An Effective Theory of Type Refinements

Yitzhak Mandelbaum
Princeton University

David Walker [*]
Princeton University

Robert Harper [†]
Carnegie Mellon University

ABSTRACT

We develop an explicit two level system that allows programmers to reason about the behavior of effectful programs. The first level is an ordinary ML-style type system, which confers standard properties on program behavior. The second level is a conservative extension of the first that uses a *logic of type refinements* to check more precise properties of program behavior. Our logic is a fragment of intuitionistic linear logic, which gives programmers the ability to reason *locally* about changes of program state. We provide a generic resource semantics for our logic as well as a sound, decidable, syntactic refinement-checking system. We also prove that refinements give rise to an optimization principle for programs. Finally, we illustrate the power of our system through a number of examples.

Categories and Subject Descriptors

D.3.1 [Programming Languages] : Formal Definitions and Theory—semantics; F.3.1 [Logic and Meanings of Programs] : Specifying and Verifying and Reasoning about Programs—pre- and post-conditions, invariants; F.3.2 [Logic and Meanings of Programs] : Semantics of Programming Languages–operational semantics

General Terms

Languages, Design, Theory

Keywords

Effectful Computation, Type Refinement, Linear Logic, Local Reasoning, Type Theory

[*]Supported in part by NSF Trusted Computing grant CCR-0208601, NSF Career grant CCR-0238328 (Programming Languages for Secure and Reliable Component Software) and a generous gift from Microsoft Research.

[†]Supported by the National Science Foundation under grant CCR-0204248: Type Refinements from September 1, 2002 until August 31, 2005.

1. INTRODUCTION

One of the major goals of programming language design is to allow programmers to express and enforce properties of the execution behavior of programs. Conventional type systems, especially those with polymorphism and abstract types, provide a simple yet remarkably effective means of specifying program properties. However, there remain many properties which, while apparent at compile-time, cannot be checked using conventional type systems.

For this reason, there has been substantial interest in the formulation of *refinements* of conventional types that allow programmers to specify such properties. For example, Davies and Pfenning [6] show how to extend ML with intersection types and domain-specific predicates, and Xi and Pfenning [39] popularized the use of singleton types. They also present compelling applications including static array-bounds checking [38].

A separate research thread has shown how to use "type-and-effect" to check properties of programs involving state. These properties include safe region-based memory management [35], safe locking [11], and the correctness of correspondence assertions for communication protocols [15].

However, neither refinement types nor the many type-and-effect systems attempt to provide a general-purpose logical framework for reasoning about effectful computations. Xi and Pfenning's dependent type system and related work [8, 2, 5, 6] only seek to capture properties of values and pure computations, rather than properties of *effectful* computations. For example, they are unable to describe *protocols* that require effectful functions to be used in a specified order. Therefore, these systems cannot be used to enforce important invariants such as the fact that a lock be held before a data structure is accessed or that a file is opened before being read or closed. On the other hand, with few exceptions, type-and-effect systems, which clearly take state into account, have focused on applying the type-and-effect methodology to solve specific problems, rather than on supporting a parameterized theory and general-purpose logic for encoding domain-specific program invariants.

We propose a new system of type refinements that allows the programmer to reason about program values and state in a unified manner, without being tied to a particular problem area. A type refinement combines a detailed description of the value produced by a computation with a formula describing the state of program resources after the computation's execution. The formula is written in a general logic for reasoning about state, whose atomic predicates are determined by the programmer. We extend our system to higher-order functions by including state pre- and post-conditions in refinements of function types.

Our work can be seen as a continuation of earlier work on refinement types, as well as a general language in which to design and use type-and-effect systems. It also serves to provide a semantic framework for understanding practical work in this area such as the Vault programming language [7]. Overall, the goal of our research is to provide a general, robust and extensible theory of type refinements that captures sound techniques for local reasoning about program state. We describe the main contributions of our system below.

A Two-Level System Based on Conservative Extension. We formalize the notion of a *type refinement* and construct a two level system for checking properties of programs. The first level involves simple type checking and the second level introduces our *logic of refinements* for reasoning about program properties that cannot be captured by conventional types. We establish a formal correspondence between types, which define the structure of a language, and refinements, which define domain-specific properties of programs written in a language. Only Denney [8] has explicitly considered such a two level system in the past, but he restricted his attention to pure computations.

We are careful to ensure that type refinements are a *conservative extension* of types. In other words, type refinements refine the information provided by the underlying, conventional type system rather than replace it with something different. The principle of conservative extension makes it possible for programmers to add type refinements gradually to legacy programs or to developing programs, to make these programs more robust.

A Parameterized Theory. The computational lambda calculus [21] serves as our basic linguistic framework. We parameterize this base language with a set of abstract base types, effectful operators over these types, and possible worlds. Consequently, our theorems hold for a very rich set of possible effects and effectful computations. In addition, we have separated our central type-checking rules from the specifics of the logic of refinements. Our theorems will hold for a variety of fragments of linear logic and we conjecture that similar substructural logics can be used in its place with little or no modification to the core system.

Support for Local Reasoning. It is essential that it be possible to focus attention on a single facet of program state without explicit reference to any others. For example, we may wish to reason about an I/O protocol separately from a locking protocol or a state protocol. Moreover, we may wish to reason separately about two different instances of the same state protocol. Formalisms based on classical logic are only sound for *global* reasoning about the entire state of a program. Not only is this unworkable as a practical matter, it is incompatible with modularity, which is essential for all but the simplest programs.

To support modularity it is necessary to employ a logic for *local* reasoning about independent facets of a program's state [18, 27, 32]. Non-linear logic does not permit local reasoning, essentially because it validates the structural principles of weakening and contraction. Substructural logics, such as Linear Logic [14], or Bunched Implications (BI) [26, 18], do not validate these principles, and so are good candidates for a logic supporting local reasoning. We employ a fragment of linear logic that is adequate for many practical purposes.

For local reasoning to be sound, the underlying effectful operators in the language must in fact "act locally." More precisely, we have identified a crucial *locality condition* on effectful operators that is necessary for soundness in the presence of local reasoning. We have proven the soundness of refinement checking in the presence of this locality condition. The soundness of refinement checking not only provides a means for checking certain correctness criteria, it also entails an optimization principle for effectful operators.

A Decidable System of Type Refinements. While we focus in this paper on a declarative presentation of our system of type refinements, we have developed a decidable, algorithmic refinement-checking system and proven it both sound and complete. A key aspect of the algorithmic system is the introduction of refinement annotations to the syntax of the language. These annotations allow the programmer to guide the checker so that it may search for type-refinement derivations in a deterministic fashion.

A Semantics for an Important Fragment of Vault. We provide a number of examples to demonstrate the expressiveness of our system. Based on these examples, our refinements appear to subsume the state-logic used in the Vault programming language [7] (although our idealized language does not contain the array of data structures present in Vault, nor the specialized type-inference techniques). Hence, our system suggests a semantics for an important fragment of Vault.

A Simple Example: File Access

Before digging in to the technical details of our framework, we present a simple example that introduces a number of important concepts. The example revolves around enforcing a simple resource usage protocol for file access and the following interface defines the types of each operation over files.

$$
\begin{aligned}
&\texttt{fnew} &&: \mathbf{String} \rightharpoonup \mathbf{File}\\
&\texttt{fopen} &&: \mathbf{File} \rightharpoonup \mathbf{unit}\\
&\texttt{fclose} &&: \mathbf{File} \rightharpoonup \mathbf{unit}\\
&\texttt{fwrite} &&: (\mathbf{File}, \mathbf{String}) \rightharpoonup \mathbf{unit}
\end{aligned}
$$

Informally, the protocol for using these operators requires that a file must be open before being written or closed, and should be closed before the program terminates. However, the interface, as presented, has no way to enforce this protocol. We demonstrate this point with a short function and use of that function that type check but do not meet the requirements of the file interface. The key problem is that the function makes an informal assumption about its arguments and the assumption is not satisfied before the function is used. The example is written in a syntax similar to that of Java. Note that `header` and `footer` are intended as strings defined elsewhere in the program.

```
// f must be open
saveEntry(f:File, entry:String) : unit {
  fwrite(f, header);
  fwrite(f, entry);
  fwrite(f, footer);
}

.
.
.

File myFile = fnew(``data.txt'');
saveEntry(myFile, myString); // Error: f not open.
```

To address this type of problem, we begin by specializing the file interface with type refinements that express the access protocol mentioned above.

$$\begin{array}{ll}
\texttt{fnew} & : (\textbf{String};\ \textbf{1}) \rightharpoonup \exists[f : \textbf{File}](\textbf{\textit{Its}}(f);\ \mathsf{closed}(f)) \\
\texttt{fopen} & : (\textbf{\textit{Its}}(f);\ \mathsf{closed}(f)) \rightharpoonup (\textbf{unit};\ \mathsf{open}(f)) \\
\texttt{fclose} & : (\textbf{\textit{Its}}(f);\ \mathsf{open}(f)) \rightharpoonup (\textbf{unit};\ \mathsf{closed}(f)) \\
\texttt{fwrite} & : (\textbf{\textit{Its}}(f), \textbf{String};\ \mathsf{open}(f)) \rightharpoonup (\textbf{unit};\ \mathsf{open}(f))
\end{array}$$

In the refined interface, the function arguments now have two components. The first component is either a refinement that corresponds to a conventional type or a more specific *singleton type* that specifies the exact value of the argument. The second component is a logical formula that describes the state of the system. Function results also have two components, with the added detail that results can be existentially quantified, as is the case with `fnew`. Note that all refinements are implicitly universally quantified over the free variables in the refinement. For example, `fopen` is implicitly quantified with $\forall[f : \textbf{File}]$.

Two kinds of formulas appear in the example: predicates and the formula **1**. As a precondition, **1** states that the function requires, and therefore affects, none of the existing state. As a postcondition it states that the function produces no state.

We can now explain the refined interface. The operator `fnew` takes a **String** (the file name) and requires no existing state. It returns some **File** object f and assures that f is initially closed. `fopen` takes a particular f and changes its state from closed to open. `fclose` does the reverse. Finally, `fwrite` requires that f be open, but does not change any state relating to f.

Next, we add type refinements to the `saveEntry` function and its use, and see how a refinement checker would catch the protocol violation. Notice how we can now formally check what was previously an informal assumption.

```
saveEntry(f:File, entry:String; open(f)): (unit; open(f)) {
  fwrite(f, header);  // State: open(f)
  fwrite(f, entry);   // State: open(f)
  fwrite(f, footer);  // State: open(f)
}

.
.
.

File myFile = fnew(``data.txt''); // State: closed(f)
saveEntry(myFile, myString); // Error: closed(f) != open(f)
```

Finally, we fix the use of `saveEntry` by inserting a call to `fopen` in the appropriate place, as shown below:

```
File myFile = fnew(``data.txt''); // State: closed(f)
fopen(myFile);                     // State: open(f)
saveEntry(myFile, myString);       // State: open(f)
```

Our logic of refinements is much richer than the simple fragment we have used in this example as it includes all of the multiplicative and additive connectives of linear logic and some restricted uses of the modality "!". For some more sophisticated examples, the reader may wish to skip ahead to section 4.

In the next section, we introduce our parameterized base language and its conventional type system (Section 2). In Section 3, we provide the syntax for general first-order refinements and provide a semantics for world (state) refinements. Next, we give a declarative account of refinement checking and discuss how the declarative account is converted to an algorithmic one. Finally, we show that our refinements are a conservative extension of the underlying type system, state the main soundness theorems relating to our system and discuss how refined operators may be optimized. In the last section, we indicate our current research directions and comment further on related work.

Finally, due to space considerations, we have omitted from this paper the proofs of our theorems as well as a number of details that were not critical to the presentation of our work. Readers are encouraged to see our companion technical report [20] for complete details.

2. BASE LANGUAGE

We use Moggi's computational λ-calculus [21] as a basic linguistic framework, as reformulated by Pfenning and Davies [29]. The framework is enriched with recursive functions and a base type of booleans. In order to consider a variety of different sorts of effects, we parameterize the language by a collection of abstract types **a**, constants c with type **a** and a set of multi-ary operators o over these abstract types.

2.1 Abstract Syntax

The abstract syntax of the language is defined by the following grammar:

$$\begin{array}{llcl}
\textit{Types} & A & ::= & \mathbf{a} \mid \textbf{Bool} \mid A_1 \to A_2 \mid A_1 \rightharpoonup A_2 \\
\textit{Var's} & X & ::= & x \mid y \mid \ldots \\
\textit{Values} & V & ::= & X \mid c \mid \textbf{true} \mid \textbf{false} \mid \lambda(X).M \mid \\
 & & & \textbf{fun}\ X\ (X_1{:}A_1) : A_2\ \textbf{is}\ E \\
\textit{Terms} & M & ::= & V \mid \textbf{if}\ M\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \mid M\ (M_1) \\
\textit{Exp's} & E & ::= & M \mid \mathsf{o}(M_1, \ldots, M_k) \mid \textbf{let}\ X\ \textbf{be}\ E_1\ \textbf{in}\ E_2\ \textbf{end} \mid \\
 & & & \textbf{app}(M, M_1) \mid \textbf{if}\ M\ \textbf{then}\ E_1\ \textbf{else}\ E_2
\end{array}$$

The binding conventions are as expected; we identify expressions up to consistent renaming of bound variables. The type $A_1 \to A_2$ is the type of "pure" functions, which always terminate without effect, and the type $A_1 \rightharpoonup A_2$ is the type of "impure" functions, which may not terminate and may have an effect when applied.

2.2 Abstract Resources

Our language is parameterized by a set of abstract, effectful operators, which manipulate some abstract resource or set of resources. We may reason about an instance of the language by specifying an interface for and implementation of these operators and resources. In the future, we intend to extend our language with a full-fledged module system and an internal means of defining new resources.

We summarize the language parameters in Figure 1. An interface Σ defines a set of abstract types $\mathcal{B}$, a set of constants $\mathcal{C}$, and a set of operators $\mathcal{O}$. The interface also provides a signature Σ_A that gives types to the constants and operators. When we come to checking refinements, we will do so with respect to a set of predicates $\mathcal{P}$, an interface Σ_p to specify the types of predicate arguments and finally, a signature Σ_ϕ to define the refinements of each constant or operator.

An implementation $\mathcal{M} = (\mathcal{W}, \mathcal{T})$ defines a set $\mathcal{W}$ of worlds w, and a transition function $\mathcal{T}$ between these worlds that specifies the behavior of the operators over constants of the appropriate types.

A world w is a pair $(\mathsf{Per}(w), \mathsf{Eph}(w))$ where $\mathsf{Per}(w)$ is a set of *persistent* facts and $\mathsf{Eph}(w)$ is a multiset of *ephemeral*

Interface	Contents
$\mathcal{B}$	Base Types
$\mathcal{C}$	Constant Names
$\mathcal{O}$	Operator Names
Σ_A	Constant and Operator Types
$\mathcal{P}$	Predicates
Σ_p	Predicate Types
Σ_ϕ	Constant and Operator Refinements
Implementation	**Meaning**
$\mathcal{W}$	Worlds
$\mathsf{Per}(w)$	w's Persistent Facts
$\mathsf{Eph}(w)$	w's Ephemeral Facts
$\mathcal{T}(\mathsf{o})$	o's Behavior

Figure 1: Language Parameters

facts. The persistent facts of a world w will remain true in all worlds that can be reached through a computation starting with w. The ephemeral facts of a world may or may not hold in its future worlds.

The notation $w_1 + w_2$ denotes a world containing the union of the persistent facts from w_1 and w_2, and the multi-set union of ephemeral facts from w_1 and w_2.

If an operator is given type $\mathbf{a}_1, \ldots, \mathbf{a}_n \rightharpoonup \mathbf{a}$ by an interface, then the transition function $\mathcal{T}(\mathsf{o})$ is a total function from a sequence of constants with types $\mathbf{a}_1, \ldots, \mathbf{a}_n$ and world w to a constant with type $\mathbf{a}$ and world w'. We use the symbol $\rightharpoonup$ to note that while these operators always terminate, they may have effects on the world. We require that these functions act monotonically on the persistent facts in the world. In other words, if $\mathcal{T}(\mathsf{o})(c_1, \ldots, c_n; w) = (c; w')$ then $\mathsf{Per}(w) \subseteq \mathsf{Per}(w')$.

The transition function $\mathcal{T}(\mathsf{o})$ must also obey a *locality* condition. In general, it may only have an effect on a part of the world, rather than the entire world. Most operators that one would like to define obey this locality condition. However, some useful operators do not. For example, in our system, programmers may not reason statically about a function such as $\mathsf{gc}(\mathit{roots})$, which deletes all resources except the resources referenced from the variable *roots*. We defer a formal explanation of this condition to Section 3.5 where we prove the soundness of refinement checking.

Example: File Access (continued). We now consider parameterizing the language with the file access primitives shown earlier in Section 1. We require three base types: a type for files **File**, for data **String** and the unit type **unit**. Our constants include a countable set of file handles and data (we use metavariables f and d to range over each of these sets, respectively) and a unit value (). The signature Σ_A provides the types for the operations and constants.

$$
\begin{array}{lcl}
\mathcal{T}(\texttt{fnew})((); w) & = & (f; w') \\
& & \text{where } \mathsf{Eph}(w') = \mathsf{Eph}(w) + \{\mathsf{closed}(f)\} \\
& & \text{and } f \notin \mathsf{Eph}(w) \\[1ex]
\mathcal{T}(\texttt{fopen})(f; w) & = & ((); w') \\
& & \text{if } w = w'' + \{\mathsf{closed}(f)\} \\
& & \text{and } \mathsf{Eph}(w') = \mathsf{Eph}(w'') + \{\mathsf{open}(f)\} \\[1ex]
\mathcal{T}(\texttt{fopen})(f; w) & = & ((); w) \\
& & \text{if } w \neq w'' + \{\mathsf{closed}(f)\} \\[1ex]
\mathcal{T}(\texttt{fclose})(f; w) & = & ((); w') \\
& & \text{if } w = w'' + \{\mathsf{open}(f)\} \\
& & \text{and } \mathsf{Eph}(w') = \mathsf{Eph}(w'') + \{\mathsf{closed}(f)\} \\[1ex]
\mathcal{T}(\texttt{fclose})(f; w) & = & ((); w) \\
& & \text{if } w \neq w'' + \{\mathsf{open}(f)\}
\end{array}
$$

Figure 2: Transition Function of File Operators

$$
\begin{array}{lcl}
\Sigma_A(()) & = & \mathbf{unit} \\
\Sigma_A(f) & = & \mathbf{File} \\
\Sigma_A(d) & = & \mathbf{String} \\
\Sigma_A(\texttt{fnew}) & = & \mathbf{unit} \rightharpoonup \mathbf{File} \\
\Sigma_A(\texttt{fopen}) & = & \mathbf{File} \rightharpoonup \mathbf{unit} \\
\Sigma_A(\texttt{fclose}) & = & \mathbf{File} \rightharpoonup \mathbf{unit} \\
\Sigma_A(\texttt{fwrite}) & = & (\mathbf{File}, \mathbf{String}) \rightharpoonup \mathbf{unit}
\end{array}
$$

The set of predicates $\mathcal{P}$ is exactly the predicates closed and open from our earlier example.

In the implementation component, we must specify the set of worlds and the behavior of the operators. Notice that there are no persistent predicates, and so the persistent set of any world is always empty. However, once a file f is initialized with a call to `fnew`, either $\mathsf{open}(f)$ or $\mathsf{closed}(f)$ will be in the ephemeral facts any future world. Therefore, the $\mathcal{W}$ of the implementation is the countably infinite set of worlds w, where $\mathsf{Per}(w) = \emptyset$ and $\mathsf{Eph}(w)$ describes a set of files that are all either open or closed, but not both. The transition function $\mathcal{T}$ specifies the dynamic semantics for each operator and is shown in Figure 2. Note that the side condition $f \notin \mathsf{Eph}(w)$ indicates that f does not appear in any of the predicates in $\mathsf{Eph}(w)$ (i.e. f is "fresh").

A key aspect of our definition of $\mathcal{T}$ is that each of the operators are defined to be *total* functions on the entire domain of worlds. If they were not total functions we would be unable to prove a generic soundness theorem for our language. Later (see Section 3.7), we will prove an optimization principle that allows programmers to replace these total functions with the appropriate partial functions when their program has the necessary refinement.

2.3 Semantics

Since our base language semantics is almost entirely standard, we merely state the forms of the various judgments. Complete details can be found in our companion technical report [20].

The static semantics is given by the following two judgement forms.

$$
\begin{array}{ll}
\Gamma \vdash_{\mathsf{M}} M : A & \textit{Term } M \textit{ has type } A \textit{ in } \Gamma \\
\Gamma \vdash_{\mathsf{E}} E : A & \textit{Expression } E \textit{ has type } A \textit{ in } \Gamma
\end{array}
$$

The meta-variable Γ ranges over finite functions from variables x to types A.

The dynamic semantics is given by the following two evaluation judgements.

$M \Downarrow V$	*the term M evaluates to value V*
$E \,@\, w \Downarrow V \,@\, w'$	*in w the expression E evaluates to V and changes to w'*

The following rule defines evaluation for effectful operators using the language parameter $\mathcal{T}$.

$$\frac{M_i \Downarrow c_i \qquad (\text{for } 1 \leq i \leq n) \qquad \mathcal{T}(\mathsf{o})(c_1, \ldots, c_n; w) = c'; w'}{\mathsf{o}(M_1, \ldots, M_n) \,@\, w \Downarrow c \,@\, w'} \quad (\text{D-E-OP})$$

Aside from this rule, the dynamic semantics is entirely standard. The language as a whole satisfies the standard type safety theorem [20].

3. REFINEMENTS

In order to define and check further, more specific, properties of values and computations than supported by the type system alone, we introduce a *logic of refinements* that may be layered on top of the computational lambda calculus described in the previous section.

3.1 Syntax

Our logic contains three classes of refinements. A *term refinement* is a predicate over a type, describing a more specific property of a term than a type alone. A *world refinement* is a formula describing the (implicit) state of a world. An *expression refinement* is a predicate over both the type of an expression and the state of the world after the expression is executed. The table below describes the syntax of term, world and expression refinements.

Binding	b	$::=$	$c{:}\mathbf{a}$
Term Refs	ϕ	$::=$	$\boldsymbol{a} \mid \boldsymbol{Bool} \mid \boldsymbol{Its}(c) \mid \pi$
Function Refs	π	$::=$	$\phi_1 \rightarrow \phi_2 \mid (\phi;\psi) \rightharpoonup \eta \mid \forall b \cdot \pi$
World Refs	ψ	$::=$	$p(c_1, \ldots, c_n) \mid !p(c_1, \ldots, c_n) \mid \mathbf{1} \mid \psi_1 \otimes \psi_2 \mid \psi_1 \multimap \psi_2 \mid \top \mid \psi_1 \,\&\, \psi_2 \mid \mathbf{0} \mid \psi_1 \oplus \psi_2$
Expr. Refs	η	$::=$	$\exists[\vec{b}](\phi;\psi)$

Since we are concentrating on properties of effectful computations, we have chosen a minimalist logic of term refinements. There is a refinement that corresponds to each type in the base language as well as *singleton types* denoted $\boldsymbol{Its}(c)$. Partial functions are refined in order to specify a precondition for the state of the world on input and a postcondition consisting of an expression refinement. The precondition for a partial function could also have been an (existentially quantified) expression refinement, but this extension provides no gain in expressive power. We allow function refinements to be prefixed with first-order universal quantification.

The world refinements consist of the multiplicative-additive fragment of linear logic augmented with intuitionistic predicates $!p(c_1, \ldots, c_n)$. The connectives $\mathbf{1}$, $\otimes$ and $\multimap$ form the multiplicative fragment of the logic whereas the connectives $\top$, $\&$, $\mathbf{0}$, and $\oplus$ are known as the additives. Both $\otimes$ and $\&$ are forms of conjunction. Intuitively, a world can be described by the formula $\psi_1 \otimes \psi_2$ if it can be split into two disjoint parts such that one part can be described by ψ_1 and the other part can be described by ψ_2. On the other hand, a world satisfies $\psi_1 \& \psi_2$ if it can be described by both

$$\frac{}{\vec{b} \vdash \boldsymbol{Bool} \sqsubseteq \mathbf{Bool}} \qquad \frac{}{\vec{b} \vdash \boldsymbol{a} \sqsubseteq \mathbf{a}} \qquad \frac{\Sigma_A(c) = \mathbf{a} \quad \text{or} \quad c{:}\mathbf{a} \in \vec{b}}{\vec{b} \vdash \boldsymbol{Its}(c) \sqsubseteq \mathbf{a}}$$

$$\frac{\vec{b} \vdash \phi_1 \sqsubseteq A_1 \quad \vec{b} \vdash \phi_2 \sqsubseteq A_2}{\vec{b} \vdash \phi_1 \rightarrow \phi_2 \sqsubseteq A_1 \rightarrow A_2} \qquad \frac{\vec{b} \vdash \phi_1 \sqsubseteq A_1 \quad \vec{b} \vdash \eta_2 \sqsubseteq_E A_2}{\vec{b} \vdash (\phi_1;\psi_1) \rightharpoonup \eta_2 \sqsubseteq A_1 \rightharpoonup A_2}$$

$$\frac{\vec{b}, c{:}\mathbf{a} \vdash \pi \sqsubseteq A}{\vec{b} \vdash \forall c{:}\mathbf{a} \cdot \pi \sqsubseteq A} \qquad \frac{\vec{b} \vdash \phi \sqsubseteq A}{\vec{b} \vdash (\phi;\psi) \sqsubseteq_E A} \qquad \frac{\vec{b}, c{:}\mathbf{a} \vdash \exists[\vec{b}_1](\phi;\psi) \sqsubseteq_E A}{\vec{b} \vdash \exists[c{:}\mathbf{a}, \vec{b}_1](\phi;\psi) \sqsubseteq_E A}$$

Figure 3: A Refinement of a Type

ψ_1 and ψ_2 simultaneously. The formulas $\mathbf{1}$ and $\top$ are the identities for $\otimes$ and $\&$ respectively. The formula $\oplus$ is a disjunction and $\mathbf{0}$ is its identity.

When $\vec{b}$ is the empty sequence in some expression refinement $\exists[\vec{b}](\phi;\psi)$, we often use the abbreviation $(\phi;\psi)$. We use the notation $\mathsf{FV}(\phi)$ to denote the set of free variables appearing in the term refinement ϕ. We use a corresponding notation for world and expression refinements. We use the notation $[c'/b]X$ to denote capture-avoiding substitution of c' for c in term or world refinement X when $b = (c{:}\mathbf{a})$ and $\Sigma_A(c') = \mathbf{a}$. We extend this notation to substitution for a sequence of bindings as in $[c'_1, \ldots, c'_n/\vec{b}]X$ or $[\vec{b}'/\vec{b}]X$. In either case, constants substituted for variables must have the correct type and the sequences must have the same length or else the substitution is undefined. We also extend substitution to persistent and ephemeral contexts in the ordinary way.

Every refinement refines a particular type. To formalize this relationship, we define two new judgments $\vec{b} \vdash \phi \sqsubseteq A$ and $\vec{b} \vdash \eta \sqsubseteq_E A$, which indicate that a term or expression refinement refines the type A given the set of bindings $\vec{b}$. Figure 3 defines these relations. Below, we present a lemma stating that any refinement refines a unique type.

Lemma 1

- *If $\vec{b} \vdash \phi \sqsubseteq A_1$ and $\vec{b} \vdash \phi \sqsubseteq A_2$ then $A_1 = A_2$.*
- *If $\vec{b} \vdash \eta \sqsubseteq_E A_1$ and $\vec{b} \vdash \eta \sqsubseteq_E A_2$ then $A_1 = A_2$.*

Finally, for every type A, there is a trivial refinement $\mathsf{triv}(A)$ that refines it.

$$\begin{array}{lcl}
\mathsf{triv}(\mathbf{Bool}) & = & \boldsymbol{Bool} \\
\mathsf{triv}(\mathbf{a}) & = & \boldsymbol{a} \\
\mathsf{triv}(A_1 \rightarrow A_2) & = & \mathsf{triv}(A_1) \rightarrow \mathsf{triv}(A_2) \\
\mathsf{triv}(A_1 \rightharpoonup A_2) & = & (\mathsf{triv}(A_1);\top) \rightharpoonup (\mathsf{triv}(A_2);\top)
\end{array}$$

3.2 Semantics of Refinements

We were inspired to define a semantics for our world refinements by the work of Ishtiaq and O'Hearn [18]. The semantics appears in Figure 4. The model (world) w used in the semantics is that described earlier in section 2.2. The fragment of the logic without the modality ! is an instance of Simon Ambler's resource semantics [1, p. 30-32]. It relies upon an abstract relation $\lesssim$ which defines the relationship between primitive facts. For example, in a system containing arithmetic predicates such as $\mathsf{less(x,y)}$, the relation would include $\mathsf{less}(\mathsf{x}, 3) \lesssim \mathsf{less}(\mathsf{x}, 5)$. In most of our examples, the

$w \vDash \psi$ if and only if

- $\psi = p(c_1, \ldots, c_n)$, $\mathsf{Eph}(w) = \{X\}$ and $X \lesssim p(c_1, \ldots, c_n)$
- $\psi = !p(c_1, \ldots, c_n)$, $X \in \mathsf{Per}(w)$ and $X \lesssim p(c_1, \ldots, c_n)$ and $\mathsf{Eph}(w) = \emptyset$
- $\psi = \mathbf{1}$ and $\mathsf{Eph}(w) = \emptyset$
- $\psi = \psi_1 \otimes \psi_2$ and there exist w_1, w_2, such that $w = w_1 + w_2$ and $w_1 \vDash \psi_1$ and $w_2 \vDash \psi_2$
- $\psi = \psi_1 \multimap \psi_2$ and for all worlds w_1 such that $w_1 \vDash \psi_1$, $w_1 + w \vDash \psi_2$
- $\psi = \top$ (and no other conditions need be satisfied)
- $\psi = \psi_1 \,\&\, \psi_2$ and $w \vDash \psi_1$ and $w \vDash \psi_2$
- $\psi = \mathbf{0}$ and false (this refinement can never be satisfied).
- $\psi = \psi_1 \oplus \psi_2$ and either $w \vDash \psi_1$ or $w \vDash \psi_2$.

$w \vDash \Omega$ iff $\mathsf{Per}(w) \supseteq \Omega$.
$w \vDash \cdot$ iff $\mathsf{Eph}(w) = \emptyset$.
$w \vDash \psi_1, \ldots, \psi_n$ iff there exist $w_1, \ldots, w_n$ such that

- $w = w_1 + \cdots + w_n$ and
- $w_1 \vDash \psi_1, \ldots, w_n \vDash \psi_n$.

$w \vDash \Omega; \Delta$ iff $w \vDash \Omega$ and $w \vDash \Delta$.

Figure 4: Semantics of World Refinements and Contexts

relation $\lesssim$ will simply be the identity relation. In other words, our predicates are usually left uninterpreted.

The semantics of world refinements is extended to closed persistent contexts Ω (lists of predicates $p(\vec{c})$) and ephemeral contexts Δ (lists of world refinements), and appears in Figure 4. We treat both kinds of contexts as equivalent up to reordering of their elements.[1]

We will show later that linear logical entailment is sound with respect to our semantics. However, as noted by Ambler [1, p. 32], there is no sense in which linear logical reasoning is complete with respect to this semantics. Despite this deficiency, linear logic has proven to be very useful for many applications. We leave definition of a sound and complete logic for our resource semantics to future work.

3.3 Declarative Refinement-Checking

In this section, we give a declarative account of how to check that a (possibly open) term or expression has a given refinement. Refinement checking of open terms will occur within a context of the following form. Whenever we consider the semantics of refinements or refinement checking, we presuppose that the values, terms and expressions in question are well-formed with an appropriate type.

Persistent Ctxt	Ω	$::=$	$\cdot \mid \Omega, c{:}\mathbf{a} \mid \Omega, x{:}\phi \mid \Omega, p(\vec{c})$
Ephemeral Ctxt	Δ	$::=$	$\cdot \mid \Delta, \psi$

Furthermore, we define a derivative form of context, Ω_b to be a vector, $\vec{b}$, consisting of all elements in Ω of the form $c{:}\mathbf{a}$.

Persistent contexts are constrained so that the variables c and x appear at most once to the left of any : in the context and no variables c may shadow any of the constants in $\mathcal{C}$. When necessary, we will implicitly alpha-vary bound

[1]When we extend Ω to open contexts which include constant declarations, reordering must respect the dependencies introduced by such declarations (see Section 3.3).

$\vdash \Sigma$ ok	*Signature Σ is well-formed*
$\vdash \Omega$ ok	*Context Ω is well-formed*
$\Omega \vdash \Delta$ ok	*Context Δ is well-formed in Ω*
$\Omega \vdash \phi$ ok	*Refinement ϕ is well-formed in Ω*
$\Omega \vdash \psi$ ok	*World ref. ψ is well-formed in Ω*
$\Omega \vdash \eta$ ok	*Expression ref. η is well-formed in Ω*
$\Omega \gg_M M : \phi$	*Term M has refinement ϕ in Ω*
$\Omega; \Delta \gg_E E : \eta$	*Expression E has ref. η in $\Omega; \Delta$*
$\Omega; \phi \Longrightarrow_M \phi'$	*Term refinement ϕ entails ϕ' in Ω*
$\Omega; \Delta \Longrightarrow_W \psi$	*Context Δ entails ψ in Ω*
$\Omega; \Delta; \eta \Longrightarrow_E \eta'$	*Expression ref. η entails η' in $\Omega; \Delta$*
$\Omega; \Delta \rightsquigarrow (\Omega_i; \Delta_i)_n$	*Context $\Omega; \Delta$ reduces to the context list $(\Omega_i; \Delta_i)_n$ in one step*
$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n$	*Context $\Omega; \Delta$ reduces to the context list $(\Omega_i; \Delta_i)_n$ in 0 or more steps*

Figure 5: Refinement-Checking Judgments

variables to maintain this invariant. We treat contexts that differ only in the order of the elements as equivalent and do not distinguish them (provided both contexts in question are well formed; in other words, reordering must respect dependencies.). Occasionally, we call the persistent context *unrestricted* and the ephemeral context *linear*. Both contraction and weakening hold for the unrestricted context while neither of these structural properties holds for the linear context.

Declarative refinement checking is formulated using the judgment forms in Figure 5. All but the first judgment are implicitly parameterized by a fixed, well-formed interface Σ.

The first six judgments in the list are relatively standard. They simply check that each sort of type or context is well-formed in the context Ω. This check amounts to the fact that constants and variables that appear in a type or context appear bound previously in the context or in the signature. The formal rules appear in the technical report.

The next two judgments form the heart of the system. They check terms and expressions to ensure that they have appropriate refinements. The term refinement-checking rules may be found in Figure 6. We point out a few important details. In rule (R-T-CONST) constants c are given very precise *singleton types*, following work by Xi and Pfenning [39]. Also, rule (R-T-TAPP) does not consider the case that the function in an application has a polymorphic refinement. This possibility is taken care of by the (R-T-SUB) rule, which instantiates universal quantifiers implicitly. Such instantiations can be resolved by standard first-order unification. Furthermore, rule (R-T-IF) does not check that the first term M has a boolean refinement, because we assume that refinement checking is preceded by ordinary type checking.

The expression refinement-checking rules appear in Figure 7. Rule (R-E-TERM) defines the interface between pure and effectful computations. As pure terms neither require nor produce state, they are checked in an empty ephemeral context and given the world refinement $\mathbf{1}$. We use the (R-E-SUB) rule (discussed in more detail below) to properly check terms within a non-empty ephemeral context. The rule for checking operators requires that we guess a sequence of con-

$$\frac{}{\Omega, x:\phi \gg_M x:\phi} \qquad (\text{R-T-Var})$$

$$\frac{c \in \mathsf{Dom}(\Sigma_\phi)}{\Omega \gg_M c : \textbf{\textit{Its}}(c)} \qquad (\text{R-T-Const})$$

$$\frac{}{\Omega \gg_M \textbf{true} : \textbf{\textit{Bool}}} \qquad (\text{R-T-True})$$

$$\frac{}{\Omega \gg_M \textbf{false} : \textbf{\textit{Bool}}} \qquad (\text{R-T-False})$$

$$\frac{\begin{array}{c}\Omega_b, \vec{b} \vdash \phi_1 \sqsubseteq A \qquad \Omega, \vec{b} \vdash \phi_1\ \mathsf{ok} \\ \Omega, \vec{b}, x{:}\phi_1 \gg_M M : \phi_2 \quad (\phi = \forall \vec{b} \cdot \phi_1 \to \phi_2)\end{array}}{\Omega \gg_M \lambda(x{:}A).M : \phi} \qquad (\text{R-T-Lam})$$

$$\frac{\begin{array}{c}\Omega_b \vdash \phi \sqsubseteq A_1 \rightharpoonup A \quad \Omega \vdash \phi\ \mathsf{ok} \\ \Omega, x{:}\phi, \vec{b}, x_1{:}\phi_1; \psi_1 \gg_E E : \eta \\ (\phi = \forall \vec{b} \cdot (\phi_1; \psi_1) \rightharpoonup \eta)\end{array}}{\Omega \gg_M \textbf{fun}\ x\ (x_1{:}A_1) : A\ \textbf{is}\ E : \phi} \qquad (\text{R-T-Fun})$$

$$\frac{\Omega \gg_M M_1 : \phi \quad \Omega \gg_M M_2 : \phi}{\Omega \gg_M \textbf{if}\ M\ \textbf{then}\ M_1\ \textbf{else}\ M_2 : \phi} \qquad (\text{R-T-If})$$

$$\frac{\Omega \gg_M M : \phi_1 \to \phi_2 \quad \Omega \gg_M M_1 : \phi_1}{\Omega \gg_M M\,(M_1) : \phi_2} \qquad (\text{R-T-TApp})$$

$$\frac{\Omega \gg_M M : \phi \quad \Omega; \phi \Longrightarrow_M \phi'}{\Omega \gg_M M : \phi'} \qquad (\text{R-T-Sub})$$

Figure 6: Refinement-Checking for Terms

$$\frac{\Omega \gg_M M : \phi}{\Omega; \cdot \gg_E M : (\phi, \mathbf{1})} \qquad (\text{R-E-Term})$$

$$\frac{\begin{array}{c}\Omega \gg_M M_i : [\vec{c}/\vec{b}]\phi_i \quad (\text{for } 1 \le i \le n) \\ (\Sigma_\phi(\mathsf{o}) = \forall \vec{b} \cdot (\phi_1, \ldots, \phi_n; \psi_1) \rightharpoonup \eta)\end{array}}{\Omega; [\vec{c}/\vec{b}]\psi_1 \gg_E \mathsf{o}(M_1, \ldots, M_n) : [\vec{c}/\vec{b}]\eta} \qquad (\text{R-E-Op})$$

$$\frac{\begin{array}{c}\Omega; \Delta \gg_E E_1 : \eta_1 \\ \Omega, \vec{b}_1, x{:}\phi_1; \psi_1 \gg_E E_2 : \eta_2 \\ (\vec{b}_1 \notin \mathsf{FV}(\eta_2))\end{array}}{\Omega; \Delta \gg_E \textbf{let}\ x\ \textbf{be}\ E_1\ \textbf{in}\ E_2\ \textbf{end} : \eta_2}\ (\eta_1 = \exists[\vec{b}_1](\phi_1; \psi_1)) \qquad (\text{R-E-Let})$$

$$\frac{\Omega \gg_M M : (\phi_1; \psi_1) \rightharpoonup \eta \quad \Omega \gg_M M_1 : \phi_1}{\Omega; \psi_1 \gg_E \textbf{app}(M, M_1) : \eta} \qquad (\text{R-E-PApp})$$

$$\frac{\Omega; \Delta \gg_E E_1 : \eta \quad \Omega; \Delta \gg_E E_2 : \eta}{\Omega; \Delta \gg_E \textbf{if}\ M\ \textbf{then}\ E_1\ \textbf{else}\ E_2 : \eta} \qquad (\text{R-E-If})$$

$$\frac{\begin{array}{c}\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n \\ \Omega_i; \Delta_i \gg_E E : \eta \quad (\text{for } 1 \le i \le n)\end{array}}{\Omega; \Delta \gg_E E : \eta} \qquad (\text{R-E-Context})$$

$$\frac{\Omega; \Delta_2 \Longrightarrow_W \psi \quad \Omega; \Delta_1, \psi \gg_E E : \eta}{\Omega; \Delta_1, \Delta_2 \gg_E E : \eta} \qquad (\text{R-E-Cut})$$

$$\frac{\Omega; \Delta_1 \gg_E E : \eta \quad \Omega; \Delta_2; \eta \Longrightarrow_E \eta'}{\Omega; \Delta_1, \Delta_2 \gg_E E : \eta'} \qquad (\text{R-E-Sub})$$

Figure 7: Refinement-Checking for Expressions

stants to substitute for the polymorphic parameters in the operator refinement. Given this substitution, we check that operator arguments may be given refinements equal to their corresponding formal parameter.

There are three expression checking rules that are not syntax-directed. (R-E-Sub) merits special attention as it is the key to local reasoning. The rule splits the context into two disjoint parts, Δ_1 and Δ_2, where Δ_1 is used to check the expression E, and Δ_2 passes through unused. As a result, the computation may be written in ignorance of the total global state. It need only know how to process the local state in Δ_1. In fact, in the case that Δ_1 is empty, the computation may be completely pure. Additionally, (R-E-Sub) serves as a conventional subsumption rule in which we check that one expression refinement entails the other. (R-E-Cut) is the logical cut rule: If we can prove some intermediary result (ψ) which in turn makes it possible to demonstrate our final goal ($E : \eta$) then we should be able to prove our final goal from our original premises. Since Δ contains linear hypotheses that must not be duplicated, we split the context into two parts Δ_1 and Δ_2, one part for each premise in the rule.

Finally, since proofs in substructural logics require careful manipulation of the context, we introduce a new rule (R-E-Context) to control context evolution during type checking. This rule depends upon the judgment $\Omega; \Delta \rightsquigarrow (\Omega_i; \Delta_i)_n$ which encodes the action of all natural left rules from the sequent calculus for linear logic. The notation $(\Omega_i; \Delta_i)_n$ stands for a list of (possibly zero) contexts $(\Omega_1; \Delta_1)$, …, $(\Omega_n; \Delta_n)$. We specifically use the word *reduces* since every valid judgment of this form reduces the number of connectives in the context when read from left to right. Most of the rules produce one context. However, the rule for disjunction produces two contexts (and E must have the same refinement in both of them) and the rule for falsehood produces no context (and we can choose any well-formed expression refinement for E without further checking). We extend the one-step context reduction judgment to its reflexive and transitive closure, which we denote $\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n$.

The last five judgments involved in refinement checking specify the logical component of the system. We have already discussed the context reduction judgments. This judgment is combined with the right rules from the sequent calculus and the cut rule in the judgment $\Omega; \Delta \Longrightarrow_W \psi$ to provide a full proof system for our fragment of linear logic. The judgment $\Omega; \phi \Longrightarrow_M \phi'$ is the corresponding proof system for term refinements. Notice that these rules do not depend upon the linear context Δ. Since terms are pure, their refinements should not depend upon ephemeral state. Finally, the judgment for expression refinement entailment $\Omega; \Delta; \eta \Longrightarrow_E \eta'$ combines the world and term proof systems with rules for existentials. These judgments are formally defined in Figures 8, 9, 10 and 11.

3.4 Algorithmic Refinement-Checking

We have developed an algorithmic refinement-checking system that is both sound and complete with respect to the

$$\frac{}{\Omega; \boldsymbol{a} \Longrightarrow_M \boldsymbol{a}} \qquad \text{(L-T-Base)}$$

$$\frac{}{\Omega; \textbf{\textit{Bool}} \Longrightarrow_M \textbf{\textit{Bool}}} \qquad \text{(L-T-Bool)}$$

$$\frac{}{\Omega; \textbf{\textit{Its}}(c) \Longrightarrow_M \textbf{\textit{Its}}(c)} \qquad \text{(L-T-Its)}$$

$$\frac{\Sigma_A(c) = \mathbf{a} \text{ or } \Omega(c) = \mathbf{a}}{\Omega; \textbf{\textit{Its}}(c) \Longrightarrow_M \boldsymbol{a}} \qquad \text{(L-T-ItsBase)}$$

$$\frac{\Omega; \phi_1' \Longrightarrow_M \phi_1 \quad \Omega; \phi_2 \Longrightarrow_M \phi_2' \quad \Omega \vdash \phi_1' \text{ ok}}{\Omega; \phi_1 \to \phi_2 \Longrightarrow_M \phi_1' \to \phi_2'} \qquad \text{(L-T-TArr)}$$

$$\frac{\begin{array}{lll} \Omega; \phi_1' \Longrightarrow_M \phi_1 & \Omega; \psi_1' \Longrightarrow_W \psi_1 & \Omega; \cdot; \eta \Longrightarrow_E \eta' \\ \Omega \vdash \phi_1' \text{ ok} & \Omega \vdash \psi_1' \text{ ok} & \end{array}}{\Omega; (\phi_1; \psi_1) \rightharpoonup \eta \Longrightarrow_M (\phi_1'; \psi_1') \rightharpoonup \eta'} \qquad \text{(L-T-PArr)}$$

$$\frac{\Omega; [c'/c{:}\mathbf{a}]\pi \Longrightarrow_M \pi'}{\Omega; \forall c{:}\mathbf{a} \cdot \pi \Longrightarrow_M \pi'} \qquad \text{(L-T-AllL)}$$

$$\frac{\Omega, c{:}\mathbf{a}; \pi \Longrightarrow_M \pi'}{\Omega; \pi \Longrightarrow_M \forall c{:}\mathbf{a} \cdot \pi'} \qquad \text{(L-T-AllR)}$$

Figure 8: Entailment for Term Refinements

$$\frac{}{\Omega; \Delta, !p(c_1, \ldots, c_n) \rightsquigarrow \Omega, p(c_1, \ldots, c_n); \Delta} \qquad \text{(CR-!)}$$

$$\frac{}{\Omega; \Delta, \mathbf{1} \rightsquigarrow \Omega; \Delta} \qquad \text{(CR-1)}$$

$$\frac{}{\Omega; \Delta, \psi_1 \otimes \psi_2 \rightsquigarrow \Omega; \Delta, \psi_1, \psi_2} \qquad \text{(CR-MAnd)}$$

$$\frac{\Omega; \Delta_1 \Longrightarrow_W \psi_1}{\Omega; \Delta_1, \Delta_2, \psi_1 \multimap \psi_2 \rightsquigarrow \Omega; \Delta_2, \psi_2} \qquad \text{(CR-Imp)}$$

$$\frac{}{\Omega; \Delta, \psi_1 \,\&\, \psi_2 \rightsquigarrow \Omega; \Delta, \psi_1} \qquad \text{(CR-And1)}$$

$$\frac{}{\Omega; \Delta, \psi_1 \,\&\, \psi_2 \rightsquigarrow \Omega; \Delta, \psi_2} \qquad \text{(CR-And2)}$$

$$\frac{}{\Omega; \Delta, \mathbf{0} \rightsquigarrow} \qquad \text{(CR-Zero)}$$

$$\frac{}{\Omega; \Delta, \psi_1 \oplus \psi_2 \rightsquigarrow (\Omega; \Delta, \psi_1), (\Omega; \Delta, \psi_2)} \qquad \text{(CR-Or)}$$

$$\frac{}{\Omega; \Delta \rightsquigarrow^* \Omega; \Delta} \qquad \text{(CR*-Reflex)}$$

$$\frac{\Omega; \Delta \rightsquigarrow (\Omega_j; \Delta_j)_m \quad \Omega_j; \Delta_j \rightsquigarrow^* (\Omega_{j_k}; \Delta_{j_k})_{n_j} \quad (\text{for } 1 \le j \le m)}{\Omega; \Delta \rightsquigarrow^* (\Omega_{1_k}; \Delta_{1_k})_{n_1}, \cdots, (\Omega_{m_k}; \Delta_{m_k})_{n_m}} \qquad \text{(CR*-Trans)}$$

Figure 9: Context Reduction and Its Closure

$$\frac{}{\Omega; \psi \Longrightarrow_W \psi} \qquad \text{(L-E-Hyp)}$$

$$\frac{}{\Omega, p(c_1, \ldots, c_n); \cdot \Longrightarrow_W !p(c_1, \ldots, c_n)} \qquad \text{(L-E-!R)}$$

$$\frac{}{\Omega; \cdot \Longrightarrow_W \mathbf{1}} \qquad \text{(L-E-1R)}$$

$$\frac{\Omega; \Delta_1 \Longrightarrow_W \psi_1 \quad \Omega; \Delta_2 \Longrightarrow_W \psi_2}{\Omega; \Delta_1, \Delta_2 \Longrightarrow_W \psi_1 \otimes \psi_2} \qquad \text{(L-E-MAndR)}$$

$$\frac{\Omega; \Delta, \psi_1 \Longrightarrow_W \psi_2 \quad \Omega \vdash \psi_1 \text{ ok}}{\Omega; \Delta \Longrightarrow_W \psi_1 \multimap \psi_2} \qquad \text{(L-E-ImpR)}$$

$$\frac{}{\Omega; \Delta \Longrightarrow_W \top} \qquad \text{(L-E-TR)}$$

$$\frac{\Omega; \Delta \Longrightarrow_W \psi_1 \quad \Omega; \Delta \Longrightarrow_W \psi_2}{\Omega; \Delta \Longrightarrow_W \psi_1 \,\&\, \psi_2} \qquad \text{(L-E-AndR)}$$

$$\frac{\Omega; \Delta \Longrightarrow_W \psi_1}{\Omega; \Delta \Longrightarrow_W \psi_1 \oplus \psi_2} \qquad \text{(L-E-OrR1)}$$

$$\frac{\Omega; \Delta \Longrightarrow_W \psi_2}{\Omega; \Delta \Longrightarrow_W \psi_1 \oplus \psi_2} \qquad \text{(L-E-OrR2)}$$

$$\frac{\begin{array}{c} \Omega; \Delta \rightsquigarrow (\Omega_i; \Delta_i)_n \\ \Omega_i; \Delta_i \Longrightarrow_W \psi \quad (\text{for } 1 \le i \le n) \end{array}}{\Omega; \Delta \Longrightarrow_W \psi} \qquad \text{(L-E-Left)}$$

$$\frac{\Omega; \Delta_2 \Longrightarrow_W \psi_1 \quad \Omega; \Delta_1, \psi_1 \Longrightarrow_W \psi}{\Omega; \Delta_1, \Delta_2 \Longrightarrow_W \psi} \qquad \text{(L-E-Cut)}$$

Figure 10: Entailment for World Refinements

$$\frac{\Omega; \phi \Longrightarrow_M \phi' \quad \Omega; \Delta, \psi \Longrightarrow_W \psi'}{\Omega; \Delta; (\phi; \psi) \Longrightarrow_E (\phi'; \psi')} \qquad \text{(L-ER-Base)}$$

$$\frac{\Omega; \Delta; \eta \Longrightarrow_E [c'/c{:}\mathbf{a}]\exists[\vec{b}](\phi; \psi)}{\Omega; \Delta; \eta \Longrightarrow_E \exists[c{:}\mathbf{a}, \vec{b}](\phi; \psi)} \qquad \text{(L-ER-ExistsR)}$$

$$\frac{\Omega, c{:}\mathbf{a}; \Delta; \exists[\vec{b}](\phi; \psi) \Longrightarrow_E \eta}{\Omega; \Delta; \exists[c{:}\mathbf{a}, \vec{b}](\phi; \psi) \Longrightarrow_E \eta} \qquad \text{(L-ER-ExistsL)}$$

Figure 11: Entailment for Expression Refinements

system presented above. There is one typing rule for each expression or term construct and all premises in the rules are fully determined, except those of the context-reduction judgment. We developed the system in two steps, outlined below.

The first step is cut elimination. We eliminate the two cut rules (the cut rule for expression refinement-checking and the cut rule for linear logic entailment) and show that the resulting system is sound and complete with respect to the original refinement-checking specification. We carry out the proof by modifying and extending the logical cut elimination proof in earlier work by Pfenning [28].

In the second step we eliminate the subsumption rule and introduce annotations in order to eliminate two critical sources of non-determinism present in the previous system. The first source is the non-syntax-directedness of the subsumption rule. We therefore incorporate the subsumption rule into the language in a syntax-directed manner, and modify the expression rules so that the context-splitting of the subsumption rule is deterministic.

The second source of non-determinism is the need for the refinement-checker to "guess" the refinement of a given term or expression when the refinement cannot be deduced. We therefore introduce type refinement annotations into the language, allowing the programmer to supply the checker with the missing refinement. In order to reduce the annotation burden, we have defined a bi-directional refinement checking algorithm. The essence of this system is the introduction of two new refinement-checking judgments: one for refinement inference and one for refinement checking. The former judgment infers a refinement for the given term or expression and produces it as an output. The latter judgment takes a refinement as input and checks the given term or expression against the refinement.

Our system also requires a second form of annotation to guide the use of the context rule in expressions. Essentially, these annotations specify when the refinement-checker must use the disjunctive left rule, which, when applied, causes the annotated expression to be rechecked in two different logical contexts. Rechecking the program text has the potential to be very expensive, so we place this facility under the control of the programmer. For a more detailed explanation of our type checking algorithm and full proofs of soundness and completeness with respect to the declarative system, we refer the reader to our technical report [20].

The algorithmic refinement-checking system is decidable modulo the three following aspects of the system:

1. Resolution of first-order existential variables.
2. Resource management.
3. Theorem proving in first-order MALL.

These sources of nondeterminism do not cause the system to be undecidable, as each can be solved independently (and has in the past). First, resolution of first-order existential variables can be done via either explicit instantiation or unification. Second, we must solve the resource, or context, management problem. This problem includes the issue of deciding how to split a linear context into parts in multiplicative rules such as the $\otimes$-right rule, (L-E-MANDR), and (R-E-SUB). There are several known approaches to efficient resource management in linear logic [3]. Third, theorem proving in the multiplicative-additive fragment of linear logic (MALL) has been proven decidable [19]. However, while our system is decidable, finding an efficient decision procedure for all three of the above problems will be challenging. We believe that further investigation should be done in the setting of a practical implementation.

3.5 Soundness

The proof of soundness of refinement checking requires the following soundness condition on the primitive operators.

Condition 2 (Soundness of Primitives)
Suppose

$$\Sigma_\phi(\mathsf{o}) = \forall\vec{b}_1 \cdot ((\phi_{1,1}, \ldots, \phi_{1,n}; \psi_1) \rightarrow \exists[\vec{b}_2](\phi_2; \psi_2))$$

If $w \vDash \Omega; [\vec{c}_1/\vec{b}_1]\psi_1$, *and for* $1 \leq i \leq n$, $\Omega \gg_M c'_i : [\vec{c}_1/\vec{b}_1]\phi_{1,i}$, *and* $\mathcal{T}(\mathsf{o})(c'_1, \ldots, c'_n; w + u) = c'; w'$ *then there exist* Ω' *and* $\vec{c}_2$ *such that*

1. $w' = w'' + u$;
2. $\Omega' \gg_M c' : [\vec{c}_2/\vec{b}_2][\vec{c}_1/\vec{b}_1]\phi_2$;
3. $w'' \vDash \Omega'; [\vec{c}_2/\vec{b}_2][\vec{c}_1/\vec{b}_1]\psi_2$.
4. $\Omega \subseteq \Omega'$

Informally, this condition states that the operator must behave as predicted by its type refinement, and, importantly, can have no effect on a part of the world that is not specified in the precondition of its refinement. Above, w satisfies the precondition of o's refinement. Consequently, no extension, u, of the world may be modified during the operation of o at world $w + u$.

The following lemma expresses the relationship between our world semantics and logical judgments, stating that logical deduction respects the semantics of formulas.

Lemma 3 (Soundness of Logical Judgments)
If $w \vDash \Omega; \Delta$ *and* $\Omega; \Delta \Longrightarrow_W \psi$ *then* $w \vDash \psi$.

Finally, we may state and prove our refinement preservation theorem.

Theorem 4 (Refinement Preservation)

1. *If* $\Omega \gg_M M : \phi$ *and* $M \Downarrow V$ *then* $\Omega \gg_M V : \phi$.
2. *If* $\Omega; \Delta \gg_E E : \exists[\vec{b}](\phi; \psi)$, $w \vDash \Omega; \Delta$, *and* $E \,@\, w + u \Downarrow V \,@\, w'$, *then there exist* Ω' *and* $\vec{c}$ *such that* $\Omega' \gg_M V : [\vec{c}/\vec{b}]\phi$, $w' = w'' + u$, $\Omega \subseteq \Omega'$ *and* $w'' \vDash \Omega'; [\vec{c}/\vec{b}]\psi$.

The following canonical forms theorem expresses the properties of values that refinement checking provides.

Theorem 5 (Refinement Canonical Forms)
If $\cdot \vdash V : A$ *and* $\Omega \gg_M V : \phi$ *(with* Ω *containing only bindings and predicates) then one of the following holds:*

1. $\phi = \textbf{\textit{Bool}}$ *and* $V = \textbf{true}$ *or* $V = \textbf{\textit{false}}$;
2. $\phi = \textbf{\textit{Its}}(c)$ *and* $V = c$;
3. $\phi = \textbf{\textit{a}}$, $V = c$ *and* $\Sigma_A(c) = \mathbf{a}$;
4. $\phi = \forall\vec{b} \cdot \phi_1 \rightarrow \phi_2$ *and* $V = \lambda(x_1{:}A_1).M$;
5. $\phi = \forall\vec{b} \cdot (\phi_1; \psi_1) \rightharpoonup \eta$ *and* $V = \textbf{fun}\ x\ (x_1{:}A_1) : A_2\ \textbf{is}\ E$.

3.6 Conservative Extension

To capture the notion that refinements are a conservative extension of the type system, we present the theorems below. The first theorem states that any refinement given to a term (or expression) in our refinement checking system will always refine the type given to the term (or expression) in the type checking system. In this theorem, we define $\mathsf{type}(\Omega)$ as the typing context Γ mapping all variables $x \in \mathsf{Dom}(\Omega)$ to the type refined by their refinement in Ω. That is, if $x{:}\phi \in \Omega$ and $\Omega_b \vdash \phi \sqsubseteq A$ then $x{:}A \in \Gamma$.

Theorem 6
If $\Omega \gg_M M : \phi$ and $\mathsf{type}(\Omega) \vdash_{\mathsf{M}} M : A$ then $\Omega_b \vdash \phi \sqsubseteq A$. Similarly, if $\Omega; \Delta \gg_E E : \eta$ and $\mathsf{type}(\Omega) \vdash_{\mathsf{E}} E : A$ then $\Omega_b \vdash \eta \sqsubseteq A$.

The next theorem states that for any well-typed term, M (or expression, E), with type A, there exists a refinement-checking derivation for which M (E) has the trivial type associated with A. That is, any well-typed term (expression) can also be shown to be well-refined. In this theorem, $\mathsf{triv}_\Gamma(\Gamma)$ is defined as the persistent context mapping elements $x \in \Gamma$ to the trivial refinement of their type in Γ. Also, $\mathsf{triv}_\Sigma(\Sigma_A)$ is defined as the refinement interface containing the trivial refinements of the elements of Σ_A.

Theorem 7
If $\Gamma \vdash_{\mathsf{M}} M : A$ and $\Sigma_\phi = \mathsf{triv}_\Sigma(\Sigma_A)$ then $\mathsf{triv}_\Gamma(\Gamma) \gg_M M : \mathsf{triv}(A)$. Similarly, if $\Gamma \vdash_{\mathsf{E}} E : A$ and $\Sigma_\phi = \mathsf{triv}_\Sigma(\Sigma_A)$ then $\mathsf{triv}_\Gamma(\Gamma); \top \gg_E E : (\mathsf{triv}(A); \top)$.

3.7 Optimization

As well as helping programmers document and check their programs for additional correctness criteria, refinements provide language or library implementors with a sound optimization principle. When programs are checked to determine their refined type, implementors may replace the total function, $\mathcal{T}(\mathsf{o})$, implementing operator o, with a partial function, $\hat{\mathcal{T}}(\mathsf{o})$, that is only defined on the refined domain given by the refinement signature Σ_ϕ.

To be precise, we define the optimized function $\hat{\mathcal{T}}(\mathsf{o})$ as follows.

$$\begin{array}{rcl}
\hat{\mathcal{T}}(\mathsf{o})(c_1,\ldots,c_n;w) & = & \mathcal{T}(\mathsf{o})(c_1,\ldots,c_n;w) \\
& & \text{if } \Sigma_\phi(\mathsf{o}) = \forall\vec{b}\cdot(\phi_1,\ldots,\phi_n;\psi) \rightharpoonup \eta, \\
& & w \vDash \Omega; [\vec{c}/\vec{b}]\psi \\
& & \text{and } \Omega \gg_M c_i : [\vec{c}/\vec{b}]\phi_i \text{ for } 1 \leq i \leq n \\
\\
\hat{\mathcal{T}}(\mathsf{o})(c_1,\ldots,c_n;w) & = & \text{undefined otherwise}
\end{array}$$

We use the notation $\hat{\Downarrow}$ to denote the optimized evaluation of expressions with the transition function $\mathcal{T}$ replaced by $\hat{\mathcal{T}}$ (that is, all operator implementations replaced by their optimized versions). We are able to prove that optimized and unoptimized evaluation are equivalent and therefore that it is safe for implementors to replace operator implementations with their optimized version.

Theorem 8 (Optimization)
If $\Omega; \Delta \gg_E E : \eta$ and $w \vDash \Omega; \Delta$ then $E \mathbin{@} w \Downarrow V \mathbin{@} w'$ if and only if $E \mathbin{@} w \hat{\Downarrow} V \mathbin{@} w'$.

4. EXAMPLES

In this section, we provide a number of examples that demonstrate the expressive power of our language. Our technical report presents several more examples including ML-style integer references, alias types and recursion counts. We omit leading universal quantifiers in our examples as they may easily be inferred in a manner similar to the way the Twelf system [30] infers leading quantifiers.

4.1 File Access Modes

We can extend our file I/O example from the introduction by tracking the access modes of open files and checking them before calling the read or write operators. We add the following access modes to our signature and change open to a binary predicate over file names and access modes. If $\mathsf{open}(f, a)$ is true at a particular program point, then file f is open in access mode a.

r	:	$\mathbf{Mode}$	Read Access
w	:	$\mathbf{Mode}$	Write Access
rw	:	$\mathbf{Mode}$	Read and Write Access
open	:	$(\mathbf{File}, \mathbf{Mode}) \to \mathbf{prop}$	Open Predicate

Below are the modified `fopen` and `fwrite` operators as well as a new `fread` operator. The refinement of `fopen` now captures the access mode a passed to `fopen` in the open predicate. In the refinements of `fread` and `fwrite`, we let $\mathsf{writeable}(f)$ and $\mathsf{readable}(f)$ abbreviate $\mathsf{open}(f, \mathsf{w}) \oplus \mathsf{open}(f, \mathsf{rw})$ and $\mathsf{open}(f, \mathsf{r}) \oplus \mathsf{open}(f, \mathsf{rw})$, respectively. We use the $\&$ connective in the refinement of `fread` in order to simultaneously ensure that f is readable before a call to `fread` and that whatever the access mode, it is preserved across the call. The $\&$ in the refinement of `fwrite` serves a parallel purpose.

$$\begin{array}{ll}
\texttt{fopen} : & (\mathbf{\mathit{Its}}(f), \mathbf{\mathit{Its}}(a);\ \mathsf{closed}(f)) \rightharpoonup (\mathbf{unit};\ \mathsf{open}(f,a)) \\
\texttt{fread} : & (\mathbf{\mathit{Its}}(f);\ \mathsf{open}(f,a) \,\&\, \mathsf{readable}(f)) \rightharpoonup \\
& (\mathbf{String};\ \mathsf{open}(f,a)) \\
\texttt{fwrite}: & (\mathbf{\mathit{Its}}(f), \mathbf{String};\ \mathsf{open}(f,a) \,\&\, \mathsf{writeable}(f)) \rightharpoonup \\
& (\mathbf{unit};\ \mathsf{open}(f,a))
\end{array}$$

4.2 File Buffering

To demonstrate the utility of persistent predicates, we add to our original file-I/O example support for layering a buffer on top of a file. We introduce three new predicates for this example: $\mathsf{bfs}(b, f)$ to indicate that a buffer b buffers a file f; fl and ufl to conservatively estimate whether a buffer is flushed or unflushed (that is, contains unwritten data).

bfs	:	$(\mathbf{Buffer}, \mathbf{File}) \to \mathbf{prop}$	Buffers Relation
fl	:	$(\mathbf{Buffer}) \to \mathbf{prop}$	Flushed Predicate
ufl	:	$(\mathbf{Buffer}) \to \mathbf{prop}$	Unflushed Predicate

Correct use of a buffer requires that the file beneath the buffer be open when a program attempts to write to that buffer and that the buffer is flushed before being freed. We enforce the first requirement in two steps. First, upon creating a new buffer b over a file f, we assert a persistent fact $\mathsf{bfs}(b, f)$, indicating that b buffers f. Next, when attempting to write to a buffer b, we check that b buffers some file f and that f is open. As bfs is invariant over the lifetime of a given buffer, it is natural to represent it as a persistent fact. In addition, doing so allows us to reuse the fact without explicitly preserving it and forget about it once it is no longer needed.

Below is the buffer interface. Calling `bnew` on a file f creates a new, flushed buffer over that file. Calling `bwrite` on a buffer b acts as expected, but requires that the file buffered by b be open, although b can be either flushed or unflushed. In either case, we conservatively estimate that b is not flushed upon return from `bwrite`. The operator `bflush` flushes a buffer, changing its state from ufl to fl. Finally, `bfree` takes a flushed buffer and frees it.

$$
\begin{array}{ll}
\texttt{bnew} & : (\textbf{\textit{Its}}(f);\ \mathbf{1}) \rightharpoonup \exists[b:\textbf{Buffer}](\textbf{\textit{Its}}(b);\ !\mathsf{bfs}(b,f)\otimes \mathsf{fl}(b))\\
\texttt{bwrite} & : (\textbf{\textit{Its}}(b);\ !\mathsf{bfs}(b,f)\otimes \mathsf{open}(f)\otimes(\mathsf{fl}(b)\oplus \mathsf{ufl}(b))) \rightharpoonup\\
 & \quad ((); \mathsf{open}(f)\otimes \mathsf{ufl}(b))\\
\texttt{bflush} & : (\textbf{\textit{Its}}(b);\ \mathsf{ufl}(b)) \rightharpoonup ((); \mathsf{fl}(b))\\
\texttt{bfree} & : (\textbf{\textit{Its}}(b);\ \mathsf{fl}(b)) \rightharpoonup ((); \mathbf{1})
\end{array}
$$

4.3 Interrupt Levels

For their study of Windows device drivers, DeLine and Fahndrich extend Vault with a special mechanism for specifying "capability states" which are arranged in a partial order [7]. They use the partial order and bounded quantification to specify preconditions on kernel functions. Here we give an alternate encoding and reason logically about the same kernel functions and their preconditions.

First, we assume a signature with abstract constants that correspond to each interrupt level and also a predicate L over these levels. If $\mathsf{L}(c)$ is true at a particular program point then the program executes at interrupt level c at that point.

pass	:	**level**	Passive Level
apc	:	**level**	APC Level
dis	:	**level**	Dispatch Level
dirql	:	**level**	DIRQL Level
L	:	**level** → **prop**	Level Predicate

Next we consider a variety of kernel functions and their type refinements. First, the `KeSetPriorityThread` function requires that the program be at Passive Level when it is called and also returns in Passive Level. The function takes arguments with type **thread** and **pr**, which we assume are defined in the current signature.

`KeSetPriorityThread`:
$$(\mathbf{thread}, \mathbf{pr}; \mathsf{L}(\mathsf{pass})) \rightharpoonup (\mathbf{pr}; \mathsf{L}(\mathsf{pass}))$$

Function `KeReleaseSemaphore` is somewhat more complex since it may be called in Passive, APC or Dispatch level and it preserves its level across the call. We let $\mathsf{less}(\mathsf{dis})$ abbreviate the formula $\mathsf{L}(\mathsf{pass}) \oplus \mathsf{L}(\mathsf{apc}) \oplus \mathsf{L}(\mathsf{dis})$.

`KeReleaseSemaphore`:
$$(\mathbf{sem}, \mathbf{pr}, \mathbf{long}; \mathsf{L}(l) \otimes (\mathsf{L}(l) \multimap \mathsf{less}(\mathsf{dis}))) \rightharpoonup (\mathbf{pr}; \mathsf{L}(l))$$

Finally, `KeAcquireSpinLock` also must be called in one of three states. However, it returns in the Dispatch state and also returns an object representing the initial state (l) that the function was called in.

`KeAcquireSpinLock`:
$$(\mathbf{sem}, \mathbf{pr}, \mathbf{long}; \mathsf{L}(l) \otimes (\mathsf{L}(l) \multimap \mathsf{less}(\mathsf{dis}))) \rightharpoonup (\textbf{\textit{Its}}(l); \mathsf{L}(\mathsf{dis}))$$

5. DISCUSSION

5.1 Related Work

A number of researchers have recently proposed strategies for checking that programs satisfy sophisticated safety properties. Each system brings some strengths and some weaknesses when compared with our own. Here are some of the most closely related systems.

Refinement Types. Our initial inspiration for this project was derived from work on refinement types by Davies and Pfenning[6] and Denney [8] and the practical dependent types proposed by Xi and Pfenning [38, 39]. Each of these authors proposed to sophisticated type systems that are able to specify many program properties well beyond the range of conventional type systems such as those for Java or ML. However, none of these groups considered the ephemeral properties that we are able to specify and check.

Safe Languages. CCured [24], CQual [13], Cyclone [16], ESC [9, 12], and Vault [7, 10] are all languages designed to verify particular safety properties. CCured concentrates on showing the safety of mostly unannotated C programs; Cyclone allows programmers to specify safe stack and region memory allocation; ESC facilitates program debugging by allowing programmers to state invariants of various sorts and uses theorem proving technology to check them; and Vault and CQual make it possible to check resource usage protocols. Vault has been applied to verification of safety conditions in device drivers and CQual has been applied to find locking bugs in the Linux kernel. One significant difference between our work and the others is that we have chosen to use a general substructural logic to encode program properties. Vault is the most similar since its type system is derived from the capability calculus [36] and alias types [34, 37], which is also an inspiration for this work. However, the capability logic is somewhat ad hoc whereas we base our type system directly on linear logic. As far as we are aware, the semantics of Vault has not been fully formalized. We hope this work is an effective starting point in that endeavour.

Igarashi and Kobayashi's resource usage analysis [17] is another piece of work in this vein. They define a complex type system that keeps track of the "uses" of resources. They have a general trace-based semantics as opposed to our possible worlds-style resource semantics. It is difficult to compare the expressive power of our two proposals precisely as the set of formulas involved is quite different. They have some interesting modal operators and a recursion operator, but their logic is propositional whereas ours is first-order.

Proof-Carrying Type Systems. Shao et al. [33] and Crary and Vanderwaart [4] have both developed powerful type languages that include a fully general logical framework within the type structure. Both languages were inspired by Necula and Lee's work on proof carrying code [23, 22] and are designed as a very general framework for coupling low-level programs with their proofs of safety. In contrast, our language is intended to be a high-level language for programmers. Hence, the design space is quite different. Our specification language is less general than either of these, but it does not require programmers to write explicit proofs that their programs satisfy the safety properties in question. Moreover, neither of these logics contain the full complement of linear logic's left-asynchronous connectives ($\mathbf{1}$, $\otimes$, $\mathbf{0}$, $\oplus$, $\exists$), which we find the most useful in our applications.

Hoare Logic. Recent efforts on the reasoning about pointers in Hoare logic [18, 31] by Ishtiaq, O'Hearn and Reynolds, provided guidance in construction of our semantic model of refinements. However, they use bunched logic in their work whereas we use a subset of linear logic. One important difference between the logics is that linear logic contains the modality !, which we use to reason about *persistent* facts. A notion of persistence seems essential to allow one to reason about values, which, by their nature, remain unchanged throughout the computation. Also, since our work is based on type theory, it naturally applies to higher-order programs, which is not the case for Hoare logic. Moreover, programmers who use Hoare logic have no automated support whereas our system has a decidable type checking algorithm.

5.2 Future Work

There are many directions for future work. We have begun to investigate the following issues. Our most immediate concern is the development of an implementation of the ideas presented in this paper. One of the authors (Mandelbaum) has developed a preliminary implementation for small core subset of Java. The current implementation is built using Polyglot [25], an extensible compiler infrastructure for Java, and allows programmers to reason with a minimalist subset of the logic that includes $\mathbf{1}$, $\otimes$ and $\top$. We are currently developing a surface language based on ML that incorporates all of the features presented in this paper.

One of the limitations of the current proposal is the inability to check the membership of an ephemeral fact in the ephemeral context while preserving the other facts in that context. For example, consider a function `writeBoth` that takes two file arguments and writes to both of them. Below is a possible refinement of such a function:

$$\begin{array}{l}\texttt{writeBoth} : \\ \quad (\boldsymbol{Its}(f), \boldsymbol{Its}(g);\ \mathsf{open}(f) \otimes \mathsf{open}(g)) \rightharpoonup \\ \qquad ((); \ \mathsf{open}(f) \otimes \mathsf{open}(g))\end{array}$$

However, this refinement precludes the possibility that f and g are equal, as $\mathsf{open}(f) \otimes \mathsf{open}(g)$ describes two distinct facts. Therefore, `writeBoth` cannot be passed the same file for both of its arguments. An alternative precondition, $(\mathsf{open}(f) \otimes \top) \,\&\, (\mathsf{open}(g) \otimes \top)$, does not depend on the distinctness of f and g but fails to preserve information about the remaining members of the ephemeral context, due to the imprecision of $\top$.

One solution to this problem would be to provide a way to view an ephemeral fact as temporarily persistent, as we can check the membership of a fact in the persistent context while preserving the remaining persistent facts. Then, we could successfully check calls to `writeBoth` with aliased arguments based on the following refinement:

$$\begin{array}{l}\texttt{writeBoth} : \\ \quad (\boldsymbol{Its}(f), \boldsymbol{Its}(g);\ !\mathsf{open}(f) \otimes !\mathsf{open}(g)) \rightharpoonup ((); \mathbf{1})\end{array}$$

We are investigating a sharing principle, similar to the world-splitting allowed by the subsumption rule, that would provide a safe method of viewing ephemeral facts as temporarily persistent.

An alternative solution would be to add a limited form of second-order quantification to the logic. We could then refine `writeBoth` as follows:

$$\begin{array}{l}\texttt{writeBoth} : \\ \quad (\boldsymbol{Its}(f), \boldsymbol{Its}(g);\ (\mathsf{open}(f) \otimes \top) \,\&\, (\mathsf{open}(g) \otimes \top) \,\&\, w) \rightharpoonup \\ \qquad ((); w)\end{array}$$

In essence, the refinement checks that $\mathsf{open}(f)$ and $\mathsf{open}(g)$ are members of the world w, while ensuring that w is preserved.

The next item of interest to us is the encoding of type-and-effect systems in our language. We believe that our language provides a general framework in which to encode many type-and-effect systems. Based on the encoding shown above, we have devised a translation from a variant of a well-known type-and-effect system concerning lock types for static enforcement of mutual exclusion [11], into our language (extended with second-order quantification). We thereby show that our refinements are at least as powerful. Our translation also helps us understand the connection between types and effects and recent research on sophisticated substructural type systems such as the one implemented in Vault [7].

Finally, our language is parameterized by a single interface and implementation that enables us to consider reasoning about a variety of different sorts of effects. The next step in the development of this project is to extend the language with an advanced module system that allows programmers to define their own logical safety policies and to reason compositionally about their programs.

Acknowledgments

David Walker would like to thank Frank Pfenning for teaching him about the structure of linear logic—without Frank's instruction, it would have been impossible to write this paper. We would like to thank Nate Nystrom, Michael Clarkson and Andrew Myers, the implementers of Polyglot [25], for giving us access to their software. Polyglot dramatically increased the speed at which we were able to develop a preliminary implementation of our ideas for Java. We are also very grateful to Dan Grossman, who gave us some important feedback on an earlier draft of this paper. Finally, we would like to thank the anonymous reviewers for their helpful comments. In particular, we thank them for bringing to our attention the `writeBoth` example, discussed in the previous section.

6. REFERENCES

[1] Simon Ambler. *First-order Linear Logic in Symmetric Monoidal Closed Categories.* PhD thesis, University of Edinburgh, 1991.

[2] Leonart Augustsson. Cayenne — a language with dependent types. In *ACM International Conference on Functional Programming*, pages 239–250, Baltimore, September 1999. ACM Press.

[3] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the Fifth International Workshop on Extensions of Logic Programming — ELP'96*, pages 67–81, Leipzig, Germany, March 1996.

[4] Karl Crary and Joe Vanderwaart. An expressive, scalable type theory for certified code. In *ACM International Conference on Functional Programming*, Pittsburgh, October 2002. ACM Press.

[5] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[6] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ACM International Conference*

on Functional Programming, pages 198–208, Montreal, September 2000. ACM Press.

[7] Rob Deline and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM Press.

[8] Ewen Denney. *A Theory of Program Refinement.* PhD thesis, University of Edinburgh, Edinburgh, 1998.

[9] David L. Detlefs. An overview of the extended static checking system. In *The First Workshop on Formal Methods in Software Practice*, pages 1–9. ACM(SIGSOFT), January 1996.

[10] Manuel Fähndrich and Rob Deline. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, June 2002. ACM Press.

[11] Cormac Flanagan and Martin Abadi. Types for safe locking. In S.D. Swierstra, editor, *Lecture Notes in Computer Science*, volume 1576, pages 91–108, Amsterdam, March 1999. Springer-Verlag. Appeared in the Eighth European Symposium on Programming.

[12] Cormac Flanagan, Rustan Leino, Mark Lillibridge, Greg Nelsonand James Saxes, and Raymie Stata. Extended static checking for java. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[13] Jeffrey Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[14] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[15] Andrew Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics 17*. Elsevier, 2001.

[16] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[17] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *ACM Symposium on Principles of Programming Languages*, pages 331–342, Portland, Oregon, January 2002. ACM Press.

[18] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, January 2001.

[19] Patrick Lincoln and Andre Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.

[20] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. Technical Report TR-656-02, Princeton University, December 2002.

[21] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[22] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[23] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[24] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, London, January 2002. ACM Press.

[25] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *12th International Conference on Compiler Construction*, April 2003. to appear.

[26] Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[27] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL '01*, pages 1–19, Paris, 2001.

[28] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[29] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[30] Frank Pfenning and Carsten Schürmann. system description: Twelf — a metalogical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, number 1632 in LNAI, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[31] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

[32] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science'02*, pages 55–74, 2002.

[33] Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *ACM Symposium on Principles of Programming Languages*, London, January 2002. ACM Press.

[34] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[35] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[36] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, May 2000.

[37] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, September 2000.

[38] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[39] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.