Formalizing Soundness of Contextual Effects

Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks, and Iulian Neamtiu

University of Maryland, College Park, MD 20742

Abstract. A *contextual effect* system generalizes standard type and effect systems: where a standard effect system computes the effect of an expression e, a contextual effect system additionally computes the *prior* and *future* effect of e, which characterize the behavior of computation prior to and following, respectively, the evaluation of e. This paper describes the formalization and proof of soundness of contextual effects, which we mechanized using the Coq proof assistant. Contextual effect soundness is an unusual property because the prior and future effect of a term e depends not on e itself (or its evaluation), but rather on the evaluation of the context in which e appears. Therefore, to state and prove soundness we must "match up" a subterm in the original typing derivation with the possibly-many evaluations of that subterm during the evaluation of the program, in a way that is robust under substitution. We do this using a novel typed operational semantics. We conjecture that our approach could prove useful for reasoning about other properties of derivations that rely on the context in which that derivation appears.

1 Introduction

Type and effect systems are used to reason about a program's computational effects [5, 8, 11]. Such systems have various applications in program analysis, e.g., to compute the set of memory accesses, I/O calls, function calls or new threads that occur in any given part of the program. Generally speaking, a type and effect system proves judgments of the form $\varepsilon; \Gamma \vdash e : \tau$ where ε is the effect of expression e. Recently, we proposed generalizing such systems to track what we call *contextual effects*, which capture the effects of the context in which an expression occurs [7]. In our contextual effect system, judgments have 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 effects of the program evaluation prior to and after computing e, respectively.

Our prior work explored the utility of contextual effects by studying two applications, one related to dynamic software updating correctness, and the other to analysis of multi-threaded programs. This paper presents the formalization and proof of soundness of contextual effects, which we have mechanized using the Coq proof assistant [2]. Intuitively, for all subexpressions e of a given program e_p, a contextual effect $[\alpha; \varepsilon; \omega]$ is sound for e if (1) α contains the actual, run-time effect of evaluating e_p prior to evaluating e, (2) ε contains the run-time effect of evaluating e itself, and (3) ω contains the run-time effect of evaluating the remainder of e_p after e's evaluation has finished. (Discussed in Section 2.)

There are two main challenges with formalizing this intuition to prove that our contextual effect system is sound. First, we must find a way to define what constitute the *actual* prior and future effects of e when it is evaluated as part of e_p. Interestingly, these effects cannot be computed compositionally (i.e., by considering the subterms of e), as they depend on the relative position of the evaluation of e within the evaluation of e_p, and not on the evaluation of e itself. Moreover, the future effect of e models the evaluation after e has reduced to a value. In a small-step semantics, specifying the future effect by finding the end of e's computation would be possible but awkward. Thus we opt for a big-step operational semantics, in which we can easily and naturally define the prior, standard, and future effect of every subterm in a derivation. (Section 3)

The second challenge, and the main novelty of our proof, is specifying how to match up the contextual effect Φ of e, as determined by the *original* typing derivation of $\Phi_p; \Gamma \vdash e_p : \tau_p$, with the run-time effects of e recorded in the evaluation derivation. The difficulty here is that, due to substitution, e may appear many times and in different forms in the evaluation of e_p. In particular, a value containing e may be passed to a function $\lambda x.e'$ such that x occurs several times in e', and thus after evaluating the application, e will be duplicated. Moreover, variables within e itself could be substituted away by other reductions. Thus we cannot just syntactically match a subterm e of the original program e_p with its corresponding terms in the evaluation derivation.

To solve this problem, we define a *typed operational semantics* in which each subderivation is annotated with two typing derivations, one for the term under consideration and one for its final value. Subterms in the original program e_p are annotated with subderivations of the original typing derivation $\Phi_p; \Gamma \vdash e_p : \tau_p$. As subterms are duplicated and have substitutions applied to them, our semantics propagates the typing derivations in the natural way to the new terms. In particular, if Φ is the contextual effect of subterm e of e_p, then all of the terms derived from e will also have contextual effect Φ in the typed operational semantics. Given this semantics, we can now express soundness formally, namely that in every subderivation of the typed evaluation of a program, the contextual effect Φ in its typing contains the run-time prior, standard, and future effects of its computation. (Section 4)

We mechanized our proof using the Coq proof assistant, starting from the framework developed by Aydemir et al [1]. We found the mechanization process worthwhile, because our proof structure, while conceptually clear, required getting a lot of details right. Most notably, typing derivations are nested inside of evaluation derivations in the typed operational semantics, and thus the proofs of each case of the lemmas are somewhat messy. Using a proof assistant made it easy to ensure we had not missed anything. We found that, modulo some typos, our paper proof was correct, though the mechanization required that we precisely define the meaning of "subderivation." (Section 5)

We believe that our approach to proving soundness of contextual effects could be useful for other systems, in particular ones in which properties of subderivations depend on their position within the larger derivation in which they appear.

Expressions	$e ::= v \mid x \mid e\ e \mid \mathsf{ref}^L\ e \mid !\,e \mid e := e$
Values	$v ::= n \mid \lambda x.e \mid r_L$
Effects	$\alpha, \varepsilon, \omega ::= \emptyset \mid 1 \mid \{L\} \mid \varepsilon \cup \varepsilon$
Contextual Effects	$\Phi ::= [\alpha; \varepsilon; \omega]$
Types	$\tau ::= int \mid ref^{\varepsilon}\ \tau \mid \tau \longrightarrow^{\Phi} \tau$
Environments	$\Gamma ::= \cdot \mid (\Gamma, x \mapsto \tau) \mid (\Gamma, r \mapsto \tau)$
Labels	L

Fig. 1. Syntax

2 Background: Contextual Effects

This section reviews our type and effect system, and largely follows our previous presentation [7]. Readers familiar with the system can safely skip this section.

2.1 Language

Figure 1 presents our source language, a simple calculus with expressions that consist of values v (integers, functions or pointers), variables and call-by-value function application. Our language also includes updateable references, created with $\mathsf{ref}^L\ e$, along with dereference and assignment. 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. Evaluating $\mathsf{ref}^L\ e$ creates a pointer r_L, where r is a fresh name in the heap and L is the declared label. Dereferencing or assigning to r_L during evaluation has effect $\{L\}$. Note that pointers r_L do not appear in the syntax of the program, but only during its evaluation. For simplicity we do not model recursive functions directly, but they can be encoded using references. Also, due to space constraints we omit let and if. They are included in the mechanized proof; supporting them is straightforward.

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 $[\alpha; \varepsilon; \omega]$. 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 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, $\alpha \cup \omega$ is the effect of the context in which e' appears, and therefore $\alpha \cup \varepsilon \cup \omega$ is the effect of evaluating 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. For brevity, whenever it is clear we will refer to contextual effects simply as *effects*.

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

$$(\text{TLam})\ \frac{\Phi; \Gamma, x : \tau' \vdash e : \tau}{\Phi_\emptyset; \Gamma \vdash \lambda x.e : \tau' \longrightarrow^{\Phi} \tau} \qquad (\text{TApp})\ \frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \\ \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\text{TRef})\ \frac{\Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}^L\ e : ref^{\{L\}}\ \tau} \qquad (\text{TDeref})\ \frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : ref^{\varepsilon}\ \tau \\ \Phi_2^{\varepsilon} = \varepsilon \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !e : \tau}$$

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

$$(\text{TLoc})\ \frac{\Gamma(r) = \tau}{\Phi_\emptyset; \Gamma \vdash r_L : ref^{\{L\}}\ \tau} \qquad (\text{TSub})\ \frac{\begin{array}{c}\Phi'; \Gamma \vdash e : \tau' \\ \tau' \leq \tau \quad \Phi' \leq \Phi\end{array}}{\Phi; \Gamma \vdash e : \tau}$$

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

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

$$(\text{SFun})\ \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad \Phi \leq \Phi'}{\tau_1 \longrightarrow^{\Phi} \tau_2 \leq \tau_1' \longrightarrow^{\Phi'} \tau_2'} \qquad (\text{SCtxt})\ \frac{\alpha_2 \subseteq \alpha_1 \quad \varepsilon_1 \subseteq \varepsilon_2 \quad \omega_2 \subseteq \omega_1}{[\alpha_1; \varepsilon_1; \omega_1] \leq [\alpha_2; \varepsilon_2; \omega_2]}$$

Fig. 2. Typing

2.2 Typing

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.

Types τ, listed in Figure 1, include the integer type *int*; reference types $ref^{\varepsilon}\ \tau$, which denote a reference to memory location 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. Environments Γ, defined in Figure 1, are maps from variable names or (unlabeled) pointers to types.

The first two rules, (TInt) and (TVar), assign the expected types and the empty effect, since values have no effect. Rule (TLam) types the function body e and annotates the function's type with the effect of e. The expression as a whole has no effect, since the function produces no run-time effects until it is actually called. Rule (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. We specify the sequencing of effects with the combinator $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$, defined by (XFLOW-CTXT). Since e_1 evaluates 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 is evaluated just after e_1. Lastly, the effect Φ of the entire expression has α_1 as its prior effect, since e_1 is evaluated first; ω_2 as its future effect, since e_2 is evaluated 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 $(\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi') \wedge (\Phi' \rhd \Phi_3 \hookrightarrow \Phi)$.

(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 $\mathit{ref}^{\varepsilon}\ \tau$. 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. (TLOC) gives a pointer r_L the type of a reference to the type of r in Γ.

Finally, (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 ω_1 should contain (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 should contain the locations that were accessed in the past, we can use e in any context that accessed at most locations in α_1.

3 Operational Semantics

As discussed in the introduction, to establish the soundness of the static semantics we must address two concerns. First, we must give an operational semantics that specifies the run-time contextual effects of each subterm e appearing in the evaluation of a term e_p. Second, we must find a way to match up subterms e that arise in the evaluation of e_p with the corresponding terms e' in the unevaluated e_p, to see whether the effects ascribed to the original terms e' by the type system approximate the actual effects of the subterms e. This section defines an operational semantics that addresses the first concern, and the next section augments it to address the second concern, allowing us to prove our system sound.

3.1 The Problem of Future Effects

Consider an expression e appearing in program e_p. We write $e_p = C[e]$ for a context C, to make this relationship more clear. Using a small-step operational semantics, we can intuitively view the contextual effects of e as follows:

$$\underbrace{C[e] \to \cdots \to}_{\text{prior effect } \alpha} C'[e] \overbrace{\underbrace{\to C'[e'] \to \cdots \to}_{\text{standard effect } \varepsilon}}^{\text{evaluation of } e} C'[v] \underbrace{\to \cdots \to v_p}_{\text{future effect } \omega}$$

(The evaluation of e_p could contain several evaluations of e, each of which could differ from e according to previous substitutions of e's free variables, but we ignore these difficulties for now and consider them in the next section.)

For this evaluation, the actual, run-time prior effect α of e is the effect of the evaluation that occurs before e starts evaluating, the actual standard effect ε of e is the effect of the evaluation of e to a value v, and the actual future effect ω of e is the effect of the remainder of the computation. For every expression in the program, there exist similar partitions of the evaluation to define the appropriate contextual effects.

However, while this picture is conceptually clear, formalizing contextual effects, particularly future effects, is awkward in small-step semantics. Suppose we have some contextual effect Φ associated with subterm e in the context $C'[e]$ above. Then Φ^ω, the future effect of subterm e, models everything that happens after we evaluate to $C'[v]$—but that happens some arbitrary number of steps after we begin evaluating $C'[e]$, making it difficult to associate with the subterm e. We could solve this problem by inserting "brackets" into the semantics to identify the end of a subterm's evaluation, but that adds complication, especially since there are many different subterms whose contextual effects we wish to track and prove sound.

Our solution to this problem is to use big-step semantics, since in big-step semantics, each subderivation is a full evaluation. This lets us easily identify both the beginning and the end of each sub-evaluation in the derivation tree, and gives us a natural specification of contextual effects.

3.2 Big-Step Semantics

Figure 3 shows key rules in 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 that point in the evaluation, respectively; H is the heap (a map from locations r to values); and e is the expression to be evaluated. Evaluations have the form

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

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

Heaps $H ::= \emptyset \mid H, r \mapsto v$

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

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

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

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

$$[\textsc{Call-W}]\ \frac{\langle \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle \alpha', \omega', H', v\rangle \qquad v \neq \lambda x.e}{\langle \alpha, \omega, H, e_1\ e_2\rangle \longrightarrow_{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$[\textsc{Deref-H-W}]\ \frac{\langle \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \qquad r \notin \mathrm{dom}(H')}{\langle \alpha, \omega, H, !\,e\rangle \longrightarrow_{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

$$[\textsc{Deref-L-W}]\ \frac{\langle \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', r_L\rangle \qquad r \in \mathrm{dom}(H') \qquad L \notin \omega'}{\langle \alpha, \omega, H, !\,e\rangle \longrightarrow_{\emptyset} \langle \alpha, \omega, H, \mathbf{err}\rangle}$$

Fig. 3. Operational Semantics

where ε is the effect of evaluating e and R is the result of reduction, either a value v or **err**, indicating evaluation failed. Intuitively, as evaluation proceeds, labels move from the future effect ω to the past effect α.

With respect to the definitions of Section 3.1, the prior effect α in Section 3.1 corresponds to α here, and the future effect ω in Section 3.1 corresponds to ω' here. The future effect ω before the evaluation of e contains both the future and the standard effect of e, i.e., $\omega = \omega \cup \varepsilon$. Similarly, the past effect α' after the evaluation of e contains the past effect α and the effect of e, i.e., $\alpha' = \alpha \cup \varepsilon$. We prove below that our semantics preserves this property.

The reduction rules are straightforward. [Id] reduces a value to itself without changing the state or the effects. [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 future effect after evaluating e, but prior to dereferencing the result. Then L is added to α' in the output configuration of the rule. Notice that $\omega' \cup \{L\}$ is a standard union, hence L may also be in ω', which allows the same location to be accessed multiple times. Also note that we require L to be in the future effect just after the evaluation of e, but do not require that it be in ω. However, this will actually hold—below we prove that $\omega = \omega' \cup \{L\} \cup \varepsilon$, and in general when the semantics takes a step, effects move from the future to the past. [Assign] behaves similarly to [Deref]. [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. Our semantics also includes rules [Call-W], [Deref-H-W] and [Deref-L-W] that produce **err** when the program tries to access a location that is not in the future effect of the input, or when values are used at the wrong type. Our system includes similar error rules for assignment (not shown).

3.3 Standard Effect Soundness

We can now prove standard effect soundness. First, we prove an *adequacy* property of our semantics that helps ensure they make sense:

Lemma 1 (Adequacy of Semantics). *If* $\langle \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle \alpha', \omega', H', v\rangle$*, then* $\alpha' = \alpha \cup \varepsilon$ *and* $\omega = \omega' \cup \varepsilon$*.*

This lemma formalizes our intuition that labels move from the future to prior effect during evaluation.

We can then prove that the static Φ^ε associated to a term by our type and effect system soundly approximates the actual effect ε of an expression. We ignore actual effects α and ω by setting them to 1. We say heap H is well-typed under Γ, written $\Gamma \vdash H$, if $\mathrm{dom}(\Gamma) = \mathrm{dom}(H)$ and for every $r \in \mathrm{dom}(H)$, we have $\Phi_\emptyset; \Gamma \vdash H(r) : \Gamma(r)$. The standard effect soundness lemma is:

Theorem 1 (Standard Effect Soundness). *If*

1. $\Phi; \Gamma \vdash e : \tau$,
2. $\Gamma \vdash H$ *and*
3. $\langle 1, 1, H, e\rangle \longrightarrow_\varepsilon \langle 1, 1, H', R\rangle$

then there is a Γ' *such that:*

1. R *is a value* v *for which* $\Phi_\emptyset; (\Gamma', \Gamma) \vdash v : \tau$,
2. $(\Gamma', \Gamma) \vdash H'$ *and*
3. $\varepsilon \subseteq \Phi^\varepsilon$.

Here (Γ', Γ) is the concatenation of environments Γ' and Γ. The proof of this theorem is by induction on the evaluation derivation, and follows traditional type-and-effect system proofs, adapted for our semantics.

Next, we prove that if the program evaluates to a value, then there is a *canonical evaluation* in which the program evaluates to the same value, but starting with an empty α and ending with an empty ω. This will produce an evaluation derivation with the *most precise* α and ω values for every configuration, which we can then prove we soundly approximate using our type and effect system.

Lemma 2 (Canonical Evaluation). *If* $\langle 1, 1, H, e\rangle \longrightarrow_\varepsilon \langle 1, 1, H', v\rangle$ *then there exists a derivation* $\langle \emptyset, \varepsilon, H, e\rangle \longrightarrow_\varepsilon \langle \varepsilon, \emptyset, H', v\rangle$*.*

4 Contextual Effect Soundness

Now we turn to proving contextual effect soundness. We aim to show that the prior and future effect of some subterm e of a program e_p approximate the evaluation of e_p before and after, respectively, the evaluation of e. Suppose for the moment that e_p contains no function applications. As a result, an evaluation derivation D_p of e_p according to the operational semantics in Figure 3 will be isomorphic to a typing derivation T_p of e_p according to the rules in Figure 2. In this situation, soundness for contextual effects is easy to define. For any subterm e of e_p, we have an evaluation derivation D and a typing derivation T:

$$D :: \langle \alpha, \omega, H, e \rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', v \rangle \qquad\qquad T :: \Phi; \Gamma \vdash e : \tau$$

where D is a subderivation of D_p and T is a subderivation of T_p. Then the prior and future effects computed by our contextual effect system are sound if $\alpha \subseteq \Phi^{\alpha}$ (the effect of the evaluation before e is contained in Φ^{α}) and $\omega' \subseteq \Phi^{\omega}$ (the effect of the evaluation after v is contained in Φ^{ω}).

For example, consider the evaluation of $!\,(\mathsf{ref}^L\ n)$.

$$\text{(Deref)}\ \dfrac{\text{(Ref)}\ \dfrac{\text{(Id)}\ \dfrac{}{\langle \emptyset, \emptyset \cup \{L\}, H, n \rangle \longrightarrow \langle \emptyset, \emptyset \cup \{L\}, H, n \rangle}}{\langle \emptyset, \emptyset \cup \{L\}, H, \mathsf{ref}^L\ n \rangle \longrightarrow \langle \emptyset, \emptyset \cup \{L\}, (H, r_L \mapsto n), r_L \rangle}}{\langle \emptyset, \emptyset \cup \{L\}, H, !\,(\mathsf{ref}^L\ n) \rangle \longrightarrow_{\{L\}} \langle \emptyset \cup \{L\}, \emptyset, (H, r_L \mapsto n), n \rangle}$$

Here is the typing derivation (where we have rolled a use of (TSub) into (TInt)):

$$\text{(TDeref)}\ \dfrac{\text{(TRef)}\ \dfrac{\text{(TInt')}\ \dfrac{}{[\emptyset; \emptyset; \{L\}]; \cdot \vdash n : \mathit{int}}}{[\emptyset; \emptyset; \{L\}]; \cdot \vdash \mathsf{ref}^L\ n : \mathit{ref}^L\ \mathit{int}} \quad [\emptyset; \{L\}; \emptyset]^{\varepsilon} = \{L\} \quad [\emptyset; \emptyset; \{L\}] \rhd [\emptyset; \{L\}; \emptyset] \hookrightarrow [\emptyset; \{L\}; \emptyset]}{[\emptyset; \{L\}; \emptyset]; \cdot \vdash !\,(\mathsf{ref}^L\ n) : \mathit{int}}$$

We can see that these derivations are isomorphic, and thus it is easy to read the contextual effect from the typing derivation for $\mathsf{ref}^L\ n$ and to match it up with the actual effect of the corresponding subderivation of the evaluation derivation.

Unfortunately, function applications add significant complication because D_p and T_p are no longer isomorphic. Indeed, a subterm e of the original program e_p may appear multiple times in D_p, possibly with substitutions applied to it. For example, consider the term $(\lambda x.\,!\,x; !\,x)\ \ \mathsf{ref}^L\ n$ (where we introduce the sequencing operator ; with the obvious semantics, for brevity), typed as:

$$\text{(TApp)}\ \dfrac{\text{(TLam)}\ \dfrac{\Phi_f; \Gamma, x : \mathit{ref}^{\{L\}}\ \mathit{int} \vdash !\,x; !\,x : \mathit{int}}{\Phi_{\emptyset}; \Gamma \vdash \lambda x.\,!\,x; !\,x : \mathit{ref}^{\{L\}}\ \mathit{int} \longrightarrow^{\Phi_f} \mathit{int} \quad (T_1)} \quad \Phi_2; \Gamma \vdash \mathsf{ref}^L\ n : \mathit{ref}^{\{L\}}\ \mathit{int} \quad (T_2) \quad \Phi_{\emptyset} \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi}{\Phi; \Gamma \vdash (\lambda x.\,!\,x; !\,x)\ \ \mathsf{ref}^L\ n : \mathit{int}}$$

The evaluation derivation has the following structure:

$$\text{(CALL)}\;\frac{\begin{array}{c}\langle\emptyset,\emptyset\cup\{L\},H,(\lambda x.\,!\,x;!\,x)\rangle \longrightarrow \langle\emptyset,\emptyset\cup\{L\},H,(\lambda x.\,!\,x;!\,x)\rangle \quad (1)\\ \langle\emptyset,\emptyset\cup\{L\},H,\mathsf{ref}^L\ n\rangle \longrightarrow \langle\emptyset,\emptyset\cup\{L\},H',r_L\rangle \quad (2)\\ \langle\emptyset,\emptyset\cup\{L\},H',(!\,x;!\,x)[x\mapsto r_L]\rangle \longrightarrow_{\{L\}} \langle\emptyset\cup\{L\},\emptyset,H',n\rangle \quad (3)\end{array}}{\langle\emptyset,\emptyset\cup\{L\},H,(\lambda x.\,!\,x;!\,x)\ \ \mathsf{ref}^L\ n\rangle \longrightarrow_{\{L\}} \langle\emptyset\cup\{L\},\emptyset,H',n\rangle}$$

where $H' = (H, r_L \mapsto n)$. Subderivations *(1)* and *(2)* correspond to the two subderivations (T_1) and (T_2) of (TAPP), but there is no analogue for subderivation *(3)*, which captures the actual evaluation of the function. Clearly this relates to the function's effect Φ_f, but how exactly is not structurally apparent from the derivation. Returning to our example, we must match up the effect in the typing derivation for $!\,x$, which is part of the typing of the function $(\lambda x.\,!\,x;!\,x)$, with evaluation of $!\,r_L$ that occurs when the function evaluates in subderivation *(3)*.

To do this, we instrument the big-step semantics from Figure 3 with typing derivations, and define exactly how to associate a typing derivation with each derived subterm in an evaluation derivation. The key property of the resulting *typed operational semantics* is that the contextual effect Φ associated with a subterm e in the original typing derivation T_p is also associated with all terms derived from e via copying or substitution. In the example, the relevant typing subderivation for $!\,x$ in T_p will be copied and substituted according to the evaluation so that it can be matched with $!\,r_L$ in subderivation *(3)*.

4.1 Typed Operational Semantics

In our typed operational semantics, evaluations have the form:

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

where T is a typing derivation for the expression e, and T' is a typing derivation for v:

$$T :: \Phi;\Gamma \vdash e : \tau \qquad\qquad T' :: \Phi_\emptyset;(\Gamma',\Gamma) \vdash v : \tau$$

Note that we include T' in our rules mostly to emphasize that v is well-typed with the same type as e. The only information from T' we need that is not present in T is the new environment (Γ',Γ), which may contain the types of pointers newly allocated in the heap during the evaluation of e. Also, the environments Γ and Γ' only refer to heap locations, since e and v have no free variables and could always be typed under the empty environment.

Figure 4 presents the typed evaluation rules. New hypotheses are highlighted with a gray background. While these rules look complicated, they are actually quite easy to construct. We begin with the original rules in Figure 3, add a typing derivation to each configuration, and then specify appropriate hypotheses about each typing derivation to connect up the derivation of the whole term with the derivation of each of the subterms. We discuss this process for each of the rules.

[ID-A] is the same as [ID], except we introduce typing derivations T_v and T'_v for the left- and right-hand sides of the evaluation, respectively. T_v may be

$$[\textsc{Id-A}]\frac{T_v :: \Phi; \Gamma \vdash v : \tau \qquad T'_v :: \Phi_\emptyset; \Gamma \vdash v : \tau}{\langle T_v, \alpha, \omega, H, v\rangle \longrightarrow_\emptyset \langle T'_v, \alpha, \omega, H, v\rangle}$$

$$[\textsc{Ref-A}]\frac{\begin{array}{cc} \langle T', \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle T_v, \alpha', \omega', H', v\rangle & r \notin \mathrm{dom}(H) \\ T :: \Phi; \Gamma \vdash \mathsf{ref}^L\ e : \mathit{ref}^\varepsilon\ \tau & T' :: \Phi'; \Gamma \vdash e : \tau \\ T_v :: \Phi_\emptyset; \Gamma' \vdash v : \tau \quad T_r :: \Phi_\emptyset; (\Gamma', r \mapsto \tau) \vdash r_L : \mathit{ref}^\varepsilon\ \tau & \Phi' \leq \Phi \end{array}}{\langle T, \alpha, \omega, H, \mathsf{ref}^L\ e\rangle \longrightarrow_\varepsilon \langle T_r, \alpha', \omega', (H', r \mapsto v), r_L\rangle}$$

$$[\textsc{Deref-A}]\frac{\begin{array}{cc} \langle T', \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle T_r, \alpha', \omega' \cup \{L\}, H', r_L\rangle & r \in \mathrm{dom}(H') \\ T :: \Phi; \Gamma \vdash !\,e : \tau & T' :: \Phi_1; \Gamma \vdash e : \mathit{ref}^{\varepsilon'}\ \tau' \\ T_r :: \Phi_\emptyset; \Gamma' \vdash r_L : \mathit{ref}^{\varepsilon'}\ \tau' & T_v :: \Phi_\emptyset; \Gamma' \vdash H'(r) : \tau \\ \Phi' \leq \Phi \qquad \tau' \leq \tau & \Phi_1 \rhd [\alpha_1; \varepsilon'; \omega_1] \hookrightarrow \Phi' \end{array}}{\langle T, \alpha, \omega, H, !\,e\rangle \longrightarrow_{\varepsilon \cup \{L\}} \langle T_v, \alpha' \cup \{L\}, \omega', H', H'(r)\rangle}$$

$$[\textsc{Assign-A}]\frac{\begin{array}{cc} \multicolumn{2}{c}{\langle T_1, \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle T_r, \alpha_1, \omega_1, H_1, r_L\rangle} \\ \multicolumn{2}{c}{\langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \longrightarrow_{\varepsilon_2} \langle T_v, \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v\rangle} \\ T :: \Phi; \Gamma \vdash e_1 := e_2 : \tau & T_1 :: \Phi_1; \Gamma \vdash e_1 : \mathit{ref}^\varepsilon\ \tau' \\ T_r :: \Phi_\emptyset; \Gamma_1 \vdash r_L : \mathit{ref}^\varepsilon\ \tau' & T_2 :: \Phi_2; \Gamma_1 \vdash e_2 : \tau' \\ T_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau' & T'_v :: \Phi_\emptyset; \Gamma_2 \vdash v : \tau \\ \Phi' \leq \Phi \qquad \tau' \leq \tau & \Phi_1 \rhd \Phi_2 \rhd [\alpha_3; \varepsilon; \omega_3] \hookrightarrow \Phi' \end{array}}{\langle T, \alpha, \omega, H, e_1 := e_2\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \langle T'_v, \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v\rangle}$$

$$[\textsc{Call-A}]\frac{\begin{array}{cc} \multicolumn{2}{c}{\langle T_1, \alpha, \omega, H, e_1\rangle \longrightarrow_{\varepsilon_1} \langle T_f, \alpha_1, \omega_1, H_1, \lambda x.e\rangle} \\ \multicolumn{2}{c}{\langle T_2, \alpha_1, \omega_1, H_1, e_2\rangle \longrightarrow_{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2\rangle} \\ \multicolumn{2}{c}{\langle T_3, \alpha_2, \omega_2, H_2, e[v_2 \mapsto x]\rangle \longrightarrow_{\varepsilon_3} \langle T_v, \alpha', \omega', H', v\rangle} \\ T :: \Phi; \Gamma \vdash e_1\ e_2 : \tau & T_1 :: \Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \\ T_f :: \Phi_\emptyset; \Gamma_1 \vdash \lambda x.e : \tau_1 \longrightarrow^{\Phi_f} \tau_2 & T_2 :: \Phi_2; \Gamma_1 \vdash e_2 : \tau_1 \\ T_{v_2} :: \Phi_\emptyset; \Gamma_2 \vdash v_2 : \tau_1 & T_3 :: \Phi_f; \Gamma_2 \vdash e[x \mapsto v_2] : \tau \\ T_v :: \Phi_\emptyset; \Gamma_3 \vdash v : \tau \quad \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi' & \Phi' \leq \Phi \end{array}}{\langle T, \alpha, \omega, H, e_1\ e_2\rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T_v, \alpha', \omega', H, v\rangle}$$

Fig. 4. Typed operational semantics

any typing derivation that assigns a type to v. Here, and in the other rules in the typed operational semantics, we allow subsumption in the typing derivations on the left-hand side of a reduction. Thus T_v may type the value v under some effect Φ that is not $\Phi_\emptyset$. The output typing derivation T'_v is the same as T_v, except it uses the effect $\Phi_\emptyset$ (recall the only information we use from T'_v is the new environment, which is this case is unchanged from T_v).

[REF-A] is a more complicated case. Here the typing derivation T must (by observation of the rules in Figure 2) assign $\mathsf{ref}^L\ e$ a type $ref^{\varepsilon}\ \tau$ and some effect Φ. By inversion, then, we know that T must in fact assign the subterm e the type τ as witnessed by some typing derivation T', which we use in the typed evaluation of e. We allow $\Phi' \leq \Phi$ to account for subsumption applied to the term $\mathsf{ref}^L\ e$. Note that this rule does not specify how to construct T' from T. Later on, we will prove that if there is a valid standard reduction of a well-typed term, then there is a valid typed reduction of the same term. Continuing with the rule, our semantics assigns some typing derivation T_v to v. Then the output typing derivation T_r should assign a type to r_L. Hence we take the environment Γ' from T_v, which contains types for locations in the heap allocated thus far, and extend it with a new binding for r of the correct type.

[DEREF-A] follows the same pattern as above. Given the initial typing derivation T of the term $!\,e$, we assume there exists a typing derivation T' of the appropriate shape for subterm e. Reducing e yields a new typing derivation T_r, and the final typing derivation T_v assigns the type τ to the value $H'(r)$ returned by the dereference. As above, we add subtyping constraints $\Phi' \leq \Phi$ and $\tau' \leq \tau$ to account for subsumption of the term $!\,e$. The most interesting feature of this rule is the last constraint, $\Phi_1 \rhd [\alpha_1; \varepsilon'; \omega_1] \hookrightarrow \Phi'$, which states that the effect $\Phi \geq \Phi'$ of the whole expression $!\,e$ (from typing derivation T) must contain the effect Φ_1 of e followed by some contextual effect containing standard effect ε'. Again, we will prove below that it is always possible to construct a typed derivation that satisfies this constraint, intuitively because [DEREF] from Figure 2 enforces exactly the same constraint. [ASSIGN-A] is similar to [DEREF].

[CALL-A] is the most complex of the four rules, but the approach is exactly the same as above. Starting with typing derivation T for the function application, we require that there exist typing derivations T_1 and T_2 for e_1 and e_2, where the type of e_2 is the domain type of e_1. Furthermore, T_f and T_{v_2} assign the same types as T_1 and T_2, respectively. Then by the substitution lemma, we know there exists a typing derivation T_3 that assigns type τ to the function body e in which the formal x is mapped to the actual v_2. The output typing derivation T_v assigns v the same type τ as T_3 assigns to the function body. We finish the rule with the usual effect sequencing and subtyping constraints.

4.2 Soundness

The semantics in Figure 4 precisely associate a typing derivation—and most importantly, a contextual effect—with each subterm in an evaluation derivation. We prove soundness in two steps. First, we argue that given a typing derivation of a program and an evaluation derivation according to the rules in Figure 3, we can always construct a typed evaluation derivation.

Lemma 3 (Typed evaluation derivations exist). *If $T :: \Phi; \Gamma \vdash e : \tau$ and $D :: \langle \alpha, \omega, H, e \rangle \longrightarrow_{\varepsilon} \langle \alpha'\omega', H', v \rangle$ where $\Gamma \vdash H$, then there exists T_v such that*

$$\langle T, \alpha, \omega, H, e \rangle \longrightarrow_{\varepsilon} \langle T_v, \alpha', \omega', H', v \rangle$$

The proof is by induction on the evaluation derivation D. For each case, we show we can always construct a typed evaluation by performing inversion on the typing derivation T, using T's premises to apply the corresponding typed operational semantics rule. Due to subsumption, we cannot perform direct inversion on T. Instead, we used a number of inversion lemmas (not shown) that generalize the premises of the syntax-driven typing rule that applies to e, for any number of following [TSUB] applications.

Next, we prove that if we have a typed evaluation derivation, then the contextual effects assigned in the derivation soundly model the actual run-time effects. Since contextual effects are non-compositional, we reason about the soundness of contextual effects in a derivation in relation to its context inside a larger derivation. To do that, we use $E_1 \in E_2$ to denote that E_1 is a subderivation of E_2. We define the subderivation relation inductively on evaluation derivations in the typed operational semantics, with base cases corresponding to each evaluation rule, and one inductive case for transitivity. For example, given an application of [CALL-A] (uninteresting premises omitted):

$$\frac{\begin{array}{c} \ldots \\ E_1 :: \langle T_1, \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle T_f, \alpha_1, \omega_1, H_1, \lambda x.e \rangle \\ E_2 :: \langle T_2, \alpha_1, \omega_1, H_1, e_2 \rangle \longrightarrow_{\varepsilon_2} \langle T_{v_2}, \alpha_2, \omega_2, H_2, v_2 \rangle \\ E_3 :: \langle T_3, \alpha_2, \omega_2, H_2, e[v_2 \mapsto x] \rangle \longrightarrow_{\varepsilon_3} \langle T_v, \alpha', \omega', H', v \rangle \end{array}}{E :: \langle T, \alpha, \omega, H, e_1\ e_2 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle T', \alpha', \omega', H, v \rangle}$$

we have $E_1 \in E$, $E_2 \in E$ and $E_3 \in E$. The subderivation relationship is also transitive, i.e., if $E_1 \in E_2$ and $E_2 \in E_3$ then $E_1 \in E_3$.

The following lemma states that if E_2 is an evaluation derivation whose contextual effects are sound (premises 2, 5, and 6) and E_1 is a subderivation of E_2 (premise 3), then the effects of E_1 are sound (conclusions 2 and 3).

Lemma 4 (Soundness of sub-derivation contextual effects). *If*

1. $E_1 :: \langle T_1, \alpha_1, \omega_1, H_1, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle T_{v_1}, \alpha'_1, \omega'_1, H'_1, v_1 \rangle$ *with* $T_1 :: \Phi_1; \Gamma_1 \vdash e_1 : \tau_1$,
2. $E_2 :: \langle T_2, \alpha_2, \omega_2, H_2, e_2 \rangle \longrightarrow_{\varepsilon_2} \langle T_{v_2}, \alpha'_2, \omega'_2, H'_2, v_2 \rangle$ *with* $T_2 :: \Phi_2; \Gamma_2 \vdash e_2 : \tau_2$,
3. $E_1 \in E_2$
4. $\Gamma_2 \vdash H_2$
5. $\alpha_2 \subseteq \Phi_2^{\alpha}$
6. $\omega_2 \subseteq \Phi_2^{\omega}$

then

1. $\Gamma_1 \vdash H_1$
2. $\alpha_1 \subseteq \Phi_1^{\alpha}$
3. $\omega_1 \subseteq \Phi_1^{\omega}$

The proof is by induction on $E_1 \in E_2$. The work occurs in the base cases of the $\in$ relation, and the transitivity case trivially applies induction.

The statement of Lemma 4 may seem odd: we assume a derivation's effects are sound and then prove the soundness of the effects of its subderivation(s).

Nevertheless, this technique is efficacious. If E_2 is the topmost derivation (for the whole program) then the lemma can be trivially applied for E_2 and any of its subderivations, as α_2 and ω'_2 will be $\emptyset$, and thus trivially approximated by the effects defined in Φ_2. Given this, and the fact (from Lemma 3) that typed derivations always exist, we can easily state and prove contextual effect soundness.

Theorem 2 (Contextual Effect Soundness). *Given a program e_p with no free variables, a typing derivation T and a (standard) evaluation D according to the rules in Figure 3, we can construct a typed evaluation derivation*

$$E :: \langle T, \emptyset, \varepsilon_p, \emptyset, e_p \rangle \longrightarrow_{\varepsilon_p} \langle T_v, \varepsilon_p, \emptyset, H, v \rangle$$

such that for every subderivation E' of E:

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

with $T' :: \Phi; \Gamma \vdash e : \tau$, it is always the case that $\alpha \subseteq \Phi^\alpha$, $\varepsilon \subseteq \Phi^\varepsilon$, and $\omega' \subseteq \Phi^\omega$.

This theorem follows as a corollary of Lemma 2, Lemma 3 and Lemma 4, since the initial heap and Γ are empty, and the whole program is typed under $[\emptyset; \varepsilon; \emptyset]$, where ε soundly approximates the effect of the whole program by Theorem 1.

The full (paper) proof can be found in a technical report [6].

5 Mechanization

We encoded the above formalization and soundness proof using the Coq proof assistant. The source code for the formalization and the proof scripts can be found at `http://www.cs.umd.edu/projects/PL/contextual/contextual-coq.tgz`. We were pleased that the mechanization of the system largely followed the paper proof, with only a few minor differences.

First, we used the framework developed by Aydemir et al. [1] for modeling bound and named variables, whereas the paper proof assumes alpha equivalence of all terms and does not reason about capturing and renaming.

Second, Lemma 4 states a property of all subderivations of a derivation. On paper, we had left the definition of subderivation informal, whereas we had to formally define it in Coq. This was straightforward if tedious. In Coq we defined $E \in E'$, described earlier, as an inductive relation, with one case for each premise of each evaluation rule.

While our mechanized proof is similar to our paper proof, it does have some awkwardness. Our encoding of typed operational semantics is dependent on typing derivations, and the encoding of the subderivation relation is dependent on typed evaluations. This causes the definitions of typed evaluations and subderivations to be dependent on large sets of variables, which decreases readability. We were unable to use Coq's system for implicit variables to address this issue, due to its current limitations.

In total, the formalization and proof scripts for the contextual effect system takes 5,503 lines of Coq, of which we wrote 2,692 lines and the remaining 2,811 lines came from Aydemir et al [1]. It took the first author approximately ten days to encode the definitions and lemmas and do the proofs, starting from minimal Coq experience, limited to attending a tutorial at POPL 2008. It took roughly equal time and effort to construct the encodings as to do the actual proofs. In the process of performing the proofs, we discovered some typographical errors in the paper proof, and we found some cases where we had failed to account for possible subsumption in the type and effect system. Perhaps the biggest insight we gained was that to prove Lemma 4, we needed to do induction on the subderivation relation, rather than on the derivation itself.

6 Related Work

Our original paper on contextual effects [7] presented the same type system and operational semantics shown in Sections 2 and 3, but placed scant emphasis on the details of the proof of soundness in favor of describing novel applications. Indeed, we felt that the proof technique described in the published paper was unnecessarily unintuitive and complicated, and that led us to ultimately discover the technique presented in this paper. To our knowledge, ours is the first mechanized proof of a property of typing and evaluation derivations that depends on the positions of subderivations in the super-derivation tree.

Type and effect systems [5, 8, 11] are widely used to statically enforce restrictions, check properties, or in static analysis to infer the behavior of computations [4, 9, 3, 10, 12]. Some more detailed comparisons with these systems can be found in our previous publication [7]. Talpin and Jouvelot [11] use a big-step operational semantics to prove standard effect soundness. In their system, operational semantics are not annotated with effects. Instead, the soundness property is that the static effect, unioned with a static description of the starting heap, describes the heap at the end of the computation. In addition to addressing contextual effects, our operational semantics can also be used as a definition of the *actual* effect (prior, standard, or future) of the computation, regardless of the static system used to infer or check effects. The soundness property for standard effects by Talpin and Jouvelot immediately follows for our system from Theorem 1.

7 Conclusions

This paper presents the proof of soundness for contextual effects [7]. We have mechanized and verified the proof using the Coq proof assistant.

Contextual effect soundness is interesting because the soundness of the effect of e depends on the *position* of e's evaluation within the evaluation derivation of the whole program e_p. That is, the prior and future effects of e depend not on the evaluation of e itself, but rather on the evaluation of e_p prior to, and after, evaluating e, respectively. Adding further complication, a subterm e within the original program, for which the contextual effect is computed by the type and effect system, may change during the evaluation of e_p. In particular, it may be duplicated or modified due to substitutions. To match up these modified terms with the term in the original typing derivation, we employ a novel *typed operational semantics* that correlates the relevant portion of the typing derivation with the evaluation of every subexpression in the program. In mechanizing our proof, we discovered a missing definition (subderivations) in our formal system, and we gained much more assurance that our proof, which had to carefully coordinate the many parts of typed evaluation derivations, was correct.

We conjecture that our proof technique can be used to reason about other non-compositional properties that span a derivation, such as the freshness of a name, or computations that depend on context.

Acknowledgments The authors thank the Penn PLClub for holding the Coq tutorial at POPL 2008, which got us interested in this topic. We thank the anonymous reviewers for their useful feedback. This work was funded in part by NSF grants CNS-0346989, CCF-0541036, CCF-0430118, and the University of Maryland Partnership with the Laboratory for Telecommunications Sciences.

References

1. Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL*, 2008.
2. The Coq proof assistant. `http://coq.inria.fr`.
3. Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock Inference for Atomic Sections. In *TRANSACT*, 2006.
4. Atsushi Igarashi and Naoki Kobayashi. Resource Usage Analysis. In *POPL*, 2002.
5. 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.
6. 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.
7. Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, 2008.
8. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
9. Christian Skalka, Scott Smith, and David Van Horn. Types and trace effects of higher order programs. *Journal of Functional Programming*, July 2007.
10. Fred Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, 2000.
11. Jean-Pierre Talpin and P'ierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.
12. David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *TOPLAS*, 24(4):701–771, July 2000.