A Generic Type-and-Effect System

Daniel Marino Todd Millstein

Computer Science Department
University of California, Los Angeles
{dlmarino,todd}@cs.ucla.edu

Abstract

Type-and-effect systems are a natural approach for statically reasoning about a program's execution. They have been used to track a variety of computational effects, for example memory manipulation, exceptions, and locking. However, each type-and-effect system is typically implemented as its own monolithic type system that hard-codes a particular syntax of effects along with particular rules to track and control those effects.

We present a generic type-and-effect system, which is parameterized by the syntax of effects to track and by two functions that together specify the *effect discipline* to be statically enforced. We describe how a standard form of type soundness is ensured by requiring these two functions to obey a few natural monotonicity requirements. We demonstrate that several effect systems from the literature can be viewed as instantiations of our generic type system. Finally, we describe the implementation of our type-and-effect system and mechanically checked type soundness proof in the Twelf proof assistant.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics, Syntax; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Theory

Keywords type-and-effect systems

1. Introduction

Type-and-effect systems (or simply *effect systems*) (Gifford and Lucassen 1986) are an approach for augmenting static type systems to reason about and control a program's computational effects. Such systems were originally developed to statically track the manipulation of dynamically allocated memory. However, many other kinds of computational effects can be controlled via an effect system. For example, Java's type system for checked exceptions (Gosling et al. 2005) can be formulated as an effect system. Other applications abound in the research literature. Effect systems have recently been used to enforce a locking discipline to prevent race conditions in Java (Flanagan and Freund 2000; Abadi et al. 2006) and to ensure strong atomicity for a transactional memory system (Abadi et al. 2008).

TLDI'09, January 24, 2009, Savannah, Georgia, USA.

In each of the above examples, a particular notion of effects and an associated discipline for controlling those effects is built into the language's type system. While it is intuitively clear that different effect systems have much in common, it is not obvious how to make these commonalities precise. Indeed, others have explicitly posed the formulation of a "general theory of effects" as an open problem, with the aim of "avoiding the need to create a new effect system for each new effect" (Wadler and Thiemann 2003). Such a theory would make it much easier to understand the relationships among effect systems and to experiment with new effect systems.

In this paper we take the first steps toward such a theory. We present a generic type-and-effect system with the following technical contributions:

- We describe a uniform and general approach to instrumenting a language's type system with "hooks" that can be used to track and control a set of computational effects. A particular effect system is instantiated from our generic effect system by providing a syntax of effects as well as definitions of the hooks. The hooks employ a style of representing effects dually as *privileges* (or *capabilities* (Walker et al. 2000)), which is commonly used to formalize effect systems that enforce a programming discipline (e.g., (Abadi et al. 2006, 2008; Neamtiu et al. 2008)).
- A basic notion of type soundness for any type-and-effect system requires static effect checking to conservatively approximate the checking done by a dynamic semantics that is instrumented with effects (e.g., (Talpin and Jouvelot 1992; Wadler and Thiemann 2003; Neamtiu et al. 2008)). We show that it is sufficient to impose a natural set of *monotonicity* requirements on the externally provided privilege discipline in order to guarantee this form of type soundness. The effect system designer must separately establish that this notion of type soundness guarantees the program behavior that the system is intended to ensure.
- We have formalized our generic effect system in the context of a polymorphic lambda calculus with mutable references. We have implemented this language, the generic effect system, and a mechanically verified type soundness theorem in the Twelf proof assistant (Pfenning and Schürmann 1999).
- We demonstrate that several effect systems from the literature can be viewed as instantiations of our generic effect system. These systems include the original effect system for memory manipulation (Gifford and Lucassen 1986), an effect system for tracking yields in a cooperative multitasking system (Fischer et al. 2007; Isard and Birrell 2007), and an effect system that ensures strong atomicity for software transactional memory (Abadi et al. 2008). Further, these effect systems all satisfy our monotonicity requirements that ensure type soundness.

Aside from its theoretical interest, we believe our work could serve as the foundation for a practical framework that allows programmers to easily and reliably augment their language's static

$$\frac{\Phi \cup \texttt{canThrow}; \Gamma; \Sigma \vdash e_1 : \tau \qquad \Phi; \Gamma; \Sigma \vdash e_2 : \mathit{ExnType} \to \tau}{\Phi; \Gamma; \Sigma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : \tau} \text{ (T-TRY)}$$

$$\frac{\Phi; \Gamma; \Sigma \vdash e : \mathit{ExnType} \qquad \texttt{canThrow} \in \Phi}{\Phi; \Gamma; \Sigma \vdash \texttt{throw}\ e : \tau} \text{ (T-THROW)}$$

Figure 1. Part of a type system that enforces checked exceptions. Φ is the set of held privileges. Γ and Σ are the type environment and store typing as usual.

type system to track new kinds of effects and to control these effects with programmer-specified disciplines. Such a framework would do for effects what systems like CQUAL (Foster et al. 1999, 2006) and Clarity (Chin et al. 2005) do for type qualifiers. In addition to the benefits for programmers, a programmer-definable effect system would be very useful for type systems researchers.

The rest of the paper is structured as follows. Section 2 describes our approach informally, and Section 3 presents our formal type system. Section 4 illustrates several existing effect systems that fit our model. Section 5 formalizes the dynamic semantics of our language, and Section 6 discusses our type soundness result. Section 7 details the implementation of our language, type system, and type soundness proof in Twelf. Section 8 compares with related work, and Section 9 concludes with a discussion of future work.

2. Overview

In this section, we give an informal overview of our generic effect system. We first show how a few standard effect systems can be described intuitively as granting and checking privileges. We then illustrate how we can abstract the specific privilege discipline being enforced and instrument the static semantics of any language with privilege checking in a generic way. Finally, we discuss how to achieve a standard notion of type soundness for the resulting type system.

2.1 Effect Systems as Privilege Checking

Consider an effect system that statically ensures that exceptions are eventually caught, as in Java (Gosling et al. 2005). Such a system enforces the discipline that a program may only raise an exception within the dynamic lifetime of a `try` expression. If we consider the ability to throw an exception as a privilege (assume for simplicity that there is a single kind of exception), then it is natural to describe this effect system as *granting* the privilege to throw an exception before typechecking the body of a `try` expression and *checking* that this privilege is held when typechecking a `throw` expression.

Figure 1 formalizes the discipline for exceptions described above. The typing judgment tracks a set of held privileges Φ, and we use a privilege `canThrow` to designate the capability to throw an exception. Rule T-TRY *grants* the `canThrow` privilege in the body of the `try` by adding this privilege to Φ before typechecking e_1. Rule T-THROW *checks* that the `canThrow` privilege is in Φ in order for a `throw` expression to typecheck. These two actions form the core of the effect system; the other typing rules simply ensure that this privilege discipline is enforced recursively on all subexpressions.

Table 1 describes how other standard effect systems can similarly be viewed as granting and checking privileges. The formulation of effects as privileges is natural when the intent is to enforce a particular discipline on a program rather than to simply determine the set of effects that may occur in the program. Each of these effect systems can be formalized as its own extension to a base type system, as we have shown with checked exceptions.

As in these examples, we formalize our generic effect system as granting and checking privileges, tracking a set Φ of held privileges during typechecking. However, the system is parameterized both by the syntax of privileges to track and the discipline for granting and checking privileges. Particular choices for these parameters yield a type system equivalent to our exception checker in Figure 1, but many other effect systems can be derived by making different choices.

The syntax of privileges is specified simply as a set of atomic constants. For example, to define our exception checker one would specify a single privilege `canThrow`, while a memory checker might employ privileges `read`, `write`, and `alloc`. For greater expressiveness, privileges can optionally refer to *tags*, which provide information about program values. We augment the host language to allow tagging of expressions that construct values, and the type system tracks the set of possible tags (or *tagset*) of each expression's value. We use ε to range over a set of globally defined tags and π to range over tagsets. For example, `read`(ε) could be used to denote the privilege to read any memory location tagged with ε.

The privilege discipline is specified by providing the definitions of two functions, which respectively specify how Φ should be *adjusted* (a generalization of granting that also supports privilege revocation) and *checked* during typechecking. Rather than hard-coding the privilege discipline as we did for exceptions in Figure 1, the typing rules in our generic effect system are uniformly instrumented with invocations of these two functions, which are the only parts of the type system that directly modify or inspect Φ. For example, using our approach, the rule T-TRY would invoke the adjust function to determine the privileges to use when typechecking e_1, while T-THROW would invoke the check function to determine what privileges are required upon a `throw`. These two functions are similarly invoked from the other rules in the type system, allowing any language feature to be controlled by the externally defined privilege discipline.

2.2 Checking and Adjusting Privileges, Generically

Our key contributions are a general interface for the check and adjust functions and a uniform approach to instrumenting a language's type system with invocations of the two functions. First, consider the function to check privileges. Intuitively, a particular effect system may want to disallow an arbitrary computation step (e.g., an exception being thrown, a write to memory) based on the set of held privileges. Therefore, a generic effect system should consult the check function as a premise in the typing rule for each syntactic form that represents a step of computation. Next consider the function to adjust privileges. Intuitively, a particular effect system may want to adjust privileges in any syntactic context that might arise dynamically (e.g., a `try` block, a `letregion` expression). Therefore, a generic effect system should consult the adjust function to modify Φ in typing rules before typechecking subexpressions of the current expression.

We make these ideas precise and applicable to any language by adapting two standard notions for formalizing a language's dynamic semantics: *redexes* and *evaluation contexts* (Wright and Felleisen 1994). The type system will have one call to the check function for each redex form in the language. The type system will have one call to the adjust function for each evaluation context form in the language.

The check function is invoked in the typing rule for each expression form that can evaluate to a redex, passing the current set of privileges and a *check context* that describes the information known statically about that redex. The function returns a boolean indicating whether the reduction step represented by that expression is allowed. To make things concrete, Figure 2 shows how the redexes of a lambda calculus with mutable references and `let` determine the check contexts for that language. Each value form in a redex is replaced by a tagset, π, which represents the information known

Table 1. Standard effect system enforcement as privilege checking.

Discipline	*Privilege*	*Grant*	*Check*
checked exceptions in Java	allowed to throw an exception	`try` block	`throw` statement
race condition checking in Java (Abadi et al. 2006)	allowed to access shared data	`synchronized` block	read or write to shared memory
region-based memory management (Tofte and Talpin 1994)	allowed to access memory in region	`letregion` expression	read or write to memory in region

Redex	$\Longmapsto$	*Check Context (C)*
$(\lambda x.e)v$		$\pi_1\ \pi_2$
`ref` v		`ref` π
$!l$		$!\pi$
$l := v$		$\pi_1 := \pi_2$
`let` $x = v$ `in` e		`let` $x = \pi$ `in` $\uparrow$

Figure 2. From redexes to check contexts. Metavariable v ranges over values, e over expressions, and l over memory locations.

Evaluation Context	$\Longmapsto$	*Adjust Context (A)*
Ee		$\downarrow\uparrow$
vE		$\pi\downarrow$
`ref` E		`ref` $\downarrow$
$!E$		$!\downarrow$
$E := e$		$\downarrow:=\uparrow$
$v := E$		$\pi:=\downarrow$
`let` $x = E$ `in` e		`let` $x = \downarrow$ `in` $\uparrow$

Figure 3. From evaluation contexts to adjust contexts. Metavariable E ranges over evaluation contexts.

statically about an expression's value, as determined by static typechecking. Expressions that are not necessarily values appearing in a redex are replaced by the $\uparrow$ symbol in the check context, indicating that information about this subexpression is not relevant to the reduction step being checked.

Consider our example of checked exceptions. Following the approach for formalizing a generic effect system described above, a language containing an expression `throw` e would include a check context of the form `throw` π due to the redex form `throw` v. Therefore, the typing rule for `throw` will invoke check(Φ, `throw` π), where Φ is the current set of privileges and π is the set of possible tags of e (determined by typechecking e). The following check function implements the hard-coded privilege checking behavior of T-THROW in Figure 1 and specifies that privilege checking for all other kinds of reduction steps should trivially succeed:

$$\text{check}(\Phi, C) = \begin{cases} \texttt{canThrow} \in \Phi \text{ if } C = \texttt{throw}\ \pi \\ \texttt{true} \text{ otherwise} \end{cases}$$

The adjust function is invoked once per evaluation context related to the current expression form in a typing rule, passing the current set of privileges and an *adjust context* that describes the information known statically about that evaluation context. The function returns a new set of privileges to be used when typechecking a designated subexpression indicated by the adjust context. Figure 3 shows how the evaluation contexts of our extended lambda calculus determine the associated adjust contexts. As before, values are replaced by tagsets and arbitrary expressions are replaced by the $\uparrow$ symbol. Finally, wherever the evaluation context contains a recursive evaluation context, the corresponding adjust context contains a $\downarrow$ symbol indicating the subexpression for which privileges should be adjusted.

Consider again our exceptions example. A language containing an expression `try` e_1 `with` e_2 would include an adjust context of the form `try` $\downarrow$ `with` $\uparrow$ due to the evaluation context `try E with` e. Therefore the typing rule for `try` will invoke adjust(Φ, `try` $\downarrow$ `with` $\uparrow$), where Φ is the current set of privileges, to determine the new set of privileges to use when typechecking e_1. Then the following adjust function implements the hard-coded privilege granting behavior of T-TRY in Figure 1 and specifies that privileges should be passed along unchanged in all other syntactic contexts:

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi \cup \{\texttt{canThrow}\} \text{ if } A = \texttt{try} \downarrow \texttt{with} \uparrow \\ \Phi \text{ otherwise} \end{cases}$$

To summarize, defining the exception checker in a generic effect system using our approach requires simply specifying the set of privileges, {`canThrow`}, and providing the above definitions for the check and adjust functions. As we show in Section 4, many other effect systems from the literature can be specified as instantiations of our generic effect system.

2.3 Proving Type Soundness

A basic notion of type soundness for effect systems requires static effect checking to conservatively approximate an instrumented dynamic semantics that tracks effects as the program executes. In our setting, this form of soundness requires that a well-typed program will never fail a dynamic privilege check. We could provide a separate mechanism for instrumenting the dynamic semantics with effects, but it turns out that the check and adjust functions provided for static type-and-effect checking naturally support dynamic privilege checking as well. Whereas statically these functions employ the conservative information gleaned from the types of expressions, dynamically the functions are given precise information about the tags of the resulting values and the set of held privileges.

Because the check and adjust functions are completely arbitrary, there is no guarantee that a particular instantiation of these functions will lead to a sound effect system. For example, since a sound static checker underapproximates the set of privileges that will be held dynamically, it would be unsound for a check function to rely on the *absence* of a privilege in Φ. Such a check function could cause the static checker to accept a program that fails privilege checking dynamically.

However, we can define a natural set of sufficient conditions on the check and adjust functions that ensure soundness. Intuitively these conditions ensure that the check and adjust functions are *monotonic* with respect to their treatment of both privileges and tags. We have proven that our generic effect system for the language described in Section 3 does indeed lead to a sound effect system for any check and adjust functions that satisfy our four monotonicity conditions, which are described in detail in Section 6. All of the check and adjust functions for the example effect systems in this paper satisfy the four conditions.

Our soundness guarantee does not capture *extensional* notions of soundness, which directly relate static effect checking to

$$
\begin{array}{llll}
\textit{Values} & v & ::= & (\lambda x.e)_\varepsilon \mid (\text{rec } x.\lambda x'.e)_\varepsilon \mid \text{unit}_\varepsilon \mid l_\varepsilon \\
\textit{Exprs} & e & ::= & v \mid x \mid ee \mid (\text{ref } e)_\varepsilon \\
 & & \mid & !e \mid (e := e)_\varepsilon \mid \text{let } x = e \text{ in } e \\
\textit{Types} & \tau & ::= & \pi\rho \\
\textit{PreTypes} & \rho & ::= & \tau \xrightarrow{\Phi} \tau \mid \text{Unit} \mid \text{Ref } \tau \\
\textit{PrivSets} & \Phi & ::= & \{p(\varepsilon)\}
\end{array}
$$

Figure 4. The syntax of our host language. Metavariable x ranges over variables, l over memory locations, ε over tags, π over nonempty sets of tags, p over privilege classes, and Φ over sets of privileges.

the uninstrumented dynamic semantics. For example, suppose the check function for our exceptions example above were defined to always return `true`. In that case, the discipline would still satisfy our notion of soundness (and monotonicity requirements) since dynamic privilege checking cannot fail, even though a well-typed program can now have an uncaught exception. Defining and proving extensional semantics for specific type-and-effect systems, such as those that track memory access and exceptions, is challenging and the subject of current research (Benton et al. 2006; Benton and Buchlovsky 2007).

3. A Generic Type-and-Effect System

Section 2 gave an intuitive overview of how we can instrument the static semantics of a language to create a generic system for enforcing disciplines on computational effects. This section makes the ideas more concrete by showing the complete, instrumented static semantics for a core language whose syntax is shown in Figure 4. Our language is the call-by-value lambda calculus with recursive functions, a unit value, ML-style references, and a let expression. As mentioned in the previous section, the syntax also includes a notion of tags.

The language has a fixed collection of typing rules defining the type system. The rules are parameterized by a set of *privilege classes* and the check and adjust functions that specify the behavior of a particular effect system for the language.

3.1 Preliminaries

Tagged Values and Tagged Types Our language includes *tags*, denoted by metavariable ε, which are static names for a set of run-time values. Each value is annotated with its associated tag, as is each expression that constructs a new value dynamically. For example, the evaluation of an expression of the form $(\text{ref } e)_\varepsilon$ evaluates e to a (tagged) value v, creates a new memory location l in the store that maps to v, and tags l with the tag ε.

The type of an expression includes a nonempty set of tags that we call a *tagset* and denote π, which represents the set of possible tags of the expression's run-time value. As shown in the definition of τ, each level of a type includes a corresponding tagset.

Our notion of tags is standard and closely related to work on *type qualifiers* (Ørbæk and Palsberg 1997; Foster et al. 2006). More sophisticated forms of tagging exist in the literature, for example tags that refer directly to program variables (Abadi et al. 2006) and lexically scoped tags (Tofte and Talpin 1994). These extensions would increase the expressiveness of our system but are orthogonal to our main contributions.

Privilege Classes and Privileges We assume a set of *privilege classes* ranged over by metavariable p. For example, to formalize a memory checker we could define three privilege classes `read`, `write`, and `alloc`, respectively representing the capability to read, write, and allocate memory. A *privilege* is a pair of a privilege class and a tag, denoted $p(\varepsilon)$. In examples, we use the privilege class itself as the privilege when the tag is irrelevant.

The static type-and-effect system typechecks an expression under a set of privileges Φ, representing the computational effects that are allowed to occur during the expression's evaluation. In Figure 4, function types are annotated with a privilege set, denoting the privileges needed to properly execute the function. As usual for type-and-effect systems, this annotation allows for modular effect checking of a function's body separate from its callers.

Checking and Adjusting Privileges As described previously, the privilege discipline is specified through two functions, check and adjust. The check function takes the current set of held privileges along with a *check context* and returns a boolean indicating whether the expression associated with the check context satisfies privilege checking. The adjust function takes the current set of held privileges along with an *adjust context* and returns a new set of privileges to use when typechecking a distinguished subexpression within the adjust context. The syntax of the check and adjust contexts for our language are shown on the right sides of Figures 2 and 3.

Our formalism does not model the *implementation* of the check and adjust functions. They are treated as black boxes that our rules use to enforce a particular effect system's discipline. As one example, the following check function corresponds to a standard checker for memory effects:

$$
\text{check}(\Phi, C) = \begin{cases}
\forall \varepsilon \in \pi.\ \texttt{read}(\varepsilon) \in \Phi & \text{if} \quad C = !\pi \\
\forall \varepsilon \in \pi.\ \texttt{write}(\varepsilon) \in \Phi & \text{if} \quad C = \pi := \pi' \\
\texttt{alloc} \in \Phi & \text{if} \quad C = \texttt{ref } \pi \\
\texttt{true} & \text{otherwise}
\end{cases}
$$

3.2 Type-and-Effect Checking

We now define our generic type-and-effect system. The typechecking judgment has the form $\Phi; \Gamma; \Sigma \vdash e : \tau$, which says that expression e can be given type τ in the context of privilege set Φ, type environment Γ (which maps each variable in scope to its type), and store type Σ (which maps each location in the store to the type of values that it holds). The typing rules are shown in Figure 5. Note that there is no restriction on the initial set of privileges used to typecheck the top-level program. For some effect disciplines an empty initial privilege set is appropriate, while for other disciplines it is natural to begin with a set of default privileges that are revoked in certain program contexts.

Because the rules are parametric in the effect discipline, the rules in Figure 5 really define a family of type systems indexed by the discipline. The discipline is defined as a triple of the set of privilege classes, the adjust function, and the check function: $\mathcal{D} = \langle \mathcal{P}_\mathcal{D}, \text{adjust}_\mathcal{D}, \text{check}_\mathcal{D} \rangle$. An instantiation of the type system for a particular discipline $\mathcal{D}$, then, is defined by the judgment $\Phi; \Gamma; \Sigma \vdash_\mathcal{D} e : \tau$ whose inference rules invoke the functions $\text{adjust}_\mathcal{D}$ and $\text{check}_\mathcal{D}$. In order to avoid cluttering the presentation, we have omitted the discipline subscripts in our figures. However we will return to this notation to more precisely state our soundness result in Section 6.

The rules perform standard typechecking for the simply typed lambda calculus with references. The rules also perform static tag checking, which approximates the tag of each expression by a tagset. Tag checking is straightforward and is a variant of *type qualifier* checking (Ørbæk and Palsberg 1997; Foster et al. 2006).

Finally, the rules employ the adjust and check functions to perform static privilege checking. The adjust function is consulted in order to produce the appropriate privilege set to use when typechecking each subexpression of the given expression. The check function is consulted in order to decide whether an expression that

$\boxed{\Phi;\Gamma;\Sigma \vdash e : \tau}$

$$\frac{\Phi_1;\Gamma,x:\tau_1;\Sigma \vdash e : \tau_2}{\Phi;\Gamma;\Sigma \vdash (\lambda x.e)_\varepsilon : \{\varepsilon\}(\tau_1 \xrightarrow{\Phi_1} \tau_2)} \quad \text{(T-FN)}$$

$$\frac{\Phi_1;\Gamma,x:\{\varepsilon\}(\tau_1 \xrightarrow{\Phi_1} \tau_2),x':\tau_1;\Sigma \vdash e : \tau_2}{\Phi;\Gamma;\Sigma \vdash (\text{rec } x.\lambda x'.e)_\varepsilon : \{\varepsilon\}(\tau_1 \xrightarrow{\Phi_1} \tau_2)} \quad \text{(T-REC)}$$

$$\frac{}{\Phi;\Gamma;\Sigma \vdash \text{unit}_\varepsilon : \{\varepsilon\}\text{Unit}} \quad \text{(T-UNIT)}$$

$$\frac{\Sigma(l) = \tau}{\Phi;\Gamma;\Sigma \vdash l_\varepsilon : \{\varepsilon\}\text{Ref } \tau} \quad \text{(T-LOC)}$$

$$\frac{\Gamma(x) = \tau}{\Phi;\Gamma;\Sigma \vdash x : \tau} \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,\downarrow\uparrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash e_1 : \pi_1(\tau_2 \xrightarrow{\Phi_1} \tau) \\ \text{adjust}(\Phi,\pi_1\downarrow) = \Phi'' \qquad \Phi'';\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \text{check}(\Phi,\pi_1\ \pi_2) \qquad \pi_2\rho_2 <: \tau_2 \qquad \Phi_1 \subseteq \Phi\end{array}}{\Phi;\Gamma;\Sigma \vdash e_1 e_2 : \tau} \quad \text{(T-APP)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,\text{ref}\downarrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash e : \tau \\ \tau = \pi\rho \qquad \text{check}(\Phi,\text{ref}\ \pi)\end{array}}{\Phi;\Gamma;\Sigma \vdash (\text{ref } e)_\varepsilon : \{\varepsilon\}\text{Ref } \tau} \quad \text{(T-REF)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,!\downarrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash e : \pi\ \text{Ref}\ \tau \\ \text{check}(\Phi,!\pi)\end{array}}{\Phi;\Gamma;\Sigma \vdash !e : \tau} \quad \text{(T-DEREF)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,\downarrow:=\uparrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash e_1 : \pi_1\text{Ref}\ \tau_1 \\ \text{adjust}(\Phi,\pi_1 :=\downarrow) = \Phi'' \qquad \Phi'';\Gamma;\Sigma \vdash e_2 : \pi_2\rho_2 \\ \text{check}(\Phi,\pi_1 := \pi_2) \qquad \pi_2\rho_2 <: \tau_1\end{array}}{\Phi;\Gamma;\Sigma \vdash (e_1 := e_2)_\varepsilon : \{\varepsilon\}\text{Unit}} \quad \text{(T-ASGN)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,\text{let } x =\downarrow \text{ in } \uparrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash e_1 : \pi_1\rho_1 \\ \text{check}(\Phi,\text{let } x = \pi_1 \text{ in } \uparrow) \qquad \Phi;\Gamma,x:\pi_1\rho_1;\Sigma \vdash e_2 : \tau\end{array}}{\Phi;\Gamma;\Sigma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \quad \text{(T-LET)}$$

$$\frac{\begin{array}{c}\text{adjust}(\Phi,\text{let } x =\downarrow \text{ in } \uparrow) = \Phi' \qquad \Phi';\Gamma;\Sigma \vdash v_1 : \pi_1\rho_1 \\ \text{check}(\Phi,\text{let } x = \pi_1 \text{ in } \uparrow) \qquad \Phi;\Gamma;\Sigma \vdash e_2[x \mapsto v_1] : \tau\end{array}}{\Phi;\Gamma;\Sigma \vdash \text{let } x = v_1 \text{ in } e_2 : \tau} \quad \text{(T-VLET)}$$

Figure 5. Generic type-and-effect checking.

will dynamically evaluate to a redex has the appropriate privileges to be reduced.

For example, consider the rule T-ASGN for typechecking expressions of the form $e_1 := e_2$. The rule invokes the adjust function to produce the privilege set Φ' to use when typechecking e_1. The adjust function is invoked again, this time passing as context the statically determined tagset π_1 for e_1, to produce the privilege set Φ'' for typechecking e_2. The adjust function for a particular discipline could simply make $\Phi' = \Phi'' = \Phi$, thereby requiring e_1 and e_2 to be well typed under the current set of privileges. However, our rule soundly allows many other effect disciplines to be enforced. For instance, the following adjust function, used in conjunction with the check function at the end of Section 3.1, serves to enforce a particular canonical form on programs, whereby the subexpressions in an assignment expression are required to be pure:

$$\text{adjust}(\Phi,A) = \begin{cases} \emptyset & \text{if} \quad A = \downarrow:=\uparrow \quad \text{or} \quad A = \pi :=\downarrow \\ \Phi & \text{otherwise} \end{cases}$$

Finally, the T-ASGN rule passes π_1 along with e_2's statically determined tagset π_2 to the check function in order to ensure that the effect discipline for assignments is being obeyed. The premise $\text{check}(\Phi,\pi_1 := \pi_2)$ is shorthand for the requirement that this call to the check function returns true.

The rule T-FN for typechecking lambdas "guesses" an argument type τ_1 and privilege set Φ_1 for use in typechecking the function body, and similarly for the rule T-REC. Our tag system is a variant of existing approaches to user-defined type qualifiers, so we expect tag inference to follow from the work on qualifier inference (Ørbæk and Palsberg 1997; Foster et al. 2006). Privilege inference poses more of a challenge, particularly in the presence of arbitrary black-box check and adjust functions. In this paper we have chosen to keep the check and adjust functions fully general, in order to explore the expressiveness and soundness limits of our approach. It may be possible to adapt existing algorithms for effect inference (e.g., (Talpin and Jouvelot 1992)) by restricting the form of the check and adjust functions.

The rule T-APP performs privilege checking in a manner analogous to the other rules, and it additionally ensures that the privileges Φ_1 required by the function being invoked are a subset of the current set of privileges Φ.

$\boxed{\tau_1 <: \tau_2}$

$$\frac{\pi_1 \subseteq \pi_2}{\pi_1\text{Unit} <: \pi_2\text{Unit}} \quad \text{(ST-UNIT)}$$

$$\frac{\begin{array}{c}\pi_1 \subseteq \pi_2 \\ \tau_1 <: \tau_2 \qquad \tau_2 <: \tau_1\end{array}}{\pi_1\ (\text{Ref } \tau_1) <: \pi_2\ (\text{Ref } \tau_2)} \quad \text{(ST-REF)}$$

$$\frac{\begin{array}{c}\pi_1 \subseteq \pi_2 \qquad \Phi_1 \subseteq \Phi_2 \\ \tau_2 <: \tau_1 \qquad \tau_1' <: \tau_2'\end{array}}{\pi_1(\tau_1 \xrightarrow{\Phi_1} \tau_1') <: \pi_2(\tau_2 \xrightarrow{\Phi_2} \tau_2')} \quad \text{(ST-FN)}$$

Figure 6. The subtyping judgment.

For increased expressiveness, our type system includes a form of subtyping for tagged types. For example, the rule T-APP only requires the type of the actual argument expression to be a subtype of the formal argument type. The subtyping rules are shown in Figure 6. A type's tagset is required to be a subset of any supertype's tagset. This makes sense since a tagset represents the set of all possible tags of the associated expression's run-time value, so a smaller tagset denotes a stronger property than a larger tagset. As usual for soundness, nontrivial subtyping is not allowed underneath a Ref type and function argument types are contravariant. Finally, a function type's privilege set must be a subset of any supertype's privilege set, since a function that requires fewer privileges can be safely used where one requiring more privileges is expected.

Our type system also includes a form of let-polymorphism for tagged types. As others have done (e.g., (Talpin and Jouvelot 1992)), we express polymorphism formally through substitution rather than through explicit quantification over types. The rule T-VLET in Figure 5 expresses this form of polymorphism. As usual,

$$\frac{\text{adjust}(\Phi, \text{letscope } \pi_1 \text{ in } \downarrow) = \Phi' \qquad \Phi'; \Gamma; \Sigma \vdash e : \tau \qquad \tau = \pi_2 \rho_2 \qquad \text{check}(\Phi, \text{letscope } \pi_1 \text{ in } \pi_2)}{\Phi; \Gamma; \Sigma \vdash \text{letscope } \pi_1 \text{ in } e : \tau}$$

Figure 7. The typing rule for letscope.

for soundness in the presence of references we only treat values polymorphically, so the rule is specialized to that situation. As an example of polymorphism, consider our memory effects checker and a function that takes a reference cell as an argument and possibly dereferences it. With polymorphism, this function can be called multiple times with differently tagged reference cells, as long as the appropriate `read` privileges are held at each call site.

4. Examples

This section illustrates the expressiveness of our approach. We first extend our core language with a generic scoping mechanism in order to facilitate the presentation of examples and then show how several effect systems from the literature can be expressed as instantiations of our generic system.

4.1 A Generic Scoping Expression

For some effect systems there is a construct in the language where it is natural to adjust privileges, such as `try` for an exception checker. But others, like our memory checker, do not have such an expression. In these cases, we can use function annotations to control privileges. For instance, by annotating a function with the empty privilege set in our memory checker, we prevent the body of the function from accessing memory. However, we may want to have more fine-grained control. To achieve this we introduce a new expression form called *letscope*. The expression (letscope π in e) dynamically behaves like e and, like all expressions in our language, has associated check and adjust contexts. Thus we can define cases for the adjust function that grant or revoke privileges while type-checking the body of the letscope. The typing rule for letscope is shown in Figure 7.

Consider again our memory checker with `read`, `write` and `alloc` privileges. Assuming that we begin with an initial set containing all privileges, we can use letscope to require purity in certain code segments:

$$\text{adjust}(\Phi, A) = \begin{cases} \emptyset \text{ if } A = \text{letscope } \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

Using the tagset π provided to the letscope we can allow more fine-grained control and require that a code segment be pure only with respect to locations with particular tags:

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi - \{\text{write}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{read}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{alloc}\} \\ \qquad \text{if } A = \text{letscope } \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

It may be useful to have different "kinds" of letscopes for different purposes. Our formalism has only one letscope expression, but different kinds can be encoded through the tagset π provided in the expression. For example, we could modify the above rule to only revoke privileges when π contains a distinguished `pure` tag:

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi - \{\text{write}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{read}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{alloc}\} \\ \qquad \text{if } A = \text{letscope } \pi \text{ in } \downarrow \text{ and } \texttt{pure} \in \pi \\ \Phi \text{ otherwise} \end{cases}$$

We could then augment the above function to revoke only `write` and `alloc` privileges when a distinguished `readonly` tag is present in π. In this way, we can use the letscope expression to perform many different privilege operations within a program. Throughout this section, we use a syntactic sugar for this idiom and allow letscope expressions to be subscripted with a tag that we call a *letscope kind*. These kinds can be used within check and adjust functions. For example, we can rewrite the above effect discipline using our syntactic sugar as follows:

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi - \{\text{write}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{read}(\varepsilon) \mid \varepsilon \in \pi\} - \{\text{alloc}\} \\ \qquad \text{if } A = \text{letscope}_{\texttt{pure}} \ \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

4.2 Type Qualifiers

Our notion of tags is sufficient to express typical idioms involving type qualifiers. For example, a program could use tags `untainted` and `tainted` to respectively tag data that can and cannot be trusted (Ørbæk and Palsberg 1997; Shankar et al. 2001). Our type system ensures that if a function's formal parameter is declared to have type $\{\texttt{untainted}\}\ \rho$, then `tainted` data will never flow there.

The CQUAL system for type qualifiers in C (Foster et al. 2006) allows users to specify a partial order on qualifiers, which induces a subtyping relation on qualified types. For example, a CQUAL user would declare $\texttt{untainted} \leq \texttt{tainted}$ in order to allow `untainted` data to flow wherever `tainted` data is expected. While our type system does not support such a relationship between tags, our use of tagsets and the associated subtyping relation on tagged types accomplishes the same thing. In our type system both `untainted` and `tainted` data can flow to a place where a value of type $\{\texttt{untainted},\texttt{tainted}\}\ \rho$ is expected.

Finally, our check function can interact with tags to provide more expressiveness for type qualifiers, even in the absence of privileges. For example, consider a tag `readonly` that is meant to annotate memory locations that cannot be updated after initialization. The following check function enforces this behavior:

$$\text{check}(\Phi, C) = \begin{cases} \texttt{readonly} \notin \pi \text{ if } C = \pi := \pi' \\ \texttt{true} \text{ otherwise} \end{cases}$$

4.3 Memory Effects

Consider the check function at the end of Section 3.1 which implements standard tracking of memory effects. Given this definition for check, our type system will require each function's type to be annotated with a set of privileges representing the memory effects that may occur during its execution. These privileges are expressive enough to represent the four "effect classes" in the original effect system of Gifford and Lucassen (1986): a function requiring the empty set of privileges corresponds to their PURE effect class; a function requiring neither `read` nor `write` (but possibly `alloc`) privileges corresponds to their FUNCTION effect class; a function requiring no `write` privileges corresponds to their OBSERVER effect class; and a function requiring arbitrary privileges corresponds to their PROCEDURE effect class.

The letscope construct can also be used to enforce these disciplines at a finer granularity than a function. One approach is to allow all memory operations by default, by including the `alloc` privilege and the `read` and `write` privileges for all tags in the initial privilege set. Memory effects can then be restricted throughout the program by using different kinds of letscopes. The adjust function shown in Figure 8 implements this discipline. It includes three letscope kinds to mark code that falls into the different effect classes. There is no letscope kind corresponding to the PROCEDURE effect class since our initial privilege set makes this the default.

$$\text{adjust}(\Phi, A) = \begin{cases} \emptyset \text{ if } A = \text{letscope}_{\texttt{pure}}\ \pi \text{ in } \downarrow \\ \Phi - \{\text{write}(\varepsilon) \mid \text{write}(\varepsilon) \in \Phi\} - \{\text{read}(\varepsilon) \mid \text{read}(\varepsilon) \in \Phi\} \\ \quad \text{if } A = \text{letscope}_{\texttt{fun}}\ \pi \text{ in } \downarrow \\ \Phi - \{\text{write}(\varepsilon) \mid \text{write}(\varepsilon) \in \Phi\} \text{ if } A = \text{letscope}_{\texttt{obs}}\ \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

Figure 8. An adjust function for a memory checker whose initial privilege set allows everything.

$$\text{check}(\Phi, C) = \begin{cases} (\text{p} \in \pi \Rightarrow \text{P} \in \Phi) \wedge (\text{u} \in \pi \Rightarrow \text{U} \in \Phi) \text{ if } C = !\pi \\ (\text{p} \in \pi \Rightarrow \text{P} \in \Phi) \wedge (\text{u} \in \pi \Rightarrow \text{U} \in \Phi) \text{ if } C = \pi := \pi' \\ \texttt{true} \text{ otherwise} \end{cases}$$

$$\text{adjust}(\Phi, A) = \begin{cases} \{\text{P}\} \text{ if } A = \text{async } \downarrow \\ \{\text{U}\} \text{ if } A = \text{unprotected } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

Figure 9. Implementing the AME calculus effect system.

As a concrete example, suppose a programmer decides to memoize the results of a certain function in order to improve performance. For the memoization to be safe, the function must be pure. Therefore, the programmer could create a wrapper function to perform the memoization whose call to the original function is enclosed in a $\text{letscope}_{\texttt{pure}}$. Our type system would then ensure that this call can be typechecked without any memory privileges.

4.4 Strong Atomicity for Transactional Memory

Our approach naturally extends to richer languages than our simple formalism. To illustrate this, we show that an effect system defined by others for their AME calculus (Abadi et al. 2008), a lambda calculus augmented with mutable references and constructs for automatic mutual exclusion (Isard and Birrell 2007), is supported by our model. The relevant language extensions are (async e), which executes e asynchronously as a transaction, and (unprotected e), which executes e asynchronously outside of a transaction. The latter form is necessary to support side effects like I/O as well as interaction with legacy code.

The authors discuss the problem of "weak atomicity" (Martin et al. 2006), whereby implementations of software transactional memory (STM) do not prevent conflicts between transactional and non-transactional code. The resulting semantics can cause unexpected and counterintuitive behavior, but implementing STM to directly support "strong atomicity" is difficult. To address the problem, the authors show how an effect system can be used to ensure that transactional and non-transactional code never read or write the same memory locations, thereby recovering strong atomicity even when the STM is implemented in the weakly atomic style.

The effect system for the AME calculus uses effects `P` and `U` to respectively distinguish between protected and unprotected contexts. The type for references is similarly augmented with a tag (`p` or `u`) indicating whether the reference can be manipulated in protected or unprotected contexts. The effect discipline then ensures that a protected (unprotected) reference is only manipulated in a protected (unprotected) context. This effect system can be easily expressed in a generic effect system for the AME calculus created by following the approach we outlined in Section 2. Figure 9 shows check and adjust functions that enforce the discipline.

4.5 Application-Specific Effects

Our type system is also expressive enough to capture useful kinds of application-specific effects. As an example, we consider some issues in a user-level threads library. Such a library typically provides wrapped versions of all system calls that potentially cause a process to block, such as the `read` and `write` file operations. Clients of the thread library should always invoke the wrapped versions of these system calls, so that the library may schedule another thread while the caller awaits a response. Our type-and-effect system can be used to check that blocking system calls are never directly invoked.

To do so, we assume that each blocking system call is annotated with the tag `blocks`. We then define a privilege `mayblock`, along with the following check and adjust functions:

$$\text{check}(\Phi, C) = \begin{cases} \texttt{blocks} \in \pi \Rightarrow \texttt{mayblock} \in \Phi \text{ if } C = \pi\ \pi' \\ \texttt{true} \text{ otherwise} \end{cases}$$

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi \cup \{\texttt{mayblock}\} \text{ if } A = \text{letscope}_{\texttt{blk}}\ \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

The check function above requires the `mayblock` privilege to be held whenever a function potentially tagged with `blocks` is invoked, while the adjust function uses $\text{letscope}_{\texttt{blk}}$ to grant the `mayblock` privilege. Now $\text{letscope}_{\texttt{blk}}$ can be employed in the implementation of the library's wrapper functions, allowing them to directly invoke the blocking system calls. The intent is that client functions should never use $\text{letscope}_{\texttt{blk}}$.[1] Therefore, if `start_thread` is the library function that takes a client function and invokes it in a new thread, then giving `start_thread`'s formal parameter the type $\texttt{Unit} \xrightarrow{\emptyset} \texttt{Unit}$ will ensure that client functions do not directly invoke blocking system calls.

To continue the example, suppose that this user-level threads library is cooperatively scheduled: client code is not preempted but instead explicitly yields control to the scheduler, either by calling one of the wrapper functions described above or by calling a special `yield` function provided by the library. Others have used an effect system to track which functions might yield control to the scheduler, since yields represent points at which the code must be properly synchronized with other threads (Fischer et al. 2007; Isard and Birrell 2007).

To define this effect system in our theory, we assume that the wrapper functions and the `yield` function are annotated with the tag `yields`, and as above we define a privilege `mayyield`. Assuming that we check programs under an initial privilege set containing `mayyield`, the following check and adjust function cases implement the desired effect discipline:

$$\text{check}(\Phi, C) = \begin{cases} \texttt{yields} \in \pi \Rightarrow \texttt{mayyield} \in \Phi \text{ if } C = \pi\ \pi' \\ \texttt{true} \text{ otherwise} \end{cases}$$

$$\text{adjust}(\Phi, A) = \begin{cases} \Phi - \{\texttt{mayyield}\} \text{ if } A = \text{letscope}_{\texttt{atomic}}\ \pi \text{ in } \downarrow \\ \Phi \text{ otherwise} \end{cases}$$

The check function above requires the caller of a function that potentially yields to hold the `mayyield` privilege. The $\text{letscope}_{\texttt{atomic}}$ can then be placed around code blocks that must be executed atomically, in order to ensure that the execution of these code blocks does not yield to the scheduler.

It is also possible to express finer-grained versions of the above effect discipline in our type system. For example, rather than tagging all wrapper functions and `yield` with the same tag, each function could have its own tag. The `mayyield` privilege could then be augmented to be parameterized by a tag. In that way, the effect system would track not only which functions potentially yield but also *how* they yield.

[1] If our language had a module system, it would be natural to allow users to limit the visibility of a letscope kind to a particular module.

There are many other application-specific properties that could be tracked and controlled using our type-and-effect system in a manner similar to the examples shown above. For instance, the Linux kernel deallocates some data and functions after the initialization phase, and this code is tagged with the qualifier `__init` (Foster et al. 2006). It would be straightforward to introduce a privilege `mayinit` along with a check function definition to track functions that may manipulate `__init` data or call `__init` functions. A letscope could then be used to revoke the `mayinit` privilege after the initialization phase, ensuring that the kernel will not typecheck if later phases attempt to touch `__init` data or functions. As a final example, the Enterprise JavaBeans component platform for Java imposes several requirements on "bean" classes written by clients (DeMichiel 2004). One of these requirements is that beans do not spawn threads. Similar to the system calls example above, an effect system could be used to track functions that potentially spawn threads.

5. Dynamic Semantics

As mentioned earlier, our dynamic semantics is instrumented to perform privilege checking via the same check and adjust functions provided for static type-and-effect checking. As usual, dynamic checking is more precise than its static counterpart. For example, the standard check function for memory effects (shown at the end of Section 3.1) will be used dynamically to check that the appropriate `read` privilege is held for the actual memory location being dereferenced.

The operational semantics of our language is defined in Figure 10. As usual, a store μ is a finite mapping from memory locations to values. The small-step operational semantics is specified by a judgment of the form $\Phi \vdash_{\mathcal{D}} \langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle$, which says that, for a particular discipline $\mathcal{D}$, the pair of expression e and store μ takes one step of evaluation to a new pair e' and μ' in the context of privileges Φ. The rules make use of a small abuse of notation: v_ε represents an arbitrary value whose associated tag is ε. As in our rules for the type system, we omit the discipline subscripts in Figure 10 for readability.

As usual in a small-step semantics, there are two kinds of rules: *congruence* rules (whose names begin with S-), which simply take a step of evaluation in some subexpression; and *computation* rules (whose names begin with E-), which perform a reduction. The adjust and check functions exactly parallel this distinction. The adjust function is consulted in each congruence rule, to determine the set of privileges to use when evaluating the subexpression. The check function is consulted in each computation rule, to determine whether the reduction is allowed under the current set of privileges. Because each value has exactly one tag, the tagset arguments to the check and adjust contexts are always singleton sets, unlike in the static semantics.

Our instrumented dynamic semantics allows us to define the standard notion of type soundness, which is the subject of the next section. It would be unnecessary for an implementation to actually perform dynamic tagging or privilege checking on well-typed programs.

6. Type Soundness

In our setting, the standard type soundness theorem for effects requires that well-typed programs do not fail any dynamic privilege checks. This is a basic well-formedness condition that should be true of any sound effect discipline. However, since the user-defined check and adjust functions are unrestricted, it is not possible to prove type soundness once and for all. In particular, there exist disciplines $\mathcal{D}$ defined by functions $\text{check}_{\mathcal{D}}$ and $\text{adjust}_{\mathcal{D}}$ that would allow a program to typecheck but fail a privilege check dynamically. Therefore, type soundness must be proven separately for each effect discipline.

We have architected a modular approach to type soundness. Just as our language's static and dynamic semantics are parameterized by a small amount of discipline-specific information, so is our language's type soundness proof. The bulk of the proof is independent of the adjust and check functions and is provable once. Proving type soundness for a particular effect discipline merely requires proving four relatively simple lemmas which ensure natural forms of monotonicity for the adjust and check functions.

This section describes our approach to proving type soundness. First, we illustrate how an effect discipline could violate our informal notion type soundness. Next, we describe our modular proof architecture, focusing on the four lemmas that must be proven about a particular effect discipline. Finally, we formally present the type soundness result for our generic framework.

6.1 An Unsound Privilege Discipline

Type soundness does not hold in general for our language, because of the presence of arbitrary check and adjust functions. For example, consider the following cases of the check and adjust functions for a discipline $\mathcal{U}$ related to memory writes:

$$\text{check}_{\mathcal{U}}(\Phi, C) = \begin{cases} \forall \varepsilon \in \pi.\ \texttt{write}(\varepsilon) \in \Phi \ \text{ if } \ C = \pi := \pi' \\ \texttt{true} \ \text{ otherwise} \end{cases}$$

$$\text{adjust}_{\mathcal{U}}(\Phi, A) = \begin{cases} \{\texttt{write}(\varepsilon) \mid \varepsilon \in \pi\} \ \text{ if } \ A = \pi :=\downarrow \\ \Phi \ \text{ otherwise} \end{cases}$$

The check function above is standard for tracking memory writes. The adjust function above only allows the right-hand side of an assignment to update memory tagged with one of the possible tags of the left-hand location.

Consider an assignment of the form $e_1 := e_2$. With the above adjust function e_2 will be statically typechecked under the privileges $\{\texttt{write}(\varepsilon) \mid \varepsilon \in \pi\}$, where π is the top-level tagset in the type of e_1. However, this adjust function will cause e_2 to be dynamically evaluated under just the privilege $\texttt{write}(\varepsilon)$, where ε is the single tag of the resulting value of e_1. If π contains some other tag ε', then the check function above will statically allow e_2 to write to memory tagged with ε', even though dynamically this check will fail.

The fact that not all effect disciplines lead to sound type systems motivates our approach to proving type soundness, which is discussed next.

6.2 Modular Proof Architecture

We have devised a novel proof architecture for ensuring type soundness of instantiations of our generic system. The architecture has a small discipline-dependent portion, which consists of only four monotonicity lemmas about the check and adjust functions. The rest of the type soundness proof is completely independent of the check and adjust functions and can be proven once and for all. As we discuss in the next section, we have validated this proof architecture by formalizing our language, its type system, and its type soundness proof in the Twelf proof assistant (Pfenning and Schürmann 1999). Twelf verifies our type soundness proof as complete modulo the proofs of the four lemmas discussed below.

6.2.1 Privilege Monotonicity

The first two lemmas require a form of monotonicity of the check and adjust functions with respect to privilege sets. The lemmas ensure that the more privileges that are held, the more an expression is allowed to do. Intuitively, this property is necessary for soundness since a sound static typechecker underapproximates the set of privileges that will be held dynamically for the evaluation of

$\boxed{\Phi \vdash \langle e,\mu\rangle \longrightarrow \langle e',\mu'\rangle}$

$$\frac{\text{adjust}(\Phi,\downarrow\uparrow) = \Phi' \qquad \Phi' \vdash \langle e_1,\mu\rangle \longrightarrow \langle e_1',\mu'\rangle}{\Phi \vdash \langle e_1\ e_2,\mu\rangle \longrightarrow \langle e_1'\ e_2,\mu'\rangle}\ \text{(S-APP1)}$$

$$\frac{\text{adjust}(\Phi,\{\varepsilon\}\downarrow) = \Phi' \qquad \Phi' \vdash \langle e_2,\mu\rangle \longrightarrow \langle e_2',\mu'\rangle}{\Phi \vdash \langle v_\varepsilon\ e_2,\mu\rangle \longrightarrow \langle v_\varepsilon\ e_2',\mu'\rangle}\ \text{(S-APP2)}$$

$$\frac{\text{check}(\Phi,\{\varepsilon\}\ \{\varepsilon'\})}{\Phi \vdash \langle(\lambda x.e)_\varepsilon\ v_{\varepsilon'},\mu\rangle \longrightarrow \langle e[x \mapsto v_{\varepsilon'}],\mu\rangle}\ \text{(E-APP)}$$

$$\frac{\text{check}(\Phi,\{\varepsilon\}\ \{\varepsilon'\})}{\Phi \vdash \langle(\text{rec } x.\lambda x'.e)_\varepsilon\ v_{\varepsilon'},\mu\rangle \longrightarrow \langle e[x \mapsto (\text{rec } x.\lambda x'.e)_\varepsilon, x' \mapsto v_{\varepsilon'}],\mu\rangle}\ \text{(E-APP2)}$$

$$\frac{\text{adjust}(\Phi,\text{ref}\downarrow) = \Phi' \qquad \Phi' \vdash \langle e,\mu\rangle \longrightarrow \langle e',\mu'\rangle}{\Phi \vdash \langle(\text{ref } e)_\varepsilon,\mu\rangle \longrightarrow \langle(\text{ref } e')_\varepsilon,\mu'\rangle}\ \text{(S-REF)}$$

$$\frac{\text{check}(\Phi,\text{ref}\ \{\varepsilon\}) \qquad l \notin \text{domain}(\mu)}{\Phi \vdash \langle(\text{ref } v_\varepsilon)_{\varepsilon'},\mu\rangle \longrightarrow \langle l_{\varepsilon'},\mu[l \mapsto v_\varepsilon]\rangle}\ \text{(E-REF)}$$

$$\frac{\text{adjust}(\Phi,!\downarrow) = \Phi' \qquad \Phi' \vdash \langle e,\mu\rangle \longrightarrow \langle e',\mu'\rangle}{\Phi \vdash \langle !e,\mu\rangle \longrightarrow \langle !e',\mu'\rangle}\ \text{(S-DEREF)}$$

$$\frac{\text{check}(\Phi,!\{\varepsilon\}) \qquad \mu(l) = v}{\Phi \vdash \langle !l_\varepsilon,\mu\rangle \longrightarrow \langle v,\mu\rangle}\ \text{(E-DEREF)}$$

$$\frac{\text{adjust}(\Phi,\downarrow:=\uparrow) = \Phi' \qquad \Phi' \vdash \langle e_1,\mu\rangle \longrightarrow \langle e_1',\mu'\rangle}{\Phi \vdash \langle(e_1 := e_2)_\varepsilon,\mu\rangle \longrightarrow \langle(e_1' := e_2)_\varepsilon,\mu'\rangle}\ \text{(S-ASGN1)}$$

$$\frac{\text{adjust}(\Phi,\{\varepsilon\} :=\downarrow) = \Phi' \qquad \Phi' \vdash \langle e,\mu\rangle \longrightarrow \langle e',\mu'\rangle}{\Phi \vdash \langle(v_\varepsilon := e)_{\varepsilon'},\mu\rangle \longrightarrow \langle(v_\varepsilon := e')_{\varepsilon'},\mu'\rangle}\ \text{(S-ASGN2)}$$

$$\frac{\text{check}(\Phi,\{\varepsilon\} := \{\varepsilon'\})}{\Phi \vdash \langle(l_\varepsilon := v_{\varepsilon'})_{\varepsilon''},\mu\rangle \longrightarrow \langle \text{unit}_{\varepsilon''},\mu[l := v_{\varepsilon'}]\rangle}\ \text{(E-ASGN)}$$

$$\frac{\text{adjust}(\Phi,\text{let } x =\downarrow\ \text{in}\ \uparrow) = \Phi' \qquad \Phi' \vdash \langle e_1,\mu\rangle \longrightarrow \langle e_1',\mu'\rangle}{\Phi \vdash \langle\text{let } x = e_1 \text{ in } e_2,\mu\rangle \longrightarrow \langle\text{let } x = e_1' \text{ in } e_2,\mu'\rangle}\ \text{(S-LET)}$$

$$\frac{\text{check}(\Phi,\text{let } x = \{\varepsilon\}\ \text{in}\ \uparrow)}{\Phi \vdash \langle\text{let } x = v_\varepsilon \text{ in } e_2,\mu\rangle \longrightarrow \langle e_2[x \mapsto v_\varepsilon],\mu\rangle}\ \text{(E-LET)}$$

$$\frac{\text{adjust}(\Phi,\text{letscope } \pi \text{ in } \downarrow) = \Phi' \qquad \Phi' \vdash \langle e,\mu\rangle \longrightarrow \langle e',\mu'\rangle}{\Phi \vdash \langle\text{letscope } \pi \text{ in } e,\mu\rangle \longrightarrow \langle\text{letscope } \pi \text{ in } e',\mu'\rangle}\ \text{(S-LETSCOPE)}$$

$$\frac{\text{check}(\Phi,\text{letscope } \pi \text{ in } \{\varepsilon\})}{\Phi \vdash \langle\text{letscope } \pi \text{ in } v_\varepsilon,\mu\rangle \longrightarrow \langle v_\varepsilon,\mu\rangle}\ \text{(E-LETSCOPE)}$$

Figure 10. Generic operational semantics.

LEMMA 6.1. If $\Phi_1 \subseteq \Phi_2$ and $\text{check}(\Phi_1,C)$, then $\text{check}(\Phi_2,C)$.

LEMMA 6.2. If $\Phi_1 \subseteq \Phi_2$, then $\text{adjust}(\Phi_1,A) \subseteq \text{adjust}(\Phi_2,A)$.

Figure 11. Lemmas for privilege monotonicity.

LEMMA 6.3. If $C_2 \sqsubseteq C_1$ and $\text{check}(\Phi,C_1)$, then $\text{check}(\Phi,C_2)$.

LEMMA 6.4. If $A_2 \sqsubseteq A_1$, then $\text{adjust}(\Phi,A_1) \subseteq \text{adjust}(\Phi,A_2)$.

Figure 12. Lemmas for tag monotonicity.

some subexpression. The two lemmas are described formally in Figure 11.

The lemmas naturally support many desirable programming disciplines for type-and-effect systems, including all of the example check and adjust functions shown in this paper. Consider the `mayyield` example shown at the end of Section 4. The first lemma in Figure 11 holds for the given check function because an element of a set is also an element of any superset. It is easy to check that the second lemma holds for the $\text{letscope}_{\texttt{atomic}}$ adjust function case, which revokes the `mayyield` privilege. Finally, the lemma also safely allows the adjust function that grants `mayblock` within $\text{letscope}_{\texttt{blk}}$.

Intuitively the two lemmas in Figure 11 disallow check and adjust functions that depend on "negative" information about Φ. For example, a check function cannot require the absence of a particular privilege from Φ. However, in some cases it is possible to obtain the desired behavior by replacing a "negative" privilege p with a dual privilege $\overline{p}$, which is granted (revoked) wherever p would have been revoked (granted) and whose presence is required wherever p's absence would have been required. Further, some uses of negative information do satisfy the two lemmas. For example, it is perfectly sound for an adjust function to revoke a particular privilege only in the absence of another privilege.

6.2.2 Tag Monotonicity

The other two lemmas require a form of monotonicity of the adjust and check functions with respect to tagsets. The lemmas ensure that the smaller the tagsets are in the adjust and check contexts, the more an expression is allowed to do. Intuitively, this is necessary for soundness since the static typechecker overapproximates the possible tags of an expression's run-time values.

The lemmas for tag monotonicity are defined in Figure 12. The lemmas rely on partial orders on the check and adjust contexts, denoted $\sqsubseteq$. These relations are simply the $\subseteq$ partial order on tagsets lifted in the obvious way to check and adjust contexts; they are formally defined in Figure 13. The tag monotonicity lemmas naturally support many desirable programming disciplines, including all of our examples (except the unsound discipline from Section 6.1). For example, the check function for `mayyield` in Section 4 satisfies the first lemma because as π gets smaller the check gets weaker. The check function for `readonly` from that section similarly satisfies that lemma.

Consider the $\text{adjust}_{\mathcal{U}}$ function for our unsound effect discipline in Section 6.1. This function does not satisfy Lemma 6.4: as π gets smaller, the function grants *fewer* privileges rather than *more* privileges.

6.3 Type Soundness

We can now express type soundness for our framework in the traditional "progress and preservation" style (Wright and Felleisen 1994). We first define the notion of a monotonic discipline:

DEFINITION 6.1. (Monotonic Discipline) We say that $\mathcal{D}$ is *monotonic* if $\text{check}_{\mathcal{D}}$ satisfies Lemmas 6.1 and 6.3 and $\text{adjust}_{\mathcal{D}}$ satisfies Lemmas 6.2 and 6.4.

We will also require a notion of a well-typed store:

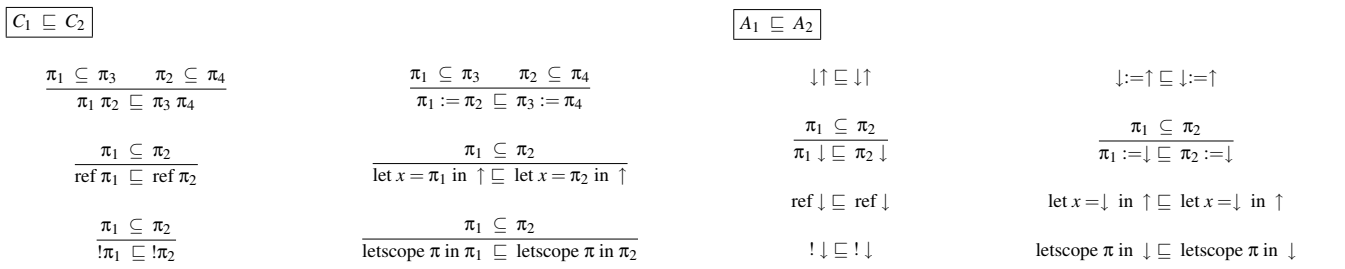

Figure 13. Partial order on check and adjust contexts.

DEFINITION 6.2. (Well-Typed Store) We say that $\Gamma, \Sigma \vdash_{\mathcal{D}} \mu$ if domain(μ) = domain(Σ) and $\forall l \in$domain(μ) we have $\emptyset; \Gamma; \Sigma \vdash_{\mathcal{D}} \mu(l) : \Sigma(l)$.

This definition is standard (Pierce 2002), except for the privilege set used in the typechecking judgment. The typechecking of values never depends on privileges, so it is sufficient to use an empty set of privileges in the definition.

Now we can state the progress and preservation theorems, which are also standard (we use $\bullet$ to denote the empty type environment):

THEOREM 6.1. (Progress) For any monotonic discipline $\mathcal{D}$, if $\Phi; \bullet; \Sigma \vdash_{\mathcal{D}} e : \tau$, then either e is a value or for all μ such that $\bullet, \Sigma \vdash_{\mathcal{D}} \mu$, there exist e' and μ' such that $\Phi \vdash_{\mathcal{D}} \langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle$.

THEOREM 6.2. (Preservation) For any monotonic discipline $\mathcal{D}$, if $\Phi; \Gamma; \Sigma \vdash_{\mathcal{D}} e : \tau$ and $\Phi \vdash_{\mathcal{D}} \langle e, \mu \rangle \longrightarrow \langle e', \mu' \rangle$ and $\Gamma, \Sigma \vdash_{\mathcal{D}} \mu$, then there exist Σ' and τ' such that $\Phi; \Gamma; \Sigma' \vdash_{\mathcal{D}} e' : \tau'$, where $\tau' <: \tau$ and $\Gamma, \Sigma' \vdash_{\mathcal{D}} \mu'$.

7. Implementation in Twelf

To validate our work, we have implemented an interpreter and typechecker for our language in the Twelf proof assistant (Pfenning and Schürmann 1999). We have also implemented a mechanically checked type soundness proof for this implementation following the architecture described in the previous section. Finally, we have implemented all of the example effect disciplines described in Section 4 except the one that relies on special language features for transactions. We have used these example disciplines to typecheck and execute some small programs and have provided mechanically checked implementations of the four monotonicity lemmas required to prove soundness for each discipline. Our Twelf implementation and examples are available at `http://www.cs.ucla.edu/~todd/research/effects.tar.gz`.

7.1 Base Semantics

First we implemented the traditional static and dynamic semantics and type soundness proof for our lambda calculus. We did so by modifying an existing Twelf implementation and type soundness proof for a simple language with mutable references called Fun (Simmons 2005). We replaced Fun's top-level function definitions with first-class (possibly recursive) lambdas, and we introduced let polymorphism as described earlier. We also inherited from Fun a slightly richer language than we presented in our formalism. For example, the language of our Twelf implementation additionally supports 32-bit integers and if-then-else expressions.

7.2 Tag and Privilege Checking

Next we augmented the static and dynamic semantics with tags and tag checking. While our formalism ensures that each value will have exactly one tag, our Twelf implementation allows a value to be tagged multiple times. We then instrumented our static and dynamic semantics with privilege checking, parameterized on the black-box adjust and check functions. For ease of maintenance we implemented static tag and privilege checking as its own pass that follows the existing typechecking pass. Similarly, we implemented dynamic privilege checking as an additional check that occurs after the program takes each ordinary small step.

In order for multiple effect disciplines to easily coexist, each declared privilege class is associated with a *privilege kind*. For example, `read`, `write`, and `alloc` privilege classes might belong to the privilege kind `memcheck`, while the `mayblock` privilege might belong to the privilege kind `userThreads`. Each privilege kind has its own associated check and adjust functions, and the static and dynamic privilege checking judgments are parameterized by the privilege kind.

The check and adjust functions are represented in Twelf as judgment forms (using the *judgments as types* principle (Harper et al. 1993)). A particular effect discipline is specified as a collection of inference rules for these judgments, which are implemented in Twelf in a logic programming style. Each rule pattern matches on a particular check or adjust context, similar to our informal examples in this paper.

We have also implemented a form of the letscope expression as described in Section 4.1. The version in our Twelf implementation has two extra parameters: a privilege kind and an integer. The first parameter allows check and adjust functions to provide different behaviors for letscope based on the privilege kind. The second parameter plays the role of the informal *letscope kinds* we have used in examples, allowing an effect discipline to easily support multiple behaviors for letscope within a privilege kind.

7.3 Modular Type Soundness Proof

Finally, we implemented our proof architecture for type soundness in Twelf. In particular, we have a mechanically checked implementation of the discipline-independent portion of the type soundness proof. This portion is parameterized by the four monotonicity lemmas described earlier. Twelf's typechecker and totality checker verify the correctness and completeness of our Progress and Preservation theorems, subject to the proofs of those four lemmas. Therefore, our proof architecture is correct: the four lemmas are sufficient to ensure type soundness for any effect discipline expressible in our theory.

The lemmas are defined as judgment forms in the typical Twelf style, as described above for the check and adjust functions. The lemmas are proven for a particular privilege kind by defining a set of proof cases, typically one (or more) per inference rule defining the check and adjust functions for that privilege kind.

8. Related Work

Many program analyses and programming disciplines have been formulated as type-and-effect systems, some of which have been

incorporated into languages (e.g., FX (Gifford et al. 1992)). The introduction discusses several examples, including tracking of memory effects, checked exceptions, and locking disciplines for concurrent programs. While these systems hard-code a particular kind of effects and an associated discipline for controlling those effects, our effect system is parameterized by both the set of effects and the associated discipline.

The CQUAL system has a generic typechecker and type inferencer for user-defined type qualifiers in C that is parameterized by a programmer-defined set of qualifiers and associated partial order (Foster et al. 2006). The Clarity system additionally allows programmers to provide explicit type rules for qualifiers (Chin et al. 2005, 2006). The tags in our generic effect system are essentially equivalent to CQUAL-style qualifiers. By themselves, type qualifiers are not expressive enough to encode computational effects: they refer to the *value* of an expression rather than its *evaluation*.

CQUAL allows programmers to declare some qualifiers as *effect qualifiers*. The type system then tracks the effect of each expression, which is a single effect qualifier that is the least upper bound of the effect qualifiers of all subexpressions. The authors used effect qualifiers to track proper initialization in the Linux kernel (Foster et al. 2006). Our theory is more expressive than CQUAL's effect qualifiers in several ways. First, our notion of effects is more general, with privileges dependent on tags. For example, it is not obvious how to encode the standard discipline for $\texttt{read}(\varepsilon)$ and $\texttt{write}(\varepsilon)$ effects in CQUAL. Second, our effect discipline is parameterized by the check and adjust functions that instrument the static semantics in a uniform manner, while CQUAL employs a fixed approach for tracking effects except for a few special cases exposed to programmers for configuration. Finally, we provide a formalized and mechanically checked proof architecture for ensuring type soundness in the presence of arbitrary check and adjust functions.

The capability calculus (Walker et al. 2000) supports a region-based memory management discipline whereby region lifetimes need not be lexically scoped. The associated type system tracks a set of static region capabilities, which indicate both which regions are live and which regions are *unique* and can therefore be safely freed. Our use of privileges as a dual to effects is directly inspired by the capability calculus. We plan to explore ways to incorporate a form of uniqueness in privileges in order to remove our lexical scoping restriction.

The Vault language (DeLine and Fahndrich 2001) generalizes the ideas in the capability calculus to support tracking of user-defined computational effects, and similar functionality can be achieved in the flow-sensitive extension of CQUAL (Foster et al. 2002). Mandelbaum et al. (2003) provide a formal account of a Vault-like type system. These systems support flow-sensitive tracking of effects, which is not possible in our type system. On the other hand, we provide a general approach for specifying arbitrary type-and-effect disciplines on any syntactic context in a language, while effect disciplines in these systems are expressible only through pre- and postconditions on functions.

Others have investigated the type-theoretic foundations of effect systems. Wadler and Thiemann (2003) show how memory effects are naturally captured by a state monad (Moggi 1989) whose type additionally maintains a set of effects, and Filinski (1999) shows how new effects can be introduced in a language as monads that are defined in terms of lower-level effects. Nanevski has pursued the use of modal operators as a logical foundation for effect systems. For example, he shows how the operator $\Box$ of modal necessity and $\Diamond$ of modal possibility can be used to respectively track read and write effects (Nanevski 2003). He also illustrates how $\Box$ provides a logical interpretation of effect consumption and uses this idea to define a type system for exception handling (Nanevski 2005).

These works are more foundational than our work, showing how to encode effects in terms of existing notions like monads and modal-logic type constructors. In these formalisms, each particular effect discipline is encoded separately through specialized typing rules. Our work, on the other hand, uses a standard style of adding effects to a type system, but identifies a general template for creating an effect system in this style that achieves a generic form of soundness. It would be interesting in future work to understand how our results translate to the monadic and modal settings.

9. Conclusion and Future Work

We have presented a generic type-and-effect system. To our knowledge, ours is the first type-and-effect system that is parameterized by both the set of effects to be tracked and the discipline to be statically enforced on these effects. Our notions of check and adjust contexts, derived from the existing notions of redexes and evaluation contexts, provide a generic way of instrumenting any language's type system to support static checking of effect disciplines. Further, we have architected a parameterized type soundness proof for our type-and-effect system which requires only four monotonicity lemmas to be proven about a discipline's check and adjust functions. We have instantiated the approach for a lambda calculus with references and implemented the language, type-and-effect system, and type soundness proof in the Twelf proof assistant.

The generic effect system presented in this paper could be extended in a number of ways to increase expressiveness. As mentioned earlier, we could employ more sophisticated forms of tagging. It might also be useful to consider employing a data structure other than a set to maintain privileges. For example, a sequence of privileges could be useful for disciplines that care about the order in which privileges are acquired. Translating our ideas to the context of the capability calculus (Walker et al. 2000) would provide a form of flow sensitivity for our effect disciplines. Finally, we plan to explore ways to incorporate proofs of extensional notions of soundness. One possible approach is to first prove standard type soundness using our strategy and then to prove a relationship between the instrumented and uninstrumented dynamic semantics.

Our ultimate goal is to use our approach for formalizing a generic effect system as the foundation for a practical framework for enforcing programmer-definable effect disciplines in a full-fledged language like ML or Java. While we have focused on making the check and adjust functions as expressive as possible while retaining type soundness, it may be useful in practice to impose additional structure. For example, restricting the form of the check and adjust functions could simplify privilege inference or ensure privilege and tag monotonicity without requiring proof.

10. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202 and CCF-0545850. Special thanks to Jeff Foster for shepherding the paper, pointers to example effect systems, and helpful comments on an earlier draft. Thanks also to Jens Palsberg for feedback on this paper and to David Walker for discussions that motivated this research direction.

References

Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.

Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 63–74. ACM, 2008.

Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 15–26. ACM Press, 2007.

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *Programming Languages and Systems, 4th Asian Symposium (APLAS 2006)*, pages 114–130. Springer, 2006.

Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming*, 2006.

Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69. ACM Press, 2001.

Linda DeMichiel. *Enterprise JavaBeans Specification, Version 3.0*. SUN Microsystems, 2004.

Andrzej Filinski. Representing layered monads. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 175–188. ACM, 1999.

Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 134–143, 2007.

Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2002.

Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst.*, 28(6):1035–1087, 2006.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38. ACM Press, 1986.

David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, MIT Laboratory for Computer Science, 1992.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *11th Workshop on Hot Topics in Operating Systems*, 2007.

Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 2003.

Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2), 2006.

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS '89: Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 207–218, 2003.

Aleksandar Nanevski. A modal calculus for exception handling. In *Intuitionistic Modal Logics and Applications Workshop*, 2005.

Iulian Neamtiu, Michael Hicks, Jeffrey Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

Rob Simmons. Twelf as a unified framework for language formalization and implementation, 2005. URL `http://www.cs.princeton.edu/~rsimmons/thesis.pdf`. Undergraduate honors thesis, Princeton University.

Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3): 245–271, 1992.

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.