Under consideration for publication in J. Functional Programming

Gradual Type-and-Effect Systems

FELIPE BAÑADOS SCHWERTER and RONALD GARCIA
University of British Columbia
(*e-mail:* {fbanados, rxg}@cs.ubc.ca)

ÉRIC TANTER
University of Chile
(*e-mail:* 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 the framework of Marino and Millstein can be automatically extended to support gradual checking.

We use gradual effect checking to develop a fully gradual type-and-effect framework, which permits interaction between static and dynamic checking for both effects and types.

Contents

1 Introduction

Type-and-effect systems enable static reasoning about the computational effects of programs. Effect systems were originally introduced to safely support mutable variables in functional languages (Gifford & Lucassen, 1986), but more recently, effect systems have been developed for a variety of effect domains, e.g., I/O, exceptions, locking, atomicity, confinement, and purity (Gosling *et al.*, 2003; Abadi *et al.*, 2006; Benton & Buchlovsky, 2007; Abadi *et al.*, 2008; Rytz *et al.*, 2012, 2013; Gordon *et al.*, 2013).

To abstract from specific effect domains and account for effect systems in general, Marino & Millstein (2009) developed a generic effect system, which we denote M&M throughout this paper. 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 over the privileges available and how they are adjusted and checked during type checking. Marino & Millstein (2009) 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 (Rytz *et al.*, 2012). Several specific effect systems for this plugin include IO effects, exceptions, and more recently, state effects (Rytz *et al.*, 2013).

Despite their obvious advantages for 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

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

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 (Siek & Taha, 2006, 2007), gradual ownership types (Sergey & Clarke, 2012), gradual typestate (Wolff *et al.*, 2011; Garcia *et al.*, 2014)), 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 (Section 3) , by formulating it in the framework of abstract interpretation (Cousot & Cousot, 1977). Using abstract interpretation, we 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 (Section 4).
- We demonstrate how an effect system formulated in the M&M framework can be immediately extended to support gradual checking by lifting existing *adjust* and *check* functions to the gradual setting (Section 3).
- We present a concrete instantiation of the generic framework to gradually check exceptions (Section 5). The resulting system is compact and provides a tangible and self-contained example of gradual effect checking.
- Our initial gradual effect checking semantics requires that all values carry tag information at runtime. We develop a conservative operational semantics which does not require values to be tagged, trading off the annotation overhead for precision with respect to the effect discipline (Section 6).
- We combine gradual effect checking and gradual typing to provide a gradual type-and-effect system that provides a programmer-directed combination of static and dynamic checking of both effects and types (Section 7).

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

This article differs from our previous ICFP article (Bañados Schwerter *et al.*, 2014) in several ways. Most importantly, the last two contributions listed above are original; they are described in detail in Section 6 and Section 7, respectively. We also fix an error in our type safety theorem for gradual effect checking.

2 Background and Motivation

In this section, we introduce static effect checking; we introduce gradual typing; and we give an intuition for how gradual effect checking is related to both. We finish with a brief introduction to the M&M generic framework for 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 are also values, 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 induce a natural notion of subtyping (Tang & Jouvelot, 1995). 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 & Millstein (2009), 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 Gradual Typing

Gradual Typing, introduced by Siek & Taha (2006), combines the flexibility of dynamic checking with the guarantees of static checking, allowing the programmer to annotate parts of the program with their types and to defer at will to runtime checking when static information is insufficient. The appeal of gradual typing has inspired the development of gradual approaches to a variety of type disciplines, including objects (Siek & Taha, 2007; Ina & Igarashi, 2011; Takikawa *et al.*, 2012), ownership types (Sergey & Clarke, 2012), typestate (Wolff *et al.*, 2011; Garcia *et al.*, 2014), and information flow typing (Disney & Flanagan, 2011).

In Siek & Taha (2006), unannotated programs are given a default unknown type (denoted ?). For example, function $(\lambda x\,.\,x)$ is automatically transformed into $(\lambda x :\, ?\,.\,x)$. Important goals of gradual typing are to provide static guarantees for fully-annotated parts of the code, to avoid runtime checks and to provide developers with the benefits of both static and dynamic checking in a single language.

The type consistency relation A standard type system depends on a set of implicit type equality restrictions. For example, a function application is valid if the type of the argument is equal to the type of the function parameter. A gradual type system loosens this restriction by replacing type equality with a type consistency relation $(\sim)$.

Intuitively, the type consistency relation accepts programs optimistically. A gradual type system only rejects those programs with contradictory static information: Through type consistency, gradual typing accepts programs missing type information, because there's a chance that they may be right (so runtime checks must be performed). For example, in function $(\lambda f :\, ?\,.\,f\ 4)$, the type of f is statically unknown, and is accepted because f *may* be a function accepting integer arguments. Type consistency is used in a typing rule such as the following:

$$\text{T-App}\frac{\Gamma;\Sigma \vdash e_1 :\, T_1 \qquad \Gamma;\Sigma \vdash e_2 :\, T_2 \qquad T_1 \sim T_2 \to T_3}{\Gamma;\Sigma \vdash e_1\ e_2 :\, T_3}$$

With a proper definition of type consistency, this rule can still be used to reject clearly incorrect programs. An application $(4\ 2)$ is rejected, because $\texttt{Int} \not\sim T_1 \to T_3$. For fully annotated programs, a gradual type system provides the same guarantees as a static type system.

Definition 1 (*Type Consistency*)
The type consistency relation is defined as follows:

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

Casts as runtime checks Whenever static information is insufficient to determine whether a program is safe or unsafe, the program must be checked at runtime. A gradual system introduces type casts to check type invariants. Type casts make explicit the optimistic static assumptions made by a gradual type system, and check them during execution.

For example, the program $((\lambda f :\, ?\,.\,f\ 4)\ 2)$ is statically accepted by a gradual type system, even though the argument 2 is not a function. This is because the function itself

may be validly applied in a different context, for example to an argument like $\lambda x : \texttt{Int} \,.\, x$. To evaluate this example program, gradual typing systems first translate the program to an internal language with type casts. The type system for the internal language is more restrictive, as it is allowed to appeal to type consistency only for casts. With this restriction, casts become the only language construct where a type inconsistency may occur. In our example, application $f\ 4$ is translated to $(\langle \texttt{Int} \to ? \Leftarrow ? \rangle f)\ 4$. At runtime, this cast triggers an execution error.

2.3 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.

For instance, the JavaUI effect system (Gordon *et al.*, 2013), 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 (Rytz *et al.*, 2012) 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.

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.

2.4 Gradual Effects in Action

Recall the function g defined in Section 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 & Taha (2006) 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 in which all effectful operations are wrapped by a :: ¿ ascription.[3] Static checking trivially succeeds because all expressions hide their required privileges. Forbidden effects are only 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.

2.5 Generic Effect Systems

To avoid re-inventing gradual effects for each possible effect discipline, we build on the generic effect framework of Marino & Millstein (2009), 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:

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

[2] In a static effect system, an effect ascription $e :: \{\texttt{read}\}$ is directly analogous to a type ascription (Pierce, 2002). Static effect ascriptions were introduced by Gifford & Lucassen (1986).

[3] This corresponds to the translation of terms from the untyped λ-calculus to the gradually-typed λ-calculus, which assigns type ? to every expression (Siek & Taha, 2006)

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 determines whether the current privileges are sufficient to evaluate non-value expressions. To achieve genericity, the $\mathbf{check}_C$ predicate is indexed by *check contexts* C, which represent the non-value expressions. The *adjust* function modifies the available privileges for 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 $\mathbf{adjust}_A$ function 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 $\mathbf{check}_C$ and $\mathbf{adjust}_A$ 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:

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

In this case, only state-manipulating expressions have interesting *check* predicates, which simply require the corresponding privilege. A tag set π that sits in an expression position of a check context represents the class of values that could appear in that position at runtime. A more complex *check* predicate could make use of the information in the tag sets, for example, to limit access to memory for an individual value (through its tag annotation). For example, we could annotate some individual references with an `imm` tag (for immutability). A predicate like

$$\mathbf{check}_{\pi_1 := \pi_2}(\Phi) \iff \texttt{write} \in \Phi \text{ and } \texttt{imm} \notin \pi_1$$

would reject programs that attempt to update these individual references, but still accept assignments to references where `imm` is guaranteed to not happen.

Since the assignment expression involves evaluating two subexpressions (the reference and the new value), there are two adjust contexts. The $\downarrow{:=}\uparrow$ context corresponds to evaluating the reference to be assigned, and the $\pi :=\downarrow$ context corresponds to evaluating the assigned value. The $\downarrow$ symbol denotes the subexpression for which privileges should be adjusted. The $\uparrow$ symbol denotes a subexpression that would be evaluated after the current expression.

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

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

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

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

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

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, $\mathbf{adjust}_{\pi_1:=\downarrow}$ knows which tags are associated with typing e_1. Finally, $\mathbf{check}_{\pi_1:=\pi_2}$ verifies that assignment is allowed with the given permissions and the subexpression tag sets. Subtyping is used here only to account for inclusion of privilege sets between function types.

For maximum flexibility, the framework imposes only two constraints on the definitions of **check** and **adjust**:

Property 1 (*Privilege Monotonicity*)

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

Property 2 (*Tag Monotonicity*)

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

Privilege monotonicity captures the idea that once an expression has sufficient privileges to run, one can always safely add more. This corresponds to effect subsumption in many particular effect systems. In contrast, tag monotonicity captures the idea that more tags implies more uncertainty about the source of a runtime value. The $\sqsubseteq$ relation holds when contexts have the same structure and the tagsets of the first context are subsets of the corresponding tagsets of the second context. For example, $\texttt{ref}\pi_1 \sqsubseteq \texttt{ref}\pi_2$ 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. For example, the order-reversing constraint on tags can be used in **adjust** to limit the privileges available for assignment to a particular set of locations, or to limit the side effects for the arguments of a particular set of operators. The restrictions for tags in **check** can be used to limit the set of values that can perform an effectful operation to those carrying at most a particular set of tags, and thus to introduce fine-grained "per-value" effect restrictions.

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 Section 2.3. We use abstract interpretation (Cousot & Cousot, 1977) 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 & Taha (2006), 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 Section 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 (Siek & Taha, 2007), 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 support the notions required to 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 Section 2.1:

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

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 2 (*Concretization*)
Let $\gamma : \textbf{CPrivSet} \to \mathscr{P}(\textbf{PrivSet})$ be defined as follows:

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

Reconsidering our two example consistent privilege sets, we find that

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

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

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

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

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 (Section 2.5), 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 4 (*Consistent Checking*)
Let $\mathbf{check}_C$ be a predicate on privilege sets. Then we define a corresponding *consistent check* predicate $\widetilde{\mathbf{check}}_C$ on consistent privilege sets as follows:

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

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

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

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 in terms of consistent privilege sets once we settle on a particular $\mathbf{check}_C$ predicate (cf. Section 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 $\mathbf{adjust}_A : \mathbf{PrivSet} \to \mathbf{PrivSet}$, which alter the set of available effect privileges (Section 2.5). Using abstract interpretation, we lift these to *consistent adjust* functions $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \to \mathbf{CPrivSet}$. To do so we must first complete the abstract interpretation framework.

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

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

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

Proposition 1 (*Soundness*)
$\Upsilon \subseteq \gamma(\alpha(\Upsilon))$ for all $\Upsilon \in \mathscr{P}(\mathbf{PrivSet})$.

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 6 (*Abstraction*)
Let $\alpha : \mathscr{P}(\mathbf{PrivSet}) \to \mathbf{CPrivSet}$ be defined as follows:[6]

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

In words, abstraction preserves the common concrete privileges, and adds unknown privileges to the resulting consistent set when needed. For example:

$$\begin{array}{rcl}\alpha(\{\{\texttt{read},\texttt{write}\}\}) & = & \{\texttt{read},\texttt{write}\}\\ \alpha(\{\{\texttt{read},\texttt{write}\},\{\texttt{read},\texttt{alloc}\}\}) & = & \{\texttt{read},¿\}\\ \alpha(\{\{\texttt{read}\},\{\texttt{alloc}\}\}) & = & \{¿\}\end{array}$$

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 7 (*Consistent Adjust*)

[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* (Cousot & Cousot, 1979).

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

w	::=	$\texttt{unit} \mid \lambda x :\ T\ .\ e \mid l$	Prevalues
v	::=	w_ε	Values
e	::=	$x \mid v \mid e\ e \mid e :: \Xi \mid (\texttt{ref}\ e)_\varepsilon \mid\ !e \mid (e := e)_\varepsilon$	Terms
T	::=	$\pi\ \rho$	Types
ρ	::=	$\texttt{Unit} \mid T \xrightarrow{\Xi} T \mid \texttt{Ref}\ T$	PreTypes
A	::=	$\downarrow\uparrow\ \mid\ \pi\downarrow\ \mid\ \texttt{ref}\ \downarrow\ \mid\ !\downarrow\ \mid\ \downarrow{:=}\uparrow\ \mid\ \pi :=\downarrow$	Adjust Contexts
C	::=	$\pi\ \pi \mid \texttt{ref}\ \pi\ \mid\ !\pi\ \mid\ \pi := \pi$	Check Contexts

Fig. 1. Syntax of the source language for gradual effect checking

Let $\widetilde{\mathbf{adjust}}_A : \mathbf{CPrivSet} \to \mathbf{CPrivSet}$ be defined as follows:

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

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

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

4 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 (Figure 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 annotated deeply. Each value-creating expression is annotated with a tag

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

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

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

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

$$\text{T-Eff}\frac{\Xi_1;\Gamma;\Sigma \vdash e : T \qquad \Xi_1 \sqsubseteq \Xi}{\Xi;\Gamma;\Sigma \vdash (e :: \Xi_1) : T} \qquad \text{T-Ref}\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\texttt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\rho \\ \widetilde{\textbf{check}}_{\texttt{ref}\ \pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho}$$

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

Fig. 2. Type system for the source language for gradual effect checking

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 (Section 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 redexes: function application, reference allocation, dereferencing, and assignment. They index $\widetilde{\textbf{check}}_C$ to determine which privileges are needed to perform the operation. Using tag sets, the framework can express check predicates that forbid particular values and adjust functions that allow more effect privileges for certain sets of values. Section 2.5 presents an example of using tags to refine a check predicate. Marino & Millstein (2009) use tag annotations to model more sophisticated scenarios like transactional memory, type qualifiers, and threads.

Figure 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 Section 2.5) 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. The $\widetilde{\textbf{adjust}}$ and $\widetilde{\textbf{check}}$ operations are defined in Section 3, and we use the same techniques introduced there to lift effect subtyping to a notion of *consistent subtyping*. First we lift traditional privilege set membership to *consistent membership*:

Definition 8 (*Consistent Membership*)
ϕ is a *consistent member* of Ξ, notation $\phi \mathrel{\widetilde{\in}} \Xi$, if and only if $\phi \in \Phi$ for *some* $\Phi \in \gamma(\Xi)$.

We use consistent membership to lift set containment $\Phi_1 \subseteq \Phi_2$ (defined as $\forall \phi \in \textbf{Priv} \,.\, \phi \in \Phi_1 \Rightarrow \phi \in \Phi_2$), to *consistent containment*:

Definition 9 (*Consistent Containment*)
Ξ_1 is *consistently contained* in Ξ_2, notation $\Xi_1 \mathrel{\widetilde{\sqsubseteq}} \Xi_2$, if and only if $\forall \phi \in \textbf{Priv}, \phi \mathrel{\widetilde{\in}} \Xi_1 \Rightarrow \phi \mathrel{\widetilde{\in}} \Xi_2$.[8]

Consistent containment means that in at least one plausible situation, privilege set containment could hold at runtime. 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.

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

Both relations express plausible substitutability. Consistent containment is not transitive, like the consistency relation for types in Siek & Taha (2006). As a result consistent subtyping is also not transitive. Our definition of consistent subtyping is directly analogous to consistent subtyping for gradual object systems (Siek & Taha, 2007).

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

[8] This definition is equivalent to the definition in Bañados Schwerter *et al.* (2014), which states that $\Phi_1 \subseteq \Phi_2$ for *some* $\Phi_1 \in \gamma(\Xi_1)$ and $\Phi_2 \in \gamma(\Xi_2)$. It is updated here to better reflect the relationship between consistent containment and strict containment.

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

Fig. 3. Syntax of the internal language for gradual effect checking

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 explicit. This section presents the internal language. The translation is presented in Section 4.3.

Figure 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 (Figure 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*** (Section 3.3, Definition 5). Consistent subtyping $\lesssim$ is replaced with a notion of subtyping $<:$ that is based on ordinary set containment for consistent privilege sets and tags:

$$
\boxed{T <: T} \qquad \frac{\pi_1 \subseteq \pi_2}{\pi_1 \rho <: \pi_2 \rho} \qquad \frac{T_3 <: T_1 \quad T_2 <: T_4 \quad \pi_1 \subseteq \pi_2 \quad \Xi_1 \subseteq \Xi_2}{\pi_1 T_1 \xrightarrow{\Xi_1} T_2 <: \pi_2 T_3 \xrightarrow{\Xi_2} T_4}
$$

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 strict notion of set membership that focuses on statically known permissions. A consistent privilege

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

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

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

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

$$\text{IT-Ref}\ \frac{\widetilde{\textbf{adjust}}_{\texttt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\rho \quad \textit{strict-check}_{\texttt{ref}\ \pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho} \qquad \text{IT-Deref}\ \frac{\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\texttt{Ref}\ T \quad \textit{strict-check}_{!\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !e : T}$$

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

Fig. 4. Typing rules for the internal language for gradual effect checking

set represents some number of concrete privilege sets, each containing some different privileges, but most consistent privilege sets have some reliable information. For instance, any set represented by $\Xi = \{\texttt{read}, ¿\}$ may have a variety of privileges, but any such set will surely contain the `read` privilege. We formalize this idea in terms of *strict membership*:

Definition 10 (*Strict membership*)
A privilege ϕ is a *strict member* of Ξ, denoted $\phi \mathrel{\widehat{\in}} \Xi$, if and only if $\phi \in \Phi$ for *all* $\Phi \in \gamma(\Xi)$.

Using strict membership, we define the *static part* of a consistent privilege set as the collection of its strict members.

Definition 11 (*Static Part*)
The *static part* of a consistent privilege set, $|\cdot| : \textbf{CPrivSet} \to \textbf{PrivSet}$ is defined as

$$|\Xi| = \left\{ \phi \in \textbf{Priv} \mid \phi \mathrel{\widehat{\in}} \Xi \right\}$$

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

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

Definition 12 (*Static Containment*)

$$\boxed{\Phi \vdash e \mid \mu \to e \mid \mu}$$

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

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

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

$$\text{E-Has-T}\ \frac{\Phi' \subseteq \Phi \qquad \Phi \vdash e \mid \mu \to e' \mid \mu'}{\Phi \vdash \mathtt{has}\ \Phi'\ e \mid \mu \to \mathtt{has}\ \Phi'\ e' \mid \mu'} \qquad \text{E-Has-V}\ \frac{}{\Phi \vdash \mathtt{has}\ \Phi'\ v \mid \mu \to v \mid \mu}$$

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

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

$$\text{E-App}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (\lambda x :\ T_1\ .\ e)_{\varepsilon_1}\ v_{\varepsilon_2} \mid \mu \to [v_{\varepsilon_2}/x]e \mid \mu} \qquad \text{E-Cast-Frame}\ \frac{\Phi \vdash e \mid \mu \to e' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1 \rangle e \mid \mu \to \langle T_2 \Leftarrow T_1 \rangle e' \mid \mu'}$$

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

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

Fig. 5. Small-step semantics of the internal language for gradual effect checking

Ξ_1 is *statically contained* in Ξ_2, notation $\Xi_1 \leq \Xi_2$, if and only if $\forall \phi \in \mathbf{Priv}\ .\ \phi \mathrel{\widehat{\in}} \Xi_1 \Rightarrow \phi \mathrel{\widehat{\in}} \Xi_2$.[9]

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' (Tang & Jouvelot, 1995). To establish this, we must consider properties of both ***strict-check*** and $\widetilde{\mathbf{adjust}}$. Conveniently, ***strict-check*** is monotonic with respect to consistent privilege set containment.

Lemma 3

[9] An equivalent definition, in terms of static part, is $|\Xi_1| \subseteq |\Xi_2|$.

If ***strict-check***$_C(\Xi_1)$ and $\Xi_1 \subseteq \Xi_2$ then ***strict-check***$_C(\Xi_2)$.

On the contrary, though, $\widetilde{\textbf{adjust}}$ is not monotonic with respect to set containment on consistent privilege sets. However, 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.

Proposition 5 (*Strong Effect Subsumption*)

If $\Xi_1;\Gamma;\Sigma \vdash e : T$ and $\Xi_1 \leq \Xi_2$, then $\Xi_2;\Gamma;\Sigma \vdash e : T$.

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

Corollary 6 (*Effect Subsumption*)

If $\Xi_1;\Gamma;\Sigma \vdash e : T$ and $\Xi_1 \subseteq \Xi_2$, then $\Xi_2;\Gamma;\Sigma \vdash e : T$.

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 dynamically 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 show when translating 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. $\Xi_1 \sqsubseteq_\sim \Xi_2$ if and only if $\Xi_1 \subseteq \Xi_2$ or ¿ $\in \Xi_2$.

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

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

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

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

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

To illustrate, consider the following example: $\{\mathtt{read}, \mathtt{alloc}\} \sqsubseteq \{\mathtt{read}, ¿\}$ because `alloc` *could* be in a representative of $\{\mathtt{read}, ¿\}$, but $\{\mathtt{read}, \mathtt{alloc}\} \not\sqsubseteq \{\mathtt{read}, ¿\}$ since that is not definitely true. Thus, to be sure at runtime, we must *check* for $|\{\mathtt{read}, \mathtt{alloc}\}| \setminus |\{\mathtt{read}, ¿\}| = \{\mathtt{alloc}\}$. Note that the rule [E-Cast-Fn] uses the standard

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

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

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

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

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

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

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

$$\text{C-Deref}\ \frac{\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e \Rightarrow e' : \pi\mathtt{Ref}\ T \qquad \widetilde{\mathbf{check}}_{!\pi}(\Xi) \qquad \Phi = \Delta_{!\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !e \Rightarrow \mathit{insert\text{-}has?}(\Phi, !e') : T}$$

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

Fig. 6. Translation of source programs to the internal language for gradual effect checking

approach to higher-order casts due to Findler & Felleisen (2002). 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 & Felleisen (1994). 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 capture the static information

implied by the consistent privilege set Ξ used to type the program. Under these conditions, the program does not get stuck. [11]

Definition 13 (*Satisfaction*)
We say that a privilege set Φ *satisfies* a consistent privilege set Ξ (notation $\Xi \vdash \Phi$) if a subset of Φ is represented by Ξ. Formally:

$$\Xi \vdash \Phi \iff \Phi' \subseteq \Phi \text{ for some } \Phi' \in \gamma(\Xi)$$

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 any privilege set Φ such that $\Xi \vdash \Phi$, and for any store μ such that $\emptyset \mid \Sigma \vdash \mu$.

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

Theorem 10 (*Preservation*)
If $\Xi;\Gamma;\Sigma \vdash e : T$, and
$\Phi \vdash e \mid \mu \to e' \mid \mu'$ for $\Xi \vdash \Phi$ 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. □

Corollary 11 (*Progress (with representation, as in (Bañados Schwerter* et al.*, 2014))*)
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 any privilege set Φ such that $\Phi \in \gamma(\Xi)$, and for any store μ such that $\emptyset \mid \Sigma \vdash \mu$.

Proof
Special case of Theorem 9, since $\Phi \in \gamma(\Xi) \Rightarrow \Xi \vdash \Phi$. □

Corollary 12 (*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
Special case of Theorem 10, like the previous corollary. □

4.3 Translating Source Programs to the Internal Language

Figure 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,

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

or subtyping that is not also a strict check, static containment, or static subtyping, respectively, must be guarded by a `has` expression (for checks and containments) or a cast (for subtypings).

Recall from Section 4.2 that the `has` expression checks if some particular privileges are available at runtime. The translation system determines for each program point which privileges (if any) must be checked. Since the generic framework imposes only privilege and tag monotonicity restrictions on the **check** and **adjust** functions, deducing these checks can be subtle.

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

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

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

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

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

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

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 14 (*Minimal Privilege Check*)
Let C be some checking context. Then define Δ_C : **CPrivSet** $\rightarrow$ **PrivSet** as follows:

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

The Δ_C function transforms a given consistent privilege set into the minimal conservative set of additional privileges needed to safely pass the $\mathbf{check}_C$ function[13]. For instance, the [C-App] translation rule uses it to guard a function application, if need be, with a runtime

[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] In principle, an infinite domain of effects could induce an uncomputable Δ_C function. In practice this has not been an issue. For instance, Toro & Tanter (2015) extended this approach to polymorphic effects while retaining computability.

privilege check. These checks are introduced by the *insert-has?* metafunction.

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

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.[14]

An important property of our translation system is that it preserves typing.

Theorem 13 (*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 $\textbf{\textit{strict-check}}_C(\Xi)$ 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. □

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 `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, `try` e_1 `handle` $s_T(x).e_2$, attempts to evaluate the expression e_1. If successful, its result is returned; if e_1 raises an 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

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

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

Fig. 7. Syntax for a Gradual Effect System with Exceptions

$$\boxed{\Phi \vdash e \mid \mu \to e \mid \mu}$$

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

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

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

$$\text{E-Try-F}\ \frac{\mathbf{check}_{\texttt{raise}\ s_{T_1}(\{\bullet\})}(\Phi)}{\Phi \vdash \texttt{try}\ \texttt{raise}\ s_{T_1}(v)\ \texttt{handle}\ s_{T_2}(x).e \mid \mu \to \texttt{raise}\ s_{T_1}(v) \mid \mu}$$

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

algorithm. Following M&M, we define a new check context for each new redex and a new adjust context for each new evaluation frame.

Figure 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 Figure 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 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

$$\boxed{\Xi;\Gamma;\Sigma \vdash e : T} \qquad \frac{\widetilde{\mathbf{adjust}}_{\mathtt{raise}\ s_T(\downarrow)}(\Xi);\Gamma;\Sigma \vdash e : T \quad \widetilde{\mathbf{check}}_{\mathtt{raise}\ s_T(\{\bullet\})}(\Xi)}{\Xi;\Gamma;\Sigma \vdash \mathtt{raise}\ s_T(e) : T'}$$

$$\frac{\widetilde{\mathbf{adjust}}_{\mathtt{try}\downarrow\mathtt{handle}\ s_T\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : T_1 \quad \widetilde{\mathbf{check}}_{\mathtt{try}\ \{\bullet\}\ \mathtt{handle}\ s_T\uparrow}(\Xi) \quad \Xi;\Gamma,x : T;\Sigma \vdash e_2 : T_2 \quad T_2 \lesssim T_1}{\Xi;\Gamma;\Sigma \vdash \mathtt{try}\ e_1\ \mathtt{handle}\ s_T(x).e_2 : T_1}$$

Fig. 9. Source language typing rules for exceptions

framework: $\widetilde{\mathbf{check}}$ 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 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:

$$\mathbf{check}_{\mathtt{raise}\ s_T(\pi)}(\Phi) \Longleftrightarrow s_T \in \Phi$$
$$\mathbf{check}_C(\Phi) \text{ holds for all other } C$$

$$\begin{aligned} \mathbf{adjust}_{\mathtt{try}\downarrow\mathtt{handle}\ s_T\uparrow}(\Phi) &= \Phi \cup \{s_T\} \\ \mathbf{adjust}_A(\Phi) &= \Phi \text{ otherwise} \end{aligned}$$

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.

Type safety also holds for this extended language. Since we extended the framework core language with new features, and also extended the structure of the typing judgment and the evaluation semantics, we technically require a new proof of type safety. However, we observe that we can preserve the statements of theorems 9, 10 and 13. Furthermore, the proof of type safety conservatively extends the proof for our base system, as each case of the original proof holds for the extended language in essentially the same way. Where necessary, appeals to the induction hypotheses account for any differences introduced by the new language features.

As intended by Marino & Millstein (2009), the framework can often be extended in this generic way to account for common language extensions. Suitable language features introduce new cases to the proofs, but otherwise do not interfere with the structure of the proof. If some candidate language extension changes the structure of the theorems or the individual proof cases, then more explicit work is needed to establish soundness. Delaware *et al.* (2013) study approaches to formally modularizing such proofs of type safety in the context of mechanized metatheory.

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

$$\boxed{\Xi;\Gamma;\Sigma \vdash e \Rightarrow e : T} \qquad \frac{\Xi;\Gamma;\Sigma \vdash e \Rightarrow e' : T_1 \quad \{s_{T_1}\} \subseteq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{raise}\ s_{T_1}(e) \Rightarrow \texttt{raise}\ s_{T_1}(e') : T_2}$$

$$\frac{\Xi;\Gamma;\Sigma \vdash e \Rightarrow e' : T_1 \quad \{s_{T_1}\} \not\subseteq \Xi \quad \{s_{T_1}\} \sqsubseteq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{raise}\ s_{T_1}(e) \Rightarrow \texttt{has}\ \{s_{T_1}\}\ \texttt{raise}\ s_{T_1}(e') : T_2}$$

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

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

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:

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

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 (*squared 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 rejects the application(*squared 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:

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

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

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

In this program, application (*squared 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* is never executed. The higher-order cast for f in the body of *squared* never fails because rule [E-Cast-Fn] only introduces `restrict` $\emptyset$ `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:

$$\begin{array}{l}\textbf{let}\ \mathit{positive} = \lambda x : \texttt{Int}\ .\ (\texttt{if}\ x \geq 0\ \texttt{then}\ x\ \texttt{else}\ \texttt{raise}\ s_{\texttt{Int}}(x)) :: \{¿\} \\ \quad \mathit{nonzero} = \lambda x : \texttt{Int}\ .\ \texttt{if}\ x = 0\ \texttt{raise}\ s_{\texttt{Int}}\ (x)\ \texttt{else}\ x \\ \textbf{in}\ \texttt{try} \\ \quad\ \mathit{nonzero}\ ((\mathit{positive}\ (-1)) :: \emptyset) \\ \quad \texttt{handle}\ s_{\texttt{Int}}(x) \\ \quad\ \texttt{print}\ \text{“0 is an invalid argument”}\end{array}$$

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 A Conservative Operational Semantics

The operational semantics of Marino & Millstein (2009) embeds check predicates that confirm the safety of effectful operations. A powerful property of that system is that a type safety proof, which states that no terms get stuck, implies type soundness: all effectful operations at runtime are performed in an effect context with sufficient permissions. Since every runtime check of the effect discipline will succeed, a real implementation can skip them.

By making our system gradual, we have compromised this property: while the built-in runtime checks may still be skipped, we must keep the privilege contexts for gradual effect checking to work. Privilege contexts depend on **adjust** operations, which in turn depend on the tags that are associated to runtime values. Thus for gradual effect checking to work at runtime, we require the bookkeeping information in the operational semantics that is used to update the context. In this section, we provide an alternative operational semantics that requires a smaller amount of mandatory runtime information: we make tag annotations on values redundant, at the cost of precision: Our alternative semantic conservatively produces runtime errors for some programs that reduce to a value in our previously introduced semantics.

6.1 Safety and soundness

Marino and Millstein's operational semantics of the language includes **check** predicates as a premise for every step performing an effectful operation, and **adjust** functions as a context for every step reducing subexpressions. By introducing these predicates, their type safety theorem ensures that the operational semantics is sound with respect to the effect discipline being enforced by the type system. This property also has the useful corollary that **check** predicates at runtime are made redundant by type safety, thus an implementation can safely remove not only the predicates themselves, but all the machinery that was in place simply to support those predicates: tag annotations on values and the context of privileges.

To make our system gradual, we have introduced runtime checks that ensure soundness in the form of `has` language constructs, which verify that a set of privileges assumed when type checking the program is actually available at runtime. The `has` language construct introduced in Section 4.2 queries the set of available privileges, so an implementation cannot erase this information nor tag annotations on values, which are needed by **adjust** to generate the set of available privileges.. At the same time, since the system encapsulates all runtime checking in the `has` construct, all **check** predicates can still be safely removed in an implementation, just like in Marino & Millstein (2009).

To calculate the available privileges, the semantics introduced in Figure 5 follows the M&M approach and depends on **adjust** functions, which in turn may depend on runtime tag information. For example, to reduce a function application $v_\varepsilon\ e$, the privileges available

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

Fig. 11. Conservative Language Syntax

to reduce e may depend on the tag ε of the function. [15] As presented so far, an implementation can remove neither the privilege context Φ nor the tags in values.

In this section we introduce an alternative semantics that uses statically approximated adjust contexts for **adjust** functions, reducing privilege precision to make tag annotations for values redundant at runtime.

We call this semantics "conservative": some programs that reduce to a value in the original semantics reduce to an Error instead. If a program reduces to a value in both operational semantics, the results are equivalent (modulo tag annotations). We provide this second semantics as a different option in the design spectrum, where the dynamic annotation overhead can be traded off against precision in the dynamic enforcement of the effect discipline

6.2 Making Tags Redundant at Runtime

In the generic operational semantics introduced in Figure 5, tag information is used only to compute **adjust** functions and **check** predicates. We first clarify why **check** predicates are redundant, and we later focus on how **adjust** functions are not redundant, as well as studying the runtime dependence of **adjust** on tag information.

The **check** predicates in the operational semantics are made redundant by the type system. In the operational semantics for the internal language presented in Figure 5, every **check** predicate uses a check context limited to the tags of the values in the expression to be reduced. For example, to reduce a $(\texttt{ref}\ \texttt{unit}_{\varepsilon_1})_{\varepsilon_2}$ expression under a privilege context Φ, rule [CE-Ref] verifies that $\mathbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi)$ holds. At the same time, typing an expression

[15] In Section 6.2.2 we show an example **adjust** function that is dependent on runtime tag information, as well as a program whose result thus depends on the tag of the function.

that reduces to $(\texttt{ref unit}_{\varepsilon_1})_{\varepsilon_2}$ requires a matching $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)$ predicate to hold, with $\varepsilon_1 \in \pi$. A $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)$ predicate ensures that for every privilege set $\Phi \in \gamma(\Xi)$, $\textbf{check}_{\texttt{ref}\ \pi}(\Phi)$ holds. By the tag monotonicity property (Property 2), $\textbf{check}_C(\Phi)$ implies $\textbf{check}_{C'}(\Phi)$ for any C' that contains less tags than C ($C' \sqsubseteq C$). Thus by statically requiring a $\textbf{\textit{strict-check}}_{\texttt{ref}\ \pi}(\Xi)$ predicate in the type system, type safety ensures that the **check** predicate in rule [CE-Ref] always holds, and therefore checking $\textbf{check}_{\texttt{ref}\ \{\varepsilon_1\}}(\Phi)$ at runtime is redundant. This monotonicity argument applies to every **check** predicate in the operational semantics, making them also redundant. Therefore, an implementation does not need to evaluate **check** predicates since they always succeed for well-typed programs, a property that also holds in Marino & Millstein (2009).

Things are not so clear for **adjust**. Since tags affect precisely which privileges are available at runtime, we must focus on where **adjust** require tag information at runtime. The only interesting case is rule [CE-Frame], which alters the set of available privileges by using **adjust** functions. The **adjust** functions depend on tag information through adjust contexts. This dependency arises only for two forms of adjust contexts in our semantics: $\pi\downarrow$ and $\pi :=\downarrow$ (both in rule [CE-Frame]), corresponding to expressions typed with rules [IT-App] and [IT-Asgn], but applies in general to any language construct in which evaluation of a subexpression could affect the privileges available for the computation of other subexpressions. The type system introduced in Section 4.2 uses an approximated tagset π obtained when typing e_1 in expressions of the form $e_1\ e_2$ and $e_1 := e_2$ to generate the respective adjust contexts $\pi\downarrow$ and $\pi :=\downarrow$, used by the $\widetilde{\textbf{adjust}}$ function to generate the context used to type e_2 in rules [IT-App] and [IT-Asgn]. The operational semantics introduced in Figure 5, on the other hand, uses *the exact tag annotations of values*. A corollary of type safety both in our system as in Marino & Millstein (2009) is the fact that the exact tag ε in the operational semantics is guaranteed to be a member of the set π used in the type derivation. With this restriction, the tag monotonicity lemmas of the generic framework ensure that the privilege information available at runtime is *always equal or greater* than the privilege information used by the type system.

In Section 6.2.1 we propose an alternative semantics that removes the need for tags on value, as well as a type-directed translation from our previous operational semantics. Instead of requiring runtime tags, the new language uses only static information, which is added statically during translation to the relevant redexes. Therefore, runtime tag annotations may be safely erased in the conservative semantics, so that the runtime system can use standard representations of values, while still supporting effect disciplines that make nontrivial uses of tag information. Removing runtime tags comes at the price of error precision: some programs with gradual effect annotations trigger an error even though the generic semantics in Figure 5 reduces them to a value. An example of a program whose observable behavior changes across both semantics is shown in Section 6.2.2.

6.2.1 The Conservative Internal Language

To define the conservative semantics, we modify the syntax of redexes. The syntax of the conservative language is introduced in Figure 11, which highlights the interesting changes from the language of Figure 3. At a high level, there is only one difference: redexes that use tag information to **adjust** privileges are annotated with a static approximation of the tags

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

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

Fig. 12. Type-directed tag addition (extract). Rules introduce the tag approximation of the generic language explicitly, to be used on adjust contexts for evaluation. Complete system in Figure A 1.

$$\text{CE-Asgn}\ \frac{\textbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash \ (l_{\varepsilon_1} := w_{\varepsilon_2})_{(\varepsilon,\pi)} \ \mid \mu \rightsquigarrow \texttt{unit}_{\varepsilon} \mid \mu[l \mapsto w_{\varepsilon_2}]}$$

$$\text{CE-Frame}\ \frac{\textbf{adjust}_{A'(f)}(\Phi) \vdash e \mid \mu \rightsquigarrow e' \mid \mu'}{\Phi \vdash f[e] \mid \mu \rightsquigarrow f[e'] \mid \mu'} \qquad \text{CE-App}\ \frac{\textbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash \ ((\lambda x :\ T_1\ .\ e)_{\varepsilon_1}\ w_{\varepsilon_2})_{\pi} \ \mid \mu \rightsquigarrow [w_{\varepsilon_2}/x]e \mid \mu}$$

Fig. 13. The conservative semantics, with special frame translation function A', that maps annotated evaluation frames to adjust contexts (extract). Complete semantics in Figure A 2.

that could arise at runtime. We translate well-typed programs from the original internal syntax to the conservative language syntax using the translation algorithm introduced in Figure 12. The translation algorithm annotates redexes by extracting tag approximations from the typing derivation. The only interesting rules are [CT-App] and [CT-Asgn], which contain sub-expressions depending on adjust contexts $\pi\downarrow$ and $\pi :=\downarrow$, respectively. We extend function applications $(e_1\ e_2)$ and assignments $((e_1 := e_2)_{\varepsilon})$ with a statically approximated tagset π, which is used by the conservative semantics instead of the tag obtained from values in the original semantics. In Section 6.3 we prove that value tag annotations in the conservative language can be safely erased in an implementation.

A new operational semantics is introduced in Figure 13. Only 3 rules differ from the semantics in Figure 5: rules [CE-App], [CE-Asgn] and [CE-Frame]. Rules [CE-App] and [CE-Asgn] do not change their semantics, but they now operate on expressions with extra syntactic information. Upon reduction, this extra information is discarded.

Rule [CE-Frame] differs from the original semantics in how the adjust context for a particular frame is obtained. In the system introduced in Figure 5, rule [CE-Frame] uses function A to infer the proper adjust context for an evaluation frame. In the case for frames $w_{\varepsilon}\ \square$ and $(w_{\varepsilon} := \square)_{\varepsilon'}$, it used the tag ε in w_{ε} to create a singleton tagset and produce

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

Fig. 14. New frame translation function A', mapping annotated evaluation frames to adjust contexts.

the corresponding $\{\varepsilon\}\downarrow$ or $\{\varepsilon\} :=\downarrow$ context. In the conservative semantics, we instead use function A' as defined in Figure 14. A' uses the extended evaluation frames of the conservative language, which includes frames of the form $(v\ \square)_{\pi}$ and $(v := \square)_{(\varepsilon,\pi)}$ that carry an extra tagset π, used to produce the corresponding $\pi\ \downarrow$ or $\pi :=\downarrow$ adjust context. No tag information from values is used to infer adjust contexts, making tag information redundant at runtime.

6.2.2 Example of a Program with Observable Differences

Do the changes proposed affect the result of evaluating a program? In this section, we construct an example program that produces different results when evaluated in each semantics:

$$((\lambda f :\ \{\varepsilon_1,\varepsilon_2\}T_1 \xrightarrow{\emptyset} T_2\ .\ f\ \texttt{has}\ \{\texttt{read}\}!l_{\varepsilon})_{\varepsilon}\ (\lambda x :\ T_1\ .\ x)_{\varepsilon_1})_{\{\varepsilon\}} \tag{1}$$

In particular, we find that this program runs to completion in the standard semantics but produces a runtime error in the conservative semantics. As Section 6.3 shows, this is the only possible difference.

We first focus on the difference between both semantics that might cause different behavior. The differences only affect programs that perform runtime checks.

Tag monotonicity ensures that when tags are *removed* from a set π, **adjust** can only *increase* the privileges of the resulting set. Therefore, for programs to have different behavior between both semantics, the **adjust** functions must make strict use of this condition, and by the same argument, depend on the adjust context.

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

In our language, this only happens for adjust contexts that use the tagsets, $\pi\downarrow$ and $\pi :=\downarrow$. We therefore propose, as an example, the following definition for $\mathbf{adjust}_{\pi\downarrow}$:

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

For any other kind of adjust context, we use the identity function. We also define predicate $\mathbf{check}_{!\pi}(\Phi) \iff \texttt{read} \in \Phi$, and $\mathbf{check}_{C}(\Phi)$ to always hold for any other check context.

Let us build an example program in a language using this effect discipline and produces different results in both semantics. Since the only language construct that changes behavior with different Φ's is $\texttt{has}\ \Phi\ e$, and we require an adjust context of a form $\pi\ \downarrow$ or $\pi :=\downarrow$ to appear to observe differences, we introduce a `has` construct in the argument position of an application: a program of the form $(e_1\ \texttt{has}\ \Phi' e_2)_{\{\varepsilon_1,\varepsilon_2\}}$ in the conservative semantics. We introduce an annotation $\{\varepsilon_1,\varepsilon_2\}$ as minimal set that exhibits different behavior based only on tags. To have a $\{\varepsilon_1,\varepsilon_2\}$ frame annotation, expression e_1 needs to be typed as $\{\varepsilon_1,\varepsilon_2\}T_1 \xrightarrow{\Xi} T_2$, and therefore can only reduce to an abstraction, which may have either tag ε_1 or ε_2.

To define a concrete program that follows these restrictions, we just write e_1 as a variable f with the appropriate type, bound in a λ-abstraction. We use the familiar `read` privilege as the singleton element of set Φ', and build an e_2 whose meaning in the source language could eventually require a `has` check: $!l_\varepsilon$ for some location l. While a simpler expression like `unit` would suffice as e_2, it would make the example unrealistic.

$$e_3 = (\lambda f :\ \{\varepsilon_1,\varepsilon_2\}T_1 \xrightarrow{\emptyset} T_2\ .\ f\ \texttt{has}\ \{\texttt{read}\}!l_\varepsilon)_\varepsilon$$

Given our definition of **adjust**, the example in Program (1) will present observable differences.

This program can be typed with $\Xi = \{¿\}$ and evaluates to different results in each semantics. To focus on the interesting step of evaluation, we first apply substitution of the argument in the body of the function. Then we get the following cases for evaluation with $\Phi = \emptyset$. In the conservative semantics (Figure 13):

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

and in the original semantics (Figure 5):

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

This example shows that some programs which reduce to a value in the original semantics trigger errors instead in the conservative semantics. In the next section, we prove that this is the only difference between both semantics.

6.3 Conservative Semantics is a Conservative Approximation

When we label the semantics in Figure 13 as conservative, we mean it is a *conservative approximation* of the original semantics: if a program reduces to a value and a store in the modified semantics, it reduces to the same value and store in the original language (modulo tag annotations). If a program reduces to a runtime error in the conservative semantics, the program either reduces to an error or to a value in the original semantics.

We establish the relation between both semantics formally in the Conservative Approximation Theorem (Theorem 16). To introduce the theorem, we first define two auxiliary

notions: A tagset erasure function ($\mathscr{E}_{\mathbf{C}}$) that maps programs from the conservative language syntax to the original syntax by removing tag annotations, and a *simulation relation* that captures the relation between both languages. Theorem 16 establishes that evaluation preserves the simulation relation throughout.

Definition 15 (*Tagset erasure function* $\mathscr{E}_{\mathbf{C}}$)
We define function $\mathscr{E}_{\mathbf{C}} : \mathbf{Expr}_{\text{Conservative}} \to \mathbf{Expr}_{\text{Generic}}$ as follows:

$$\begin{array}{rcl}
\mathscr{E}_{\mathbf{C}}(\texttt{unit}_{\varepsilon}) & = & \texttt{unit}_{\varepsilon} \\
\mathscr{E}_{\mathbf{C}}(l_{\varepsilon}) & = & l_{\varepsilon} \\
\mathscr{E}_{\mathbf{C}}((\lambda x:\ T\ .\ e)_{\varepsilon}) & = & (\lambda x:\ T\ .\ \mathscr{E}_{\mathbf{C}}(e))_{\varepsilon} \\
\mathscr{E}_{\mathbf{C}}((e_1\ e_2)_{\pi}) & = & \mathscr{E}_{\mathbf{C}}(e_1)\ \mathscr{E}_{\mathbf{C}}(e_2) \\
\mathscr{E}_{\mathbf{C}}(\langle T_1 \Leftarrow T_0\rangle e) & = & \langle T_1 \Leftarrow T_0\rangle \mathscr{E}_{\mathbf{C}}(e) \\
\mathscr{E}_{\mathbf{C}}(\texttt{has}\ \Phi\ e) & = & \texttt{has}\ \Phi\ \mathscr{E}_{\mathbf{C}}(e) \\
\mathscr{E}_{\mathbf{C}}(\texttt{restrict}\ \Xi\ e) & = & \texttt{restrict}\ \Xi\ \mathscr{E}_{\mathbf{C}}(e) \\
\mathscr{E}_{\mathbf{C}}(\mathsf{Error}) & = & \mathsf{Error} \\
\mathscr{E}_{\mathbf{C}}((\texttt{ref}\ e)_{\varepsilon}) & = & (\texttt{ref}\ \mathscr{E}_{\mathbf{C}}(e))_{\varepsilon} \\
\mathscr{E}_{\mathbf{C}}(!e) & = & !\mathscr{E}_{\mathbf{C}}(e) \\
\mathscr{E}_{\mathbf{C}}\left((e_1 := e_2)_{(\varepsilon,\pi)}\right) & = & (\mathscr{E}_{\mathbf{C}}(e_1) := \mathscr{E}_{\mathbf{C}}(e_2))_{\varepsilon}
\end{array}$$

This tagset erasure function inverts the tag addition function $\Rrightarrow$ introduced in Figure 12. The only interesting cases are for application and assignment (highlighted). In both cases, we remove the extra π annotations that were introduced by the translation defined in Figure 12. Other cases simply apply the function recursively to subexpressions.

Simulation Relation. The simulation relation characterizes how we relate programs from the conservative semantics with programs in the original semantics. To avoid confusion, we underscore with a $_{\mathbf{C}}$ relations that should hold in the conservative semantics, and with an $_{\mathbf{O}}$ relations that should hold in the original semantics introduced in Section 4.2.

Definition 16 (*Simulation Relation*)

$$\frac{\begin{array}{c}\Xi;\Gamma;\Sigma \vdash_{\mathbf{C}} e_2 :\ T_2 \\ e_1 = \mathscr{E}_{\mathbf{C}}(e_2) \\ \Gamma;\Sigma \vDash_{\mathbf{O}} \mu_1 \qquad \Gamma;\Sigma \vDash_{\mathbf{C}} \mu_2 \\ \mu_1 = \mathscr{E}_{\mathbf{C}} \circ \mu_2\end{array}}{\Xi;\Gamma;\Sigma \ \Vdash\ (e_1,\mu_1) \sim (e_2,\mu_2)}$$

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

1. e_2 can be typed in the conservative type system using the context (There exists a type T such that $\Xi;\Gamma;\Sigma \vdash_{\mathbf{C}} e_2 :\ T$).
2. e_1 and e_2 correspond to the same expression modulo tag information ($e_1 = \mathscr{E}_{\mathbf{C}}(e_2)$)
3. Both stores μ_1 and μ_2 are consistent with the context, and μ_1 is equivalent to μ_2 without the extra tagset information ($\forall x, \mu_1(x) = \mathscr{E}_{\mathbf{C}}(\mu_2(x))$)

Theorem 14 (*Strong Conservative Approximation*)

.

Let $\Xi;\Gamma;\Sigma \Vdash (e_1,\mu_1) \sim (e_2,\mu_2)$ and $\Xi \vdash \Phi$. If $\Phi \vdash e_2 \mid \mu_2 \rightsquigarrow e_2' \mid \mu_2'$, then for any $\Xi \vdash \Phi'$, either:

- $\Phi' \vdash e_2' \mid \mu_2' \rightsquigarrow^* \mathsf{Error} \mid \mu_2'$
- $\exists e_1'$ and μ_1' such that $\Phi' \vdash e_1 \mid \mu_1 \rightarrow e_1' \mid \mu_1'$ and $\exists \Sigma' \supseteq \Sigma$ such that $\Xi;\Gamma;\Sigma' \Vdash (e_1',\mu_1') \sim (e_2',\mu_2')$.

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

The key step in the proof is that to use the induction hypothesis with rule [CE-Frame], we must ensure that the Φ used in the premise of original semantics' [E-Frame] rule ($\mathbf{adjust}_{A(f_\mathbf{O})}(\Phi)$) simulates the Ξ of the simulation relation ($\widetilde{\mathbf{adjust}}_{A'(f_\mathbf{C})}(\Xi) \vdash \mathbf{adjust}_{A(f_\mathbf{O})}(\Phi)$), which is non trivial since there are different adjust contexts used in each semantics. To do so we first establish a principle of well-formedness for frames (Definition 17). We then prove that well-formed frames is admissible from typing in the conservative language, and thus admissible from the simulation relation (Lemma 15). We then prove that simulation ensures well-formed frames, and that there is a partial ordering between the adjust contexts used in the conservative semantics and those in the original semantics ($A(f_\mathbf{O}) \sqsubseteq A'(f_\mathbf{C})$).

This partial ordering ensures that if Φ is a valid simulation privilege set for Ξ ($\Xi \vdash \Phi$), then also $\widetilde{\mathbf{adjust}}_{A'(f_\mathbf{C})}(\Xi) \vdash \mathbf{adjust}_{A(f_\mathbf{O})}(\Phi)$. This relation enables usage of the induction hypothesis. □

Definition 17 (*Well-formed conservative frames*)
We say that a frame $f_\mathbf{C}$ is well-formed if either:

- $f_\mathbf{C}$ is syntactically equivalent to a frame in the original semantics $f_\mathbf{O}$ (for example, $!\Box$)
- $f_\mathbf{C}$ is equivalent to a frame in the original semantics $f_\mathbf{O}$ *plus extra tagset information* that conservatively approximates the required tag information for value subexpressions. i.e. frame $(w_\varepsilon\ \Box)_\pi$ is well-formed only if $\varepsilon \in \pi$, and frame $(w_{\varepsilon_1} := \Box)_{(\varepsilon,\pi)}$ is well-formed only if $\varepsilon_1 \in \pi$.

Lemma 15 (*Typing in the conservative semantics ensures well-formed frames*)
If $\Xi;\Gamma;\Sigma \vdash e : T$, and $e = f_C[e']$, then f_C is a well-formed frame.

Proof
By cases on the final rule of the type derivation. □

Theorem 16 (*Conservative Approximation*)
Let $\Xi;\Gamma;\Sigma \vdash e_1 \Rrightarrow e_2 : T$, μ_1 and μ_2 such that $\Xi;\Gamma;\Sigma \Vdash (e_1,\mu_1) \sim (e_2,\mu_2)$, and $\Xi \vdash \Phi$. If $\Phi \vdash e_2 \mid \mu_2 \rightsquigarrow^* v_2 \mid \mu_2'$, then $\exists v_1$ and μ_1' such that $\Phi \vdash e_1 \mid \mu_1 \rightarrow^* v_1 \mid \mu_1'$ and $\exists \Sigma' \supseteq \Sigma$ such that $\Xi;\Gamma;\Sigma' \Vdash (v_1,\mu_1') \sim (v_2,\mu_2')$.

Proof
To prove this theorem, we establish an intermediate strong conservative approximation lemma (Theorem 14). Then this theorem reduces to the reflexive-transitive closure of Theorem 14. □

$$\text{CIT-App}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_0\left(T_1 \xrightarrow{\Xi_1} T_3\right) \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \textit{strict-check}_{\pi_1\,\pi_2}(\Xi) \qquad \pi_0\left(T_1 \xrightarrow{\Xi_1} T_3\right) <: \pi_1\left((\pi_2\rho_2) \xrightarrow{\Xi} T_3\right)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1\ e_2)_{\pi_1} : T_3}$$

$$\text{CIT-Asgn}\ \frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_0 \texttt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \textit{strict-check}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <: T_1 \qquad \pi_0 \subseteq \pi_1\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_{(\varepsilon,\pi_1)} : \{\varepsilon\}\texttt{Unit}}$$

Fig. 15. Type system for the conservative language (extract). Complete system in Figure A 3.

6.4 Type Safety of the Conservative Semantics

In this section we prove type safety for the conservative language. We introduce its type system in Figure 15. The key difference with the type system in Figure 4 is that it uses tagsets from the extra annotations in redexes instead of the tagsets from the typing of e_1 in rules [CIT-App] and [CIT-Asgn] (which is highlighted in boxes in Figure 15). This change also requires explicit subsumption of tagsets to relate the tagset in the redex annotation back to the tagset in the type of e_1.

Theorem 17 (*Progress*)
Suppose $\Xi;\emptyset;\Sigma \vdash e : T$. Then either e is a value v, an Error, or $\Phi \vdash e \mid \mu \rightsquigarrow e' \mid \mu'$ for all privilege sets $\Phi \in \gamma(\Xi)$ and for any store μ such that $\emptyset \mid \Sigma \vdash \mu$.

Proof
By structural induction on type derivations. Most cases are analogous to the original language's proof of progress, since most typing rules are the same. However, this theorem does not depend on effect satisfaction: representation is sufficient. This happens because rules [CIT-App] and [CIT-Asgn] use the same adjust contexts at runtime and statically, so now the representation condition ($\Phi \in \gamma(\Xi)$) is sufficient and, unlike in the original language, we do not need to appeal to privilege set satisfaction. □

Theorem 18 (*Preservation*)
If $\Xi;\Gamma;\Sigma \vdash e : T$, and $\Phi \vdash e \mid \mu \rightsquigarrow e' \mid \mu'$ for $\Phi \in \gamma(\Xi)$ with $\Gamma \mid \Sigma \vdash \mu$, then $\exists \Sigma' \supseteq \Sigma$ such that $\Gamma \mid \Sigma' \vdash \mu'$ and $\Xi;\Gamma;\Sigma' \vdash e' : T'$ with $T' <: T$.

Proof
By structural induction on type derivations and the applicable evaluation rules. Most cases follow analogously to the original language, except for:

- Typing rule [CIT-App] with evaluation rule [CE-Frame] using $f = (v\Box)_\pi$.
- Typing rule [CIT-Asgn] with evaluation rule [CE-Frame] using $f = (v := \Box)_{(\varepsilon,\pi)}$.

Both cases are simpler in the sense that they do not require the tag monotonicity restrictions from Property 2 that were used in the original language to account for the difference between the tagset used at evaluation and the one used for typing. □

6.5 Redundancy of Tags in the Conservative Semantics

We introduced the conservative semantics to avoid carrying tag information at runtime. To formalize the claim that tag information is redundant and thus it may be discarded at runtime, we introduce a simulation argument. We can easily define an operational semantics $\hookrightarrow$ based on the $\rightsquigarrow$ conservative semantics defined in Figure 13, except that we remove every **check** predicate (so that they don't have to be checked at runtime), and also define a tag-removal function $\mathscr{E}_V\,()$ as follows

$$\begin{array}{rcl}
\mathscr{E}_V\,(\mathtt{unit}_\varepsilon) & = & \mathtt{unit} \\
\mathscr{E}_V\,(l_\varepsilon) & = & l \\
\mathscr{E}_V\,((\lambda x:\ T\ .\ e)_\varepsilon) & = & (\lambda x:\ T\ .\ \mathscr{E}_V\,(e)) \\
\mathscr{E}_V\,((e_1\ e_2)_\pi) & = & (\mathscr{E}_V\,(e_1)\ \mathscr{E}_V\,(e_2))_\pi \\
\mathscr{E}_V\,(\langle T_1 \Leftarrow T_0\rangle e) & = & \langle T_1 \Leftarrow T_0\rangle \mathscr{E}_V\,(e) \\
\mathscr{E}_V\,(\mathtt{has}\ \Phi\ e) & = & \mathtt{has}\ \Phi\ \mathscr{E}_V\,(e) \\
\mathscr{E}_V\,(\mathtt{restrict}\ \Xi\ e) & = & \mathtt{restrict}\ \Xi\ \mathscr{E}_V\,(e) \\
\mathscr{E}_V\,(\mathsf{Error}) & = & \mathsf{Error} \\
\mathscr{E}_V\,((\mathtt{ref}\ e)_\varepsilon) & = & \mathtt{ref}\ \mathscr{E}_V\,(e) \\
\mathscr{E}_V\,(!e) & = & !\mathscr{E}_V\,(e) \\
\mathscr{E}_V\left((e_1 := e_2)_{(\varepsilon,\pi)}\right) & = & (\mathscr{E}_V\,(e_1) := \mathscr{E}_V\,(e_2))_\pi
\end{array}$$

We can then state the following theorem relating both semantics:

Theorem 19 (**check** *and tags are redundant in* $\rightsquigarrow$)
If $\Xi;\Gamma;\Sigma \vdash e:\ T$ and $\Phi \vdash e \mid \mu \rightsquigarrow e' \mid \mu'$ for $\Phi \in \gamma(\Xi)$ and $\Gamma \mid \Sigma \vDash \mu$, then also $\Phi \vdash \mathscr{E}_V\,(e) \mid \mathscr{E}_V\,(\mu) \hookrightarrow \mathscr{E}_V\,(e') \mid \mathscr{E}_V\,(\mu')$.

Proof
The **check** predicates in the operational semantics always hold due to the typing hypothesis, since we know by Definition 5 that ***strict-check***$_A(\Xi)$ implies **check**$_A(\Phi)$ $\forall \Phi \in \gamma(\Xi)$. We can thus remove the **check** predicates from the conservative semantics and ensure that reduction behaves identically. After removing **check** predicates, ε tag annotations do not affect reduction, so if $\hookrightarrow$ is defined for programs both with and without value tag annotations, we can prove that $\Phi \vdash e \mid \mu \hookrightarrow e' \mid \mu' \iff \Phi \vdash \mathscr{E}_V\,(e) \mid \mathscr{E}_V\,(\mu) \hookrightarrow \mathscr{E}_V\,(e') \mid \mathscr{E}_V\,(\mu')$.
□

7 Gradual Typing and Gradual Effects: Gradual Type-and-Effect Systems

The gradual effects framework empowers the programmer to decide when and where to introduce effect annotations, automatically adding required checks to ensure safety. However, this flexibility does not yet apply to types: we require fully statically checked types (except for effect annotations). In this section we introduce gradual typing in our framework, giving the programmer full flexibility over both effect and type annotations.

$$\varepsilon \in \mathbf{Tags} \,.\, \pi \in \mathscr{P}(\mathbf{Tags})$$

w	::=	$\mathtt{unit} \mid \lambda x : T \,.\, e \mid l$	Prevalues
v	::=	w_ε	Values
e	::=	$x \mid v \mid e\ e \mid (\mathtt{ref}\ e)_\varepsilon \mid\ !e \mid (e := e)_\varepsilon$	Terms
T	::=	$\pi\ \rho \mid\ ?$	Types
ρ	::=	$\mathtt{Unit} \mid T \xrightarrow{\Xi} T \mid \mathtt{Ref}\ T$	PreTypes

Fig. 16. Syntax of the source language for gradual typing with tags

Marino and Millstein's framework for generic type-and-effect systems accepts values with tag annotations. Static reasoning about tags is achieved by annotating types with sets of tags. For example, the type $(\pi\ \mathtt{Unit})$ only denotes values $\mathtt{unit}_\varepsilon$ with $\varepsilon \in \pi$. However, since the unknown type ? in gradual typing is not tagged, some tag set must be assumed for ?. To introduce complete gradual annotations for types and effects in the M&M framework, we take a two-step approach: We first introduce gradual typing for a simple language with tags, where we assume that all tags are available for ?, corresponding to the universe set of tags (**Tags**). Then we move on to introduce gradual typing, observing that gradual typing and gradual effects are *almost* orthogonal: the same assumption that all tags are available for ? applies to our generic gradual effects framework as the final link to achieve a completely gradual type-and-effect system.

7.1 Extending Gradual Typing for Tag Annotations

We first focus on the interactions between tags and gradual typing. To do so, we introduce tags to the gradually-typed lambda calculus introduced by (Siek & Taha, 2006).

The syntax of the language is introduced in Figure 16; it adds tags to the gradually-typed lambda calculus with references. To do so, we redefine types to consist of both a tag set π and a pretype ρ.

? is still a type When adding tags, we face a design choice: to consider ? as a pretype (ρ) or as a type (T). Type ? was introduced in gradual typing to permit programs missing type information, which leads us to consider ? as a type. If ? were a pretype instead, every program would require type tag annotations, forcing static checking. This requirement defeats the purpose of gradual typing.

Since tags change our definition of types, we must update the definitions of other notions based on types. In particular, we must update both type consistency and consistent subtyping.

Definition 18 (*Type Consistency*)
We define type consistency as the reflexive and symmetric relation introduced by Siek & Taha (2006), extending rule [C-Fun] to be reflexive on the set of tags.

$$\text{C-Fun}\frac{T'_1 \sim T_1 \qquad T'_2 \sim T_2}{\big(\ \pi\ (T'_1 \to T'_2)\big) \sim \big(\ \pi\ (T_1 \to T_2)\big)}$$

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

$$\text{G-App}\frac{\Gamma;\Sigma \vdash e_1 : T \quad \Gamma;\Sigma \vdash e_2 : T_2 \quad T \lesssim \pi(T_2 \to T_3)}{\Gamma;\Sigma \vdash e_1\ e_2 : T_3} \qquad \text{G-Deref}\frac{\Gamma;\Sigma \vdash e : T_1 \quad T_1 \sim \pi\texttt{Ref}\ T}{\Gamma;\Sigma \vdash !e : T}$$

$$\text{G-Asgn}\frac{\Gamma;\Sigma \vdash e_1 : T \quad T \sim \pi\texttt{Ref}\ T_1 \quad \Gamma;\Sigma \vdash e_2 : T_2 \quad T_2 \lesssim T_1}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Fig. 17. Typing rules for the source language for gradual typing with tags. Complete definition in Figure B 1

7.1.1 Consistent subtyping

Tag annotations in types are sets, leading to a natural notion of subtyping. Siek & Taha (2007) combined gradual typing and subtyping in the context of object oriented languages, and we follow their strategy to combine the notion of subtyping induced by tags with type consistency:

Definition 19 (*Consistent Subtyping*)
Following the work of (Siek & Taha, 2007), we define consistent subtyping ($\lesssim$) as follows

$$a \lesssim b \triangleq \exists \alpha \sim a\ .\ \alpha <: b$$

In our gradually-typed lambda calculus with tags and references, the subtyping relation is defined as follows:

Definition 20 (*Subtyping relation*)

$$\boxed{T <: T} \qquad \text{ST-Id}\frac{\pi_1 \subseteq \pi_2}{\pi_1\ \rho <: \pi_2\ \rho} \qquad \text{ST-Dyn}\frac{}{? <: ?}$$

$$\text{ST-Abs}\frac{T_3 <: T_1 \quad T_2 <: T_4 \quad \pi_1 \subseteq \pi_2}{\pi_1\ T_1 \to T_2 <: \pi_2\ T_3 \to T_4}$$

The language in Siek & Taha (2007) defined consistent subtyping by equivalently using consistency on either side of the subtyping relation (but not both). With our definitions of type consistency $\sim$ and subtyping, we prove this very property as the following corollary:

Corollary 20 (*Consistent subtyping equivalence*)

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

Proof
By structural induction over the type consistency definition ($\sim$). □

Using consistent subtyping, we define a type system for the source language in Figure 17.

$$
\begin{array}{lcll}
v & ::= & \ldots \mid \langle ? \Leftarrow T \rangle v, \text{ if } T \neq ? & \text{Values} \\
e & ::= & \ldots \mid \mathsf{Error} \mid \langle T \Leftarrow T \rangle e & \text{Terms} \\
f & ::= & \Box\, e \mid v\, \Box \mid (\mathtt{ref}\ \Box)_\varepsilon \mid !\Box \mid (\Box := e)_\varepsilon \mid (w_\varepsilon := \Box)_\varepsilon & \text{Frames} \\
g & ::= & f \mid \langle T_2 \Leftarrow T_1 \rangle \Box & \text{Error Frames}
\end{array}
$$

Fig. 18. Syntax of the internal language for gradual typing with tags

$$
\boxed{\Gamma;\Sigma \vdash e : T}
\qquad
\text{GT-App}\ \frac{\Gamma;\Sigma \vdash e_1 : \pi\,(T_1 \to T_3) \quad \Gamma;\Sigma \vdash e_2 : T_2 \quad \pi\,(T_1 \to T_3) <: \pi\,(T_2 \to T_3)}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3}
$$

$$
\text{GT-Asgn}\ \frac{\Gamma;\Sigma \vdash e_1 : \pi \mathtt{Ref}\ T_1 \quad \Gamma;\Sigma \vdash e_2 : T_2 \quad T_2 <: T_1}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}}
$$

Fig. 19. Typing rules of interest for the internal language for gradual typing with tags. Complete definition in Figure B 2.

7.1.2 Internal language

The syntactic extensions required for the internal language are introduced in Figure 18. A type system for the internal language is introduced in Figure 19, which differs from the source language only in depending directly on subtyping instead of consistent subtyping. The operational semantics introduced in Figure 20 is the semantics of the gradually typed lambda-calculus with references plus tags. Standard rules [GE-Ref], [GE-Asgn], [GE-Deref] and [GE-App] are augmented to consider tag-annotated values. Rule [GE-Cast-Id]

$$
\boxed{e \mid \mu \to e \mid \mu}
$$

$$
\text{GE-Ref}\ \frac{l \notin \mathrm{dom}\,(\mu)}{(\mathtt{ref}\ v)_\varepsilon \mid \mu \to l_\varepsilon \mid \mu[l \mapsto v]}
\qquad
\text{GE-Deref}\ \frac{\mu(l) = v}{!l_\varepsilon \mid \mu \to v \mid \mu}
$$

$$
\text{GE-Asgn}\ \frac{}{\left(l_{\varepsilon_1} := v\right)_\varepsilon \mid \mu \to \mathtt{unit}_\varepsilon \mid \mu[l \mapsto v]}
\qquad
\text{GE-App}\ \frac{}{(\lambda x : T_1\,.\,e)_{\varepsilon_1}\ v \mid \mu \to [v/x]e \mid \mu}
$$

$$
\text{GE-Cast-Id}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\langle \pi_2 \rho \Leftarrow \pi_1 \rho \rangle w_\varepsilon \mid \mu \to w_\varepsilon \mid \mu}
\qquad
\text{GE-Cast-Dyn}\ \frac{}{\langle ? \Leftarrow ? \rangle v \mid \mu \to v \mid \mu}
$$

Fig. 20. Interesting rules for small-step semantics of the internal language for gradual typing with tags. Complete definition in Figure B 3.

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

CG-Ref-2
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : ?}{\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon \Rightarrow (\texttt{ref}\ e')_\varepsilon : \{\varepsilon\}\texttt{Ref}\ ?}$$

CG-Deref-2
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : ?}{\Gamma;\Sigma \vdash !e \Rightarrow\ !\langle \mathbf{Tags}(\texttt{Ref}\ ?) \Leftarrow ? \rangle e' : T}$$

CG-Asgn-2
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1 \texttt{Ref}\ T_1 \quad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \quad e''_2 = \langle T_1 \Leftarrow ? \rangle e_2'}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow (e_1' := e''_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

CG-Asgn-3
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \quad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : T \quad e''_1 = (\langle \mathbf{Tags}(\texttt{Ref}\ T) \Leftarrow ? \rangle e_1')}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow (e''_1 := e_2')_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

C-App-2
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \to T_3) \quad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \quad e''_1 = (\langle\!\langle \pi_1(T_1 \to T_3) \Leftarrow \pi_1(T_1 \to T_3) \rangle\!\rangle e_1')}{\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow e''_1\ \left((\langle\!\langle T_1 \Leftarrow ? \rangle\!\rangle e_2') \right) : T_3}$$

C-App-3
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \quad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : T \quad e''_1 = (\ \langle\!\langle \mathbf{Tags}(T \to ?) \Leftarrow ? \rangle\!\rangle\ e_1')}{\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow e''_1\ e_2' : ?}$$

Fig. 21. Extract of translation of source programs to the internal language for gradual typing with tags. Complete rules in Figure B 4 and Figure B 5.

handles casts for tagged types, verifying that the subtyping condition for π_1 and π_2 holds. A separate rule [GE-Cast-Dyn] handles identity casts for ?.

7.1.3 Tags interact with the translation algorithm

The translation algorithm introduces the runtime *checks* necessary to ensure that a program is safe, making the optimistic assumptions of the original language explicit by inserting type casts. But what tags are valid when we do not statically know the type of an expression (i.e., the case of type ?)?

To define a safe system, we make the conservative assumption that an expression with a type ? may at runtime have *any* tag annotation. Therefore, we assume that an expression with type ? must provide the *universe set of tag annotations* (denoted **Tags**). This assumption is consistent with the tag monotonicity restrictions of gradual effect checking, inherited from the generic M&M framework, so no extra assumptions are required to combine tagged gradual typing with gradual effect checking: If we use the set **Tags**, these monotonicity restrictions ensure that the required restrictions for ***strict-check*** predicates and $\widetilde{\textbf{adjust}}$ functions always hold, hence ensuring type safety.

Of course, using **Tags** as assumption is very conservative. A different approach is to introduce a notion of graduality among tags. We plan to explore this approach in future work, but since tags are not fundamental to effects, we consider this approximation sufficient as it makes definitions simpler.

7.1.4 Type safety

Theorem 21 (*Progress*)

$$\begin{array}{lcll} T & ::= & \pi\,\rho \mid ? & \text{Types} \\ \rho & ::= & \texttt{Unit} \mid T \xrightarrow{\Xi} T \mid \texttt{Ref}\ T & \text{PreTypes} \end{array}$$

Fig. 22. Syntax addition to the source language for gradual type-and-effect systems from Figure 1. A complete syntax is available in Figure B 6.

Suppose $\emptyset;\Sigma \vdash e : T$. Then either e is a value or for any store μ such that $\emptyset \mid \Sigma \vDash \mu$, $e \mid \mu \to e; \mid \mu'$.

Proof
By structural induction over derivations of $\emptyset;\Sigma \vdash e : T$. The only interesting cases arise for rule [GT-Cast]. For rule [GT-Cast], the proof proceeds by cases over the presence (or lack) of consistent subtyping between T_1 and T_2. □

Theorem 22 (*Preservation*)
If $\Gamma;\Sigma \vdash e : T$, and $e \mid \mu \to e' \mid \mu'$ for $\Gamma \mid \Sigma \vDash \mu$, then $\Gamma \mid \Sigma' \vDash \mu'$ and $\Gamma;\Sigma' \vdash e' : T'$ for some $T' <: T$ and $\exists \Sigma' \supseteq \Sigma$.

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

Theorem 23 (*Translation preserves typing*)
If $\Gamma;\Sigma \vdash e \Rightarrow e' : T$ in the source language then $\Gamma;\Sigma \vdash e' : T$ in the internal language.

Proof
By structural induction over the translation rules. □

7.2 Combining Gradual Typing and Gradual Effect Checking

Once we have extended gradual typing with tag annotations, we can easily combine gradual typing and gradual effect checking to deliver a system that provides static and dynamic type-and-effect checking. The syntax for this language is provided in Figure 22. It combines the typing, translation, and evaluation rules from gradual typing with tags and gradual effect checking. In this section, we focus on the modifications required to combine both systems: we build a notion of "effect consistency" from our definition of consistent containment, our definition of subtyping changes to encompass both tag annotations and effects, and we reuse the assumptions required to introduce tags into gradual typing to define a new translation relation.

7.2.1 Type and effect consistency

Gradual typing uses type consistency to statically accept expressions with type ? (or whose type is partially unknown) whenever expressions with particular type are required, like in a function application. Type consistency acts as a relaxed form of equality: whenever a certain type T_1 is required for a program to be valid, a program with an unknown type must also be statically accepted, because the type system is not able to distinguish between them. The definition of type consistency introduced by Siek & Taha (2006) does not consider effect annotations, so we must provide a type consistency relation that does.

Our analysis of gradual effect checking did not require a notion of "effect consistency". Instead, we based gradual effect checking on consistent containment, which acted as a relaxed form of set containment. We define effect consistency again using abstract interpretation:

Definition 21 (*Effect consistency*)
The consistent privilege sets Ξ_1 and Ξ_2 are *consistent*, denoted $\Xi_1 \simeq \Xi_2$, if and only if $\Phi \in \gamma(\Xi_1)$ *and* $\Phi \in \gamma(\Xi_2)$ for some $\Phi \in$ **PrivSet**.

Like for consistent containment, we also provide a simple direct characterization. As with standard sets we may prove that two sets a and b are equal if $a \subseteq b$ and $b \subseteq a$, we prove that two sets are consistent if they are mutually contained consistently:

Proposition 24
$\Xi_1 \simeq \Xi_2 \iff \Xi_1 \sqsubseteq_{\sim} \Xi_2$ and $\Xi_2 \sqsubseteq_{\sim} \Xi_1$.

Proof
($\Leftarrow$) By cases on the definition of γ. ($\Rightarrow$) direct from Definition 21. □

We use effect consistency to extend type consistency to handle effects. As in Siek & Taha (2006), reference cell types are only consistent with themselves (since type consistency is reflexive). We define type consistency as follows:

$$\boxed{T \sim T} \qquad \text{C-Refl}\,\frac{}{T \sim T} \qquad \text{C-UnR}\,\frac{}{T \sim ?} \qquad \text{C-UnL}\,\frac{}{? \sim T}$$

$$\text{C-Fun}\,\frac{T'_1 \sim T_1 \qquad T'_2 \sim T_2 \qquad \Xi' \simeq \Xi}{\pi\left(T'_1 \xrightarrow{\Xi'} T'_2\right) \sim \pi\left(T_1 \xrightarrow{\Xi} T_2\right)}$$

7.2.2 Static Semantics for the Source Language

In this section we introduce the type system for the source language, presented in Figure 23. This type system, as was the case for gradual effect checking, depends on two relations, subtyping ($<:$) and consistent subtyping ($\lesssim$), which we now define.

Subtyping The subtyping relation introduced in Section 7.1 suffices to define gradual typing (?), but does not consider effects. We build a sufficient definition of subtyping

for gradual typing with gradual effect checking by extending the subtyping definition introduced in Section 4.2 to account for reflexivity of type ? with the following rule:

$$\frac{}{? <: ?}$$

Consistent Subtyping We migrate the definition of consistent subtyping from gradual typing and gradual effect checking, using the new definitions of subtyping and type consistency we have presented.

Definition 22 (*Consistent Subtyping*)
Consistent subtyping ($\lesssim$) is defined as

$$a \lesssim b \text{ if and only if } \exists \alpha \sim a \,.\, \alpha <: b \text{ (if and only if } \exists \beta \sim b \,.\, a <: \beta)$$

We use type consistency and consistent subtyping in the type system for our language as presented in Figure 23. The internal language syntax is introduced in Figure 24, and only differs from the syntax from Figure 18 in the usage of effect annotations from gradual effect checking.

We consider this definition of consistent subtyping an extension of the definition presented in Section 4. The formal relationship between both definitions is established by the following theorem:

Theorem 25
Let $\lesssim_{GE}$ be the consistent subtyping definition in Section 4 and $\lesssim_{GT}$ consistent subtyping as introduced in Definition 22. If $T_1 \lesssim_{GE} T_2$, then also $T_1 \lesssim_{GT} T_2$ (i.e., $\lesssim_{GE} \subseteq \lesssim_{GT}$)

Proof
By structural induction on the definition of subtyping in gradual effect checking ($\lesssim_{GE}$), using the following lemma:

Lemma 26
$\Xi_1 \sqsubseteq_{\sim} \Xi_2$ if and only if there exists a Ξ' such that $\Xi_1 \simeq \Xi'$ and $\Xi' \subseteq \Xi_2$

□

Type consistency in the type system The type system in Figure 23 uses type consistency to retrieve a tag set π for $\widetilde{\textbf{check}}$ predicates. Upon translation, we make a conservative approximation (using the universe set **Tags**), that ensures safety and is in line with the assumptions required to combine gradual typing and tag annotations, as discussed in Section 7.1.

7.2.3 Internal Language

For our system with gradual typing, we introduce internal language extensions in Figure 24, which are similar to the ones in Figure 3.

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

$$\text{GT-App}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : T \quad \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : T_2 \quad T_2 \sim \pi_2\rho_2 \quad T \lesssim \pi_1(T_2 \xrightarrow{\Xi} T_3) \quad \widetilde{\textbf{check}}_{\pi_1\pi_2}(\Xi)}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3}$$

$$\text{GT-Ref}\ \frac{\widetilde{\textbf{adjust}}_{\texttt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e : T \quad T \sim \pi\rho \quad \widetilde{\textbf{check}}_{\texttt{ref}\ \pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho}$$

$$\text{GT-Deref}\ \frac{\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e : T_1 \quad T_1 \sim \pi\texttt{Ref}\ T \quad \widetilde{\textbf{check}}_{!\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !e : T}$$

$$\text{GT-Asgn}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : T \quad T \sim \pi_1\texttt{Ref}\ T_1 \quad \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : T_2 \quad T_2 \sim \pi_2\rho_2 \quad \widetilde{\textbf{check}}_{\pi_1:=\pi_2}(\Xi) \quad T_2 \lesssim T_1}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Fig. 23. Extract of typing rules for the source language for gradual type-and-effect systems. Complete judgment in Figure B 7.

v	::=	$\ldots \mid \langle ? \Leftarrow T \rangle v$, if $T \neq ?$	Values
e	::=	$\ldots \mid \mathsf{Error} \mid \langle T \Leftarrow T \rangle e \mid \texttt{has}\ \Phi\ e \mid \texttt{restrict}\ \Xi\ e$	Terms
f	::=	$\square\ e \mid v\ \square \mid (\texttt{ref}\ \square)_\varepsilon \mid !\square \mid (\square := e)_\varepsilon \mid (w_\varepsilon := \square)_\varepsilon$	Frames
g	::=	$f \mid \langle T_2 \Leftarrow T_1 \rangle \square \mid \texttt{has}\ \Phi\ \square \mid \texttt{restrict}\ \Xi\ \square$	Error Frames

Fig. 24. Syntax of the internal language for gradual type-and-effect systems

Type System The type system introduced in Figure 25 only differs from the type system in Figure 4 in rule [GIT-Cast], which must be more flexible to encompass type cast failures. In gradual effect checking, we could be more restrictive with the types in the casts and require $T_1 \lesssim T_2$ because type casts would never fail due to type inconsistencies, but would instead be reduced to a combination of effect-related `restrict` and `has` constructs that may or may not fail depending on the available privileges. When we introduce type ?, we require more flexibility to preserve safety, in particular for type preservation: A program $\langle \texttt{Nat} \Leftarrow ? \rangle \langle ? \Leftarrow \texttt{Unit} \rangle \texttt{unit}$ should reduce to a cast $\langle \texttt{Nat} \Leftarrow \texttt{Unit} \rangle \texttt{unit}$ that would fail in a later step, but cast $\langle \texttt{Nat} \Leftarrow \texttt{Unit} \rangle$ cannot be typed if rule [GIT-Cast] includes a consistent subtyping restriction between `Unit` and `Nat`, breaking preservation[16].

Operational Semantics We reuse all the rules of the operational semantics introduced in Figure 5 to provide the operational semantics for gradual typing detailed in Figure 26,

[16] In most of the examples of this section, we elide tags (both in types and terms) for clarity and focus.

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

$$\text{GIT-Ref}\ \frac{\widetilde{\textbf{adjust}}_{\texttt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\rho \qquad \textit{strict-check}_{\texttt{ref}\ \pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ e)_{\varepsilon} : \{\varepsilon\}\texttt{Ref}\ \pi\rho} \qquad \text{GIT-Deref}\ \frac{\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\texttt{Ref}\ T \qquad \textit{strict-check}_{!\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !e : T}$$

$$\text{GIT-Asgn}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_1\texttt{Ref}\ T_1 \quad \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \quad \textit{strict-check}_{\pi_1:=\pi_2}(\Xi) \quad \pi_2\rho_2 <: T_1}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_{\varepsilon} : \{\varepsilon\}\texttt{Unit}}$$

$$\text{GIT-App}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_1\left(T_1 \xrightarrow{\Xi_1} T_3\right) \quad \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \quad \textit{strict-check}_{\pi_1\pi_2}(\Xi) \quad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <: \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3}$$

Fig. 25. Typing rules for the internal language for gradual type-and-effect systems

$$\boxed{\Phi \vdash e \mid \mu \to e \mid \mu}$$

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

$$\text{GTE-Cast-Id}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2\rho \Leftarrow \pi_1\rho\rangle w_{\varepsilon} \mid \mu \to w_{\varepsilon} \mid \mu} \qquad \text{GTE-Cast-Merge}\ \frac{}{\Phi \vdash \langle T_2 \Leftarrow ?\rangle\langle ? \Leftarrow T_1\rangle v \mid \mu \to \langle T_2 \Leftarrow T_1\rangle v \mid \mu}$$

$$\text{GTE-Cast-Dyn}\ \frac{}{\Phi \vdash \langle ? \Leftarrow ?\rangle v \mid \mu \to v \mid \mu} \qquad \text{GTE-Cast-Bad}\ \frac{T_1 \not<: T_2}{\Phi \vdash \langle T_2 \Leftarrow T_1\rangle v \mid \mu \to \mathsf{Error} \mid \mu}$$

Fig. 26. Extract of the small-step semantics of the internal language for gradual type-and-effect systems. These rules extend the semantics in Figure 5 with type casts. Complete semantics in Figure B 9.

which is extended with more rules for cast evaluation. Unlike gradual effect checking, type casts in this semantics may fail for reasons other than effect restrictions, since type consistency is not transitive. Consider the program$(\lambda f : ?\ .\ (f\ \texttt{unit}))\ \texttt{unit}$. After translation and substitution, this program steps to

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

First, both casts are merged into a $\langle \texttt{Unit} \xrightarrow{\Xi} ? \Leftarrow \texttt{Unit}\rangle$ cast, by rule [GTE-Cast-Merge]. This cast then fails because both types are not consistent subtypes, by rule [GTE-Cast-Bad]. The operational semantics also includes a reduction rule for identity ? casts, rule [GTE-Cast-Dyn].

7.2.4 Translation Algorithm

In this section, we introduce the translation algorithm for gradual type-and-effect systems proposed in Figure 27. The translation algorithm is where the interesting connections between gradual typing and gradual effect checking arise, because some translation rules use the tag set of a type to generate an adjust or a check context. Following the assumptions introduced when combining gradual typing with tag annotations, we assume the universe of tags (set **Tags**) whenever the translation has to extract a tag set from an expression with an unknown type (?).

Whenever possible, we use the tag information available on types instead of recoursing to the **Tags** assumption. To do so, we introduce two separate translation rules for $(\texttt{ref}\ e)$ and $!e$ constructs (when e has type ? or does not, respectively), and four separate translation rules for application and assignment expressions (when e_1 and e_2 have type ? or do not, respectively).

In the case of `ref` e and $!e$ constructs, rules [C-GT-Ref-1], [C-GT-Ref-2], [C-GT-Deref-1] and [C-GT-Deref-2] handle type and tag assumptions. In gradual typing, there was no need to introduce separate rules for `ref` e constructs. We introduce separate rules to limit the case where assumptions for the check context are required. In rule [C-GT-Ref-2], when e has type ?, the translation rule uses a check context `ref` **Tags** for the $\widetilde{\textbf{check}}$ predicate and for the Δ function that collects the missing privileges (if any) to perform allocation. For $!e$ constructs, like in gradual typing, we have two separate translation rules. Rule [C-GT-Deref-2] makes explicit the assumption that e should have type $\widetilde{\pi\texttt{Ref}\ T}$ for some π and T and the assumptions required for the check contexts used in $\widetilde{\textbf{check}}$ and Δ as in rule [C-GT-Ref-2].

There are four translation rules for function applications. Rule [C-GT-App-1] is exactly the same rule [C-GT-App-1] used in the generic gradual effect system, and applies when types for both the operator and argument are known. Rule [C-GT-App-2] applies when the argument type is unknown (?), and inserts a cast for the argument from ? to the type of the operator domain. It assumes that e_2' might hold any set of tags, so the **check** and Δ functions use the universe of tags available for the argument. Rule [C-GT-App-2] needs a cast on the operator for gradual effect checking, which performs an effect coercion ensuring that the privileges required for the function type (Ξ_1) are actually available in the context Ξ.

Rule [C-GT-App-3] applies when the operator has type ?. We assume that the type of the argument is known, leaving the case of both elements having type ? for rule [C-GT-App-4]. In rule [C-GT-App-3], we do not have tag information for the function type, information that is needed to adjust the privileges available to translate e_2. As in rule [C-GT-App-2], we assume the maximum set of tags. The assumed set is also used for the **check** and Δ functions, and for the cast that ensures that e_1 is a function at all. By our definition of subtyping, a function which has any set of tags π_1 (and the appropriate privilege set and parameter and return types) is accepted since in the cast $\pi_1 \subseteq$ **Tags** always holds.

Rule [C-GT-App-4] makes the assumptions from rule [C-GT-App-3] explicit, but also assumes that the argument has type ?. We have two separate rules because if e_2' has type ?, we must make tag assumptions also for the argument to generate the check contexts required by **check** and Δ.

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

$$\text{C-GT-Ref-2}\ \frac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\downarrow}(\Xi);\Gamma;\Sigma\vdash e\Rightarrow e' : ?\qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\ \mathbf{Tags}}(\Xi)\qquad \Phi = \Delta_{\mathtt{ref}\ \mathbf{Tags}}(\Xi)}{\Xi;\Gamma;\Sigma\vdash (\mathtt{ref}\ e)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}(\Phi,(\mathtt{ref}\ e')_\varepsilon) : \{\varepsilon\}\mathtt{Ref}\ ?}$$

$$\text{C-GT-Deref-2}\ \frac{\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma\vdash e\Rightarrow e' : ?\qquad \widetilde{\mathbf{check}}_{!\mathbf{Tags}}(\Xi)\qquad \Phi = \Delta_{!\mathbf{Tags}}(\Xi)}{\Xi;\Gamma;\Sigma\vdash !e \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\ !\langle \mathbf{Tags}(\mathtt{Ref}\ ?)\Leftarrow ?\rangle e'\right) : T}$$

$$\text{C-GT-Asgn-2}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : \pi_1\mathtt{Ref}\ T_1\\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : ?\qquad \widetilde{\mathbf{check}}_{\pi_1:=\mathbf{Tags}}(\Xi)\qquad \Phi = \Delta_{\pi_1:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash (e_1:=e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\left(e_1' := \langle T_1\Leftarrow ?\rangle e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

$$\text{C-GT-Asgn-3}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : ?\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : \pi_2\rho_2\qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\pi_2}(\Xi)\qquad \Phi = \Delta_{\mathbf{Tags}:=\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash (e_1:=e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\left(\ (\langle \mathbf{Tags}(\mathtt{Ref}\ ?)\Leftarrow ?\rangle e_1')\ := e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

$$\text{C-GT-Asgn-4}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : ?\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : ?\qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi)\qquad \Phi = \Delta_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash (e_1:=e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\left(\ (\langle \mathbf{Tags}(\mathtt{Ref}\ ?)\Leftarrow ?\rangle e_1')\ := e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

$$\text{C-GT-App-2}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : \pi_1(T_1\xrightarrow{\Xi_1}T_3)\qquad \Xi_1\sqsubseteq_\sim \Xi\\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : ?\qquad \widetilde{\mathbf{check}}_{\pi_1\mathbf{Tags}}(\Xi)\qquad \Phi = \Delta_{\pi_1\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\ \left(\langle\!\langle \pi_1(T_1\xrightarrow{\Xi}T_3)\Leftarrow \pi_1(T_1\xrightarrow{\Xi_1}T_3)\rangle\!\rangle e_1'\right)\ \left(\langle\!\langle T_1\Leftarrow ?\rangle\!\rangle e_2'\right)\right) : T_3}$$

$$\text{C-GT-App-3}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : ?\qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}\pi_2}(\Xi)\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : \pi_2\rho_2\qquad \Phi = \Delta_{\mathbf{Tags}\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\left(\ \langle\!\langle \mathbf{Tags}(\pi_2\rho_2\xrightarrow{\Xi}?)\Leftarrow ?\rangle\!\rangle\ e_1'\right)\ e_2'\right) : T_3}$$

$$\text{C-GT-App-4}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma\vdash e_1\Rightarrow e_1' : ?\qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}\mathbf{Tags}}(\Xi)\\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi);\Gamma;\Sigma\vdash e_2\Rightarrow e_2' : ?\qquad \Phi = \Delta_{\mathbf{Tags}\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma\vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi,\left(\ \langle\!\langle \mathbf{Tags}(?\xrightarrow{\Xi}T_3)\Leftarrow ?\rangle\!\rangle\ e_1'\right)\ e_2'\right) : T_3}$$

Fig. 27. Extract of translation from source programs to the internal language for gradual type-and-effect systems. Complete system in Figure B 10 and Figure B 11.

Analogous assumptions are made for translation of assignment expressions in rules [C-GT-Asgn-1], [C-GT-Asgn-2], [C-GT-Asgn-3] and [C-GT-Asgn-4].

Choosing the right privilege set in application rules Rule [C-GT-App-2] introduces a cast $\langle T_2 \xrightarrow{\Xi} ? \Leftarrow ? \rangle$. In gradual typing, an application where e_1 has type ? requires a cast to ensure that e_1 is actually a function. While it seems that for this restriction *any* privilege set would be valid, like $\{¿\}$, this set is not sufficient. Using a privilege set $\{¿\}$ does not take into account the restriction arising from type-and-effect systems that a function that is applied cannot generate more side effects than those allowed in the context of application. To make this assumption explicit, the cast must restrict the privileges to the context Ξ of available privileges.

The following program demonstrates improper behavior if a cast to $\{¿\}$ is used instead of a cast to Ξ:

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

In this program, *effectful-argument* represents a properly typed function in scope that generates a `write` effect ($\Gamma(\textit{effectful-argument}) = \texttt{Unit} \xrightarrow{\{\texttt{write}\}} \texttt{Unit}$, for example). The translation should introduce enough runtime checks to make this program fail, since the context where f is applied does not allow side effects ($(f\ \texttt{unit}) :: \emptyset$). If rule [C-GT-App-2] used $\{¿\}$ instead of Ξ, then this program would not produce the required runtime error for using *effectful-argument*.

To enforce this invariant consistently, we use the same privilege set for the casts inserted by rules [C-GT-App-3] and [C-GT-App-4].

7.2.5 Type safety

In this section we prove that the combined gradual type-and-effect system is type safe and that the translation preserves typing.

Theorem 27 (*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 Φ such that $\Xi \vdash \Phi$ and for any store μ such that $\emptyset \mid \Sigma \vDash \mu$.

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

Theorem 28 (*Preservation*)
If $\Xi;\Gamma;\Sigma \vdash e : T$, and $\Phi \vdash e \mid \mu \to e' \mid \mu'$ for $\Xi \vdash \Phi$ 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
As in preservation Theorem 22, by structural induction over the typing derivation and the applicable evaluation rules. □

Theorem 29 (*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 rules. The tagset assumptions introduced ensure that **check** and **check** predicates always hold, and that **adjust** and **adjust** functions produce minimal privilege sets, which combined with effect subsumption ensures type preservation.
□

8 Related Work

In the realm of effect systems, the most closely related work is the generic framework of Marino & Millstein (2009), which is discussed extensively in this paper, because we build upon it to formulate gradual effect checking in a generic setting.

Rytz *et al.* (2012) 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 (Rytz *et al.*, 2013). 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 Section 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. Recently, Toro & Tanter (2015) extended our gradual effect checking approach to support lightweight effect polymorphism (Rytz *et al.*, 2012), combined with a DSL called EffScript to declaratively define and apply effect disciplines. The resulting system is implemented as a compiler plugin for Scala.

While there is a long history in the area of combining static and dynamic checking, the gradual typing approach of Siek & Taha (2006) 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 checking. Originally developed for functional languages, it has been extended in several directions, including structural objects (Siek & Taha, 2007) and generics (Ina & Igarashi, 2011). 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.* (2011) 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 & Clarke (2012) propose gradual ownership types. 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 & Flanagan (2011) 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. Fennell & Thiemann (2013) extend

the notion of security labels to state. However, their notion of gradual security is closer to quasi-static typing than to gradual typing, as unknown security information is statically treated as the top of the security lattice, requiring explicit downcasts in source programs.

Thiemann & Fennell (2014) propose gradual typing for annotated type systems. Their work is closer to the use of tags than effect annotations in the framework of Marino & Millstein (2009), as it focuses on annotations on values and on atomic types instead of function types, with the goal of gradualizing the annotation part of annotated type systems.

Extensions to contract systems for higher-order functions (Findler & Felleisen, 2002), such as computational contracts (Scholliers *et al.*, 2015) and temporal contracts (Disney *et al.*, 2011), 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.

None of the prior approaches to gradual checking relies on abstract interpretation to develop an account of uncertainty. Garcia *et al.* (2016) extends the idea of using abstract interpretation to systematically derive a dynamic semantics for gradual programs by proof reduction on gradual typing derivations. We have yet to investigate how this new framework could be used to derive a dynamic semantics for gradual effects.

9 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, parallel work by Toro & Tanter (2015), adapts and implements gradual effects in Scala.

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 also extended our system to support full gradual type-and-effect systems, which depend on gradual effects as an initial step, and explored tradeoffs in the operational semantics for gradual effect checking. In future work, we intend to investigate effects in the framework of Garcia *et al.* (2016), to identify any novel approach that may arise in the derivation of a dynamic semantics for gradual effect checking.

Bibliography

Abadi, Martín, Flanagan, Cormac, & Freund, Stephen N. (2006). Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and*

Systems, **28**(2), 207–255.

Abadi, Martín, Birrell, Andrew, Harris, Tim, & Isard, Michael. (2008). Semantics of Transactional Memory and Automatic Mutual Exclusion. *Pages 63–74 of: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*. ACM Press.

Bañados Schwerter, Felipe, Garcia, Ronald, & Tanter, Éric. (2014). A Theory of Gradual Effect Systems. *Pages 283–295 of: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014)*. ACM Press.

Benton, Nick, & Buchlovsky, Peter. (2007). Semantics of an Effect Analysis for Exceptions. *Pages 15–26 of: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI 2007)*. ACM Press.

Cousot, Patrick, & Cousot, Radhia. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of: Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. ACM Press.

Cousot, Patrick, & Cousot, Radhia. (1979). Systematic Design of Program Analysis Frameworks. *Pages 269–282 of: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1979)*. ACM Press.

Delaware, Benjamin, Keuchel, Steven, Schrijvers, Tom, & Oliveira, Bruno C.d.S. (2013). Modular Monadic Meta-theory. *Pages 319–330 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. ACM Press.

Disney, Tim, & Flanagan, Cormac. (2011). Gradual Information Flow Typing. *International workshop on scripts to programs*.

Disney, Tim, Flanagan, Cormac, & McCarthy, Jay. (2011). Temporal Higher-Order Contracts. *Pages 176–188 of: Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011)*. ACM Press.

Fennell, Luminous, & Thiemann, Peter. 2013 (June). Gradual security typing with references. *Pages 224–239 of: Computer security foundations symposium (csf), ieee*.

Findler, Robert Bruce, & Felleisen, Matthias. (2002). Contracts for Higher-Order Functions. *Pages 48–59 of: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*. ACM Press.

Garcia, Ronald, Tanter, Éric, Wolff, Roger, & Aldrich, Jonathan. (2014). Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, **36**(4), 12:1–12:44.

Garcia, Ronald, Clark, Alison M., & Tanter, Éric. (2016). Abstracting Gradual Typing. *Pages 429–442 of: Proceedings of the 43rd annual acm sigplan-sigact symposium on principles of programming languages (popl 2016)*. ACM Press.

Gifford, David K., & Lucassen, John M. (1986). Integrating Functional and Imperative Programming. *Pages 28–38 of: Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM Press.

Gordon, Colin S., Dietl, Werner, Ernst, Michael D., & Grossman, Dan. (2013). JavaUI: Effects for Controlling UI Object Access. *Pages 179–204 of: Proceedings of the 27th European Conference on Object-oriented Programming (ECOOP 2013)*. Springer-Verlag.

Gosling, James, Joy, Bill, Steele, Guy, & Bracha, Gilad. (2003). *The Java Language Specification, Third Edition*. Addison-Wesley.

Ina, Lintaro, & Igarashi, Atsushi. (2011). Gradual Typing for Generics. *Pages 609–624 of: Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2011)*. ACM Press.

Marino, Daniel, & Millstein, Todd. (2009). A Generic Type-and-Effect System. *Pages 39–50 of: Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation*.

Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press.

Rytz, Lukas, Odersky, Martin, & Haller, Philipp. (2012). Lightweight Polymorphic Effects. *Pages 258–282 of: Proceedings of the 26th European Conference on Object-oriented Programming (ECOOP 2012)*. Springer-Verlag.

Rytz, Lukas, Amin, Nada, & Odersky, Martin. (2013). A Flow-Insensitive, Modular Effect System for Purity. *Pages 4:1–4:7 of: Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*. ACM Press.

Scholliers, Christophe, Tanter, Éric, & De Meuter, Wolfgang. (2015). Computational Contracts. *Science of computer programming*, **98**(3), 360–375.

Sergey, Ilya, & Clarke, Dave. (2012). Gradual Ownership Types. *Pages 579–599 of: Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012)*. Springer-Verlag.

Siek, Jeremy, & Taha, Walid. (2007). Gradual Typing for Objects. *Pages 2–27 of: Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007)*. Springer-Verlag.

Siek, Jeremy G., & Taha, Walid. 2006 (Sept.). Gradual Typing for Functional Languages. *Pages 81–92 of: Proceedings of the Scheme and Functional Programming Workshop*.

Takikawa, Asumu, Strickland, T. Stephen, Dimoulas, Christos, Tobin-Hochstadt, Sam, & Felleisen, Matthias. (2012). Gradual Typing for First-Class Classes. *Pages 793–810 of: Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*. ACM Press.

Tang, Yan Mei, & Jouvelot, Pierre. (1995). Effect Systems with Subtyping. *Pages 45–53 of: Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM 1995)*. ACM Press.

Thiemann, Peter, & Fennell, Luminous. (2014). Gradual Typing for Annotated Type Systems. *Pages 47–66 of: Programming languages and systems*. Lecture Notes in Computer Science, vol. 8410. Springer Berlin Heidelberg.

Toro, Matías, & Tanter, Éric. (2015). Customizable Gradual Polymorphic Effects for Scala. *Submission to the 30th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2015)*.

Wolff, Roger, Garcia, Ronald, Tanter, Éric, & Aldrich, Jonathan. (2011). Gradual Typestate. *Pages 459–483 of: Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011)*. Springer-Verlag.

Wright, Andrew K., & Felleisen, Matthias. (1994). A Syntactic Approach to Type Soundness. *Journal of Information and Computation*, **115**(1), 38–94.

A Detailed definitions of Section 6

In this appendix we include complete descriptions of some relations used to describe the conservative semantics.

$$\boxed{\Xi;\Gamma;\Sigma \vdash e \Rrightarrow e : T} \qquad \text{CT-Fn}\ \frac{\Xi_1;\Gamma,x : T_1;\Sigma \vdash e \Rrightarrow e' : T_2}{\Xi;\Gamma;\Sigma \vdash (\lambda x : T_1 . e)_\varepsilon \Rrightarrow (\lambda x : T_1 . e')_\varepsilon : \{\varepsilon\}T_1 \xrightarrow{\Xi_1} T_2}$$

$$\text{CT-Unit}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \mathtt{unit}_\varepsilon \Rrightarrow \mathtt{unit}_\varepsilon : \{\varepsilon\}\mathtt{Unit}}$$

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

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

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

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

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

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

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

$$\text{CT-Deref}\ \frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e \Rrightarrow e' : \pi\mathtt{Ref}\ T \\ \textit{strict-check}_{!\pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !e \Rrightarrow !e' : T}$$

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

Fig. A 1. Complete type-directed tag addition. It introduces the tag approximation of the generic language explicitly, to be used on adjust contexts for evaluation.

$$\boxed{\Phi \vdash e \mid \mu \rightsquigarrow e \mid \mu} \qquad \text{CE-Ref}\ \frac{l \notin \mathrm{dom}(\mu) \qquad \mathbf{check}_{\mathtt{ref}\ \{\varepsilon_1\}}(\Phi)}{\Phi \vdash (\mathtt{ref}\ w_{\varepsilon_1})_{\varepsilon_2} \mid \mu \rightsquigarrow l \mid \mu[l \mapsto w]}$$

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

$$\text{CE-Asgn}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash \left(l_{\varepsilon_1} := w_{\varepsilon_2}\right)_{(\varepsilon,\pi)} \mid \mu \rightsquigarrow \mathtt{unit}_{\varepsilon} \mid \mu[l \mapsto w_{\varepsilon_2}]}$$

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

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

$$\text{CE-App}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash \left((\lambda x :\ T_1\ .\ e)_{\varepsilon_1}\ w_{\varepsilon_2}\right)_{\pi} \mid \mu \rightsquigarrow [{}^{w_{\varepsilon_2}}/_{x}]\, e \mid \mu}$$

$$\text{CE-Cast-Frame}\ \frac{\Phi \vdash e \mid \mu \rightsquigarrow e' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1 \rangle e \mid \mu \rightsquigarrow \langle T_2 \Leftarrow T_1 \rangle e' \mid \mu'}$$

$$\text{CE-Has-T}\ \frac{\Phi' \subseteq \Phi \qquad \Phi \vdash e \mid \mu \rightsquigarrow e' \mid \mu'}{\Phi \vdash \mathtt{has}\ \Phi'\ e \mid \mu \rightsquigarrow \mathtt{has}\ \Phi'\ e' \mid \mu'}$$

$$\text{CE-Has-V}\ \frac{}{\Phi \vdash \mathtt{has}\ \Phi'\ w \mid \mu \rightsquigarrow w \mid \mu} \qquad \text{CE-Has-F}\ \frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \mathtt{has}\ \Phi'\ e \mid \mu \rightsquigarrow \mathsf{Error} \mid \mu}$$

$$\text{CE-Rst}\ \frac{\Phi'' = \max\{\Phi' \in \gamma(\Xi) \mid \Phi' \subseteq \Phi\} \qquad \Phi'' \vdash e \mid \mu \rightsquigarrow e' \mid \mu'}{\Phi \vdash \mathtt{restrict}\ \Xi\ e \mid \mu \rightsquigarrow \mathtt{restrict}\ \Xi\ e' \mid \mu'}$$

$$\text{CE-Rst-V}\ \frac{}{\Phi \vdash \mathtt{restrict}\ \Xi\ w \mid \mu \rightsquigarrow w \mid \mu}$$

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

$$\text{CE-Cast-Fn}\ \frac{\pi_1 \subseteq \pi_2}{\begin{array}{l}\Phi \vdash \langle \pi_2 T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow \pi_1 T_{11} \xrightarrow{\Xi_1} T_{12} \rangle\, (\lambda x :\ T_{11}\ .\ e) \mid \mu \rightsquigarrow \\ (\lambda x :\ T_{21}\ .\ \langle T_{22} \Leftarrow T_{12} \rangle \mathtt{restrict}\ \Xi_2\ \mathtt{has}\ (|\Xi_1| \setminus |\Xi_2|)\ \ [{}^{(\langle T_{11} \Leftarrow T_{21} \rangle x)}/_{x}]\, e) \mid \mu\end{array}}$$

Fig. A 2. The complete conservative semantics, using special frame translation function A', that maps annotated evaluation frames to adjust contexts.

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

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

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

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

$$\text{CIT-App}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_0\left(T_1 \xrightarrow{\Xi_1} T_3\right) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1\,\pi_2}(\Xi) \qquad \pi_0\left(T_1 \xrightarrow{\Xi_1} T_3\right) <: \pi_1\left((\pi_2\rho_2) \xrightarrow{\Xi} T_3\right)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1\ e_2)_{\pi_1} : T_3}$$

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

$$\text{CIT-Rst}\frac{\Xi_1;\Gamma;\Sigma \vdash e : T \qquad \Xi_1 \leq \Xi}{\Xi;\Gamma;\Sigma \vdash \mathtt{restrict}\ \Xi_1\ e : T} \qquad \text{CIT-Error}\frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error} : T}$$

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

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

$$\text{CIT-Asgn}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_0\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \textbf{\textit{strict-check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 <: T_1 \qquad \pi_0 \subseteq \pi_1\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_{(\varepsilon,\pi_1)} : \{\varepsilon\}\mathtt{Unit}}$$

Fig. A 3. Complete type system for the conservative language

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

$$\text{G-Fn}\frac{\Gamma, x : T_1;\Sigma \vdash e : T_2}{\Gamma;\Sigma \vdash (\lambda x : T_1 \,.\, e)_\varepsilon : \{\varepsilon\}T_1 \to T_2} \qquad \text{G-Unit}\frac{}{\Gamma;\Sigma \vdash \mathtt{unit}_\varepsilon : \{\varepsilon\}\mathtt{Unit}}$$

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

$$\text{G-App}\frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 : T \\ \Gamma;\Sigma \vdash e_2 : T_2 \\ T \lesssim \pi(T_2 \to T_3)\end{array}}{\Gamma;\Sigma \vdash e_1\ e_2 : T_3} \qquad \text{G-Ref}\frac{\Gamma;\Sigma \vdash e : T}{\Gamma;\Sigma \vdash (\mathtt{ref}\ e)_\varepsilon : \{\varepsilon\}\mathtt{Ref}\ T'}$$

$$\text{G-Deref}\frac{\begin{array}{c}\Gamma;\Sigma \vdash e : T_1 \\ T_1 \sim \pi\mathtt{Ref}\ T\end{array}}{\Gamma;\Sigma \vdash !e : T} \qquad \text{G-Asgn}\frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 : T \qquad T \sim \pi\mathtt{Ref}\ T_1 \\ \Gamma;\Sigma \vdash e_2 : T_2 \\ T_2 \lesssim T_1\end{array}}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}}$$

Fig. B 1. Complete typing rules for the source language for gradual typing with tags.

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

$$\text{GT-Fn}\frac{\Gamma, x : T_1;\Sigma \vdash e : T_2}{\Gamma;\Sigma \vdash \lambda x : T_1 \,.\, e : T_1 \to T_2} \qquad \text{GT-Unit}\frac{}{\Gamma;\Sigma \vdash \mathtt{unit}_\varepsilon : \{\varepsilon\}\mathtt{Unit}}$$

$$\text{GT-App}\frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 : \pi(T_1 \to T_3) \\ \Gamma;\Sigma \vdash e_2 : T_2 \\ \pi(T_1 \to T_3) <: \pi(T_2 \to T_3)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3} \qquad \text{GT-Loc}\frac{\Sigma(l) = T}{\Gamma;\Sigma \vdash l_\varepsilon : \{\varepsilon\}\mathtt{Ref}\ T}$$

$$\text{GT-Var}\frac{\Gamma(x) = T}{\Gamma;\Sigma \vdash x : T} \qquad \text{GT-Cast}\frac{\Gamma;\Sigma \vdash e : T_0 \qquad T_0 <: T_1}{\Gamma;\Sigma \vdash \langle T_2 \Leftarrow T_1 \rangle e : T_2}$$

$$\text{GT-Error}\frac{}{\Gamma;\Sigma \vdash \mathsf{Error} : T} \qquad \text{GT-Ref}\frac{\Gamma;\Sigma \vdash e : T_0 \qquad T_0 <: T_1}{\Gamma;\Sigma \vdash (\mathtt{ref}\ e)_\varepsilon : \{\varepsilon\}\mathtt{Ref}\ T_1}$$

$$\text{GT-Deref}\frac{\Gamma;\Sigma \vdash e : \pi\mathtt{Ref}\ T}{\Gamma;\Sigma \vdash !e : T}$$

$$\text{GT-Asgn}\frac{\Gamma;\Sigma \vdash e_1 : \pi\mathtt{Ref}\ T_1 \qquad \Gamma;\Sigma \vdash e_2 : T_2 \qquad T_2 <: T_1}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\mathtt{Unit}}$$

Fig. B 2. Complete typing rules for the internal language for gradual typing with tags

B Detailed definitions of Section 7

$\boxed{e \mid \mu \to e \mid \mu}$

$$\text{GE-Ref}\ \frac{l \notin \mathrm{dom}(\mu)}{(\texttt{ref}\ v)_{\ \varepsilon} \mid \mu \to l_{\ \varepsilon} \mid \mu[l \mapsto v]}$$

$$\text{GE-Asgn}\ \frac{}{\left(l_{\ \varepsilon_1} := v\right)_{\ \varepsilon} \mid \mu \to \texttt{unit}_{\ \varepsilon} \mid \mu[l \mapsto v]} \qquad \text{GE-Deref}\ \frac{\mu(l) = v}{!l_{\ \varepsilon} \mid \mu \to v \mid \mu}$$

$$\text{GE-Frame}\ \frac{e \mid \mu \to e' \mid \mu'}{f[e] \mid \mu \to f[e'] \mid \mu'} \qquad \text{GE-Error}\ \frac{}{g[\mathsf{Error}] \mid \mu \to \mathsf{Error} \mid \mu}$$

$$\text{GE-App}\ \frac{}{(\lambda x :\ T_1\ .\ e)_{\ \varepsilon_1}\ v \mid \mu \to [v/x]\, e \mid \mu} \qquad \text{GE-Cast-Frame}\ \frac{e \mid \mu \to e' \mid \mu'}{\langle T_2 \Leftarrow T_1 \rangle e \mid \mu \to \langle T_2 \Leftarrow T_1 \rangle e' \mid \mu'}$$

$$\text{GE-Cast-Id}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\langle \pi_2 \rho \Leftarrow \pi_1 \rho \rangle w_\varepsilon \mid \mu \to w_\varepsilon \mid \mu} \qquad \text{GE-Cast-Merge}\ \frac{}{\langle T_2 \Leftarrow ? \rangle \langle ? \Leftarrow T_1 \rangle v \mid \mu \to \langle T_2 \Leftarrow T_1 \rangle v \mid \mu}$$

$$\text{GE-Cast-Dyn}\ \frac{}{\langle ? \Leftarrow ? \rangle v \mid \mu \to v \mid \mu} \qquad \text{GE-Cast-Bad}\ \frac{T_1 \not\precsim T_2}{\langle T_2 \Leftarrow T_1 \rangle v \mid \mu \to \mathsf{Error} \mid \mu}$$

$$\text{GE-Cast-Fn}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\langle \pi_2 T_{21} \to T_{22} \Leftarrow \pi_1 T_{11} \to T_{12} \rangle\, (\lambda x :\ T_{01}\ .\ e)_\varepsilon \mid \mu \to (\lambda x :\ T_{21}\ .\ \langle T_{22} \Leftarrow T_{12} \rangle\, [(\langle T_{11} \Leftarrow T_{21} \rangle x)/x]\, e)_\varepsilon \mid \mu}$$

Fig. B 3. Complete small-step semantics of the internal language for gradual typing with tags

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

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

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

$$\text{C-Loc}\ \frac{\Sigma(l) = T}{\Gamma;\Sigma \vdash l_\varepsilon \Rightarrow l_\varepsilon : \{\varepsilon\}\texttt{Ref}\ T} \qquad \text{C-App-1}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \to T_3) \\ \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \\ e_1'' = (\langle\!\langle \pi_1(\pi_2\rho_2 \to T_3) \Leftarrow \pi_1(T_1 \to T_3) \rangle\!\rangle e_1') \\ \pi_1(T_1 \to T_3) \lesssim \pi_1(\pi_2\rho_2 \to T_3)\end{array}}{\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow e_1''\ e_2' : T_3}$$

$$\text{C-App-2}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \to T_3) \\ \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \\ e_1'' = (\langle\!\langle \pi_1(T_1 \to T_3) \Leftarrow \pi_1(T_1 \to T_3) \rangle\!\rangle e_1')\end{array}}{\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow e_1''\ \left((\langle\!\langle T_1 \Leftarrow ? \rangle\!\rangle e_2') \right) : T_3}$$

$$\text{C-App-3}\ \frac{\begin{array}{c}\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \\ \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : T \\ e_1'' = (\ \langle\!\langle \mathbf{Tags}(T \to ?) \Leftarrow ? \rangle\!\rangle\ e_1')\end{array}}{\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow e_1''\ e_2' : ?}$$

Fig. B 4. Complete translation of source programs to the internal language for gradual typing with tags, Part I

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

CG-Ref-1
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : \pi\rho}{\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon \Rightarrow \left(\texttt{ref}\ e'\right)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho}$$

CG-Ref-2
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : ?}{\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon \Rightarrow \left(\texttt{ref}\ e'\right)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ ?}$$

CG-Deref-1
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : \pi\texttt{Ref}\ T}{\Gamma;\Sigma \vdash !e \Rightarrow !e' : T}$$

CG-Deref-2
$$\frac{\Gamma;\Sigma \vdash e \Rightarrow e' : ?}{\Gamma;\Sigma \vdash !e \Rightarrow\ !\langle \mathbf{Tags}(\texttt{Ref}\ ?) \Leftarrow ? \rangle e' : T}$$

CG-Asgn-1
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1\texttt{Ref}\ T_1 \qquad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \qquad \pi_2\rho_2 \lesssim T_1}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \left(e_1' := e_2'\right)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

CG-Asgn-2
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1\texttt{Ref}\ T_1 \qquad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ?}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \left(e_1' :=\ \langle T_1 \Leftarrow ? \rangle e_2'\right)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

CG-Asgn-3
$$\frac{\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \qquad \Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : T}{\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \left(\ (\langle \mathbf{Tags}(\texttt{Ref}\ T) \Leftarrow ? \rangle e_1')\ := e_2'\right)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Fig. B 5. Complete translation of source programs to the internal language for gradual typing with tags, Part II

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

$$\varepsilon \in \mathbf{Tags} \,.\, \pi \in \mathscr{P}(\mathbf{Tags})$$

w	$::=$	$\mathtt{unit} \mid \lambda x : T \,.\, e \mid l$	Prevalues
v	$::=$	w_ε	Values
e	$::=$	$x \mid v \mid e\ e \mid e :: \Xi$	Terms
		$\mid (\mathtt{ref}\ e)_\varepsilon \mid\ !e \mid (e := e)_\varepsilon$	
T	$::=$	$\pi\ \rho \mid\ ?$	Types
ρ	$::=$	$\mathtt{Unit} \mid T \xrightarrow{\Xi} T \mid \mathtt{Ref}\ T$	PreTypes
A	$::=$	$\downarrow\uparrow\ \mid \pi\downarrow \mid \mathtt{ref}\ \downarrow\ \mid\ !\downarrow$	Adjust Contexts
		$\mid\ \downarrow{:=}\uparrow\ \mid\ \pi :=\downarrow$	
C	$::=$	$\pi\ \pi \mid \mathtt{ref}\pi\ \mid\ !\pi\ \mid\ \pi := \pi$	Check Contexts

Fig. B 6. Complete syntax of the source language for gradual type-and-effect systems

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

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

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

$$\text{GT-Var}\frac{\Gamma(x) = T}{\Xi;\Gamma;\Sigma \vdash x : T} \qquad \text{GT-App}\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : T \\ \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : T_2 \\ T_2 \sim \pi_2\rho_2 \qquad T \lesssim \pi_1(T_2 \xrightarrow{\Xi} T_3) \qquad \widetilde{\textbf{check}}_{\pi_1\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3}$$

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

$$\text{GT-Deref}\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e : T_1 \\ T_1 \sim \pi\texttt{Ref}\ T \qquad \widetilde{\textbf{check}}_{!\pi}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash !e : T}$$

$$\text{GT-Asgn}\frac{\begin{array}{c}\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : T \qquad T \sim \pi_1\texttt{Ref}\ T_1 \\ \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : T_2 \qquad T_2 \sim \pi_2\rho_2 \\ \widetilde{\textbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad T_2 \lesssim T_1\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Fig. B 7. Complete typing rules for the source language for gradual type-and-effect systems

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

$$\text{GIT-App}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_1\left(T_1 \xrightarrow{\Xi_1} T_3\right) \quad \widetilde{\textbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \quad \textit{strict-check}_{\pi_1\pi_2}(\Xi) \quad \pi_1 T_1 \xrightarrow{\Xi_1} T_3 <: \pi_1\pi_2\rho_2 \xrightarrow{\Xi} T_3}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 : T_3}$$

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

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

$$\text{GIT-Error}\ \frac{}{\Xi;\Gamma;\Sigma \vdash \mathsf{Error} : T} \qquad \text{GIT-Rst}\ \frac{\Xi_1;\Gamma;\Sigma \vdash e : T \quad \Xi_1 \leq \Xi}{\Xi;\Gamma;\Sigma \vdash \texttt{restrict}\ \Xi_1\ e : T}$$

$$\text{GIT-Ref}\ \frac{\widetilde{\textbf{adjust}}_{\texttt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\rho \quad \textit{strict-check}_{\texttt{ref}\ \pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\texttt{ref}\ e)_\varepsilon : \{\varepsilon\}\texttt{Ref}\ \pi\rho} \qquad \text{GIT-Deref}\ \frac{\widetilde{\textbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e : \pi\texttt{Ref}\ T \quad \textit{strict-check}_{!\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash !e : T}$$

$$\text{GIT-Asgn}\ \frac{\widetilde{\textbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 : \pi_1\texttt{Ref}\ T_1 \quad \widetilde{\textbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \quad \textit{strict-check}_{\pi_1:=\pi_2}(\Xi) \quad \pi_2\rho_2 <: T_1}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\texttt{Unit}}$$

Fig. B 8. Complete typing rules for the internal language for gradual type-and-effect systems

$\boxed{\Phi \vdash e \mid \mu \to e \mid \mu}$

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

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

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

$$\text{GTE-Error}\ \frac{}{\Phi \vdash g[\mathsf{Error}] \mid \mu \to \mathsf{Error} \mid \mu} \qquad \text{GTE-Has-V}\ \frac{}{\Phi \vdash \mathtt{has}\ \Phi'\ v \mid \mu \to v \mid \mu}$$

$$\text{GTE-Has-T}\ \frac{\Phi' \subseteq \Phi \qquad \Phi \vdash e \mid \mu \to e' \mid \mu'}{\Phi \vdash \mathtt{has}\ \Phi'\ e \mid \mu \to \mathtt{has}\ \Phi'\ e' \mid \mu'}$$

$$\text{GTE-Rst-V}\ \frac{}{\Phi \vdash \mathtt{restrict}\ \Xi\ v \mid \mu \to v \mid \mu} \qquad \text{GTE-Has-F}\ \frac{\Phi' \not\subseteq \Phi}{\Phi \vdash \mathtt{has}\ \Phi'\ e \mid \mu \to \mathsf{Error} \mid \mu}$$

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

$$\text{GTE-App}\ \frac{\mathbf{check}_{\{\varepsilon_1\}\{\varepsilon_2\}}(\Phi)}{\Phi \vdash (\lambda x :\ T_1\ .\ e)_{\varepsilon_1}\ v_{\varepsilon_2} \mid \mu \to [v_{\varepsilon_2}/x]e \mid \mu} \qquad \text{GTE-Cast-Frame}\ \frac{\Phi \vdash e \mid \mu \to e' \mid \mu'}{\Phi \vdash \langle T_2 \Leftarrow T_1 \rangle e \mid \mu \to \langle T_2 \Leftarrow T_1 \rangle e' \mid \mu'}$$

$$\text{GTE-Cast-Id}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\Phi \vdash \langle \pi_2\rho \Leftarrow \pi_1\rho \rangle w_{\varepsilon} \mid \mu \to w_{\varepsilon} \mid \mu} \qquad \text{GTE-Cast-Merge}\ \frac{}{\Phi \vdash \langle T_2 \Leftarrow ? \rangle \langle ? \Leftarrow T_1 \rangle v \mid \mu \to \langle T_2 \Leftarrow T_1 \rangle v \mid \mu}$$

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

$$\text{GTE-Cast-Fn}\ \frac{\varepsilon \in \pi_1 \qquad \pi_1 \subseteq \pi_2}{\begin{array}{l}\Phi \vdash \langle \pi_2 T_{21} \xrightarrow{\Xi_2} T_{22} \Leftarrow \pi_1 T_{11} \xrightarrow{\Xi_1} T_{12} \rangle\, (\lambda x :\ T_{01}\ .\ e)_{\varepsilon} \mid \mu \to \\ (\lambda x :\ T_{21}\ .\ \langle T_{22} \Leftarrow T_{12} \rangle \mathtt{restrict}\ \Xi_2\ \mathtt{has}\ (|\Xi_1| \setminus |\Xi_2|)\ \ [(\langle T_{01} \Leftarrow T_{21} \rangle x)/x]e)_{\varepsilon} \mid \mu\end{array}}$$

Fig. B 9. Complete small-step semantics of the internal language for gradual type-and-effect systems

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

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

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

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

$$\text{C-GT-App-1}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \\ e''_1 = (\langle\!\langle \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \rangle\!\rangle e_1') \\ \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \lesssim \pi_1(\pi_2\rho_2 \xrightarrow{\Xi} T_3) \\ \widetilde{\mathbf{check}}_{\pi_1\pi_2}(\Xi) \qquad \Phi = \Delta_{\pi_1\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, e''_1\ e'_2\right) : T_3}$$

$$\text{C-GT-App-2}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \\ \widetilde{\mathbf{adjust}}_{\pi_1\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \\ e''_1 = (\langle\!\langle \pi_1(T_1 \xrightarrow{\Xi} T_3) \Leftarrow \pi_1(T_1 \xrightarrow{\Xi_1} T_3) \rangle\!\rangle e_1') \\ \Xi_1 \sqsubseteq_\sim \Xi \qquad \widetilde{\mathbf{check}}_{\pi_1\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\pi_1\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, e''_1\ (\langle\!\langle T_1 \Leftarrow ? \rangle\!\rangle e'_2)\right) : T_3}$$

$$\text{C-GT-App-3}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \\ e''_1 = (\ \langle\!\langle \mathbf{Tags}(\pi_2\rho_2 \xrightarrow{\Xi} ?) \Leftarrow ? \rangle\!\rangle\ e_1') \\ \widetilde{\mathbf{check}}_{\mathbf{Tags}\pi_2}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, e''_1\ e'_2\right) : T_3}$$

$$\text{C-GT-App-4}\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \\ e''_1 = (\ \langle\!\langle \mathbf{Tags}(? \xrightarrow{\Xi} T_3) \Leftarrow ? \rangle\!\rangle\ e_1') \\ \widetilde{\mathbf{check}}_{\mathbf{Tags}\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash e_1\ e_2 \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, e''_1\ e'_2\right) : T_3}$$

$$\text{C-GT-Eff}\frac{\Xi_1;\Gamma;\Sigma \vdash e \Rightarrow e' : T \qquad \Xi_1 \sqsubseteq_\sim \Xi \qquad \Phi = (|\Xi_1| \setminus |\Xi|)}{\Xi;\Gamma;\Sigma \vdash (e :: \Xi_1) \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \mathtt{restrict}\ \Xi_1\ e'\right) : T}$$

Fig. B 10. Complete translation of source programs to the internal language for gradual type-and-effect systems, part I

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

C-GT-Ref-1
$$\frac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e \Rightarrow e' : \pi\rho \qquad \widetilde{\mathbf{check}}_{\mathtt{ref}\,\pi}(\Xi) \qquad \Phi = \Delta_{\mathtt{ref}\,\pi}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\mathtt{ref}\ e)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(\mathtt{ref}\ e'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Ref}\ \pi\rho}$$

C-GT-Ref-2
$$\frac{\widetilde{\mathbf{adjust}}_{\mathtt{ref}\downarrow}(\Xi);\Gamma;\Sigma \vdash e \Rightarrow e' : ? \qquad \widetilde{\mathbf{check}}_{\mathtt{ref\ Tags}}(\Xi) \qquad \Phi = \Delta_{\mathtt{ref\ Tags}}(\Xi)}{\Xi;\Gamma;\Sigma \vdash (\mathtt{ref}\ e)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(\mathtt{ref}\ e'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Ref}\ ?}$$

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

C-GT-Deref-2
$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{!\downarrow}(\Xi);\Gamma;\Sigma \vdash e \Rightarrow e' : ? \\ \widetilde{\mathbf{check}}_{!\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{!\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash\, !e \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, !\langle \mathbf{Tags}(\mathtt{Ref}\ ?) \Leftarrow ? \rangle e'\right) : T}$$

C-GT-Asgn-1
$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \qquad \widetilde{\mathbf{check}}_{\pi_1:=\pi_2}(\Xi) \qquad \pi_2\rho_2 \lesssim T_1 \qquad \Phi = \Delta_{\pi_1:=\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(e_1' := e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

C-GT-Asgn-2
$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : \pi_1\mathtt{Ref}\ T_1 \\ \widetilde{\mathbf{adjust}}_{\pi_1:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \qquad \widetilde{\mathbf{check}}_{\pi_1:=\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\pi_1:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(e_1' := \langle T_1 \Leftarrow ? \rangle e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

C-GT-Asgn-3
$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : \pi_2\rho_2 \qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\pi_2}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}:=\pi_2}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(\left(\langle \mathbf{Tags}(\mathtt{Ref}\ ?) \Leftarrow ? \rangle e_1'\right) := e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

C-GT-Asgn-4
$$\frac{\begin{array}{c}\widetilde{\mathbf{adjust}}_{\downarrow:=\uparrow}(\Xi);\Gamma;\Sigma \vdash e_1 \Rightarrow e_1' : ? \\ \widetilde{\mathbf{adjust}}_{\mathbf{Tags}:=\downarrow}(\Xi);\Gamma;\Sigma \vdash e_2 \Rightarrow e_2' : ? \qquad \widetilde{\mathbf{check}}_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi) \qquad \Phi = \Delta_{\mathbf{Tags}:=\mathbf{Tags}}(\Xi)\end{array}}{\Xi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon \Rightarrow \mathit{insert\text{-}has?}\left(\Phi, \left(\left(\langle \mathbf{Tags}(\mathtt{Ref}\ ?) \Leftarrow ? \rangle e_1'\right) := e_2'\right)_\varepsilon\right) : \{\varepsilon\}\mathtt{Unit}}$$

Fig. B 11. Complete translation of source programs to the internal language for gradual type-and-effect systems, part II