Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming *

Iulian Neamtiu Michael Hicks Jeffrey S. Foster Polyvios Pratikakis

Department of Computer Science
University of Maryland
College Park, MD 20742, USA
{neamtiu,mwh,jfoster,polyvios}@cs.umd.edu

Abstract

This paper presents a generalization of standard effect systems that we call *contextual effects*. A traditional effect system computes the effect of an expression e. Our system additionally computes the effects of the computational context in which e occurs. More specifically, we compute the effect of the computation that has already occurred (the *prior effect*) and the effect of the computation yet to take place (the *future effect*).

Contextual effects are useful when the past or future computation of the program is relevant at various program points. We present two substantial examples. First, we show how prior and future effects can be used to enforce *transactional version consistency* (TVC), a novel correctness property for dynamic software updates. TVC ensures that programmer-designated transactional code blocks appear to execute entirely at the same code version, even if a dynamic update occurs in the middle of the block. Second, we show how future effects can be used in the analysis of multi-threaded programs to find thread-shared locations. This is an essential step in applications such as data race detection.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; C.4 [*Performance of Systems*]: Reliability, availability, and serviceability; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Languages, Reliability, Theory, Verification

Keywords Contextual effects, computation effects, type and effect systems, version consistency, dynamic software updating, data race detection

1. Introduction

Type and effect systems provide a framework for reasoning about the possible side effects of a program's executions. Effects traditionally consider assignments or allocations, but can also track other events, such as functions called or operations performed. A standard type and effect system (Lucassen 1987; Nielson et al. 1999) proves judgments $\varepsilon; \Gamma \vdash e : \tau$, where ε is the effect of the expression e. For many applications, knowing the effect of the *context* in which e appears is also useful. For example, if e includes a security-sensitive operation, then knowing the effect of execution prior to evaluating e could be used to support history-based access control (Abadi and Fournet 2003; Skalka et al. 2007). Conversely, knowing the effect of execution following e could be used for some forms of static garbage collection, e.g., to free initialization functions once initialization is complete (Foster et al. 2006).

The core idea of this paper is a generalization of standard effect systems to compute what we call *contextual effects*. Our contextual effect system proves judgments of the form $\Phi; \Gamma \vdash e : \tau$, where Φ is a tuple $[\alpha; \varepsilon; \omega]$ containing ε, the standard effect of e, and α and ω, the *prior effect* and *future effect*, respectively, of e's context. For example, in an application $e_1\ e_2$, the prior effect of e_2 includes the effect of e_1, and likewise the future effect of e_1 includes the effect of e_2. The first contribution of this paper is a system for computing contextual effects and a proof that the system is sound (Section 2).

The inspiration for contextual effects arose from experience with two research projects. The first considers *dynamic software updating* (DSU), a technique by which running software can be updated on-the-fly with new code and data. In prior work we formalized and implemented Ginseng, a compiler and tool suite that supports DSU for C programs (Neamtiu et al. 2006; Stoyle et al. 2007). Updates are at the granularity of function calls, meaning that following an update, active code continues with the old version, while subsequent calls are to the new version. A key consideration for update correctness is timing, since, applied at the wrong time, the changes due to an update could conflict with processing in flight. For example, suppose the original code defined function `h() { f(); g(); }`, but then is changed to move the call to `g` from `h` to `f`, i.e., `h() { f(); }` and `f() { ...; g(); }`. Suppose the update occurs just prior to the original pair of calls. The call to `f` will be to the new version that calls `g`, but then returns to its caller, the *old* `h`, which then calls `g` again, potentially leading to an error.

We address this problem with a novel correctness property called *transactional version consistency* (TVC), the second contribution of this paper. In this approach, programmers designate blocks of code as *transactions* whose execution must always be attributable to a single program version. Thus an update is only allowed within a transaction if the transaction's execution still appears due to either the old or new program version. The problematic update above would be ruled out by placing the body of `h` in a transaction. We formalize this idea in a small language Proteus-tx, which extends our prior dynamic updating calculus Proteus (Stoyle et al. 2007) with transactions. Proteus-tx's type system uses con-

* Slightly revised, January 2008. This paper differs slightly from the version appearing in the official proceedings in its formulation of the proof of contextual effect soundness.

POPL'08, January 7–12, 2007, San Francisco, California, USA.

textual effects at candidate update points within a transaction to determine what updates are safe to apply. We have proven that Proteus-tx enforces transactional version consistency, and we have developed a preliminary implementation for C. In our prior work, we manually specified that updates could occur at one or two *quiescent* program points, typically at the conclusion of event processing loops (Neamtiu et al. 2006). Using these quiescent points to identify transaction boundaries, we discovered many additional version-consistent update points within transactions, which can be used to reduce the time from when an update becomes available to when it is applied (Section 3).

The second research effort from which contextual effects arose is Locksmith (Hicks et al. 2006; Pratikakis et al. 2006), a tool that can automatically detect data races in C programs. We found that we could use contextual effects to compute what memory locations are shared among threads in a multi-threaded program. The basic idea is that thread-shared locations are exactly those in the intersection of the standard effect of a child thread and the *future* effect of the parent thread at the point the child is created. Locations accessed prior to creating the child but not afterward are not shared. The final contribution of this paper is a presentation of our algorithm for computing shared locations as an extension to contextual effects. We used this analysis to compute shared locations in a range of C programs. Our algorithm finds that many locations are thread-local, and hence cannot have races (Section 4).

We believe that contextual effects have many other uses, in particular any application in which the past or future computation of the program is relevant at various program points. The remainder of the paper presents our core contextual type and effect system, followed by its application to transactional version consistency and thread sharing analysis.

2. Contextual effects

To begin, we present a core contextual effect system for a simple imperative calculus and prove it sound. Figure 1 presents our source language, which contains expressions e that consist of values v (integers or functions); variables; let binding; function application; and the conditional if0, which tests its integer-valued guard against 0. Our language also includes updatable references $\mathsf{ref}^L\ e$ along with dereference and assignment. Here we annotate each syntactic occurrence of ref with a *label* L, which serves as the abstract name for the locations allocated at that program point. We use labels to define contextual effects. For simplicity we do not model recursive functions directly in our language, but they can be encoded using references.

Our system uses two kinds of effect information. An *effect*, written α, ε, or ω, is a possibly-empty set of labels, and may be 1, the set of all labels. A *contextual effect*, written Φ, is a tuple $[\alpha;\varepsilon;\omega]$. In our system, if e' is a subexpression of e, and e' has contextual effect $[\alpha;\varepsilon;\omega]$, then

- The *current effect* ε is the effect of evaluating e' itself.
- The *prior effect* α is the effect of evaluating e up until we begin evaluating e'.
- The *future effect* ω is the effect of the remainder of the evaluation of e after e' is fully evaluated.

Thus ε is the effect of e' itself, and $\alpha \cup \omega$ is the effect of the context in which e' appears—and therefore $\alpha \cup \varepsilon \cup \omega$ contains all locations accessed during the entire reduction of e.

To make contextual effects easier to work with, we introduce some shorthand. We write Φ^α, Φ^ε, and Φ^ω for the prior, current, and future effect components, respectively, of Φ. We also write $\Phi_\emptyset$ for the empty effect $[1;\emptyset;1]$—by subsumption, discussed below, an expression with this effect may appear in any context. In what follows, we refer to contextual effects simply as *effects*, for brevity.

Expressions	e	::=	$v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e\ e$
		|	$\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		|	$\mathsf{ref}^L\ e \mid !\,e \mid e := e$
Values	v	::=	$n \mid \lambda x.e$
Effects	$\alpha, \varepsilon, \omega$	::=	$\emptyset \mid 1 \mid \{L\} \mid \varepsilon \cup \varepsilon$
Contextual Effs.	Φ	::=	$[\alpha;\varepsilon;\omega]$
Types	τ	::=	$int \mid ref^{\varepsilon}\ \tau \mid \tau \longrightarrow^{\Phi} \tau$
Labels	L		

Figure 1. Contextual effects source language

2.1 Typing

We now present a type and effect system to determine the contextual effect of every subexpression in a program. Types τ, listed at the end of Figure 1, include the integer type int; reference types $ref^{\varepsilon}\ \tau$, which denote a reference to memory of type τ where the reference itself is annotated with a label $L \in \varepsilon$; and function types $\tau \longrightarrow^{\Phi} \tau'$, where τ and τ' are the domain and range types, respectively, and the function has contextual effect Φ.

Figure 2 presents our contextual type and effect system. The rules prove judgments of the form $\Phi; \Gamma \vdash e : \tau$, meaning in type environment Γ, expression e has type τ and contextual effect Φ. The first two rules, (TINT) and (TVAR), assign the expected types and the empty effect, since values have no effect.

(TLET) types subexpressions e_1 and e_2, which have effects Φ_1 and Φ_2, respectively, and requires that these effects combine to form Φ, the effect of the entire expression. We use a call-by-value semantics, and hence the effect of the let should be the effect of e_1 followed by the effect of e_2. We specify the sequencing of effects with the combinator $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$, defined by (XFLOW-CTXT) in the middle part of Figure 2. Since e_1 happens before e_2, this rule requires that the future effect of e_1 be $\varepsilon_2 \cup \omega_2$, i.e., everything that happens during the evaluation of e_2, captured by ε_2, plus everything that happens after, captured by ω_2. Similarly, the past effect of e_2 must be $\varepsilon_1 \cup \alpha_1$, since e_2 happens just after e_1. Lastly, the effect Φ of the entire expression has α_1 as its prior effect, since e_1 happens first; ω_2 as its future effect, since e_2 happens last; and $\varepsilon_1 \cup \varepsilon_2$ as its current effect, since both e_1 and e_2 are evaluated. We write $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ as shorthand for $(\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi') \wedge (\Phi' \rhd \Phi_3 \hookrightarrow \Phi)$.

(TIF) requires that its branches have the same type τ and effect Φ_2, which can be achieved with subsumption (below), and uses $\rhd$ to specify that Φ_1, the effect of the guard, occurs before either branch. (TREF) types memory allocation, which has no effect but places the annotation L into a singleton effect $\{L\}$ on the output type. This singleton effect can be increased as necessary by using subsumption.

(TDEREF) types the dereference of a memory location of type $ref^{\varepsilon}\ \tau$. In a standard effect system, the effect of $!\,e$ is the effect of e plus the effect ε of accessing the pointed-to memory. Here, the effect of e is captured by Φ_1, and because the dereference occurs after e is evaluated, (TDEREF) puts Φ_1 in sequence just before some Φ_2 such that Φ_2's current effect is ε. Therefore by (XFLOW-CTXT), Φ^{ε} is $\Phi_1^{\varepsilon} \cup \varepsilon$, and e's future effect Φ_1^{ω} must include ε and the future effect of Φ_2. On the other hand, Φ_2^{ω} is unconstrained by this rule, but it will be constrained by the context, assuming the dereference is followed by another expression. (TASSIGN) is similar to (TDEREF), combining the effects Φ_1 and Φ_2 of its subexpressions with a Φ_3 whose current effect is ε.

(TLAM) types the function body e and sets the effect on the function arrow to be the effect of e. The expression as a whole has

Typing

$$(\textsc{TInt})\ \frac{}{\Phi_\emptyset; \Gamma \vdash n : int} \qquad (\textsc{TVar})\ \frac{\Gamma(x) = \tau}{\Phi_\emptyset; \Gamma \vdash x : \tau}$$

$$(\textsc{TLet})\ \frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \quad \Phi_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$(\textsc{TIf})\ \frac{\Phi_1; \Gamma \vdash e_1 : int \quad \Phi_2; \Gamma \vdash e_2 : \tau \quad \Phi_2; \Gamma \vdash e_3 : \tau \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$(\textsc{TRef})\ \frac{\Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}^L\ e : ref^{\{L\}}\ \tau}$$

$$(\textsc{TDeref})\ \frac{\Phi_1; \Gamma \vdash e : ref^{\varepsilon}\ \tau \quad \Phi_2^{\varepsilon} = \varepsilon \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash !\,e : \tau}$$

$$(\textsc{TAssign})\ \frac{\Phi_1; \Gamma \vdash e_1 : ref^{\varepsilon}\ \tau \quad \Phi_2; \Gamma \vdash e_2 : \tau \quad \Phi_3^{\varepsilon} = \varepsilon \quad \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi}{\Phi; \Gamma \vdash e_1 := e_2 : \tau}$$

$$(\textsc{TLam})\ \frac{\Phi; \Gamma, x : \tau' \vdash e : \tau}{\Phi_\emptyset; \Gamma \vdash \lambda x.e : \tau' \longrightarrow^{\Phi} \tau}$$

$$(\textsc{TApp})\ \frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \quad \Phi_2; \Gamma \vdash e_2 : \tau_1 \quad \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\textsc{TSub})\ \frac{\Phi'; \Gamma \vdash e : \tau' \quad \tau' \leq \tau \quad \Phi' \leq \Phi}{\Phi; \Gamma \vdash e : \tau}$$

Effect combinator

$$(\textsc{XFlow-Ctxt})\ \frac{\Phi_1 = [\alpha_1; \varepsilon_1; (\varepsilon_2 \cup \omega_2)] \quad \Phi_2 = [(\varepsilon_1 \cup \alpha_1); \varepsilon_2; \omega_2] \quad \Phi = [\alpha_1; (\varepsilon_1 \cup \varepsilon_2); \omega_2]}{\Phi_1 \rhd \Phi_2 \ \hookrightarrow\ \Phi}$$

Subtyping

$$(\textsc{SInt})\ \frac{}{int \leq int} \qquad (\textsc{SRef})\ \frac{\tau \leq \tau' \quad \tau' \leq \tau \quad \varepsilon \subseteq \varepsilon'}{ref^{\varepsilon}\ \tau \leq ref^{\varepsilon'}\ \tau'}$$

$$(\textsc{SFun})\ \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad \Phi \leq \Phi'}{\tau_1 \longrightarrow^{\Phi} \tau_2 \leq \tau_1' \longrightarrow^{\Phi'} \tau_2'}$$

$$(\textsc{SCtxt})\ \frac{\alpha_2 \subseteq \alpha_1 \quad \varepsilon_1 \subseteq \varepsilon_2 \quad \omega_2 \subseteq \omega_1}{[\alpha_1; \varepsilon_1; \omega_1] \leq [\alpha_2; \varepsilon_2; \omega_2]}$$

Figure 2. Contextual effects type system

no effect, since the function produces no run-time effects until it is actually called. (TApp) types function application, which combines Φ_1, the effect of e_1, with Φ_2, the effect of e_2, and Φ_f, the effect of the function.

The last rule in our system, (TSub), introduces subsumption on types and effects. The judgments $\tau' \leq \tau$ and $\Phi' \leq \Phi$ are defined at the bottom of Figure 2. (SInt), (SRef), and (SFun) are standard, with the usual co- and contravariance where appropriate. (SCtxt) defines subsumption on effects, which is covariant in the current effect, as expected, and contravariant in both the prior and future effects. To understand the contravariance, first consider an expression e with future effect ω_1. Since future effects should

Values	v	$::=$	$\ldots \mid r_L$
Heaps	H	$::=$	$\emptyset \mid H, r \mapsto v$
Environments	Γ	$::=$	$\emptyset \mid \Gamma, x : \tau \mid \Gamma, r : \tau$

$$[\textsc{Id}]\ \frac{}{\langle \alpha, \omega, H, v\rangle \longrightarrow_\emptyset \langle \alpha, \omega, H, v\rangle}$$

$$[\textsc{Call}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e\rangle \quad \langle \alpha_1, \omega_1, H_1, e_2\rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2\rangle \quad \langle \alpha_2, \omega_2, H_2, e[x \mapsto v_2]\rangle \longrightarrow_{\varepsilon_3} \langle \alpha', \omega', H', v\rangle}{\langle \alpha, \omega, H, e_1\ e_2\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v\rangle}$$

$$[\textsc{Ref}]\ \frac{\langle \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', v\rangle \quad r \notin \mathrm{dom}(H')}{\langle \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

$$[\textsc{Deref}]\ \frac{\langle \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L\rangle \quad r \in \mathrm{dom}(H')}{\langle \alpha, \omega, H, !\,e\rangle \longrightarrow_{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

$$[\textsc{Assign}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L\rangle \quad \langle \alpha_1, \omega_1, H_1, e_2\rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v\rangle}{\langle \alpha, \omega, H, e_1 := e_2\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v\rangle}$$

$$[\textsc{If-T}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \quad v_1 = 0 \quad \langle \alpha_1, \omega_1, H_1, e_2\rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v\rangle}$$

$$[\textsc{If-F}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \quad v_1 = n \neq 0 \quad \langle \alpha_1, \omega_1, H_1, e_3\rangle \longrightarrow_{\varepsilon_3} \langle \alpha_3, \omega_3, H_3, v\rangle}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_3} \langle \alpha_3, \omega_3, H_3, v\rangle}$$

$$[\textsc{Let}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1\rangle \quad \langle \alpha_1, \omega_1, H_1, e_2[x \mapsto v_1]\rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2\rangle}{\langle \alpha, \omega, H, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2\rangle}$$

Figure 3. Contextual effects operational semantics (partial)

soundly approximate (i.e., be a superset of) the locations that may be accessed in the future, we can use e in any context that accesses *at most* locations in ω_1. Similarly, since past effects approximate locations that were accessed in the past, we can use e in any context that accessed at most locations in α_1.

2.2 Semantics and Soundness

We now prove that our contextual effect system is sound. The top of Figure 3 gives some basic definitions needed for our operational semantics. We extend values v to include the form r_L, which is a run-time heap location r annotated with label L. We need to track labels through our operational semantics to formulate and prove soundness, but these labels need not exist at run-time. We define heaps H to be maps from locations to values. Finally, we extend typing environments Γ to assign types to heap locations.

The bottom part of Figure 3 defines a big-step operational semantics for our language. Reductions operate on *configurations* $\langle \alpha, \omega, H, e\rangle$, where α and ω are the sets of locations accessed before and after evaluation of e, respectively; H is the heap; and e is the expression to be evaluated. Evaluations have the form

$$\langle \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', R\rangle$$

where ε is the effect of evaluating e and R is the result of reduction, either a value v or $\mathbf{err}$, indicating evaluation failed. Intuitively, α records a trace of what has happened in the past, and ω is a *capability* describing what locations may be accessed in the future. As evaluation proceeds, labels move from the capability ω to the trace α.

The reduction rules are straightforward. [ID] reduces a value to itself without changing the state or the effects. [CALL] evaluates the first expression to a function, the second expression to a value, and then the function body with the formal argument replaced by the actual argument. [REF] generates a fresh location r, which is bound in the heap to v and evaluates to r_L. [DEREF] reads the location r in the heap and adds L to the standard evaluation effect. This rule requires that the future effect after evaluating e have the form $\omega' \cup \{L\}$, i.e., L must be in the capability after evaluating e, but prior to dereferencing the result. Then L is added to α' in the the output configuration of the rule. Notice that $\omega' \cup \{L\}$ is a standard union, and so L may also be in ω'. This allows the same location can be accessed multiple times. [ASSIGN] behaves similarly to [DEREF].

Lastly, [IF-T] and [IF-F] give the two cases for conditionals, and [LET] binds x to the result of evaluating e_1 inside of e_2. Our semantics also includes rules (not shown) that produce **err** when the program tries to access a location that is not in the input capability, or when values are used at the wrong type.

Given this operational semantics, we can now prove that the contextual effect system in Figure 2 is sound. Due to limited space, we only state our lemmas and theorems. The proofs can be found in full in our companion technical report (Neamtiu et al. 2007).

We begin with a standard definition of heap typing.

Definition 2.1 (Heap Typing)**.** *We say heap H is well-typed under Γ, written $\Gamma \vdash H$, if $\mathrm{dom}(\Gamma) = \mathrm{dom}(H)$ and if for every $r \in \mathrm{dom}(H)$, we have $\Phi_\emptyset; \Gamma \vdash H(r) : \Gamma(r)$.*

Given this definition, we show the standard effect soundness theorem, which states that the program does not go wrong and that the standard effect Φ^ε captures the effect of evaluation.

Theorem 2.2 (Standard Effect Soundness)**.** *If $\Phi; \Gamma \vdash e : \tau$ and $\Gamma \vdash H$ and $\langle 1, 1, H, e\rangle \longrightarrow_\varepsilon \langle 1, 1, H', R\rangle$, then there is a $\Gamma' \supseteq \Gamma$ such that R is a value v for which $\Phi_0; \Gamma' \vdash v : \tau$ where $\Gamma' \vdash H'$ and $\varepsilon \subseteq \Phi^\varepsilon$.*

Next, we show the operational semantics is adequate, in that it moves effects from the future to the past during evaluation.

Lemma 2.3 (Adequacy of Semantics)**.** *If $\langle \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle \alpha', \omega', H', v\rangle$ then $\alpha' = \alpha \cup \varepsilon$ and $\omega = \omega' \cup \varepsilon$.*

Next we must define what it means for the statically-ascribed contextual effects of some expression e to be sound with respect to the effects of e's evaluation. Suppose that e_p is a program that is well-typed according to typing derivation $\mathcal{T}$ and evaluates to some value v as witnessed by an evaluation derivation D. Observe that each term e_1 that is reduced in a subderivation of D is either a subterm of e_p, or is derived from a subterm e_2 of e_p via reduction; in the latter case it is sound to give e_1 the same type and effect that e_2 has in $\mathcal{T}$. To reason about the soundness of the effects, therefore, we must track the static effect of expression e_2 as it is evaluated.

We do this by defining a new *typed operational semantics* that extends standard configurations with a typing derivation of the term in that configuration. The key property of this semantics is that it preserves the effect Φ of a term throughout its evaluation, and we prove that given standard evaluation and typing derivations of the original program, we can always construct a corresponding typed operational semantics derivation.

Finally, we prove that given a typed operational semantics derivation, the effect Φ in the typing in each configuration conservatively approximates the actual prior and future effect.

Theorem 2.4 (Prior and Future Effect Soundness)**.** *If*

$$E :: \langle T, \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle T_v, \alpha', \omega', H', v\rangle$$

where $T :: \Phi; \Gamma \vdash e : \tau$, $\alpha \subseteq \Phi^\alpha$ and $\omega' \subseteq \Phi^\omega$ then for all sub-derivations E_i of E, $E_i :: \langle T_i, \alpha_i, \omega_i, H_i, e_i\rangle \longrightarrow_\varepsilon \langle T_{v_i}, \alpha_i', \omega_i', H_i', v_i\rangle$ where $T_i :: \Phi_i; \Gamma_i \vdash e_i : \tau_i$, it will hold that $\alpha_i \subseteq \Phi_i^\alpha$ and $\omega_i' \subseteq \Phi_i^\omega$.

The proof of the above theorem is by induction on the derivation, starting at the root and working towards the leaves, and relying on Theorem 2.2 and Lemma 2.3.

2.3 Contextual Effect Inference

The typing rules in Figure 2 form a checking system, but we would prefer to infer effect annotations rather than require the programmer to provide them. Here we sketch the inference process, which is straightforward and uses standard constraint-based techniques.

We change the rules in Figure 2 into inference rules by making three modifications. First, we make the rules syntax-driven by integrating (TSUB) into the other rules (Mitchell 1991); second, we add variables χ to represent as-yet-unknown effects; and third, we replace implicit equalities with explicit equality constraints.

The resulting rules are mostly as expected, with one interesting difference for (TAPP). We might expect inlining subsumption into (TAPP) to yield the following rule:

$$(*)\ \frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1' \qquad \tau_1' \leq \tau_1 \qquad \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

However, this would cause the inferred Φ_f effect to be larger than necessary if there are multiple calls to the same function. For example, consider the following code, where f is some one-argument function, x, y, and z are references, and A and B identify two program points:

```
( if0 ... then /*A*/ (f 1; !x) else /*B*/ (f 2; !y)); !z
```

If we used rule (*), then from branch A, we would require $\{\mathsf{x}, \mathsf{z}\} \subseteq \Phi_f^\omega$, and from branch B, we would require $\{\mathsf{y}, \mathsf{z}\} \subseteq \Phi_f^\omega$, where Φ_f is the effect of function f. Putting these together, we would thus have $\Phi_f^\omega = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$. This result is appropriate, since any of those locations may be accessed after some call to f. However, consider the future effect Φ_A^ω at program point A. By (XFLOW-CTXT), Φ_A^ω would contain Φ_f^ω, and yet y will not be accessed once we reach A, since that access is on another branch. The analogous problem happens at program point B, whose future effect is polluted by x.

The problem is that our effect system conflates all calls to f. One solution would be to add Hindley-Milner style parametric polymorphism, which would address this particular example. However, even with Hindley-Milner polymorphism we would suffer the same problem at indirect function calls, e.g., in C, calls through function pointers would be monomorphic.

The solution is to notice that inlining subsumption into (TAPP) should not yield (*), but instead results in the following rule:

$$(\text{TAPP}')\ \frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1' \qquad \Phi_f \leq \Phi_f' \qquad \tau_1' \leq \tau_1 \qquad \Phi_f' \text{ fresh} \qquad \Phi_1 \rhd \Phi_2 \rhd \Phi_f' \hookrightarrow \Phi}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

Applied to the above example, (TAPP′) results in two constraints on the future effect of Φ_f:

$$\Phi_f^\omega \supseteq \Phi_{fA}^\omega = \{\mathsf{x}, \mathsf{z}\} \qquad \Phi_f^\omega \supseteq \Phi_{fB}^\omega = \{\mathsf{y}, \mathsf{z}\}$$

Here Φ_{fA} and Φ_{fB} are the fresh function effects at the call to f in A and B, respectively. Notice that we have $\Phi_f^\omega = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$, as before, since f is called in both contexts. But now Φ_{fA}^ω need not contain y, and Φ_{fB} need not contain x. Thus with (TAPP′), a function's effect summarizes all of its contexts, but does not cause the prior and future effects from different contexts to pollute each other.

```
1  int main() { ...
2   conn = accept_loop();
3    init_log ();
4    handle_sess (conn);
5   ... }
6
7  void handle_sess (conn) {
8    ...
9    while (1) {
10    tx {
11     cmd = get(conn.fd);
12     if (cmd == "LIST") {
13       handle_list (cmd);
14     } else ...
15     /* new: log_log () */
16    } /* end tx */
17   }
18 }
```

```
19 conn accept_loop() {
20   ...
21   while (1) {
22    tx {
23     addr = accept();
24     if (! fork ())
25      /* child */
26      return conn;
27     }
28    } /* end tx */
29   }
30 }
31
32 void handle_list (cmd) {
33    start_log_entry ();
34    ...
35    log_log (); /* old */
36 }
```

Figure 4. High-level structure of the `vsftpd` server

To perform type inference, we apply our inference rules, viewing them as generating the *constraints* C in their hypotheses, given by the following grammar:

$$C ::= \tau \leq \tau' \mid \Phi \leq \Phi' \mid \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$$

We can then solve the constraints by performing graph reachability to find, for each variable χ, the set of base effects $\{L\}$ or 1 that reach it. In practice, these constraints can be solved very efficiently using a toolkit such as Banshee (Kodumal and Aiken 2005), which also can be used to build a context-sensitive version of inference using context-free language reachability (Pratikakis et al. 2006).

3. Transactional Version Consistency for DSU

In prior work we developed Ginseng, a dynamic software updating (DSU) system for C programs (Neamtiu et al. 2006; Stoyle et al. 2007). To use Ginseng, programmers construct *dynamic patches* that contain new or updated functions and data. Ginseng applies these patches to the running program, and then subsequent function calls and data accesses apply to the new versions. A key novelty of Ginseng is that it ensures dynamic updates do not violate type safety while still allowing patches to change function and data types (Neamtiu et al. 2006; Stoyle et al. 2007), which is often necessary (Baumann et al. 2007; Neamtiu et al. 2005).

However, while type safety is important, it is only one aspect of program correctness. In this section we introduce *transactional version consistency* (TVC), a new property that lets programmers reason more easily about the safety of updates, and show how contextual effects can enforce TVC.

To illustrate the problem with an example, consider Figure 4, which sketches the structure of `vsftpd`, an FTP server which we have dynamically updated using Ginseng (Neamtiu et al. 2006). For the moment, ignore the tx{} annotations in the code. In this program, main (lines 1–5) first calls accept_loop, whose body (lines 19–30) contains an infinite loop. Each iteration of the loop accepts a connection request (line 23) and forks a child process to handle the requested session (line 24). If forking succeeds, the child returns (line 26) to main, which initializes the session's log (line 3, function not shown) and processes the session (line 4) by calling handle_sess (lines 7–18). In turn, handle_sess processes client commands until the session is closed. We show one command processor, handle_list (lines 32–36), which is called on line 13. This function creates a log entry (line 33) that is flushed when processing is finished (line 35).

Consider the following update, inspired by actual changes to `vsftpd`. Rather than flush the log entry at the end of each command processing function (like handle_list), the update changes the code to do so in handle_sess instead, e.g., the call to log_log on line 35 is moved to line 15. Once this update is applied, the next iteration of the loop will execute the new version of the code.[1]

This update is type safe, but notice that its behavior may be incorrect depending on where it occurs. Suppose Ginseng applies the update after line 15, meaning that future calls to handle_list and the handle_sess loop body go to the new version. Then everything happens correctly: we have just called the old version of handle_list, so logging occurred, and we finish executing the old version of the handle_sess loop, and thus do not call log_log. The next iteration uses all new code, and so logging continues as usual.

In contrast, suppose the update was applied at line 11. At this point we are executing the old handle_sess, but we will call the *new* handle_list, which no longer calls log_log. Moreover, handle_list returns to its caller, i.e., the old handle_sess, which also does not call log_log. Thus no log entry is produced for the command.

This example reveals a violation of what we call *version consistency*: due to the timing of the update, we execute part old code and part new code, and the overall result is inconsistent. It is easy to construct other problematic updates that violate program semantics in various ways depending where the update is applied.

Prior to the current work, we maintained version consistency in Ginseng by only allowing updates at programmer-specified positions. In `vsftpd`, we chose update points on lines 16 and 28, at the end of the connection acceptance and request processing code, just prior to the next iteration of the loop. However, this approach of having one or two well-placed update points may result in less *update availability*, meaning it may take longer for updates to occur once submitted to the program. While this may be reasonable for single-threaded programs with low-latency event processing code, in a multi-threaded program—like an operating system or embedded system—the problem could be quite serious. In particular, we would have to require that all threads reach their update points simultaneously for an update to take place. Since this is unlikely to happen naturally, we could treat update points as synchronization barriers, waiting until all threads have blocked before applying the update (Stoyle et al. 2007). However, this will degrade service while waiting for threads to block, and at worst could introduce deadlock.

3.1 Version Consistency via Contextual Effects

To increase update availability while ensuring version consistency, we draw inspiration from recent work on making multi-threaded programming simpler by using *atomic blocks* (Harris and Fraser 2003). The key benefit of atomic blocks is that the programmer can consider them in isolation, because the language guarantees they will be serializable with respect to the rest of the computation.

Analogously, we allow programmers to specify *transactions* within their code, and we enforce *transactional version consistency* (TVC), meaning that transactions execute as if they were entirely the old version or entirely the new version, no matter where an update actually occurs. For example, in Figure 4, we have marked the bodies of the two event-processing loops as transactions by placing them within tx{} blocks (lines 10 and 22), and thus the programmer can consider updates as occurring at lines 10 or 16 in handle_sess or lines 22 or 28 in accept_loop, which are equivalent to the manually specified locations we used earlier.

[1] Updates to functions in Ginseng take effect the next time the function is called. To update the body of a long-running loop, programmers specify the loop should be treated as a tail-recursive function, which can then be

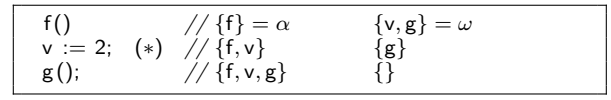

Figure 5. A sample transaction

We can use contextual effects to enforce TVC while still allowing updates to occur within transactions. To see how, consider the code snippet in Figure 5. Comments show the prior and future effects after each statement, where we include both locations accessed and functions called in effects. Suppose we are running this code within a transaction and have just executed line 2, marked with a star, when an update becomes available. At this point the transaction has called f and written to v ($\alpha = \{\mathsf{f}, \mathsf{v}\}$), and will call g in the future ($\omega = \{\mathsf{g}\}$). Consider the possible outcomes depending on the effect ε of the update, where an update's effect consists of the set of functions and global variables that it adds or changes.

1. If $\alpha \cap \varepsilon = \emptyset$, e.g., the update only modifies g ($\varepsilon = \{\mathsf{g}\}$), then the update is safe. In our example, this would result in using the new versions of f and v, which are the same as the old versions, with the new version of g. It is as if we applied the update at the beginning of the transaction, and the entire transaction runs under the new version.
2. If $\omega \cap \varepsilon = \emptyset$, e.g., the update only modifies f ($\varepsilon = \{\mathsf{f}\}$), then the update is also safe. In our example, this would result in using the old versions of f, v, and g. Notice that although we have called f ($\alpha \cap \varepsilon \neq \emptyset$), we never call it again after the update. It is as if we applied the update at the end of the transaction, and the whole transaction runs under the old version.
3. If $\alpha \cap \varepsilon \neq \emptyset$ and $\omega \cap \varepsilon \neq \emptyset$, e.g., the update modifies f and g, then we cannot apply the update, because that would result in using some old code (f) and some new code (g).

Putting these together, we can apply an update with effect ε anywhere in a transaction such that $\alpha \cap \varepsilon = \emptyset$ or $\omega \cap \varepsilon = \emptyset$, where α and ω are the prior and future effects at the update point.

Our transactions are enforced using statically-determined contextual effects as defined in Section 2. Alternatively, we could log a transaction's actual effects at run-time and use these to check version consistency. Similar to common implementations of transactional memory (Harris and Fraser 2003; Herlihy and Moss 1993), we could optimistically apply an update when it becomes available, and then commit it or roll it back at the end of a transaction depending how the transaction's effect intersects with the effect of the update. This might improve update availability, since contextual effects are conservative approximations of the actual run-time effects. Nevertheless, we believe the benefits of a static approach outweigh the drawbacks. First, the static approach does not pay the overhead of logging, which is unnecessary most of the time since updates are infrequent. Second, there are no restrictions on the use of I/O within transactions; notice that network I/O occurs in both transactions in Figure 4. The optimistic approach generally must avoid I/O in transactions to properly roll back if version consistency is violated.

3.2 Syntax

Figure 6 presents Proteus-tx, which extends the language from Section 2 to model transactionally version-consistent dynamic updates, adapting the ideas of Proteus, our prior dynamic updating calculus (Stoyle et al. 2007). A Proteus-tx program is a definition d,

updated using the normal function update mechanism. Thus the update takes effect at the next iteration of the loop (Neamtiu et al. 2006).

Definitions	d	::=	$\mathsf{main}\ e$
		|	$\mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d$
		|	$\mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d$
Expressions	e	::=	$v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e\ e$
		|	$\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		|	$\mathsf{ref}\ e \mid !\,e \mid e := e$
		|	$\mathsf{tx}\ e \mid \mathsf{update}^{\alpha,\omega}$
Values	v	::=	$n \mid \mathsf{z}$
Effects	$\alpha, \omega, \varepsilon$	::=	$\emptyset \mid 1 \mid \{\mathsf{z}\} \mid \varepsilon \cup \varepsilon$
Global symbols	$\mathsf{f}, \mathsf{g}, \mathsf{z}$	$\in$	$GSym$
Dynamic updates	upd	::=	$\{chg, add\}$
Additions	add	$\in$	$GSym \rightharpoonup (\tau \times b)$
Changes	chg	$\in$	$GSym \rightharpoonup (\tau \times b)$
Bindings	b	::=	$v \mid \lambda x.e$

Figure 6. Proteus-tx syntax, effects, and updates

which consists of an expression $\mathsf{main}\ e$, possibly preceded by definitions of *global symbols*, written f, g, or z and drawn from a set $GSym$. The definition $\mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d$ binds mutable variable g to v within the scope of d, and the definition $\mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d$ binds f to a (possibly-recursive) function with formal parameter x and body e.

Expressions e in Proteus-tx have several small differences from the language of Figure 1. We add global symbols z to the set of values v. We also remove anonymous lambda bindings to keep things simpler, for reasons discussed in Section 3.4. To mark transactions, we add a form $\mathsf{tx}\ e$ for a transaction whose body is e.

We specify program points where dynamic updates may occur with the term $\mathsf{update}^{\alpha,\omega}$, where the annotations α and ω specify the prior and future effects at the update point, respectively. When evaluation reaches $\mathsf{update}^{\alpha,\omega}$, an available update is applied if its contents do not conflict with the future and prior effect annotations; otherwise evaluation proceeds without updating.

A *dynamic update* upd consists of a pair of partial functions chg and add that describe the changes and additions, respectively, of global symbol bindings. The range of these functions is pairs (τ, b), where b is the new or replacement value (which may be a function $\lambda x.e$) and τ is its type. Note that Proteus-tx disallows type-altering updates, though Section 3.6 explains how they can be supported by employing ideas from our earlier work (Stoyle et al. 2007). Also, although our implementation allows state initialization functions, for simplicity we do not model them in Proteus-tx.

Finally, effects in Proteus-tx consist of sets of global symbol names z, which represent either a dereference of or assignment to z (if it is a variable) or a call to z (if it is a function name). Because updates in Proteus-tx can only change global symbols (and do not read or write through their contents), we can ignore the effects of the latter (we use syntax $\mathsf{ref}\ e$ instead of $\mathsf{ref}^L\ e$).

3.3 Typing

Figure 7 extends the core contextual effect typing rules from Figure 2 to Proteus-tx. The first three rules define the judgment $\Gamma \vdash d$, meaning definition d is well-typed in environment Γ. (TMAIN) types e in Γ, where e may have any effect and any type. (TDVAR) types the value v, which has the empty effect (since it is a value), and then types d with g bound to a reference to v labeled with effect $\{\mathsf{g}\}$. The last definition rule, (TDFUN), constructs a new environment Γ' that extends Γ with a binding of f to the function's type. The function body e is type checked in Γ', to allow for recursive calls. This rule also requires that f appear in all components of the function's effect Φ, written $\{\mathsf{f}\} \subseteq \Phi$. We add f to the prior effect because f must have been called for its entry to be reached. We add f to the current effect so that it is included in the effect at a call site.

$$(\text{TMAIN})\ \frac{\Phi; \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{main}\ e}$$

$$(\text{TDVAR})\ \frac{\Phi_\emptyset; \Gamma \vdash v : \tau \quad \Gamma, \mathsf{g} : \mathit{ref}^{\{\mathsf{g}\}}\ \tau \vdash d}{\Gamma \vdash \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d}$$

$$(\text{TDFUN})\ \frac{\begin{array}{c}\Gamma' = \Gamma, \mathsf{f} : \tau \longrightarrow^{\Phi} \tau' \quad \Phi; \Gamma', x : \tau \vdash e : \tau' \\ \Gamma' \vdash d \qquad\qquad \{\mathsf{f}\} \subseteq \Phi\end{array}}{\Gamma \vdash \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d}$$

$$(\text{TGVAR})\ \frac{\Gamma(\mathsf{f}) = \tau}{\Phi_\emptyset; \Gamma \vdash \mathsf{f} : \tau}$$

$$(\text{TUPDATE})\ \frac{\Phi^\alpha \subseteq \alpha' \quad \Phi^\omega \subseteq \omega'}{\Phi; \Gamma \vdash \mathsf{update}^{\alpha', \omega'} : \mathit{int}}$$

$$(\text{TTRANSACT})\ \frac{\Phi_1; \Gamma \vdash e : \tau \quad \Phi^\alpha \subseteq \Phi_1^\alpha \quad \Phi^\omega \subseteq \Phi_1^\omega}{\Phi; \Gamma \vdash \mathsf{tx}\ e : \tau}$$

Figure 7. Proteus-tx typing (extends Figure 2)

Lastly, we add f to the future effect because f is on the call stack and we consider its continued execution to be an effect. Note that this prohibits updates to main(), which is always on the stack. However, we can solve this problem by extracting portions of main() into separate functions, which can then be updated; Ginseng provides support to automate this process (Neamtiu et al. 2006). The next rule, (TGVAR), types global variables, which are bound in Γ. The last two rules type the dynamic updating-related elements of Proteus-tx. (TUPDATE) types update by checking that its prior and future effect annotations are supersets of (and thus conservatively approximate) the prior and future effects of the context.

Finally, (TTRANSACT) types transactions. A key design choice here is deciding how to handle nested transactions. In (TTRANSACT), we include the prior and future effects of Φ, from the outer context, into those of Φ_1, from the transaction body. This ensures that an update within a child transaction does not violate version consistency of its parent. However, we do not require the reverse—the components of Φ_1 need not be included in Φ. This has two consequences. First, sequenced transactions are free to commit independently. For example, consider the following code

```
tx { tx { /*A*/ }; /*B*/ tx { /*C*/ } }
```

According to (TTRANSACT), the effect at B is included in the prior and future effects of C and A, respectively, but not the other way around. Thus neither transaction's effect propagates into the other, and therefore does not influence any update operations in the other.

The second consequence is that version consistency for a parent transaction ignores the effects of its child transactions. This resembles *open nesting* in concurrency transactions (Ni et al. 2007). For example, suppose in the code above that A and C contain calls to a hash table T. Without the inner transaction markers, an update to T available at B would be rejected, because due to A it would overlap with the prior effect, and due to C it would overlap with the future effect. With the inner transactions in place, however, the update would be allowed. As a result, the parent transaction could use the old version of the hash table in A and the new version in C.

This treatment of nested transactions makes sense when inner transactions contain code whose semantics is largely independent of the surrounding context, e.g., the abstraction represented by a hash table is independent of where, or how often, it is used. Baumann et al. (2007) have applied this semantics to successfully partition dynamic updates to the K42 operating system into independent, object-sized chunks. While we believe open nesting makes sense, we can see circumstances in which closed nesting might be more natural, so we expect to refine our approach with experience.

3.4 Operational Semantics

Figure 8 defines a small-step operational semantics that reduces configurations $\langle n; \Sigma; H; e\rangle$, where n defines the current program version (a successful dynamic update increments n), Σ is the *transaction stack* (explained shortly), H is the heap, and e is the active program expression. Reduction rules have the form $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$, where the event η on the arrow is either μ, a dynamic update that occurred (discussed below), or ε, the effect of the evaluation step.

In our semantics, heaps map references r and global variables z to triples (τ, b, ν) consisting of a type τ, a binding b (defined in Figure 6), and a *version set* ν. The first and last components are relevant only for global symbols; the type τ is used to ensure that dynamic updates do not change the types of global bindings, and the version set ν contains all the program versions up to, and including, the current version since the corresponding variable was last updated. When an update occurs, new or changed bindings are given only the current version, while all other bindings have the current version added to their version set (i.e., we preserve the fact that the same binding was used in multiple program versions).

As evaluation proceeds, we maintain a transaction stack Σ, which is a list of pairs (n, σ) that track the currently active transactions. Here n is the version the program had when the transaction began, and σ is a *trace*. A trace is a set of pairs (z, ν), each of which represents a global symbol access paired with its version set at the time of use. The traces act as a log of dynamic events, and we track them in our semantics so we can prove that all global symbols accessed in a transaction come from the same version.

To evaluate a program d, we first compute $\mathcal{C}(\emptyset, d)$ using the function $\mathcal{C}$ shown at the top of Figure 8, which yields a pair $H; e$. This function implicitly uses the types derived by typing d using the rules in Figure 7. Then we begin regular evaluation in the configuration $\langle 0; (0, \emptyset); H; e\rangle$, i.e., we evaluate e at version 0, with initial transaction stack $(0, \emptyset)$, and with the declared bindings H. This causes the top-level expression e in main e to be treated as if it were enclosed in a transaction block.

The first several reduction rules in Figure 8 are straightforward. [LET], [REF], [DEREF], [ASSIGN], [IF-T], and [IF-F] are small-step versions of the rules in Figure 3, though normal references no longer have effects. None of these rules affects the current version or transaction stack. [CONG] is standard.

[GVAR-DEREF], [GVAR-ASSIGN], and [CALL] each have effect $\{\mathsf{z}\}$ and add (z, ν) to the current transaction's trace, where ν is z's current version set. Notice that [CALL] performs dereference and application in one step, finding z in the heap and performing substitution. Since dynamic updates modify heap bindings, this ensures that every function call is to the most recent version. Notice that although both functions and variables are stored in the heap, we assign regular function types to functions ((TDFUN) in Figure 7) so that they cannot be assigned to within a program. Including λ-terms in the expression language would either complicate function typing or make it harder to define function updates so we omit them to keep things simpler.

The next several rules handle transactions. [TX-START] pushes the pair $(n, \emptyset)$ onto the right of the transaction stack, where n is the current version and $\emptyset$ is the empty trace. The expression tx e is reduced to intx e, which is a new form that represents an actively-evaluating transaction. The form intx e does not appear in source programs, and its type rule matches that of tx e (see Figure 9).

Next, [TX-CONG-1] and [TX-CONG-2] perform evaluation within an active transaction intx e by reducing e to e'. The latter rule applies if e's reduction does not include an update, in which

Definitions

Heaps	H	::=	$\emptyset \mid r \mapsto (\cdot, b, \nu), H$
		$\mid$	$\mathsf{z} \mapsto (\tau, b, \nu), H$
Version sets	ν	::=	$\emptyset \mid \{n\} \cup \nu$
Traces	σ	::=	$\emptyset \mid (\mathsf{z}, \nu) \cup \sigma$
Transaction stacks	Σ	::=	$\emptyset \mid (n, \sigma), \Sigma$
Expressions	e	::=	$\ldots \mid r \mid \mathsf{intx}\ e$
Events	η	::=	$\varepsilon \mid \mu$
Update Direction	dir	::=	$bck \mid fwd$
Update Bundles	μ	::=	(upd, dir)

Compilation

$$\begin{array}{ll} \mathcal{C}(H; \mathsf{main}\ e) & = H; e \\ \mathcal{C}(H; \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d) & = \mathcal{C}(H, \mathsf{f} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda x.e, \{0\}); d) \\ \mathcal{C}(H; \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d) & = \mathcal{C}(H, \mathsf{g} \mapsto (\tau, v, \{0\}); d) \end{array}$$

Evaluation Contexts

$$\begin{array}{rcl} \mathbb{E} & ::= & [\,] \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \\ & \mid & \mathsf{ref}\ \mathbb{E} \mid !\ \mathbb{E} \mid \mathbb{E} := e \mid r := \mathbb{E} \mid \mathsf{g} := \mathbb{E} \\ & \mid & \mathsf{if0}\ \mathbb{E}\ \mathsf{then}\ e\ \mathsf{else}\ e \end{array}$$

Computation

$$\begin{array}{rll}
[\textsc{let}] & \langle n; (n', \sigma); H; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e[x \mapsto v]\rangle & \\
[\textsc{ref}] & \langle n; (n', \sigma); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H[r \mapsto (\cdot, v, \emptyset)]; r\rangle & r \notin \mathrm{dom}(H) \\
[\textsc{deref}] & \langle n; (n', \sigma); H; !\, r\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; v\rangle & H(r) = (\cdot, v, \emptyset) \\
[\textsc{assign}] & \langle n; (n', \sigma); H; r := v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H[r \mapsto (\cdot, v, \emptyset)]; v\rangle & r \in \mathrm{dom}(H) \\
[\textsc{if-t}] & \langle n; (n', \sigma); H; \mathsf{if0}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e_1\rangle & \\
[\textsc{if-f}] & \langle n; (n', \sigma); H; \mathsf{if0}\ n''\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e_2\rangle & n'' \neq 0 \\
[\textsc{cong}] & \langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; \mathbb{E}[e']\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; e'\rangle \\
[\textsc{gvar-deref}] & \langle n; (n', \sigma); H; !\, \mathsf{z}\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H; v\rangle & H(\mathsf{z}) = (\tau, v, \nu) \\
[\textsc{gvar-assign}] & \langle n; (n', \sigma); H; \mathsf{z} := v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H[\mathsf{z} \mapsto (\tau, v, \nu)]; v\rangle & H(\mathsf{z}) = (\tau, v', \nu) \\
[\textsc{call}] & \langle n; (n', \sigma); H; \mathsf{z}\ v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H; e[x \mapsto v]\rangle & H(\mathsf{z}) = (\tau, \lambda x.e, \nu) \\
[\textsc{tx-start}] & \langle n; (n', \sigma); H; \mathsf{tx}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma), (n, \emptyset); H; \mathsf{intx}\ e\rangle & \\
[\textsc{tx-cong-1}] & \langle n; (n'', \sigma), \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\mu} \langle n'; \mathcal{U}[(n'', \sigma)]^{\mu}_{n'}, \Sigma'; H'; \mathsf{intx}\ e'\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n'; \Sigma'; H'; e'\rangle \\
[\textsc{tx-cong-2}] & \langle n; \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\emptyset} \langle n'; \Sigma'; H'; \mathsf{intx}\ e'\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma'; H'; e'\rangle \\
[\textsc{tx-end}] & \langle n; ((n', \sigma'), (n'', \sigma'')); H; \mathsf{intx}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma'); H; v\rangle & \mathit{traceOK}(n'', \sigma'') \\
[\textsc{update}] & \langle n; (n', \sigma); H; \mathsf{update}^{\alpha,\omega}\rangle \longrightarrow_{(upd,dir)} \langle n+1; \mathcal{U}[(n', \sigma)]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; 1\rangle & \mathit{updateOK}(upd, H, (\alpha, \omega), dir) \\
[\textsc{no-update}] & \langle n; (n', \sigma); H; \mathsf{update}^{\alpha,\omega}\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; 0\rangle &
\end{array}$$

Update Safety

$$\begin{array}{l} \mathit{updateOK}(upd, H, (\alpha, \omega), dir) = \\ \quad dir = bck \Rightarrow \alpha \cap \mathrm{dom}(upd^{chg}) = \emptyset \\ \wedge \quad dir = fwd \Rightarrow \omega \cap \mathrm{dom}(upd^{chg}) = \emptyset \\ \wedge \quad \Gamma = \mathit{types}(H) \\ \wedge \quad \Gamma_{upd} = \Gamma, \mathit{types}(upd^{add}) \\ \wedge \quad \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in upd^{chg}. \\ \qquad \big(\Phi_{\emptyset}; \Gamma_{upd} \vdash b : \tau \wedge \mathit{heapType}(\tau, \mathsf{z}) = \Gamma(\mathsf{z})\big) \\ \wedge \quad \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in upd^{add}. \\ \qquad \big(\Phi_{\emptyset}; \Gamma_{upd} \vdash b : \tau \wedge \mathsf{z} \notin \mathrm{dom}(H)\big) \end{array}$$

Trace Safety

$$\mathit{traceOK}(n, \sigma) = (\forall (\mathsf{z}, \nu) \in \sigma.\ n \in \nu)$$

Heap Updates

$$\mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H)]^{upd}_{n} = \begin{cases} \mathsf{z} \mapsto (\tau, b', \{n\}), \mathcal{U}[H]^{upd}_{n} \\ \quad \text{if } upd^{chg}(\mathsf{z}) \mapsto (\tau, b') \\ \mathsf{z} \mapsto (\tau, b, \nu \cup \{n\}), \mathcal{U}[H]^{upd}_{n} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{U}[(r \mapsto (\cdot, b, \emptyset), H)]^{upd}_{n} = (r \mapsto (\cdot, b, \emptyset)), \mathcal{U}[H]^{upd}_{n}$$

$$\mathcal{U}[\emptyset]^{upd}_{n} = \{\mathsf{z} \mapsto (\tau, b, \{n\}) \mid \mathsf{z} \mapsto (\tau, b) \in upd^{add}\}$$

Heap Typing Environments

$$\begin{array}{rcl} \mathit{types}(\emptyset) & = & \emptyset \\ \mathit{types}(\mathsf{z} \mapsto (\tau, b, \nu), H') & = & \mathsf{z} : \mathit{heapType}(\tau, \mathsf{z}), \mathit{types}(H') \\ \mathit{heapType}(\tau_1 \longrightarrow^{\Phi} \tau_2, \mathsf{z}) & = & \tau_1 \longrightarrow^{\Phi} \tau_2 \quad \mathsf{z} \in \Phi \\ \mathit{heapType}(\tau, \mathsf{z}) & = & \mathit{ref}^{\{\mathsf{z}\}}\ \tau \qquad \tau \neq (\tau_1 \longrightarrow^{\Phi} \tau_2) \end{array}$$

Trace Stack Updates

$$\begin{array}{rcl} \mathcal{U}[(n', \sigma)]^{upd,fwd}_{n} & = & (n', \sigma) \\ \mathcal{U}[(n', \sigma)]^{upd,bck}_{n} & = & (n, \mathcal{U}_t[\sigma]^{upd}_{n}) \\ \mathcal{U}_t[\sigma]^{upd}_{n} & = & \{(\mathsf{z}, \nu \cup \{n\} \mid \mathsf{z} \notin \mathrm{dom}(upd^{chg})\} \\ & \cup & \{(\mathsf{z}, \nu) \mid \mathsf{z} \in \mathrm{dom}(upd^{chg})\} \end{array}$$

Figure 8. Proteus-tx operational semantics

case the effect ε of reducing e is treated as $\emptyset$ in the outer transaction. This corresponds to our model of transaction nesting, which does not consider the effects of inner transactions when updating outer transactions. Otherwise, if an update occurs, then [TX-CONG-1] applies, and we use the function $\mathcal{U}$ to update version numbers on the outermost entry of the transaction stack. $\mathcal{U}$ is discussed shortly.

The key property guaranteed by Proteus-tx, that transactions are version consistent, is enforced by [TX-END], which gets stuck unless $\mathit{traceOK}(n'', \sigma'')$ holds. This predicate, defined just below the reduction rules, states that every element (z, ν) in the transaction's trace σ'' satisfies $n'' \in \nu$, meaning that when z was used, it could be attributed to version n'', the version of the transaction. If this predicate is satisfied, [TX-END] strips off intx and pops the top (rightmost) entry on the transaction stack.

The last two rules handle dynamic updates. When $\mathsf{update}^{\alpha,\omega}$ is in redex position, these rules try to apply an available update bundle μ, which is a pair (upd, dir) consisting of an update (from Figure 6) and a *direction* dir that indicates whether we should consider the update as occurring at the beginning or end of the transaction, respectively. If $\mathit{updateOK}(upd, H, (\alpha, \omega), dir)$ is satisfied for some dir, then [UPDATE] applies and the update occurs. Otherwise [NO-UPDATE] applies, and the update must be delayed.

If [UPDATE] applies, we increment the program's version number and update the heap using $\mathcal{U}[H]^{upd}_{n+1}$, defined in the middle-right

$$\text{TIntrans}\ \frac{\Phi_1; \Gamma \vdash e : \tau \quad \Phi^\alpha \subseteq \Phi_1^\alpha \quad \Phi^\omega \subseteq \Phi_1^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau}$$

$$\frac{\begin{array}{l}\mathrm{dom}(\Gamma) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau \longrightarrow^\Phi \tau', \lambda x.e, \nu) \in H. \\ \quad \Phi; \Gamma, x : \tau \vdash e : \tau' \ \wedge\ \Gamma(\mathsf{z}) = \tau \longrightarrow^\Phi \tau' \ \wedge\ \mathsf{z} \in \Phi \\ \forall \mathsf{z} \mapsto (\tau, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge\ \Gamma(\mathsf{z}) = \mathit{ref}^\varepsilon\ \tau \ \wedge\ \mathsf{z} \in \varepsilon \\ \forall r \mapsto (\cdot, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge\ \Gamma(r) = \mathit{ref}^\varepsilon\ \tau \\ \forall \mathsf{z} \mapsto (\tau, b, \nu) \in H. \quad n \in \nu\end{array}}{n; \Gamma \ \vdash\ H}$$

$$\frac{\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \quad \mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f})}{[\alpha; \varepsilon; \omega], \cdot; H \vdash (n, \sigma)} \qquad \frac{\Phi', \mathcal{R}; H \vdash \Sigma \quad \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \quad \mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f})}{[\alpha; \varepsilon; \omega], \Phi', \mathcal{R}; H \vdash (n, \sigma), \Sigma}$$

$$\text{where} \quad \mathit{ver}(H, \mathsf{f}) = \nu \ \mathit{iff}\ H(\mathsf{f}) = (\tau, b, \nu)$$

Figure 9. Proteus-tx typing extensions for proving soundness

of Figure 8. This function replaces global variables and adds new bindings according to the update. New and replaced bindings' version sets contain only the current version, while unchanged bindings add the current version to their existing version sets.

The $\mathit{updateOK}()$ predicate is defined just below the reduction rules in Figure 8. The first two conjuncts enforce the update safety requirement discussed in Section 3.1. There are two cases. If $\mathit{dir} = \mathit{bck}$, then we require that the update not intersect the prior effects, so that the update will appear to have happened at the beginning of the transaction. In this case, we need to update the version number of the transaction to be the new version, and any elements in the trace not modified by the update can have the new version added to their version sets, i.e., the past effect can be attributed to the new version. To do this, [UPDATE] applies the function $\mathcal{U}[(n', \sigma)]_{n+1}^{\mathit{upd}, \mathit{dir}}$, defined on the bottom right of Figure 8, with $\mathit{dir} = \mathit{bck}$. The update applies to outer transactions as well, and thus [TX-CONG-1] applies this same version number replacement process across the transaction stack.

In the other case, if $\mathit{dir} = \mathit{fwd}$, we require that the remainder of the transaction not be affected by the update, so the update will appear to have happened at the end of the transaction. In this case we need not modify the transaction stack, and hence $\mathcal{U}[(n', \sigma)]_n^{\mathit{upd}, \mathit{dir}}$ with $\mathit{dir} = \mathit{fwd}$ simply returns (n', σ).

The remaining premises of $\mathit{updateOK}()$ determine whether the update itself is well-formed: each replacement binding must have the same type as the original, and new and added bindings must type check in the context of the updated heap.

3.5 Soundness

We have proven that well-typed Proteus-tx programs are version-consistent. The main result is that a well-typed, well-formed program either reduces to a value or evaluates indefinitely while preserving typing and version consistency. To prove this we need two additional judgments, shown in Figure 9. Heap typing $n; \Gamma \vdash H$ extends Definition 2.1 from the core system, where the additional conditions ensure that global symbols are well-typed, have well-formed effects, and include version n (presumed to be the current version) in their version sets.

Stack well-formedness $\mathcal{R}; H \vdash \Sigma$ checks that a transaction stack Σ is correctly approximated by a *transaction effect* $\mathcal{R}$, which consists of a list of contextual effects Φ, one for each nested transaction. $\mathcal{R}$ is computed from a typing derivation in a straightforward way according to the function $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$, extracting Φ_1 from each occurrence of (TINTRANS) recursively; the rules are not shown due to space constraints. Stack well-formedness ensures two properties. First, it ensures that each element in the trace σ is included in the corresponding prior effect α (i.e., $\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha$). As a result, we know that bck updates that rewrite the stack will add the new version to all elements of the trace, since none have changed. Second, it ensures that elements in each transaction's current effect (i.e., the part yet to be executed) have the version of that transaction: $\mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f})$.

With this we can prove the core result:

Theorem 3.1 (Single-step Soundness)**.** *If $\Phi; \Gamma \vdash e : \tau$ where $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$; and $n; \Gamma \vdash H$; and $\Phi, \mathcal{R}; H \vdash \Sigma$; and $\mathit{traceOK}(\Sigma)$, then either e is a value, or there exist n', H', Σ', Φ', e', and η such that $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$ and $\Phi'; \Gamma' \vdash e' : \tau$ where $[\![\Phi'; \Gamma' \vdash e' : \tau]\!] = \mathcal{R}'$; and $n'; \Gamma' \vdash H'$; and $\Phi', \mathcal{R}'; H' \vdash \Sigma'$; and $\mathit{traceOK}(\Sigma')$ for some $\Phi', \Gamma', \mathcal{R}'$.*

The proof is based on progress and preservation lemmas, as is standard. Details are in our technical report (Neamtiu et al. 2007).

From this lemma we can prove soundness:

Corollary 3.2 (Soundness)**.** *If $\Phi; \Gamma \vdash e : \tau$ and $0; \Gamma \vdash H$ then $\langle 0; (0, \emptyset); H; e\rangle \rightsquigarrow_A \langle n'; (n'', \sigma); H'; v\rangle$ for some value v or else evaluates indefinitely, where $\rightsquigarrow_A$ is the reflexive, transitive closure of the $\longrightarrow_\eta$ relation such that A is a set of events η.*

3.6 Transactional Version Consistency for C Programs

We extended Ginseng to implement transactional version consistency for C using contextual effects. In our implementation, transactional blocks are indicated with a special label, and are written __DSU_TX : {...}. Candidate update points can be inserted either manually or, in our experiment, automatically, as discussed below. To perform effect inference, we first compute a context-sensitive points-to analysis using CIL (Necula et al. 2002). Then we generate (context-insensitive) effect constraints (following Section 2.3) using labels derived from the points-to analysis, and we solve the constraints with Banshee (Kodumal and Aiken 2005).

After computing the contextual effects, Ginseng transforms the program to make it updatable, and transforms each update point into a call to a function $\mathsf{update}(\Delta, \alpha, \omega)$. Here α and ω are the prior and future effects at the update point, pre-computed by our contextual effect inference, and Δ is a set of type names whose definitions cannot be modified and variables whose types cannot be modified. More specifically, Δ contains all of those entities that could be accessed—functions f that might be called, variables g that might be dereferenced, and types t whose values might be destructed—by code possibly on the stack at the time of the update, since that code will expect the old version's type (Neamtiu et al. 2006; Stoyle et al. 2007). When update is called at run time, it checks to see whether an update is available and, if so, applies the update if it is both type safe (i.e., no variable or type in Δ has been changed by the update to have a different type) and version consistent (given α and ω). If an update is not safe, it is delayed and execution continues at the old version.

Type-altering Updates Since a function f's type is annotated with its contextual effect Φ, a modification to the program that causes f's effect Φ to change must be considered a change to f's type. This can occur even when f's code has not changed, e.g., if f calls g and an update changes g's effect. Our implementation handles such changes following the approach of our earlier work (Stoyle et al. 2007). In particular, if a variable f's type changes from τ to τ' due to an update, then if either $\tau' \leq \tau$ or $\tau' \not\leq \tau$ and $\mathsf{f} \notin \Delta$, the update is safe and can be applied. In the latter case, although f's type changes in an incompatible way, no active code depends on its type. On the other hand, if $\tau' \not\leq \tau$ and $\mathsf{f} \in \Delta$ then the update may be unsafe, since active code may rely on its type, and thus the update must be delayed.

State Transformation Our version consistency condition is slightly more complicated in practice due to *state transformers*. A state transformer is an optional function, supplied by the programmer, that is called at update time to transform old program state into new program state. The programmer writes the state transformer as if it will be run at the beginning or end of a transaction, and our system must ensure that this appearance is true. That is, to allow an update to occur within a transaction, we must ensure that (1) the writes performed by the state transformer do not violate the version consistency of the current program transactions, and (2) the effects of the current transactions do not violate the version consistency of the state transformer itself. We achieve both ends by considering the update changes $(\mathrm{dom}(upd^{chg}))$ *and* the state transformer's current effect ε_{xf} as the effect of the update when performing the usual checks for version consistency.

For example, if an update point $\mathtt{update}(\Delta, \alpha, \omega)$ is reached within a transaction, then if $\omega \cap (\varepsilon_{xf} \cup \mathrm{dom}(upd^{chg})) = \emptyset$ then the remaining actions of the transaction will not affect the state transformer, and vice versa, and so it is as if the update occurred at the end of the transaction. Likewise, if $\alpha \cap (\varepsilon_{xf} \cup \mathrm{dom}(upd^{chg})) = \emptyset$ then the effect of the transaction to this point has no bearing on the execution of the state transformer, and vice versa, so it is as if the update occurred at the beginning of the transaction. Note that because state transformers can also access the heap from global variables we need to include accesses to standard heap references (i.e., names L as in Section 2) in our effects.

Non-updatable transactions When writing a state transformer, the programmer needs to anticipate where it might be applied in the program, i.e., the transformer might need to do different things depending on which transactions have completed (as evidenced by the current state). Thus we have found it is convenient to rule out updates in some transactions, to limit the amount of location-dependent code in a state transformer. For example, in Figure 4, we would like to forbid updates from the program start up to the first transaction on line 22, and from the end of the transaction on line 28 to the beginning of the transaction on line 10. Since this code is not run very often, prohibiting transactions in it should not significantly reduce update availability.

Formally, we could support this notion by adding a new form $\mathtt{tx}^*\ e$ that has the same type rule and operational semantics as a transaction $\mathtt{tx}\ e$, but in which no updates are allowed at run time. In the example, however, the regions to which we would like to apply $\mathtt{tx}^*$ are non-lexically scoped. We could refactor the code to form lexical blocks, but ultimately we plan to support non-lexical transactions in our implementation. For our experiments below, we simulate $\mathtt{tx}^*\ e$ transactions by simply not flowing the prior and future effect of the outer context into the $\mathtt{tx}\ e$ blocks (i.e., eliminating the constraints in the hypothesis of (TTRANSACT)). This produces the result we want and is safe because the $\mathtt{tx}\ e$ blocks in `vsftpd` are only nested inside the transaction for the top-level expression, and everything else in that expression is effectively in a $\mathtt{tx}^*$ block.

3.7 Experiments

We measured the potential benefits of transactional version consistency by analyzing 12 dynamic updates to `vsftpd`. The updates correspond to versions 1.1.0 through 2.0.2. For our experiment, we modified Ginseng to seed the transactions in each `vsftpd` version with candidate update points. While we could conceivably insert update points at every statement, we found through manual examination that inserting update points just before the `return` statement of any function reachable from within a transaction provides good coverage. Then we used Ginseng to infer the contextual effects and type modification restrictions at each update point, and computed at how many of them we could safely apply the update.

Version	LoC	Time (s)	Upds	Type-safe	VC-safe
1.1.0	10,157	193	344	300	33
1.1.1	10,245	196	346	19	9
1.1.2	10,540	234	350	25	8
1.1.3	10,723	238	354	19	8
1.2.0	12,027	326	413	31	9
1.2.1	12,662	264	438	368	146
1.2.2	12,691	278	439	32	9
2.0.0	13,465	440	471	392	9
2.0.1	13,478	420	471	459	9
2.0.2pre2	13,531	632	471	471	9
2.0.2pre3	14,712	686	484	484	8
2.0.2	17,386	649	471	468	9

Figure 10. Version consistency analysis results

We conducted our experiments on an Athlon 64 X2 dual core 4600 machine with 4GB of RAM, running Debian, kernel version 2.6.18. Figure 10 summarizes our results. For each version, we list its size, the time Ginseng takes to pre-compute contextual effects and type modification restrictions, and the number of candidate update points that were automatically inserted. The analysis takes around 10 minutes for the largest example, and we expect that time could be reduced with more engineering effort. The last two columns indicate how many update points are type safe, and how many are both type safe and version consistent, with respect to the update from the version in that row to the next version. Note that determining whether an update is type safe and version consistent is very fast, and so we do not report the time for that computation.

Recall that in our prior work with `vsftpd` we manually added two update points. From the table, we can see that several additional update points are type safe and version consistent. We manually examined all of these update points. For all program versions except 1.1.0, 1.2.1, and 2.0.2pre2, we found that roughly one-third of the VC-safe update points occur somewhere in the middle of a transaction, providing better potential update availability. Another third occur close to or just before the end of a transaction, and the last third occur in dead code, providing no advantage. For the remaining versions, 1.1.0, 1.2.1, and 2.0.2pre2, we found that roughly 10% of the update points are in the middle of transactions, and almost all the remaining ones are close to the end of a transaction, with a few more in dead code.

One reason so many update points tend to occur toward the end of the transaction is due to the field-insensitivity of the alias analysis we used. In `vsftpd`, the type `vsf_session` contains a multitude of fields and is used pervasively throughout the code. The field-insensitive analysis causes spurious conflicts when one field is accessed early in the transaction but others are accessed later on, as is typical. This pushes the update points to the end of the transaction, following `vsf_session`'s last use. We plan to integrate a field-sensitive alias analysis into Ginseng to remedy this problem.

Interestingly, there are generally far more updates that are exclusively type safe than those that are both type safe and version consistent. We investigated some of these, and we found that the reasons for this varied with the update. For example, the updates that do not change `vsf_session` (e.g., 1.1.0) have a high number of type-safe update points, while those that do (e.g., 1.1.1) have far fewer. This makes sense, given `vsf_session`'s frequent use.

In summary, these results show that many update points are both type safe and version consistent, providing greater availability of updates than via manual placement. We expect still more update availability with a more accurate alias analysis.

4. Thread Sharing Analysis

Data race detectors (Savage et al. 1997; Flanagan and Freund 2000; Pratikakis et al. 2006; Naik et al. 2006) and other static analysis

```
1   let x = ref 0, y = ref 1,
2       z = ref 2, w = ref 3 in
3     y := 4;
4     fork (!x; !y; !z);
5     x := 5;
6     fork (z := 2);
7     while (...) fork(w := (!w) + 1)
```

$$(\text{TFORK})\ \frac{\Phi_{ei}; \Gamma \vdash e : \tau \qquad \Phi^{\varepsilon}_{ei} \subseteq \Phi^{\varepsilon}_{i}}{\Phi_i; \Gamma \vdash \mathsf{fork}^i\ e : \tau}$$

Figure 11. Thread sharing analysis: example and type rule

tools often perform *thread sharing analysis* to identify memory locations that may be accessed by multiple threads (and conversely those locations that are purely thread-local) in a concurrent program. This is useful because only shared locations need to be protected from concurrent access. In this section we show how contextual effects can be used to implement a thread sharing analysis.

We illustrate our analysis with an example. Suppose we have a language construct fork e, which creates a new thread that evaluates e, and consider the code in the top of Figure 11, written in an ML-like language with a while loop. This program creates four updatable references and then manipulates them in the parent thread and various child threads.

One simple but incorrect method for computing sharing would be to compute the (standard) effects of each thread and then intersect them; any location in the intersection of two threads would be considered thread-shared. In this case the effect of the main thread is $\{\mathsf{x}, \mathsf{y}\}$ (the writes on lines 3 and 5), and the effects of the threads on lines 4, 6, and 7 are $\{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$, $\{\mathsf{z}\}$, and $\{\mathsf{w}\}$, respectively. Thus we would compute that x, y, and z are shared, and that w is not.

The most obvious problem here is that w is determined thread-local, even though it is shared among the several threads created on line 7. We could solve this problem by performing some kind of analysis to determine which calls to fork might be invoked multiple times, but that adds complexity and does not solve another problem involving precision.

Observe that although x and y are accessed both by the main thread and the thread created on line 4, their sharing is different. The main thread writes to x on line 5 after the child thread on line 4 may have started, hence the read and write may be simultaneous. On the other hand, the write to y on line 3 happens before (Lamport 1978; Manson et al. 2005) the child thread is created on line 4, and since there is no other write in the parent thread, we can consider y to be thread-local—there are no possible concurrent accesses from different threads.

We can solve both of these problems by using contextual effects rather than regular effects to determine sharing. The idea is simple: a location is thread-shared if it may be accessed by a child thread and by the parent thread, but only *after* the child thread is created—and we can use the future effect to compute what happens after thread creation. This takes care of the problems with loops, since the future effect of a fork in a loop will include the back-edge in the loop; and it allows the parent to modify data before a child thread is created without forcing that data to be considered shared. This technique is even more useful when we distinguish read and write effects, which we do in practice, in which case data need only be considered shared if at least one of the potential simultaneous accesses is a write.

4.1 Typing

The bottom of Figure 11 gives the new type rule (TFORK) needed for sharing analysis. This rule types thread creation, where each syntactic occurrence $\mathsf{fork}^i\ e$ has been named with an index i. In this rule, we compute the effect Φ_{ei} of the child thread separately from the effect Φ_i of the parent thread. Once we have all such effects, we can compute the set of shared locations as $\mathit{shared} = \bigcup_i (\Phi^{\varepsilon}_{ei} \cap \Phi^{\omega}_{i})$. In other words, a location is shared if it is accessed both in the child thread and in the parent thread after a call to fork.

There is one catch, however. Consider z from the example program in Figure 11. This particular location is not accessed by the parent thread after it is created, and hence is not (yet) in Φ^{ω}_{i}, and thus will not be considered shared. To handle this case, sharing among two child threads, we simply add the child's effects to the parent's effects with the constraint $\Phi^{\varepsilon}_{ei} \subseteq \Phi^{\varepsilon}_{i}$. Considering the example again, this causes the write to z on line 6 to be added to the parent thread's effect on the same line, and therefore it will be in the parent's future effect on line 4.

Name	LoC	Time (s)	Shared	Total	%
aget	1,914	0.40	60	338	18%
ctrace	2,212	0.25	21	307	7%
engine	2,608	0.49	10	390	3%
knot	1,985	0.35	29	319	9%
pfscan	1,948	0.25	26	238	11%
smtprc	8,630	2.46	128	1077	12%
eql	16,568	1.68	22	240	9%
3c501	17,441	0.64	23	353	7%
plip	19,141	0.88	64	402	16%
sundance	19,951	1.05	25	633	4%
wavelan	20,085	1.14	123	660	19%
hp100	20,368	1.16	24	450	5%
synclink	24,691	2.65	219	1158	19%

Figure 12. Sharing analysis results

4.2 Implementation and Experiments

We have incorporated our thread sharing analysis into Locksmith (Pratikakis et al. 2006), a static analysis tool we developed to find data races in C programs. Locksmith works by enforcing the *guarded-by* pattern: Every shared location in the program must be consistently guarded by some lock. Locksmith requires essentially no annotations, and uses several other analyses in addition to the thread sharing analysis described above. Previous presentations of Locksmith's sharing analysis were informal (Pratikakis et al. 2006) or used a different formulation (Hicks et al. 2006) (see Section 5), and did not report the effectiveness of the analysis.

Figure 12 shows the results. We measure the running time, the number of shared locations, and the total number of locations. Here locations include all global variables, syntactic occurrences of malloc, local variables whose address is taken, or fields of locations (our analysis is field sensitive). The results show that contextual effect inference runs very quickly, and is able to determine that many locations in the program are thread-local. On average, only 11% of the total locations are determined to be shared. Thus Locksmith can safely assume that accesses to the remaining 89% of locations need not be guarded by locks, greatly improving the precision of race detection.

5. Related Work

5.1 Effect Systems

Several researchers have proposed extending standard effect systems (Lucassen 1987; Nielson et al. 1999) to model more complex properties. One common approach is to use traces of actions for effects rather than sets of actions. These traces can be used to check that resources are accessed in the correct order (Igarashi and Kobayashi 2002), to statically enforce history-based access control (Skalka et al. 2007), and to check communication sequenc-

ing (Nielson et al. 1999). While these systems can model the ordering of events, they do not compute the prior or future effect at a program point. We believe we could combine trace effects with our approach to create a contextual trace effect system, which we leave for future work.

In prior work (Hicks et al. 2006) we introduced *continuation effects* γ, which resemble the union $\varepsilon \cup \omega$ of our standard and future effects. Judgments in this system have the form $\gamma; \Gamma \vdash e : \tau; \gamma'$, where γ' describes the effect of e's continuation in the remainder of the program, and γ is equivalent to $\varepsilon \cup \gamma'$ where ε is the standard effect of e. The drawback of this formulation is that the standard effect ε of e cannot be recovered from γ, since $(\varepsilon \cup \gamma') - \gamma' \neq \varepsilon$ when $\varepsilon \cap \gamma' \neq \emptyset$. This system also does not include prior effects.

Capabilities in Alias Types (Smith et al. 2000) and region systems like CL (Walker et al. 2000) are likewise related to our standard and future effects. A capability consists of static approximation of the memory locations that are live in the program, and thus may be dereferenced in the current expression or in later evaluation. Because these systems assume their inputs are in continuation passing style (CPS), the effect of a continuation is equivalent to our future effects. The main differences are that we compute future effects at every program point (rather than only for continuations), that we compute prior effects, and that we do not require the input program to be CPS-converted.

5.2 Correctness of Dynamic Software Updating

Several systems for on-line updates have been proposed. Here we focus on how prior work controls an update's timing to assure its effects are correct.

Most work disallows updates to code that is active, i.e., actually running or referenced by the call stack. The simplest approach to updating active code, taken by several recent systems (Gilmore et al. 1997; Makris and Ryu 2007; Chen et al. 2006), is to passively wait for it to become inactive. This can be problematic for multi-threaded programs, since there is a greater possibility that active threads reference a to-be-updated object. To address this problem, Soules et al. (2003) developed a quiescence protocol to support dynamic updating in the object-oriented K42 operating system (Baumann et al. 2005, 2007). Once an update for an object is proposed, an adaptor object is interposed to block subsequent callers of the object. Once the active threads have exited, the object is upgraded and the blocked callers are resumed. The danger is that dependencies between updated objects could result in a deadlock. While code inactivity is useful, it is not sufficient for ensuring higher-level properties like version consistency. In particular, version consistency may require delaying an update if to-be-updated objects are not currently active but were during the transaction.

Lee (1983) proposed a generalization of the quiescence condition by allowing programmers to specify timing constraints on when elements of an update should occur; recent work by Chen et al. (2007) is similar. As an example, the condition `update P, Q when P, M, S idle` specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M`, and `S` are not active. Lee provides some guidance for using these conditions. For example, if procedure `P`'s type has changed, then an update to it and its callers should occur when all are inactive. Our work uses a program analysis to discover conditions such as these to establish the higher-level transactional version consistency property.

Prior work with Ginseng focused on ensuring that a dynamic update does not introduce type errors when it is applied (Neamtiu et al. 2006; Stoyle et al. 2007). For example, the program point just before a call to a function `f` is restricted from changing the type of `f`—to allow the update would result in the old code calling the new `f` at an incompatible type. We developed a static *updatability analysis* that gathers type constraints imposed by the active (old) code at each program point and only allows an update to take place if it satisfies the constraints. This is more fine-grained than Lee's constraints—if the type of a function changes, we can update it even when its callers are active so long as they will not call the updated function directly. Our current work is complementary to this work, as a type-safe update will not necessarily be version-consistent (as illustrated by the example in Section 3), and depending on how transactions are specified the reverse may also be true.

Our use of transactions to ensure version consistency resembles work by Boyapati et al. (2003) on lazily upgrading objects in a *persistent object store* (POS). Using a type system that guarantees object encapsulation, their system ensures that an object's transformation function, used to initialize the state of a new version based on old state, sees only objects of the old version, which is similar to our version consistency property. How updates interact with application-level transactions is less clear to us. The assumption seems to be that updates to objects are largely semantically-independent, so there is less concern about version-related dependencies between objects within a transaction.

5.3 Thread Sharing Analysis

Thread sharing analysis is a key part of several tools for analyzing multi-threaded programs. Eraser (Savage et al. 1997), a dynamic data race detector, assumes locations are shared after they have been accessed by at least two threads. Our thread-sharing analysis can be seen as a static version of this approach. RCC Java (Flanagan and Freund 2000) allows programmers to manually add annotations to mark classes as thread local, so that their fields need not be guarded by locks when accessed.

Chord (Naik et al. 2006) uses a thread escape analysis to find shared locations; a location is considered shared if it is reachable from a thread object. This is more conservative than our approach, which allows data to be thread-local as long as it is not used in the parent after a child thread is forked. Chord avoids this problem by discounting constructors when determining thread sharing or data races. A newer version of Chord includes a flow-sensitive sharing analysis (Naik and Aiken 2007), but it is not described in detail.

von Praun and Gross (2003) propose a thread sharing analysis for Java. Their system determines that an object is shared if it is accessed by multiple threads, and includes additional reasoning to reduce sharing by taking thread creation and joining into account.

RacerX (Engler and Ashcraft 2003) performs deadlock and data race detection for C. RacerX uses a heuristic, statistical approach to decide whether data is likely to be shared, based on how often it is accessed when a lock is held. This is in contrast to our approach, which tries to compute thread sharing in a sound manner.

6. Conclusion

We have introduced contextual effects, which extend standard effect systems to capture the effect of the context in which each subexpression appears, i.e., the effect of evaluation both before and after the evaluation of the subexpression. We formalized a core contextual type and effect system and proved it sound. We then used extensions of our core system in two applications. First, we proposed transactional version consistency, a new correctness condition for dynamic software updates. We showed how to use contextual effects to enforce this property while allowing updates to occur more frequently within programs. Second, we used contextual effects to compute locations shared between threads in concurrent programs. We determined shared locations by intersecting the future effect of the parent at thread creation time with the effect of the child. Our experimental results show that our static contextual effect system is useful in both applications. These examples show the utility of contextual effects, and we anticipate they will also prove useful in a variety of other applications.

Acknowledgments

The authors thank Peter Sewell, Nikhil Swamy, and the anonymous referees for their insightful comments on drafts of this paper. This research was supported in part by National Science Foundation grants CCF-0541036 and CCF-0346989, and the University of Maryland Partnership with the Laboratory for Telecommunications Sciences.

References

Martin Abadi and Cedric Fournet. Access control based on execution history. In *NDSS*, 2003.

Andrew Baumann, Gernot Heiser, Jonathan Appavoo, et al. Providing dynamic update in an operating system. In *USENIX*, 2005.

Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, et al. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX*, 2007.

Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, 2003.

Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *VEE*, 2006.

Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *ICSE*, pages 271–281, 2007.

Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, 2003.

Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *PLDI*, 2000.

Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *TOPLAS*, 28(6):1035–1087, November 2006.

Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997.

Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock Inference for Atomic Sections. In *TRANSACT*, 2006.

Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis. In *POPL*, Portland, Oregon, 2002.

John Kodumal and Alexander Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, 2005.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Dept. of Computer Science, University of Wisconsin, Madison, April 1983.

John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, August 1987. MIT/LCS/TR-408.

Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. EuroSys*, March 2007.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, 2005.

John C. Mitchell. Type inference with simple subtypes. *JFP*, 1(3):245–285, July 1991.

Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, 2006.

Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *MSR'05*, 2005. URL `http://www.cs.umd.edu/~mwh/papers/evolution.pdf`.

Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. Technical Report CS-TR-4920, Dept. of Computer Science, University of Maryland, November 2007.

George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *LNCS*, 2304:213–228, 2002.

Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, et al. Open nesting in software transactional memory. In *PPoPP*, 2007.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN 3540654100.

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Context-sensitive correlation analysis for detecting races. In *PLDI*, 2006.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *SOSP*, 1997.

Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *JFP*, July 2007. Forthcoming; available online at `http://www.journals.cambridge.org`.

Fred Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, 2000.

Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, et al. System support for online reconfiguration. In *USENIX*, 2003.

Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating (full version). *TOPLAS*, 29(4):22, August 2007.

Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03*, 2003.

David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *TOPLAS*, 24(4):701–771, July 2000.