad \Phi = \Delta_{\pi_1\pi_2}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash e_1\ e_2 \Rightarrow \textit{insert-has?}(\Phi, e''_1\ e_2') : T_3}$$

$$\text{C-Eff}\ \frac{\Xi_1; \Gamma; \Sigma \vdash e \Rightarrow e' : T \qquad \Xi_1 \mathrel{\underset{\sim}{\sqsubseteq}} \Xi \qquad \Phi = (|\Xi_1| \setminus |\Xi|)}{\Xi; \Gamma; \Sigma \vdash (e :: \Xi_1) \Rightarrow \textit{insert-has?}(\Phi, \texttt{restrict}\ \Xi_1\ e') : T}$$

$$\text{C-Ref}\ \frac{\begin{array}{c} \widetilde{\textbf{adjust}}_{\texttt{ref}\,\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e \Rightarrow e' : \pi\rho \\ \widetilde{\textbf{check}}_{\texttt{ref}\ \pi}(\Xi) \qquad \Phi = \Delta_{\texttt{ref}\ \pi}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (\texttt{ref}\ e)_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, \left(\texttt{ref}\ e'\right)_\varepsilon) : \{\varepsilon\}\texttt{Ref}\ \pi\rho}$$

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

$$\text{C-Asgn}\ \frac{\begin{array}{c} \widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi)\,; \Gamma; \Sigma \vdash e_1 \Rightarrow e_1' : \pi_1\texttt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi)\,; \Gamma; \Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \\ \widetilde{\textbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1 \qquad \Phi = \Delta_{\pi_1:=\pi_2}(\Xi) \end{array}}{\Xi; \Gamma; \Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \textit{insert-has?}(\Phi, \left(e_1' := e_2'\right)_\varepsilon) : \{\varepsilon\}\texttt{Unit}}$$

Figure 6. Translation of source programs to the internal language

must be checked. Since the generic framework imposes only privilege and 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:

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

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 $\widetilde{\textbf{check}}_C(\{¿\})$ 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.[12] 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 10 (Minimal Privilege Check)**.** *Let C be some checking context. Then define Δ_C :* **CPrivSet** $\rightarrow$ **PrivSet** *as follows:*

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

The Δ_C function transforms a given consistent privilege set into the minimal conservative set of additional privileges needed to safely pass the $\textbf{check}_C$ 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.

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

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

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:

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

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.[13]

Crucially, the translation system preserves typing.

Theorem 11 (Translation preserves typing)**.** *If $\Xi; \Gamma; \Sigma \vdash e \Rightarrow e' : T$ in the source language then $\Xi; \Gamma; \Sigma \vdash e' : T$ 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. ☐

[12] 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.

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

$$\begin{array}{rcll}
e & ::= & \ldots \mid \texttt{raise}\ s_T(e) & \text{Terms} \\
 & & \mid \texttt{try}\ e\ \texttt{handle}\ s_T(x)\ .\ e & \\
f & ::= & f' \mid \texttt{try}\ \square\ \texttt{handle}\ s_T\ e & \text{Source Frames} \\
f' & ::= & \textit{(Original Source Frames)} & \text{Propagating Frames} \\
 & & \mid \texttt{raise}\ s_T(\square) & \\
C & ::= & \ldots \mid \texttt{raise}\ s_T(\pi) & \text{Check Contexts} \\
 & & \mid \texttt{try}\ \pi\ \texttt{handle}\ s_T\ \uparrow & \\
A & ::= & \ldots \mid \texttt{raise}\ s_T(\downarrow) & \text{Adjust Contexts} \\
 & & \mid \texttt{try}\ \downarrow\ \texttt{handle}\ s_T\ \uparrow &
\end{array}$$

Figure 7. Syntax for a Gradual Effect System with Exceptions

$$\text{E-Raise-Frame}\ \frac{\textbf{check}_{\texttt{raise}\ s_T(\{\bullet\})}(\Phi)}{\Phi \vdash f'[\texttt{raise}\ s_T(v)] \mid \mu \rightarrow \texttt{raise}\ s_T(v) \mid \mu}$$

$$\text{E-Try-V}\ \frac{\textbf{check}_{\texttt{try}\ \{\bullet\}\ \texttt{handle}\ s_T \uparrow}(\Phi)}{\Phi \vdash \texttt{try}\ v\ \texttt{handle}\ s_T(x).e \mid \mu \rightarrow v \mid \mu}$$

$$\text{E-Try-T}\ \frac{\textbf{check}_{\texttt{try}\ \emptyset\ \texttt{handle}\ s_T \uparrow}(\Phi)}{\Phi \vdash \texttt{try}\ \texttt{raise}\ s_T(v)\ \texttt{handle}\ s_T(x).e \mid \mu \rightarrow [{}^{v}/_{x}]\ e \mid \mu e}$$

E-Try-F

$$\frac{\textbf{check}_{\texttt{raise}\ s_{T_1}(\{\bullet\})}(\Phi)}{\Phi \vdash \texttt{try}\ \texttt{raise}\ s_{T_1}(v)\ \texttt{handle}\ s_{T_2}(x).e \mid \mu \rightarrow \texttt{raise}\ s_{T_1}(v) \mid \mu}$$

Figure 8. Evaluation rules added to the operational semantics for a system with exceptions

5. Example: Gradual Effects for Exceptions

In this section we show how to use our framework to define systems with richer language features. We extend the language with exception handling and introduce an effect discipline that verifies that every raised exception is caught by some handler. We introduce new syntax; privilege and tag domains; adjust and check operations and contexts; and typing, translation, and evaluation rules. Note that the example system is general enough to allow different effect disciplines for exceptions.

The language introduces an infinite set of exception constructors s_T, which are indexed on the type T of argument that they carry as a payload. An exception is triggered by the $\texttt{raise}\ s_T(e)$ expression, which indicates that the expression e should be evaluated to a value of type T, wrapped in the exception constructor, and raised. An exception handler, $\texttt{try}\ e_1\ \texttt{handle}\ s_T(x).e_2$, attempts to evaluate the expression e_1. If successful, its result is returned; if e_1 raises a s_T exception, it binds the payload to x and evaluates e_2.

We also introduce new adjust and check contexts. These contexts are used to parameterize different effect disciplines over the same constructs. They are used by the **adjust** and **check** functions in the operational semantics, by the type system and the translation algorithm. Following M&M, we define a new check context for each new redex and a new adjust context for each new evaluation frame.

Fig. 8 presents the semantics for exceptions in our system. Exceptions propagate out of evaluation frames by rule [E-Raise-Frame] until they are caught by a matching handler. Since handlers are also evaluation frames, we must distinguish the rest of the evaluation frames from handlers. As presented in Fig. 7, we call non-handler frames "Propagating Frames".

A `try` handler first reduces the guarded expression. If it is a value, the exception handler is discarded through rule [E-Try-V]. If

$$\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\texttt{raise}\ s_T(\downarrow)}(\Xi)\,;\Gamma;\Sigma\vdash e\colon T\\ \widetilde{\textbf{check}}_{\texttt{raise}\ s_T(\{\bullet\})}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{raise}\ s_T(e)\colon T'}$$

$$\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\texttt{try}\ \downarrow\texttt{handle}\ s_T\uparrow}(\Xi)\,;\Gamma;\Sigma\vdash e_1\colon T_1\\ \Xi;\Gamma,x\colon T;\Sigma\vdash e_2\colon T_2 \qquad T_2\lesssim T_1\\ \widetilde{\textbf{check}}_{\texttt{try}\ \{\bullet\}\texttt{handle}\ s_T\uparrow}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{try}\ e_1\ \texttt{handle}\ s_T(x).e_2\colon T_1}$$

Figure 9. Source language typing rules for exceptions

the guarded expression reduces to an exception whose constructor matches the handler, rule [E-Try-T] substitutes the payload value in the handling expression. If the constructor does not match the handler, the exception is propagated by rule [E-Try-F], and the handler discarded.

Rule [E-Try-T] uses $\emptyset$ in the check context instead of a tagset because the guarded expression produced an exception instead of a value. The type system does not relate the type of the exception payload to the type of the guarded expression, so when **check** is evaluated it cannot access tag information related to the guarded expression. We followed the most conservative strategy for this case. Thanks to the tag monotonicity property, we know that **check** holds with $\emptyset$ if it holds for any particular π because $\texttt{try}\ \emptyset\ \texttt{handle}\ s_T\uparrow\ \sqsubseteq \texttt{try}\ \pi\ \texttt{handle}\ s_T\uparrow$.

The new source language typing rules are presented in Figure 9. The corresponding typing rules for the internal language follow the same pattern as for rules in the general framework: $\widetilde{\textbf{check}}$ is replaced by ***strict-check*** and $\lesssim$ is replaced by $<:$. In the translation system, the rules introduce Δ_C and *insert-has?*.

As presented so far, our gradual effect system with exceptions does not enforce any particular effect discipline. To do so, we need to define both a domain for privileges and concrete **check** and **adjust** functions. We instantiate privileges **Priv** to be the exception constructors (of the form s_T), and provide the following definitions for **check** and **adjust**, which capture the standard effect discipline for exceptions:

$$\begin{array}{c}\textbf{check}_{\texttt{raise}\ s_T(\pi)}(\Phi) \iff s_T\in\Phi\\ \textbf{check}_C(\Phi)\ \text{ holds for all other } C\end{array}$$

$$\begin{array}{rcl}\textbf{adjust}_{\texttt{try}\ \downarrow\ \texttt{handle}\ s_T\ \uparrow}(\Phi) & = & \Phi\cup\{s_T\}\\ \textbf{adjust}_A(\Phi) & = & \Phi\ \text{otherwise}\end{array}$$

Note that this effect discipline does not require tags, so technically we use a singleton set for the universe of tags ($\varepsilon\in\{\bullet\}$). In practice the tags can be removed altogether.

Implementation With a concrete effect discipline, an instance of the general effect system can be specialized to produce concrete operational semantics, type system and translation algorithm rules, inlining the calls to **check** and **adjust**. Figure 10 presents specialized translation rules for the concrete discipline we have chosen. These rules directly incorporate the semantics of the *insert-has?* function, separating its two cases across two separate translation rules. Since the only non-trivial check context in the effect discipline is $\texttt{raise}\ s_T(\pi)$, we provide separate rules only for `raise` using the feasible values for $\Delta_{\texttt{raise}\ s_T(\pi)}$ in each case ($\emptyset$ or $\{s_T\}$).

Illustration By making the exception checking discipline gradual, we achieve a more expressive language. Consider the following function, which also uses conditionals and arithmetic expressions:

$$\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash e\Rightarrow e'\colon T_1\\ \{s_{T_1}\}\subseteq\Xi\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{raise}\ s_{T_1}(e)\Rightarrow \texttt{raise}\ s_{T_1}(e')\colon T_2}$$

$$\frac{\begin{array}{c}\Xi;\Gamma;\Sigma\vdash e\Rightarrow e'\colon T_1\\ \{s_{T_1}\}\not\subseteq\Xi \qquad \{s_{T_1}\}\mathrel{\widetilde{\sqsubseteq}}\Xi\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{raise}\ s_{T_1}(e)\Rightarrow \texttt{has}\ \{s_{T_1}\}\ \texttt{raise}\ s_{T_1}(e')\colon T_2}$$

$$\frac{\begin{array}{c}\Xi\cup\{s_T\};\Gamma;\Sigma\vdash e_1\Rightarrow e_1'\colon T_1\\ \Xi;\Gamma,x\colon T;\Sigma\vdash e_2\Rightarrow e_2'\colon T_2 \qquad T_2<:T_1\\ e'=\texttt{try}\ e_1'\ \texttt{handle}\ s_T(x).e_2'\end{array}}{\Xi;\Gamma;\Sigma\vdash \texttt{try}\ e_1\ \texttt{handle}\ s_T(x).e_2\Rightarrow e'\colon T_1}$$

Figure 10. Implementation version of the translation rules for a system with exceptions

$$\begin{array}{ll}\textbf{let} & \mathit{squared} = \lambda f\colon \texttt{Int}\xrightarrow{\Xi}\texttt{Int}\ .\ (\lambda x\colon \texttt{Int}\ .\ (f(x*x))::\emptyset)\\ & \mathit{positive} = \lambda x\colon \texttt{Int}\ .\ \texttt{if}\ x\geq 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{raise}\ s_{\texttt{Int}}(x)\\ & \textbf{in}\ (\mathit{squared}\ \mathit{positive})\end{array}$$

A key property of the *positive* function is that it never raises an exception when applied to a non-negative argument. On the other hand, function *squared* always calls f with $x*x$ as an argument, which is never negative. We therefore know that the function produced by evaluating $(\mathit{squared}\ \mathit{positive})$ never raises an exception, so we would like to type it as $\texttt{Int}\xrightarrow{\emptyset}\texttt{Int}$. A static effect system is too restrictive to do so, but a gradual effect system provides the flexibility to assign the desired type to the function.

The *squared* function's parameter is declared to have type $\texttt{Int}\xrightarrow{\Xi}\texttt{Int}$, for some Ξ. Without gradual effects, the only options for Ξ are either $\Xi=\emptyset$, in which case the type system will reject the application$(\mathit{squared}\ \mathit{positive})$ because the argument requires too many privileges, or $\{s_{\texttt{Int}}\}\subseteq\Xi$, which means the returned function cannot be typed as $\texttt{Int}\xrightarrow{\emptyset}\texttt{Int}$.

In the gradual exception system, we can annotate function *positive* to hide its side effects, delaying privilege checking to runtime, and annotate function *squared* to allow functions that may throw exceptions, as in the following:

$$\begin{array}{l}\textbf{let}\,\mathit{squared} = \lambda f\colon \texttt{Int}\xrightarrow{\{¿\}}\texttt{Int}\ .\ (\lambda x\colon \texttt{Int}\ .\ (f(x*x))::\emptyset)\\ \quad \mathit{positive} = \lambda x\colon \texttt{Int}\ .\ (\texttt{if}\ x\geq 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{raise}\ s_{\texttt{Int}}(x))::\{¿\}\\ \quad \textbf{in}\ (\mathit{squared}\ \mathit{positive})\end{array}$$

The translation algorithm then produces the following program in the intermediate language:

$$\begin{array}{lcl}\textbf{let}\ \mathit{squared} & = & \lambda\, f\colon \texttt{Int}\xrightarrow{\{¿\}}\texttt{Int}.\\ & & \ \lambda\, x\colon \texttt{Int}.\\ & & \ \ \texttt{restrict}\ \emptyset\\ & & \ \ \ (((\langle \texttt{Int}\xrightarrow{\emptyset}\texttt{Int} \Leftarrow \texttt{Int}\xrightarrow{\{¿\}}\texttt{Int}\rangle f)(x*x)))\\ \quad \mathit{positive} & = & \lambda\, x\colon \texttt{Int}.\\ & & \ \texttt{restrict}\ \{¿\}\\ & & \ \ \texttt{if}\ x\geq 0\\ & & \ \ \ \texttt{then}\ x\\ & & \ \ \ \texttt{else}\ \texttt{has}\ \{s_{\texttt{Int}}\}\ \texttt{raise}\ s_{\texttt{Int}}(x)\\ \ \textbf{in}\ (\mathit{squared}\ \mathit{positive}) & & \end{array}$$

In this program, application $(\mathit{squared}\ \mathit{positive})$ can be typed as $\texttt{Int}\xrightarrow{\emptyset}\texttt{Int}$, as desired. Given the properties of integer numbers, the `else` branch in the body of *positive* will never be executed. The higher-order cast for f in the body of *squared* never fails because rule [E-Cast-Fn] only introduces $\texttt{restrict}\ \emptyset\ \texttt{has}\ \emptyset$checks.

Effect errors are not exceptions Gradual Effects for Exceptions is more expressive than simply raising uncaught exceptions. Triggering an Error instead of propagating the exception prevents the system from following implicit exceptional control flows, where an outer handler catches an exception that was locally forbidden. The following example demonstrates how this behavior can affect evaluation of a program:

```
let positive = λx: Int . (if x ≥ 0 then x else raise s_Int(x)) :: {¿}
    nonzero = λx: Int . if x = 0 raise s_Int (x) else x
 in try
      nonzero ((positive −1) :: ∅)
     handle s_Int(x)
      print "0 is an invalid argument"
```

The handler in the **let** body is designed to catch the exceptions thrown by the body of *nonzero*. To this end, the code uses an effect ascription to ensure that the argument to *nonzero* does not throw any exception.

At the same time, the program reuses the *positive* function introduced in the previous example, but applies the function to a negative number. Given this incorrect argument, *positive* attempts to raise an exception. An effect ascription to the $\emptyset$ privilege set forces the application to not raise any exception at all. This inconsistent behavior is caught at runtime by the gradual effect discipline.

We purposely used the same label for exceptions in *positive* and in *nonzero*. If the system simply threw the uncaught exception in *positive*, the handler would take control even though it was not designed for that exception. Instead, since *positive* has no exception raising privileges, the system triggers an Error just before it would have thrown the exception. Evaluation thus terminates without control ever reaching the exception handler, which was designed for failures of *nonzero* only.

6. Related Work

In the realm of effect systems, the most closely related work is the generic framework of Marino and Millstein [15], which we have extensively discussed in this paper, because we build upon it to formulate gradual effect checking in a generic setting.

Rytz *et al.* [17] develop a notion of lightweight effect polymorphism, which lets functions be polymorphic in the effects of their higher-order arguments. The formulation is also generic like the M&M framework, although there are more technical differences; most notably, the system is formulated to infer effects instead of checking privileges. An implementation of the generic polymorphic framework has been developed for Scala, originally only with IO and exceptions as effects. More recently, a purity analysis has been integrated in the compiler plugin [18]. The effect system has been applied to a number of Scala libraries. Interestingly, Rytz *et al.* report cases where they suffer from the conservativeness of the effect analysis, similar to the example of Sec. 2. To address this, Rytz recently introduced an `@unchecked` annotation. Although it is called a cast, it is an "unsafe cast", since no dynamic checking is associated to it; i.e. it is just a mechanism to bypass static checking. We believe our work on gradual effect checking could be of direct practical use in Scala, and intend to pursue that route.

While there is a long history in the area of combining static and dynamic checking, the gradual typing approach of Siek and Taha [23] has been particularly successful and triggered many developments. Its main contribution was to identify the notion of consistency as a key to support the full spectrum of static-to-dynamic typing. Originally developed for functional languages, it has been extended in several directions, including structural objects [22] and generics [14]. Most directly related to this work is the application of the gradual typing principles to other typing disciplines, such as ownership types, typestates, and information flow typing.

Wolff *et al.* [26] develop gradual typestate checking. Typestates reflect the changing states of objects in their types. To support flexible aliasing in the face of state change, the language provides access permissions to support rely-guarantee reasoning about aliases, and state guarantees, which preserve type information for distinct aliases of shared objects.

Sergey and Clarke propose gradual ownership types [21]. Like gradual typestates, gradual ownership expresses and dynamically tracks heap properties. While typestate focuses on objects changing state, ownership controls the flow of object references.

Disney and Flanagan [6] explore the idea of gradual security with a gradual information flow type system. Data can be marked as confidential, and the runtime system ensures that it is not leaked. This dynamically-checked discipline is moved towards the static end of the spectrum by introducing security labels on types.

Extensions to contract systems for higher-order functions [9], such as computational contracts [19, 20] and temporal contracts [7], have the ability to monitor for the occurrence of specific (sequences of) execution events, in particular effectful operations. These approaches rely on full runtime monitoring; it is not clear if they could be reconciled with the pay-as-you-go model of gradual checking.

As far as we know, none of the existing approaches to gradual checking relies on abstract interpretation to develop an account of uncertainty. While it remains to be studied, it seems that the abstract interpretation approach we follow here could be used to investigate existing and as-yet unexplored notions of gradual checking.

7. Conclusion

The primary contribution of this paper is a framework for developing gradually checked effect systems for any number of effect systems that can be couched in the M&M framework. Using our approach, one can systematically transform a static effect discipline into one that supports full static checking, full dynamic checking, and any intermediate blend. We believe that gradual effect checking can facilitate the process of migrating programs toward a statically checked effect discipline, as well as bringing dynamic effect checks to languages that have no such checks whatsoever, and leaving wiggle room for programs that can only partially fit an effect discipline. To empirically evaluate this claim, we intend to implement our framework in the Scala language, and extend it to support the effect features that the language already provides.

Initially, we relied on the principles of gradual checking and our intuitions to guide the design, but found it challenging to develop and validate our concepts. We found abstract interpretation to be an effective framework in which to develop and validate our intuitions. Using it we were able to generically define the idea of consistent functions and predicates, as well as explain and define auxiliary concepts such as strict checking and static containment. We believe that, in addition to gradual effects, other gradual checking notions could be fruitfully investigated in this framework. In particular, we intend to extend our system to support full gradual type-and-effect systems, which depend on gradual effects as an initial step, and define blame tracking for effect casts.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 63–74, San Francisco, CA, USA, Jan. 2008. ACM Press.

[3] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International*

Workshop on Types in Languages Design and Implementation (TLDI 07), pages 15–26, New York, NY, USA, 2007. ACM.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, Jan. 1977. ACM Press.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 79)*, pages 269–282, New York, NY, USA, 1979. ACM.

[6] T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[7] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*, pages 176–188, Tokyo, Japan, Sept. 2011. ACM Press.

[8] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Pittsburgh, PA, USA, October 2002. ACM Press.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM Press.

[10] R. Garcia, É. Tanter, R. Wolff, and J. Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems*, 2014. To appear.

[11] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 28–38, Cambridge, MA, USA, Aug. 1986. ACM Press.

[12] C. S. Gordon, W. Dietl, M. D. Ernst, and D. Grossman. JavaUI: Effects for controlling UI object access. In G. Castagna, editor, *Proceedings of the 27th European Conference on Object-oriented Programming (ECOOP 2013)*, volume 7920 of *Lecture Notes in Computer Science*, pages 179–204, Montpellier, France, July 2013. Springer-Verlag.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2003.

[14] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*, pages 609–624, Portland, Oregon, USA, Oct. 2011. ACM Press.

[15] D. Marino and T. Millstein. A generic type-and-effect system. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 39–50, 2009.

[16] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[17] L. Rytz, M. Odersky, and P. Haller. Lightweight polymorphic effects. In J. Noble, editor, *Proceedings of the 26th European Conference on Object-oriented Programming (ECOOP 2012)*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282, Beijing, China, June 2012. Springer-Verlag.

[18] L. Rytz, N. Amin, and M. Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*, 2013. Article No.: 4.

[19] C. Scholliers, É. Tanter, and W. De Meuter. Computational contracts. In *Scheme and Functional Programming Workshop*, 2011.

[20] C. Scholliers, É. Tanter, and W. De Meuter. Computational contracts. *Science of Computer Programming (To Appear)*, Oct. 2013. URL `http://dx.doi.org/10.1016/j.scico.2013.09.005`.

[21] I. Sergey and D. Clarke. Gradual ownership types. In H. Seidl, editor, *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 579–599, Tallinn, Estonia, 2012. Springer-Verlag.

[22] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in Lecture Notes in Computer Science, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

[23] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[24] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*, pages 793–810, Tucson, AZ, USA, Oct. 2012. ACM Press.

[25] Y. M. Tang and P. Jouvelot. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 95)*, pages 45–53, New York, NY, USA, 1995. ACM. ISBN 0-89791-720-0.

[26] R. Wolff, R. Garcia, É. Tanter, and J. Aldrich. Gradual typestate. In M. Mezini, editor, *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483, Lancaster, UK, July 2011. Springer-Verlag.

[27] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Journal of Information and Computation*, 115(1):38–94, Nov. 1994.