An Effective Theory of Type Refinements

Yitzhak Mandelbaum
Princeton University

David Walker [*]
Princeton University

Robert Harper [†]
Carnegie Mellon University

December 2002

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 which uses a *logic of type refinements* to check more precise properties of program behavior. Our logic is a fragment of intuitionistic linear logic, which allows us 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.

[*]Supported in part by DARPA Grant F30602-99-1-0519, NSF Trusted Computing grant CCR-0208601 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.

Contents

1 Introduction **2**

2 Base Language **5**
- 2.1 Abstract Syntax 5
- 2.2 Abstract Resources 5
- 2.3 Static Semantics 8
- 2.4 Dynamic Semantics 8
- 2.5 Properties 9

3 Refinements **11**
- 3.1 Syntax 11
- 3.2 Semantics of Refinements 13
- 3.3 Declarative Refinement Checking 14
- 3.4 Properties of Refinement Checking Judgments 20
- 3.5 Algorithmic Refinement Checking 27
  - 3.5.1 Cut Elimination 27
  - 3.5.2 Subsumption Elimination and Annotation Introduction 27
- 3.6 Decidability 38
- 3.7 Soundness 38
- 3.8 Conservative Extension 50
- 3.9 Optimization 50

4 Examples **51**
- 4.1 ML-style integer references 51
- 4.2 Alias Types 52
- 4.3 Interrupt Levels 52
- 4.4 Recursion Counts 53

5 Discussion **53**
- 5.1 Variants and Further Extensions 53
- 5.2 Current and Future Work 54
- 5.3 Related Work 55
- 5.4 Conclusions 56

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. For instance, we can normally prove that any program with integer type will either diverge or actually return an integer, rather than a boolean or character. We can also lift such properties to terms or values with higher type, which provides us with a mechanism to reason about the behavior of functions.

In recent years, there has been substantial interest in formulating *refinements* of conventional types that allow programmers to specify more precise properties of program data than are implied by ordinary Java- or ML-style type systems. For example, Xi and Pfenning [XP99] popularized the use of singleton types to reason exactly about the values bound to variables or produced by computations. They also present compelling applications including static array-bounds checking [XP98].

In order for a system of *type refinements* to be of practical use, we claim that it must satisfy two criteria.

1. The system must be a *conservative extension* of the underlying type system. In other words, type refinements should 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. Adding type refinements to a program should not invalidate or contradict previous reasoning principles.

2. The system should support *modular* or *local reasoning.* In any given program, there will be many invariants that a programmer might need to reason about. However, any single program part might only depend upon a few of these invariants. In such program parts, it should be possible to reason about local behavior based exclusively on these few invariants rather than the many.

Systems such as Xi and Pfenning's dependent types [XP99] satisfy both these criteria. Their system is a conservative extension of ML and when reasoning about whether an array index is in bounds (for instance), one only requires refinements concerning the particular array and the index in question. One need not specify conditions about all arrays or about all integers that appear anywhere in a program.

Still, Xi and Pfenning's dependent type system and related work [Den98, Aug99, CW99, DP00] are only able 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. This property implies that these systems cannot be used to enforce important invariants such as the fact that that a lock be held before a data structure is accessed or that a file is opened, read and then closed.

Ideally, we would construct a system of refinements that specify properties of the underlying state while maintaining our two criteria. Yet, it is considerably more difficult to construct such a system than it is to construct systems such as Xi and Pfenning's, which only specifies properties of pure computations. The primary difficulty lies in the fact that semantics for stateful computations thread the entire state along the evaluation path of the program. Hence, it would seem as though any refinement of this state should capture and maintain all of the properties that may be needed at some future point in the computation. In other words, at a first glance, one might guess that refinements for state will violate our second condition.

Fortunately, the single "state" of a computation may often be viewed as a structured object with many parts. Moreover, a particular part of a computation will often depend only upon a few parts of the state and leave the rest untouched. The key to achieving a practical, modular system of refinements is to develop a logic that supports *local reasoning* about state. Conventional techniques based on classical logic require *global reasoning* in the sense that assertions are (and must be) understood as describing the *entire* state of the computation. This works fine for simple *while* programs, since then it is feasible to consider all program variables at once. But in more complex programs this is clearly infeasible, and in any case is incompatible with modularity.

To achieve local reasoning we must employ a logic that allows us to make assertions about a "piece" of the program state, without resorting to mentioning all of it. Classical logic fails to support local reasoning! The reason can be traced back to the validity, in classical logic, of the entailments $A \vdash A \wedge A$ and $A \vdash \top$. The former allows the free replication of assertions such as "location l contains 7", and the latter allows us to "forget" such assertions entirely. While at first this may seem harmless, these conventional reasoning principles create fundamental difficulties for local reasoning. In particular, it is impossible to axiomatize state-changing operations such as assignment by entailments between pre- and post-conditions, creating a rift between the steps of computation and the associated steps of reasoning.

To support local reasoning about state requires an unconventional logic in which such fundamental entailments are not valid. One such logic is *linear logic* [Gir87], but others, such as *bunched implications* [OP99, IO01], have also been proposed. What these logics have in common is fine-grained control over replication and neglect of assertions so that a natural correspondence between reasoning and computation may be achieved. In this paper we employ a fragment of linear logic that is adequate for many practical purposes, as we will see in Section 4 below.

In summary, this work makes the following main contributions.

- 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 a *logic of refinements* for reasoning about program properties that cannot be captured by conventional types. We establish a formal correspondence between the level of types and our more precise level of logic of refinements. Only Denney [Den98] has explicitly considered such a two-level system in the past, but he restricted attention to pure computations.

- The computational lambda calculus [Mog91] serves as our basic linguistic framework and our logic of refinements enables programmers to reason *locally* about effectful computations. 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. We have also worked hard to separate 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.

- We have identified a crucial *locality condition* concerning the behavior of 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. We prove this optimization principle as a corollary.

- We develop an algorithmic refinement checking system that is both cut-free and subsumption-free, utilizing programmer annotations. We prove it both sound and complete with respect to our original system.

- We provide a number of examples to demonstrate the expressiveness of our system. Our refinements appear to subsume the state-logic used in the Vault programming language [DF01] (although our idealized language does not contain the array of data structures present in Vault, or the specialized type inference techniques). Hence, our system suggests a semantics for an important fragment of Vault.

Within the last year or so, several type systems for checking properties of programs involving state have been developed. However, these other proposals are either designed for very specific applications rather than general effects [WCM00, NMW02, GMJ$^+$02], are undecidable [fJ02, IO01], or use a less general logic [DF01, FTA02]. Overall, the goal of the current paper is to complement this exciting surge of research by providing a general, robust and extensible theory of type refinements that captures sound techniques for local reasoning about program state.

In the remainder of this paper, 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. Also in this section, we define a declarative system for our language and a corresponding algorithmic system for a new, annotated version of our language. The new system is used for deciding type refinement judgments on annotated terms. Finally, we show that our refinements are a conservative extension of the underlying type system, and prove that our algorithmic system is sound and that refined operators may be optimized. In Section 4 we provide a series of simple examples to show how our refinements may be used. In the last section, we discuss variants of our system and indicate our current research directions. We also comment further on related work and provide some conclusions.

2 Base Language

We use Moggi's computational λ-calculus [Mog91] as a basic linguistic framework, as reformulated by Pfenning and Davies [PD01]. The framework is enriched with a base type of booleans and recursive functions. In order to consider a variety of different sorts of effects, we parameterize the language by a collection of abstract types $\mathbf{a}$, constants c with type $\mathbf{a}$ 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:

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

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.

Other formulations of Moggi's computational lambda calculus include a suspended computation $\{E\}$. Since we include a function space with an impure body, we do not need to include suspended computations explicitly. We may encode the suspended computation $\{E\}$ with the function $\lambda x{:}\mathbf{Bool}.E$ where $x \notin \mathrm{FV}(E)$ and $\lambda x{:}A.E$ is an abbreviation for a recursive function in which the function name X does not appear in the body.

2.2 Abstract Resources

The language is parameterized by a set of operators that manipulate some abstract resource or set of resources. We may reason about an instance of the language by specifying an interface Σ for and implementation $\mathcal{M}$ 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.

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 for each constant or operator.

An implementation $\mathcal{M} = (\mathcal{W}, \mathcal{T})$ defines a set $\mathcal{W}$ of worlds w, and a transition function $\mathcal{T}$ 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* facts. Below, we will define an accessibility relation on worlds that relates a world w to its possible future worlds (*i.e.,* to any world that could be reached through a computation starting with w). It will be the case that the persistent facts of one world will remain true in all its possible future worlds. The ephemeral facts of a world may or may not hold in its future worlds.

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	**Contents**
$\mathcal{W}$	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

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. We also define the notation $w \cup S$, where S is a set of (persistent) facts, to be $(\mathsf{Per}(w) \cup S, \mathsf{Eph}(w))$. We write $S \backslash S'$ for set or multi-set difference.

If an interface specifies that an operator has type $\mathbf{a}_1, \ldots, \mathbf{a}_n \rightharpoonup \mathbf{a}$ 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 3.7 where we prove the soundness of refinement checking.

We derive an accessibility relation on worlds from the transition functions. We say that w_{n+1} is a possible future world of w_1 and write $w_1 \leq w_{n+1}$ when there exists some set of operators and constants such that

$$\begin{array}{l}\mathcal{T}(\mathsf{o}_1)(\vec{c_1}, w_1) = c'_1, w_2 \\ \ldots \\ \mathcal{T}(\mathsf{o}_n)(\vec{c_n}, w_n) = c'_n, w_{n+1}\end{array}$$

This clearly defines a pre-order. Notice that if $w \leq w'$ then $\mathsf{Per}(w) \subseteq \mathsf{Per}(w')$.

We summarize the language parameters in Figure 1.

Example: Integer References As a typical example of effectful computation, we consider parameterizing the language with allocation and assignment to integer references. We require three base types, a type for integers **int**, the type for integer references **int ref** and the unit type **unit**. Our constants include the integers i, a countable set of locations for storing integers (we use metavariable ℓ to range over locations) and a unit value (). We use **new** , **set** and **get** for allocation, assignment and dereference respectively. The signature Σ_A provides the usual types for these operations.

$$
\begin{array}{lcl}
\Sigma_A(()) & = & \mathbf{unit} \\
\Sigma_A(i) & = & \mathbf{int} \\
\Sigma_A(\ell) & = & \mathbf{int\,ref} \\
\Sigma_A(\mathbf{new}\ ) & = & (\mathbf{int}) \rightharpoonup (\mathbf{int\,ref}) \\
\Sigma_A(\mathbf{get}\ ) & = & (\mathbf{int\,ref}) \rightharpoonup (\mathbf{int}) \\
\Sigma_A(\mathbf{set}\ ) & = & (\mathbf{int}, \mathbf{int\,ref}) \rightharpoonup (\mathbf{unit})
\end{array}
$$

We use two different predicates to capture the effect of references on the world. The predicate $\mathsf{alloc}(\ell)$ indicates we have allocated location ℓ and it is in use storing an integer. We use the predicate $\mathsf{ctns}(\ell, i)$ to denote the fact that ℓ contains the integer i at a particular program point. We defer examples of possible refinements for the operators in this system until Section 4, after specifying refinement syntax and semantics in Section 3 .

In the implementation component, we must specify the set of worlds and the behavior of the operators. Once allocated, references are never deallocated since we haven't included a "free" operation. Therefore, the set of persistent facts for any world will contain $\mathsf{alloc}(\ell)$ for each location ℓ that has been previously allocated. The world will also contain an ephemeral fact $\mathsf{ctns}(\ell, i)$ for each such location and for some integer i. The transition function $\mathcal{T}$ specifies the dynamic semantics for each operator. A key aspect of this definition 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 3.9), 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.

$$
\begin{array}{lcl}
\mathcal{T}(\mathbf{new}\ )(i, w) & = & (\ell, w') \\
 & & \text{where } \mathsf{Per}(w') = \mathsf{Per}(w) \cup \{\mathsf{alloc}(\ell)\} \\
 & & \text{and } \mathsf{Eph}(w') = \mathsf{Eph}(w) + \{\mathsf{ctns}(\ell, i)\} \\
 & & \text{and } \mathsf{alloc}(\ell) \notin \mathsf{Per}(w) \\
 & & \\
\mathcal{T}(\mathbf{get}\ )(\ell, w) & = & (i, w) \\
 & & \text{if } \mathsf{ctns}(\ell, i) \in \mathsf{Eph}(w) \text{ (for any } i) \\
 & & \\
\mathcal{T}(\mathbf{get}\ )(\ell, w) & = & (0, w) \\
 & & \text{if } \mathsf{ctns}(\ell, i) \notin \mathsf{Eph}(w) \text{ (for any } i) \\
 & & \\
\mathcal{T}(\mathbf{set}\ )(i, \ell, w) & = & ((), w') \\
 & & \text{if } w = w'' + \{\mathsf{ctns}(\ell, j)\} \text{ (for any } j) \\
 & & \text{and where } \mathsf{Per}(w') = \mathsf{Per}(w) \\
 & & \text{and } \mathsf{Eph}(w') = \mathsf{Eph}(w'') + \{\mathsf{ctns}(\ell, i)\} \\
\mathcal{T}(\mathbf{set}\ )(i, \ell, w) & = & ((), w) \\
 & & \text{if } w \neq w'' + \{\mathsf{ctns}(\ell, j)\} \text{ (for any } j)
\end{array}
$$

$$\frac{}{\Gamma, x{:}A \vdash_{\mathsf{M}} x : A} \quad \text{(S-T-Var)}$$

$$\frac{(\Sigma_A(c) = \mathbf{a})}{\Gamma \vdash_{\mathsf{M}} c : \mathbf{a}} \quad \text{(S-T-Const)}$$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \quad \text{(S-T-True)}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}} \quad \text{(S-T-False)}$$

$$\frac{\Gamma, x_1{:}A_1 \vdash_{\mathsf{M}} M : A_2}{\Gamma \vdash \lambda(x_1{:}A_1).M : A_1 \to A_2} \quad \text{(S-T-Lam)}$$

$$\frac{\Gamma, x{:}A_1 \rightharpoonup A_2, x_1{:}A_1 \vdash_{\mathsf{E}} E : A_2}{\Gamma \vdash \mathbf{fun}\ x\ (x_1{:}A_1) : A\ \mathbf{is}\ E : A_1 \rightharpoonup A_2} \quad \text{(S-T-Fun)}$$

$$\frac{\Gamma \vdash_{\mathsf{M}} M : \mathbf{Bool} \quad \Gamma \vdash_{\mathsf{M}} M_1 : A \quad \Gamma \vdash_{\mathsf{M}} M_2 : A}{\Gamma \vdash_{\mathsf{M}} \mathbf{if}\ M\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : A} \quad \text{(S-T-If)}$$

$$\frac{\Gamma \vdash_{\mathsf{M}} M : A_1 \to A_2 \quad \Gamma \vdash_{\mathsf{M}} M_1 : A_1}{\Gamma \vdash_{\mathsf{M}} M\,(M_1) : A_2} \quad \text{(S-T-TApp)}$$

Figure 2: Static Semantics of Terms

2.3 Static Semantics

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

$\Gamma \vdash_{\mathsf{M}} M : A$	*Term M has type A in Γ*
$\Gamma \vdash_{\mathsf{E}} E : A$	*Expression E has type A in Γ*

The meta-variable Γ ranges over finite functions from variables x to types A. We write such functions according to the following grammar (where a variable x may appear at most once).

$$\Gamma ::= \cdot \mid \Gamma, x{:}A$$

The symbol $\cdot$ represents the function with empty domain. Normally, when the domain is not empty, we omit the initial "$\cdot$". We write $\Gamma(x)$ for the type (if any) assigned to x by Γ.

The rules defining these judgments are given in Figures 2 and 3. They are entirely standard and need little explanation except to say that the judgments are implicitly parameterized by the signature Σ_A.

2.4 Dynamic Semantics

The dynamic semantics is given by two evaluation judgments:

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

$$\frac{\Gamma \vdash_{\mathsf{M}} M : A}{\Gamma \vdash_{\mathsf{E}} M : A} \qquad \text{(S-E-Term)}$$

$$\frac{(\Sigma_A(c) = (\mathbf{a}_1, \ldots, \mathbf{a}_n) \rightharpoonup (\mathbf{a})) \quad \Gamma \vdash M_i : \mathbf{a}_i \quad (\text{for } 1 \leq i \leq n)}{\Gamma \vdash_{\mathsf{M}} \mathsf{o}(M_1, \ldots, M_n) : \mathbf{a}} \qquad \text{(S-E-Op)}$$

$$\frac{\Gamma \vdash_{\mathsf{E}} E_1 : A_1 \quad \Gamma, x{:}A_1 \vdash_{\mathsf{E}} E_2 : A_2}{\Gamma \vdash_{\mathsf{E}} \mathbf{let}\, x\, \mathbf{be}\, E_1\, \mathbf{in}\, E_2\, \mathbf{end} : A_2} \qquad \text{(S-E-Let)}$$

$$\frac{\Gamma \vdash_{\mathsf{M}} M : A_1 \rightharpoonup A_2 \quad \Gamma \vdash_{\mathsf{M}} M_1 : A_1}{\Gamma \vdash_{\mathsf{E}} \mathbf{app}(M, M_1) : A_2} \qquad \text{(S-E-PApp)}$$

$$\frac{\Gamma \vdash_{\mathsf{M}} M : \mathbf{Bool} \quad \Gamma \vdash_{\mathsf{E}} E_1 : A \quad \Gamma \vdash_{\mathsf{E}} E_2 : A}{\Gamma \vdash_{\mathsf{E}} \mathbf{if}\, M\, \mathbf{then}\, E_1\, \mathbf{else}\, E_2 : A} \qquad \text{(S-E-If)}$$

Figure 3: Static Semantics of Expressions

The rules defining the evaluation relations are given in Figures 4 and 5. Once again, the rules are entirely standard.

2.5 Properties

Since we are using an evaluation semantics, the proof of type safety is indirect. First, we prove a type preservation lemma stating that the value of a term or expression has the type of the expression itself. Second, we prove a canonical forms lemma characterizing the closed values of each type. Third, we augment the operational semantics with rules specifying that the value of an expression is a designated answer, **wrong**, in the case that the principal argument of an expression is non-canonical. From this we may conclude that well-typed expressions do not "go wrong". We will consider here only the first and second steps, the third being routine.

Theorem 1 (Type Preservation)

1. *If* $\cdot \vdash_{\mathsf{M}} M : A$ *and* $M \Downarrow V$, *then* $\cdot \vdash_{\mathsf{M}} V : A$.

2. *If* $\cdot \vdash_{\mathsf{E}} E : A$ *and* $E \mathbin{@} w \Downarrow V \mathbin{@} w'$, *then* $\cdot \vdash_{\mathsf{M}} V : A$.

Theorem 2 (Type Canonical Forms)
If $\cdot \vdash_{\mathsf{M}} V : A$, *then*

1. *if* $A = \mathbf{a}$, *then* $V = c$ *(and* $\Sigma_A(c) = \mathbf{a}$*);*

2. *if* $A = \mathbf{Bool}$, *then* $V =$ ***true*** *or* $V =$ ***false****;*

3. *if* $A = A_1 \to A_2$, *then* $V = \lambda(x_1{:}A_1).M$

4. *if* $A = A_1 \rightharpoonup A_2$, *then* $V = \mathbf{fun}\ x\ (x_1{:}A_1) : A_2\ \mathbf{is}\ E$.

$$\frac{}{V \Downarrow V} \quad \text{(D-T-Val)}$$

$$\frac{M \Downarrow \mathbf{true} \quad M_1 \Downarrow V}{\mathbf{if}\, M \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_2 \Downarrow V} \quad \text{(D-T-If-T)}$$

$$\frac{M \Downarrow \mathbf{false} \quad M_2 \Downarrow V}{\mathbf{if}\, M \,\mathbf{then}\, M_1 \,\mathbf{else}\, M_2 \Downarrow V} \quad \text{(D-T-If-F)}$$

$$\frac{M \Downarrow \lambda(x_1{:}A_1).M' \qquad M_1 \Downarrow V_1 \qquad [V_1/x_1]M' \Downarrow V}{M\,(M_1) \Downarrow V} \quad \text{(D-T-TApp)}$$

Figure 4: Dynamic Semantics of Terms

$$\frac{M \Downarrow V}{M \mathbin{@} w \Downarrow V \mathbin{@} w} \quad \text{(D-E-Term)}$$

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

$$\frac{E_1 \mathbin{@} w_1 \Downarrow V_1 \mathbin{@} w_1' \quad [V_1/x]E_2 \mathbin{@} w_1' \Downarrow V_2 \mathbin{@} w_2'}{\mathbf{let}\, x \,\mathbf{be}\, E_1 \,\mathbf{in}\, E_2 \,\mathbf{end} \mathbin{@} w_1 \Downarrow V_2 \mathbin{@} w_2'} \quad \text{(D-E-Let)}$$

$$\frac{\begin{array}{c} M \Downarrow V_0 \qquad M_1 \Downarrow V_1 \qquad [V_0/x][V_1/x_1]E \mathbin{@} w \Downarrow V' \mathbin{@} w' \\ (V_0 = \mathbf{fun}\; x\; (x_1{:}A_1) : A_2 \;\mathbf{is}\; E) \end{array}}{\mathbf{app}(M, M_1) \mathbin{@} w \Downarrow V' \mathbin{@} w'} \quad \text{(D-E-PApp)}$$

$$\frac{M \Downarrow \mathbf{true} \quad E_1 \mathbin{@} w \Downarrow V_1 \mathbin{@} w_1}{\mathbf{if}\, M \,\mathbf{then}\, E_1 \,\mathbf{else}\, E_2 \mathbin{@} w \Downarrow V_1 \mathbin{@} w_1} \quad \text{(D-E-If-T)}$$

$$\frac{M \Downarrow \mathbf{false} \quad E_2 \mathbin{@} w \Downarrow V_2 \mathbin{@} w_2}{\mathbf{if}\, M \,\mathbf{then}\, E_1 \,\mathbf{else}\, E_2 \mathbin{@} w \Downarrow V_2 \mathbin{@} w_2} \quad \text{(D-E-If-F)}$$

Figure 5: Dynamic Semantics of Expressions

3 Refinements

The canonical forms theorem specifies (some of) the properties that the type structure induces on the value space. For example, values with integer type are $0, 1, 2, 3, \ldots$. In order to define and check further, more specific, properties of values and also computations, we introduce a *logic of refinements* that may be layered on top of the computational lambda calculus described in the previous section.

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.

3.1 Syntax

A (term) *refinement*, or *property*, is a predicate over a type. A *world refinement*, or *world property*, is a predicate over the (implicit) type of the world. Finally, an *expression refinement* is a predicate over both a type and the implicit type of the world. 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 (but not other 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 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.

The multiplicative-additive fragment is decidable [LS94], and the intuitionistic predicates do not change that, but were we to add freely-generated modal formulas $!\psi$, the logic would be undecidable.

When $\vec{b}$ is the empty sequence in some expression refinement $\exists[\vec{b}](\phi, \psi)$, we often use the abbreviation (ϕ, ψ). We use the notation $\mathsf{FV_c}(\phi)$ 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. We write $\vec{b} \vdash \phi \sqsubseteq A$ and $\vec{b} \vdash \eta \sqsubseteq_E A$ to indicate that a term or expression refinement refines the type A given the set of bindings $\vec{b}$. Figure 6 defines this

$$\frac{}{\vec{b} \vdash \boldsymbol{Bool} \sqsubseteq \mathbf{Bool}} \qquad (\textsc{Refines-Bool})$$

$$\frac{}{\vec{b} \vdash \boldsymbol{a} \sqsubseteq \mathbf{a}} \qquad (\textsc{Refines-Base})$$

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

$$\frac{\vec{b} \vdash \phi_1 \sqsubseteq A_1 \quad \vec{b} \vdash \phi_2 \sqsubseteq A_2}{\vec{b} \vdash \phi_1 \to \phi_2 \sqsubseteq A_1 \to A_2} \qquad (\textsc{Refines-TArr})$$

$$\frac{\vec{b} \vdash \phi_1 \sqsubseteq A_1 \quad \vec{b} \vdash \eta_2 \sqsubseteq_E A_2}{\vec{b} \vdash (\phi_1, \psi_1) \rightharpoonup \eta_2 \sqsubseteq A_1 \rightharpoonup A_2} \qquad (\textsc{Refines-PArr})$$

$$\frac{\vec{b}, c{:}\mathbf{a} \vdash \phi \sqsubseteq A}{\vec{b} \vdash \forall c{:}\mathbf{a} \cdot \phi \sqsubseteq A} \qquad (\textsc{Refines-All})$$

$$\frac{\vec{b}, \vec{b}' \vdash \phi_i \sqsubseteq A_i \qquad (\text{for } 1 \le i \le n) \qquad \vec{b}, \vec{b}' \vdash \eta \sqsubseteq_E A}{\vec{b} \vdash \forall \vec{b}' \cdot (\phi_1, \ldots, \phi_n, \psi) \rightharpoonup \eta \sqsubseteq (A_1, \ldots, A_n) \rightharpoonup A} \qquad (\textsc{Refines-OpType})$$

$$\frac{\vec{b}, c{:}\mathbf{a} \vdash \exists[\vec{b}_1](\phi, \psi) \sqsubseteq_E A}{\vec{b} \vdash \exists[c{:}\mathbf{a}, \vec{b}_1](\phi, \psi) \sqsubseteq_E A} \qquad (\textsc{Refines-Ex})$$

$$\frac{\vec{b} \vdash \phi \sqsubseteq A}{\vec{b} \vdash (\phi, \psi) \sqsubseteq_E A} \qquad (\textsc{Refines-ER})$$

Figure 6: A Refinement of a Type

relation. The following useful lemma may be proven by induction on the structure of the refinement in question.

Lemma 3

- *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* $A_1 = A_2$*.*

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 semantic model for our world refinements by the work of Ishtiaq and O'Hearn [IO01]. Since Ishtiaq and O'Hearn work with bunched logic [OP99] whereas we use a fragment of linear logic [Gir87], their model is not appropriate for our system, although there are many similarities. 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.

The semantics appears in Figure 7. The fragment of the logic without the modality ! is an instance of Simon Ambler's resource semantics [Amb91, 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 $\mathsf{less(x,3)} \lesssim \mathsf{less(x,5)}$. In most of our examples, the 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) below. We treat both kinds of contexts as equivalent up to reordering of their elements.[1]

$$\begin{array}{l}
w \vDash_\Omega \Omega \text{ iff } \mathsf{Per}(w) \supseteq \Omega \\
w \vDash_\Delta \cdot \text{ iff } \mathsf{Eph}(w) = \emptyset \\
w \vDash_\Delta \psi_1, \ldots, \psi_n \text{ iff there exist } w_1, \ldots, w_n \text{ such that} \\
\quad w = w_1 + \cdots + w_n \\
\quad w_1 \vDash \psi_1 \cdots w_n \vDash \psi_n \\
w \vDash \Omega; \Delta \text{ iff } w \vDash_\Omega \Omega \text{ and } w \vDash_\Delta \Delta
\end{array}$$

An important point to notice in the semantic definition is that if a given world satisfies a formula or context, then we can always add more persistent facts to the world and it will continue to satisfy the given formula or context. In other words, the persistent facts satisfy the following monotonicity property.

Lemma 4 (Monotonicity of Persistent Facts)
If $(\mathsf{Per}(w), \mathsf{Eph}(w)) \vDash \psi$ *then for any set* S*,* $(\mathsf{Per}(w) \cup S, \mathsf{Eph}(w)) \vDash \psi$*.*

[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).

$w \models \psi$ if and only if

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

Figure 7: Semantics of World Refinements

However, monotonicity does not generally hold for ephemeral facts. Only $\top$ can absorb arbitrarily many ephemeral facts. The other cases for our semantic definition require that the ephemeral set of facts must be empty ($\mathbf{1}$ or $!p(\vec{c})$), a singleton set ($p(\vec{c})$) or divided between subexpressions in such a way that all resources are accounted for ($\&$, $\otimes$, $\multimap$ or $\Omega; \Delta$).

We will show later that linear logical entailment is sound with respect to our semantics. However, as noted by Ambler [Amb91, 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.

$$\begin{array}{llcl} \textit{Persistent Ctxt} & \Omega & ::= & \cdot \mid \Omega, c{:}\mathbf{a} \mid \Omega, x{:}\phi \mid \Omega, p(\vec{c}) \\ \textit{Ephemeral Ctxt} & \Delta & ::= & \cdot \mid \Delta, \psi \end{array}$$

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. When necessary, we will implicitly alpha-vary bound 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 hold for the linear context.

$\vdash \Sigma\ \mathsf{ok}$	*Signature Σ is well-formed*
$\vdash \Omega\ \mathsf{ok}$	*Context Ω is well-formed*
$\Omega \vdash \Delta\ \mathsf{ok}$	*Context Δ is well-formed in Ω*
$\Omega \vdash \phi\ \mathsf{ok}$	*Refinement ϕ is well-formed in Ω*
$\Omega \vdash \psi\ \mathsf{ok}$	*World ref. ψ is well-formed in Ω*
$\Omega \vdash \eta\ \mathsf{ok}$	*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 8: Refinement Checking Judgments

Declarative refinement checking is formulated using the judgment forms in Figure 8. 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 Figures 9, 10 and 11.

The next two judgments form the heart of the system. They check terms and expressions to ensure that they have the appropriate refinements. First, we consider the term refinement checking rules, which may be found in Figure 12. Variables and booleans are given the expected refinements. Constants c are given very precise *singleton types*, following work by Xi and Pfenning [XP99]. When such precision is unnecessary, we may use the subsumption rule (R-T-Sub) to give these constants a more general refinement corresponding to their type (*i.e.,* $\mathsf{triv}(\Sigma_A(c))$). Function definition has the usual form, but with two added conditions. The first requires that the type refinement (or a part thereof) chosen for the function refines the annotated type of the function. This requirement ensures that any term or expression's type refinement indeed refines that term's (expression's) type. This property is expressed more precisely in Lemma 33. The second condition requires the chosen refinement (or a part thereof) to be well-formed, thus ensuring that the context in the refinement checking judgment in the rule's premise is well-formed. Function application, rule (R-T-TApp), has the usual form. This rule 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. The rule (R-T-If) resembles the standard rule for if statements except that we do not bother to check that the first term M has a boolean refinement. Such a check is unnecessary because we assume refinement checking is preceded by ordinary type checking.

The expression refinement checking rules appear in Figure 13. Rule (R-E-Term) defines the interface between pure and effectful computations. Pure terms themselves do not produce state,

$$\frac{\begin{array}{c}\Sigma = (\mathcal{B}, \mathcal{C}, \mathcal{O}, \Sigma_A, \mathcal{P}, \Sigma_p, \Sigma_\phi) \\ \cdot \vdash \Sigma_\phi(c) \sqsubseteq \Sigma_A(c) \qquad \cdot \vdash \Sigma_\phi(c)\ \mathsf{ok} \qquad (\text{for } c \in \mathcal{C}) \\ \cdot \vdash \Sigma_\phi(\mathsf{o}) \sqsubseteq \Sigma_A(\mathsf{o}) \qquad \cdot \vdash \Sigma_\phi(\mathsf{o})\ \mathsf{ok} \qquad (\text{for } \mathsf{o} \in \mathcal{O})\end{array}}{\vdash \Sigma\ \mathsf{ok}} \qquad (\text{WF-Sig})$$

$$\frac{}{\vdash \cdot\ \mathsf{ok}} \qquad (\text{I-CTXT-Empty})$$

$$\frac{\vdash \Omega\ \mathsf{ok}}{\vdash \Omega, c{:}\mathbf{a}\ \mathsf{ok}}\ (c \notin \mathsf{Dom}(\Omega) \cup \mathsf{Dom}(\Sigma_\phi)) \qquad (\text{I-CTXT-Const})$$

$$\frac{\vdash \Omega\ \mathsf{ok} \quad \Omega \vdash \phi\ \mathsf{ok}}{\vdash \Omega, x{:}\phi\ \mathsf{ok}}\ (x \notin \mathsf{Dom}(\Omega)) \qquad (\text{I-CTXT-Var})$$

$$\frac{\vdash \Omega\ \mathsf{ok} \quad \Omega \vdash p(c_1, \ldots, c_n)\ \mathsf{ok}}{\vdash \Omega, p(c_1, \ldots, c_n)\ \mathsf{ok}} \qquad (\text{I-CTXT-Pred})$$

$$\frac{}{\Omega \vdash \cdot\ \mathsf{ok}} \qquad (\text{L-CTXT-Empty})$$

$$\frac{\Omega \vdash \Delta\ \mathsf{ok} \quad \Omega \vdash \psi\ \mathsf{ok}}{\Omega \vdash \Delta, \psi\ \mathsf{ok}} \qquad (\text{L-CTXT-Refs})$$

Figure 9: Well-formed Signatures and Contexts

$$\frac{}{\Omega \vdash \boldsymbol{a} \ \mathsf{ok}} \quad \text{(WF-Base)}$$

$$\frac{}{\Omega \vdash \boldsymbol{Bool} \ \mathsf{ok}} \quad \text{(WF-Bool)}$$

$$\frac{c \in \mathsf{Dom}(\Omega) \cup \mathsf{Dom}(\Sigma_\phi)}{\Omega \vdash \boldsymbol{Its}(c) \ \mathsf{ok}} \quad \text{(WF-Its)}$$

$$\frac{\Omega \vdash \phi_1 \ \mathsf{ok} \quad \Omega \vdash \phi_2 \ \mathsf{ok}}{\Omega \vdash \phi_1 \to \phi_2 \ \mathsf{ok}} \quad \text{(WF-TArr)}$$

$$\frac{\begin{array}{c}\Omega \vdash \phi \ \mathsf{ok} \quad \Omega \vdash \psi \ \mathsf{ok} \\ \Omega \vdash \eta \ \mathsf{ok}\end{array}}{\Omega \vdash (\phi, \psi) \rightharpoonup \eta \ \mathsf{ok}} \quad \text{(WF-PArr)}$$

$$\frac{\Omega, c{:}\mathbf{a} \vdash \phi \ \mathsf{ok}}{\Omega \vdash \forall c{:}\mathbf{a} \cdot \phi \ \mathsf{ok}} \ (c \notin \mathsf{Dom}(\Omega) \cup \mathsf{Dom}(\Sigma_\phi)) \quad \text{(WF-All)}$$

$$\frac{\begin{array}{c}\Omega, \vec{b} \vdash \phi_i \ \mathsf{ok} \qquad (\text{for } 1 \leq i \leq n) \\ \Omega, \vec{b} \vdash \psi \ \mathsf{ok} \qquad \Omega, \vec{b} \vdash \eta \ \mathsf{ok}\end{array}}{\Omega \vdash \forall \vec{b} \cdot (\phi_1, \ldots, \phi_n, \psi) \rightharpoonup \eta \ \mathsf{ok}} \quad \text{(WF-Op)}$$

Figure 10: Well-formed Term Refinements

$$\frac{\begin{array}{c}\Sigma_A(c_i) = \mathbf{a}_i \text{ or } \Omega(c_i) = \mathbf{a}_i \quad (\text{for } 1 \leq i \leq n) \\ (\Sigma_p(p) = (\mathbf{a}_1, \ldots, \mathbf{a}_n) \rightharpoonup \mathbf{prop})\end{array}}{\Omega \vdash p(c_1, \ldots, c_n) \ \mathsf{ok}} \qquad (\text{WF-WR-Pred})$$

$$\frac{\Omega \vdash p(c_1, \ldots, c_n) \ \mathsf{ok}}{\Omega \vdash !p(c_1, \ldots, c_n) \ \mathsf{ok}} \qquad (\text{WF-WR-!Pred})$$

$$\frac{}{\Omega \vdash \mathbf{1} \ \mathsf{ok}} \qquad (\text{WF-WR-1})$$

$$\frac{\Omega \vdash \psi_1 \ \mathsf{ok} \quad \Omega \vdash \psi_2 \ \mathsf{ok}}{\Omega \vdash \psi_1 \otimes \psi_2 \ \mathsf{ok}} \qquad (\text{WF-WR-MAnd})$$

$$\frac{\Omega \vdash \psi_1 \ \mathsf{ok} \quad \Omega \vdash \psi_2 \ \mathsf{ok}}{\Omega \vdash \psi_1 \multimap \psi_2 \ \mathsf{ok}} \qquad (\text{WF-WR-Impl})$$

$$\frac{}{\Omega \vdash \top \ \mathsf{ok}} \qquad (\text{WF-WR-Top})$$

$$\frac{\Omega \vdash \psi_1 \ \mathsf{ok} \quad \Omega \vdash \psi_2 \ \mathsf{ok}}{\Omega \vdash \psi_1 \,\&\, \psi_2 \ \mathsf{ok}} \qquad (\text{WF-WR-And})$$

$$\frac{}{\Omega \vdash \mathbf{0} \ \mathsf{ok}} \qquad (\text{WF-WR-0})$$

$$\frac{\Omega \vdash \psi_1 \ \mathsf{ok} \quad \Omega \vdash \psi_2 \ \mathsf{ok}}{\Omega \vdash \psi_1 \oplus \psi_2 \ \mathsf{ok}} \qquad (\text{WF-WR-Or})$$

$$\frac{\Omega \vdash \phi \ \mathsf{ok} \quad \Omega \vdash \psi \ \mathsf{ok}}{\Omega \vdash \exists[\,](\phi, \psi) \ \mathsf{ok}} \qquad (\text{WF-ER-Empty})$$

$$\frac{\Omega, c{:}\mathbf{a} \vdash \exists[\vec{b}](\phi, \psi) \ \mathsf{ok}}{\Omega \vdash \exists[c{:}\mathbf{a}, \vec{b}](\phi, \psi) \ \mathsf{ok}} \ (c \notin \mathsf{Dom}(\Omega) \cup \mathsf{Dom}(\Sigma_\phi)) \qquad (\text{WF-ER-Binding})$$

Figure 11: Well-formed World and Expression Refinements

and so are restricted to execute only in an empty emphemeral context and are given the corresponding world refinement **1**. Then, 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 constants to substitute for the polymorphic parameters in the operator refinement. Given this substitution, we must check that operator arguments may be given refinements equal to their corresponding formal parameter. The rules (R-E-PApp) and (R-E-If) are similar to their pure counterparts except that they produce expression refinements rather than term refinements.

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 $(\Omega_1; \Delta_1), \ldots, (\Omega_n; \Delta_n)$. The judgment may be read as saying "Context $\Omega; \Delta$ reduces to the context list $(\Omega_i; \Delta_i)_n$." 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. Hence, the number of times the rule (R-E-Context) can be applied in sequence is bounded by the number of connectives in the context. This fact is one of the keys to the decidability of our type system.

A sample valid judgment involves the modal formula $!p(c_1, \ldots, c_n)$.

$$\Omega; \Delta, !p(c_1, \ldots, c_n) \rightsquigarrow \Omega, p(c_1, \ldots, c_n); \Delta$$

It shifts the modal formula from the linear context into the unrestricted context and removes the modality. In this case, no further conditions must be checked to validate this transformation. A second example involves linear implication.

$$\frac{\Omega; \Delta_1 \Longrightarrow_W \psi_1}{\Omega; \Delta_1, \Delta_2, \psi_1 \multimap \psi_2 \rightsquigarrow \Omega; \Delta_2, \psi_2}$$

This time, we must check the condition $\Omega; \Delta_1 \Longrightarrow_W \psi_1$ in order to validate the context reduction to $\Omega; \Delta_2, \psi_2$. Most of the rules produce one context, which must be used to continue checking the expression E. 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

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

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

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

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

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

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

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

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

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

Figure 12: Refinement Checking for Terms

term proof systems with rules for existentials. These judgments are formally defined in Figures 14, 15, 16 and 17.

3.4 Properties of Refinement Checking Judgments

The following lemma expresses a number of well-formedness properties of our refinement checking judgments.

Lemma 5

1. *If* $\vdash \Omega\ \mathsf{ok}$, $\Omega \vdash \Delta\ \mathsf{ok}$ *and* $\Omega; \Delta \leadsto (\Omega_i; \Delta_i)_n$ *then* $\vdash \Omega_i\ \mathsf{ok}$ *and* $\Omega_i \vdash \Delta_i\ \mathsf{ok}$ *(for* $1 \leq i \leq n$*).*
2. *If* $\vdash \Omega\ \mathsf{ok}$, $\Omega \vdash \Delta\ \mathsf{ok}$ *and* $\Omega; \Delta \leadsto^* (\Omega_i; \Delta_i)_n$ *then* $\vdash \Omega_i\ \mathsf{ok}$ *and* $\Omega_i \vdash \Delta_i\ \mathsf{ok}$ *(for* $1 \leq i \leq n$*).*
3. *If* $\vdash \Omega\ \mathsf{ok}$, $\Omega \vdash \Delta\ \mathsf{ok}$ *and* $\Omega; \Delta \Longrightarrow_W \psi$ *then* $\Omega \vdash \psi\ \mathsf{ok}$.
4. *If* $\vdash \Omega\ \mathsf{ok}$, $\Omega \vdash \phi\ \mathsf{ok}$ *and* $\Omega; \phi \Longrightarrow_W \phi'$ *then* $\Omega \vdash \phi'\ \mathsf{ok}$.

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

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

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

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

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

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

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

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

Figure 13: Refinement Checking for Expressions

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

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

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

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

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

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

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

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

Figure 14: Entailment for Term Refinements

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

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

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

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

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

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

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

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

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

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

Figure 15: Context Reduction and Its Closure

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

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

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

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

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

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

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

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

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

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

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

Figure 16: Entailment for World Refinements

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

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

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

Figure 17: Entailment for Expression Refinements

5. *If* $\vdash \Omega$ ok, $\Omega \vdash \Delta$ ok, $\Omega \vdash \eta$ ok *and* $\Omega; \Delta; \eta \Longrightarrow_E \eta'$ *then* $\Omega \vdash \eta'$ ok.
6. *If* $\vdash \Omega$ ok *and* $\Omega \gg_M M : \phi$ *then* $\Omega \vdash \phi$ ok.
7. *If* $\vdash \Omega$ ok, $\Omega \vdash \Delta$ ok *and* $\Omega; \Delta \gg_E E : \eta$ *then* $\Omega \vdash \eta$ ok.

Proof: The proof of the first item is by inspection and the second by induction, using the first when necessary. The proof of the third item is by induction on the height of the entailment derivation, using the first when necessary. The proof of the fourth and fifth items is by simultaneous induction on the entailment derivation in each case, using the second item when necessary. The proof of the last two items is by simultaneous induction on the height of the refinement-checking derivation in each case, using the first five items when necessary. ■

The following lemma states that all term refinements related by the term entailment judgment refine the same type. It is needed in the proof of the term inversion lemma in section 3.7.

Lemma 6
If $\Omega; \phi' \Longrightarrow_M \phi$ *then* $\Omega_b \vdash \phi' \sqsubseteq A$ *iff* $\Omega_b \vdash \phi \sqsubseteq A$. *If* $\Omega; \Delta; \eta' \Longrightarrow_E \eta$ *then* $\Omega_b \vdash \eta' \sqsubseteq A$ *iff* $\Omega_b \vdash \eta \sqsubseteq A$.

Proof: The proof is by simultaneous induction on the entailment derivation, relying on the syntax-directedness of the $\sqsubseteq$ relation. ■

The following lemma expresses the relationship between our world semantics and logical judgments, stating that logical deduction respects the semantics of formulas. More specifically, item 1 expresses the property that if a set of contexts, $\Omega; \Delta$, is a satisfactory description of a world, w, then at least one of the sets of contexts to which those contexts reduce is also a satisfactory description of w. Item 2 expresses the property that if a set of contexts, $\Omega; \Delta$, is a satisfactory description of a world, w, then any formula, ψ, entailed by those contexts is itself a satisfactory description of w.

This lemma plays a critical role in our proof of preservation.

Lemma 7 (Soundness of Logical Judgments)

1. *If* $w \vDash \Omega; \Delta$ *and* $\Omega; \Delta \leadsto^* (\Omega'_i; \Delta'_i)_{i=1}^n$ *then for some* $i : 1..n$, $w \vDash \Omega'_i; \Delta'_i$.
2. *If* $w \vDash \Omega; \Delta$ *and* $\Omega; \Delta \Longrightarrow_W \psi$ *then* $w \vDash \psi$.

Proof: The proof of parts 1 and 2 is by simultaneous induction on the heights of context-reduction and entailment derivations. ■

The following are standard substitution lemmas for all but the first two refinement-checking judgments listed in Figure 8.

Lemma 8 (Substitution)
Suppose that $\Omega \gg_M c : \mathbf{a}$.

- *If* $\Omega, c':\mathbf{a}, \Omega'; \phi \Longrightarrow_M \phi'$, *then* $\Omega, [c/c']\Omega'; [c/c']\phi \Longrightarrow_M [c/c']\phi'$.
- *If* $\Omega, c':\mathbf{a}, \Omega'; \Delta \Longrightarrow_W \psi$, *then* $\Omega, [c/c']\Omega'; [c/c']\Delta \Longrightarrow_W [c/c']\psi$.
- *If* $\Omega, c':\mathbf{a}, \Omega'; \Delta; \eta \Longrightarrow_E \eta'$, *then* $\Omega, [c/c']\Omega'; [c/c']\Delta; [c/c']\eta \Longrightarrow_E [c/c']\eta'$.
- *If* $\Omega, c':\mathbf{a} \vdash \Delta\ \mathsf{ok}$ *then* $\Omega \vdash [c/c']\Delta\ \mathsf{ok}$
- *If* $\Omega, c':\mathbf{a} \vdash \phi\ \mathsf{ok}$ *then* $\Omega \vdash [c/c']\phi\ \mathsf{ok}$
- *If* $\Omega, c':\mathbf{a} \vdash \psi\ \mathsf{ok}$ *then* $\Omega \vdash [c/c']\psi\ \mathsf{ok}$
- *If* $\Omega, c':\mathbf{a} \vdash \eta\ \mathsf{ok}$ *then* $\Omega \vdash [c/c']\eta\ \mathsf{ok}$
- *If* $\Omega, c':\mathbf{a}, \Omega'; \Delta \rightsquigarrow (\Omega_i, c':\mathbf{a}, \Omega'_i; \Delta_i)_n$ *then* $\Omega, [c/c']\Omega'; [c/c']\Delta \rightsquigarrow (\Omega_i, [c/c']\Omega'_i; [c/c']\Delta_i)_n$
- *If* $\Omega, c':\mathbf{a}, \Omega'; \Delta \rightsquigarrow^* (\Omega_i, c':\mathbf{a}, \Omega'_i; \Delta_i)_n$ *then* $\Omega, [c/c']\Omega'; [c/c']\Delta \rightsquigarrow^* (\Omega_i, [c/c']\Omega'_i; [c/c']\Delta_i)_n$
- *If* $\Omega, c':\mathbf{a}, \Omega' \gg_M M : \phi$, *then* $\Omega, [c/c']\Omega' \gg_M M : [c/c']\phi$. *Similarly, if* $\Omega, c':\mathbf{a}, \Omega'; \Delta \gg_E E : \eta$, *then* $\Omega, [c/c']\Omega'; [c/c']\Delta \gg_E E : [c/c']\eta$.

Additionally, if $\Omega, x:\phi' \gg_M M : \phi$, *and* $\Omega \gg_M V : \phi'$, *then* $\Omega \gg_M [V/x]M : \phi$. *Similarly, if* $\Omega, x:\phi'; \Delta \gg_E E : \eta$ *and* $\Omega \gg_M V : \phi'$, *then* $\Omega; \Delta \gg_E [V/x]E : \eta$.

Proof: By induction on the height of the relevant derivations. ■

Lemma 9 (Reflexivity and Transitivity of Entailment)

1. $\Omega; \phi \Longrightarrow_M \phi$
2. *If* $\Omega; \phi_1 \Longrightarrow_M \phi_2$ *and* $\Omega; \phi_2 \Longrightarrow_M \phi_3$ *then* $\Omega; \phi_1 \Longrightarrow_M \phi_3$.
3. $\Omega; \Delta; \eta \Longrightarrow_E \eta$
4. *If* $\Omega; \Delta_1; \eta_1 \Longrightarrow_E \eta_2$ *and* $\Omega; \Delta_2; \eta_2 \Longrightarrow_E \eta_3$ *then* $\Omega; \Delta_1, \Delta_2; \eta_1 \Longrightarrow_E \eta_3$.

Proof: The proof of reflexivity is by simultaneous induction on term and expression entailment derivations. The proof of transitivity is by simultaneous induction on the first derivation of both term and expression entailment derivations. In the case of (L-T-AllR), we apply Lemma 8. In the case of (L-ER-Base), we apply rule (L-E-Cut) to the second premise in obtaining our conclusion. ■

Lemma 10 (Admissibility of Cut for Term Refinements)
If $\Omega; \phi_1 \Longrightarrow_M \phi_2$ *then*

- *if* $\Omega, x:\phi_2 \gg_M M : \phi$ *then* $\Omega, x:\phi_1 \gg_M M : \phi$.
- *if* $\Omega, x:\phi_2; \Delta \gg_E E : \eta$ *then* $\Omega, x:\phi_1; \Delta \gg_E E : \eta$.

Proof: By induction on refinement derivations. For the case (R-T-Var) we apply (R-T-Sub) to yield our conclusion. ■

3.5 Algorithmic Refinement Checking

We develop an algorithmic refinement checking system in two steps. The new system is algorithmic up to the resolution of linear logic theorem proving (which is itself decidable).

1. 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 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 [Pfe94].

2. Subsumption elimination and annotation introduction. In this step we eliminate two critical forms of non-determinism present in the previous system. We introduce type refinement annotations into the language, allowing the programmer to guide the checker in its search for type-refinement derivations. We furthermore 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.

   At this point, there is one typing rule for each expression or term construct. All premises in the rules are now fully determined, except those of the context-reduction judgment. We show soundness and completeness of the new system.

3.5.1 Cut Elimination

Figure 18 gives the cut-free rules for expression refinement checking. These rules rely upon five new judgments:

$\Omega \gg^{nc}_{M} M : \phi$	Cut-free term refinement checking
$\Omega; \Delta \gg^{nc}_{E} E : \eta$	Cut-free expression refinement checking
$\Omega; \phi \Longrightarrow^{nc}_{M} \phi'$	Cut-free term refinement entailment
$\Omega; \Delta \Longrightarrow^{nc}_{W} \psi$	Cut-free world refinement entailment
$\Omega; \Delta; \eta \Longrightarrow^{nc}_{E} \eta'$	Cut-free expression refinement entailment

Only the second new judgment is significantly different from the corresponding cut-containing judgment. The others are identical to the corresponding cut-containing judgment except that they are mutually dependent upon other cut-free judgments (and of course, the cut-free world refinement entailment derivations do not contain the cut rule). Hence, we do not include the rules for these other judgments.

Lemma 11 (Equivalence of Cut-Free Refinement Checking)

- *(Soundness) If* $\Omega; \Delta \gg^{nc}_{E} E : \eta$ *then* $\Omega; \Delta \gg_{E} E : \eta$.
- *(Completeness) If* $\Omega; \Delta \gg_{E} E : \eta$ *then* $\Omega; \Delta \gg^{nc}_{E} E : \eta$.

Proof: The proof of soundness is straightforward, as any cut-free derivation is an instance of the declarative system with the cut rule applied in one of three fixed places (as a hypothesis in rules NC-E-Term, NC-E-Op, or NC-E-PAPP). Similarly, the cut-free sequent calculus proofs are a subset of the cut-containing sequent calculus proofs.

The proof of completeness follows standard techniques [Pfe94]. ■

3.5.2 Subsumption Elimination and Annotation Introduction

At this point, we eliminate two critical sources of non-determinism present in the previous system. The first source arises from rules with elements in the premises that do not appear in the conclusions. These elements, then, must be "guessed" when reading the rules from the top to bottom, as would be

$$\frac{\Omega \gg^{nc}_{M} M : \phi}{\Omega; \cdot \gg^{nc}_{E} M : (\phi, \mathbf{1})} \quad (\text{NC-E-Term})$$

$$\frac{\begin{array}{c}\Omega; \Delta \Longrightarrow^{nc}_{W} [\vec{c}/\vec{b}]\psi_1 \\ \Omega \gg^{nc}_{M} M_i : [\vec{c}/\vec{b}]\phi_i \quad (\text{for } 1 \leq i \leq n) \\ (\Sigma_\phi(\mathsf{o}) = \forall \vec{b} \cdot (\phi_1, \ldots, \phi_n, \psi_1) \rightharpoonup \eta)\end{array}}{\Omega; \Delta \gg^{nc}_{E} \mathsf{o}(M_1, \ldots, M_n) : [\vec{c}/\vec{b}]\eta} \quad (\text{NC-E-Op})$$

$$\frac{\begin{array}{c}\Omega; \Delta \gg^{nc}_{E} E_1 : \eta_1 \\ \Omega, \vec{b}_1, x{:}\phi_1; \psi_1 \gg^{nc}_{E} E_2 : \eta_2 \\ (\vec{b}_1 \notin \mathsf{FV_c}(\eta_2))\end{array}}{\Omega; \Delta \gg^{nc}_{E} \mathbf{let}\, x \,\mathbf{be}\, E_1 \,\mathbf{in}\, E_2 \,\mathbf{end} : \eta_2} \; (\eta_1 = \exists[\vec{b}_1](\phi_1, \psi_1)) \quad (\text{NC-E-Let})$$

$$\frac{\Omega; \Delta \Longrightarrow^{nc}_{W} \psi_1 \quad \Omega \gg^{nc}_{M} M : (\phi_1, \psi_1) \rightharpoonup \eta \quad \Omega \gg^{nc}_{M} M_1 : \phi_1}{\Omega; \Delta \gg^{nc}_{E} \mathbf{app}(M, M_1) : \eta} \quad (\text{NC-E-PApp})$$

$$\frac{\Omega; \Delta \gg^{nc}_{E} E_1 : \eta \quad \Omega; \Delta \gg^{nc}_{E} E_2 : \eta}{\Omega; \Delta \gg^{nc}_{E} \mathbf{if}\, M \,\mathbf{then}\, E_1 \,\mathbf{else}\, E_2 : \eta} \quad (\text{NC-E-If})$$

$$\frac{\begin{array}{c}\Omega; \Delta \rightsquigarrow^{*} (\Omega_i; \Delta_i)_n \\ \Omega_i; \Delta_i \gg^{nc}_{E} E : \eta \quad (\text{for } 1 \leq i \leq n)\end{array}}{\Omega; \Delta \gg^{nc}_{E} E : \eta} \quad (\text{NC-E-Context})$$

$$\frac{\Omega; \Delta_1 \gg^{nc}_{E} E : \eta \quad \Omega; \Delta_2; \eta \Longrightarrow^{nc}_{E} \eta'}{\Omega; \Delta_1, \Delta_2 \gg^{nc}_{E} E : \eta'} \quad (\text{NC-E-Sub})$$

Figure 18: Cut-free Refinement Checking for Expressions

$$\frac{}{\Omega, x : \phi \gg^{nsc}_{M} x \uparrow \phi} \qquad \text{(NSC-T-Var)}$$

$$\frac{c \in \mathsf{Dom}(\Sigma_\phi)}{\Omega \gg^{nsc}_{M} c \uparrow \boldsymbol{Its}(c)} \qquad \text{(NSC-T-Const)}$$

$$\frac{}{\Omega \gg^{nsc}_{M} \mathbf{true} \uparrow \boldsymbol{Bool}} \qquad \text{(NSC-T-True)}$$

$$\frac{}{\Omega \gg^{nsc}_{M} \mathbf{false} \uparrow \boldsymbol{Bool}} \qquad \text{(NSC-T-False)}$$

$$\frac{\Omega \gg^{nsc}_{M} M_I \uparrow \forall \vec{b} \cdot \phi_1 \to \phi_2 \quad \Omega \gg^{nsc}_{M} M_C \downarrow [\vec{c}/\vec{b}]\phi_1}{\Omega \gg^{nsc}_{M} M_I\,(M_C) \uparrow [\vec{c}/\vec{b}]\phi_2} \qquad \text{(NSC-T-TApp)}$$

$$\frac{\Omega \gg^{nsc}_{M} M_C \downarrow \phi}{\Omega \gg^{nsc}_{M} M_C \lhd \phi \uparrow \phi} \qquad \text{(NSC-T-CtoI)}$$

$$\frac{\Omega \gg^{nsc}_{M} M_I \uparrow \phi' \quad \Omega; \phi' \Longrightarrow^{nc}_{M} \phi}{\Omega \gg^{nsc}_{M} M_I \downarrow \phi} \qquad \text{(NSC-T-ItoC)}$$

$$\frac{\begin{array}{cc} \Omega_b, \vec{b} \vdash \phi_1 \sqsubseteq A & \Omega, \vec{b} \vdash \phi_1 \ \mathsf{ok} \\ \multicolumn{2}{c}{\Omega, \vec{b}, x{:}\phi_1 \gg^{nsc}_{M} M_C \downarrow \phi_2 \quad (\phi = \forall \vec{b} \cdot \phi_1 \to \phi_2)} \end{array}}{\Omega \gg^{nsc}_{M} \lambda(x{:}A).M_C \downarrow \phi} \qquad \text{(NSC-T-Lam)}$$

$$\frac{\begin{array}{c} \Omega_b \vdash \phi \sqsubseteq A_1 \rightharpoonup A \quad \Omega \vdash \phi \ \mathsf{ok} \\ \Omega, x{:}\phi, \vec{b}, x_1{:}\phi_1; \psi_1 \gg^{nsc}_{E} E_C \downarrow \eta; \cdot \\ (\phi = \forall \vec{b} \cdot (\phi_1, \psi_1) \rightharpoonup \eta) \end{array}}{\Omega \gg^{nsc}_{M} \mathbf{fun}\ x\ (x_1{:}A_1) : A\ \mathbf{is}\ E_C \downarrow \phi} \qquad \text{(NSC-T-Fun)}$$

$$\frac{\Omega \gg^{nsc}_{M} M_{C1} \downarrow \phi \quad \Omega \gg^{nsc}_{M} M_{C2} \downarrow \phi}{\Omega \gg^{nsc}_{M} \mathbf{if}\, M\, \mathbf{then}\, M_{C1}\, \mathbf{else}\, M_{C2} \downarrow \phi} \qquad \text{(NSC-T-If)}$$

Figure 19: Cut-free, Subsumption-free, Bi-Directional Refinement Checking for Terms

$$\frac{\Omega \gg^{nsc}_{M} M_I \uparrow \phi}{\Omega; \Delta \gg^{nsc}_{E} M_I \uparrow (\phi, \mathbf{1}); \Delta} \qquad \text{(NSC-E-ITerm)}$$

$$\frac{\begin{array}{c}\Omega; \Delta \Longrightarrow^{nc}_{W} [\vec{c}/\vec{b}]\psi_1 \\ \Omega \gg^{nsc}_{M} M_{Ci} \downarrow [\vec{c}/\vec{b}]\phi_i \quad (\text{for } 1 \leq i \leq n) \\ (\Sigma_\phi(\mathsf{o}) = \forall \vec{b} \cdot (\phi_1, \ldots, \phi_n, \psi_1) \rightharpoonup \eta)\end{array}}{\Omega; \Delta, \Delta' \gg^{nsc}_{E} \mathsf{o}(M_{C1}, \ldots, M_{Cn}) \uparrow [\vec{c}/\vec{b}]\eta; \Delta'} \qquad \text{(NSC-E-Op)}$$

$$\frac{\begin{array}{c}\Omega; \Delta \Longrightarrow^{nc}_{W} [\vec{c}/\vec{b}]\psi_1 \\ \Omega \gg^{nsc}_{M} M_I \uparrow \forall \vec{b} \cdot (\phi_1, \psi_1) \rightharpoonup \eta \quad \Omega \gg^{nsc}_{M} M_C \downarrow [\vec{c}/\vec{b}]\phi_1\end{array}}{\Omega; \Delta, \Delta' \gg^{nsc}_{E} \mathbf{app}(M_I, M_C) \uparrow [\vec{c}/\vec{b}]\eta; \Delta'} \qquad \text{(NSC-E-PApp)}$$

$$\frac{\Omega; \Delta \gg^{nsc}_{E} E_C \downarrow \eta; \Delta'}{\Omega; \Delta \gg^{nsc}_{E} E_C \triangleleft \eta \uparrow \eta; \Delta'} \qquad \text{(NSC-E-CtoI)}$$

$$\frac{\Omega; \Delta \gg^{nsc}_{E} E_I \uparrow \eta'; \Delta', \Delta'' \quad \Omega; \Delta''; \eta' \Longrightarrow^{nc}_{E} \eta}{\Omega; \Delta \gg^{nsc}_{E} E_I \downarrow \eta; \Delta'} \qquad \text{(NSC-E-ItoC)}$$

$$\frac{\Omega \gg^{nsc}_{M} M_C \downarrow [\vec{c}/\vec{b}]\phi \quad \Omega; \Delta \Longrightarrow^{nc}_{W} [\vec{c}/\vec{b}]\psi}{\Omega; \Delta, \Delta' \gg^{nsc}_{E} M_C \downarrow \eta; \Delta'} \; (\eta = \exists[\vec{b}](\phi, \psi)) \qquad \text{(NSC-E-CTerm)}$$

$$\frac{\begin{array}{c}\Omega; \Delta \gg^{nsc}_{E} E_I \uparrow \eta_1; \Delta_1 \\ \Omega, \vec{b}, x{:}\phi_1; \Delta_1, \psi_1 \gg^{nsc}_{E} E_C \downarrow \eta_2; \Delta_2 \\ (\vec{b} \notin \mathsf{FV}_{\mathsf{c}}(\eta_2))\end{array}}{\Omega; \Delta \gg^{nsc}_{E} \mathbf{let}\, \vec{b}, x\, \mathbf{be}\, E_I\, \mathbf{in}\, E_C\, \mathbf{end} \downarrow \eta_2; \Delta_2} \; (\eta_1 = \exists[\vec{b}](\phi_1, \psi_1)) \qquad \text{(NSC-E-Let)}$$

$$\frac{\Omega; \Delta \gg^{nsc}_{E} E_{C1} \downarrow \eta; \Delta' \quad \Omega; \Delta \gg^{nsc}_{E} E_{C2} \downarrow \eta; \Delta'}{\Omega; \Delta \gg^{nsc}_{E} \mathbf{if}\, M\, \mathbf{then}\, E_{C1}\, \mathbf{else}\, E_{C2} \downarrow \eta; \Delta'} \qquad \text{(NSC-E-If)}$$

$$\frac{\begin{array}{c}\Omega; \Delta \rightsquigarrow^{*} (\Omega_i; \Delta_i)_m \\ \Omega_i; \Delta_i \gg^{nsc}_{E} /E_C/_i \downarrow \eta; \Delta' \quad (\text{for } 1 \leq i \leq m)\end{array}}{\Omega; \Delta \gg^{nsc}_{E} E_C \triangleleft m \downarrow \eta; \Delta'} \qquad \text{(NSC-E-Context)}$$

Figure 20: Cut-free, Subsumption-free, Bi-Directional Refinement Checking for Expressions

done in a refinement-checking algorithm. A classic example is the application rule (R-T-TApp), for which ϕ_1 is unspecified in the conlusion. The second source of non-determinism is the subsumption rules, (R-T-Sub) and (NC-E-Sub), as neither rule is syntax-directed.

To elmininate the first source of non-determinism, we add an annotation syntax to the language, allowing the programmer to add refinement annotations to expressions where the refinement checker would have otherwise needed to "guess" the appropriate refinement. In order to minimize the number of annotations needed, we use bi-directional refinement checking in a similar style to that used by Davies and Pfenning [DP00]. We redefine our grammar, distinguishing between terms and expressions whose refinement can or need be inferred and those whose refinement can be checked against a specified refinement. We classify as inferable those terms (expressions) whose type refinement can always be inferred or terms (expressions) which require that at least one of their subterms be inferable and are therefore themselves inferable. We classify all terms (expressions), including inferable ones, as checkable.

For example, consider function application. From the refinement of an application we cannot derive the refinement of the function itself, and, therefore, the refinement of the function must be inferred. As the function refinement contains a (quantified) result refinement from which the application refinement can be derived, function application is classified as an inferable term. For another example, consider functions and if statements. Since, generally, function refinements are not inferable and our system does not have meets and joins, functions and if statements are classified as only checkable terms.

Finally, we introduce a new form of inferable term (expression) consisting of a checkable term (expression) annotated with a type refinement. The system infers the type refinement of the term (expression) to be that specified by the annotation.

The modified grammar follows:

$$\begin{array}{lll l}
\textit{Annotations} & \alpha & ::= & \phi \mid \eta \mid m \mid \bullet \mid \{\alpha_i\}_m \\
\textit{Bindings} & \beta & ::= & \vec{b} \mid \{\beta_i\}_m \\
\textit{Values} & V_I & ::= & X \mid c \mid \mathbf{true} \mid \mathbf{false} \mid \\
 & V_C & ::= & \lambda(X).M_C \mid \mathbf{fun}\ X\ (X_1{:}A_1) : A_2\ \mathbf{is}\ E_C \\
\textit{Terms} & M_I & ::= & V_I \mid M_I\,(M_C) \mid M_C \triangleleft \alpha \\
 & M_C & ::= & M_I \mid V_C \mid \mathbf{if}\, M\, \mathbf{then}\, M_{C1}\, \mathbf{else}\, M_{C2} \\
\textit{Exp's} & E_I & ::= & M_I \mid \mathsf{o}(M_{C1}, \ldots, M_{Ck}) \mid \mathbf{app}(M_I, M_C) \mid \\
 & & & E_C \triangleleft \alpha \\
 & E_C & ::= & E_I \mid M_C \mid \mathbf{let}\, \beta, X\, \mathbf{be}\, E_I\, \mathbf{in}\, E_C\, \mathbf{end} \mid \\
 & & & \mathbf{if}\, M\, \mathbf{then}\, E_{C1}\, \mathbf{else}\, E_{C2}
\end{array}$$

Our new grammar contains five forms of annotations: term annotations (ϕ), expression annotations (η), context-rule annotations (m, with m an integer), the empty annotation ($\bullet$), and lists of annotations ($\{\alpha_i\}_m$). The annotation lists are relevant for use with the (NSC-E-Context) rule. Term and expression annotations, and lists thereof, can only be used with terms and expressions, respectively. The empty annotation only has meaning within a list. We also add an annotated form of the **let** expression, allowing programmers to bind the existential variables appearing in E_1 so that they can be used in annotations in E_2. Note that the condition term in both the term and expression **if** statements remains an unnannotated term M, as it is only examined by the type-checker and not the refinement checker.

Next, we introduce two new classes of refinement-checking judgments, one for inferable terms and expressions, and one for checkable terms and expressions. In total, we introduce four new judgments, shown below.

$\Omega \gg_M^{nsc} M_I \uparrow \phi$	Term refinement inference
$\Omega \gg_M^{nsc} M_C \downarrow \phi$	Term refinement checking
$\Omega; \Delta \gg_E^{nsc} E_I \uparrow \eta; \Delta'$	Expression refinement inference
$\Omega; \Delta \gg_E^{nsc} E_C \downarrow \eta; \Delta'$	Expression refinement checking

As a shorthand to refer to both the inference ($\uparrow$) and checking ($\downarrow$) judgments simultaneously, we use the notation $\updownarrow$. Similarly, we refer generically to M (E) when we do not wish to distinguish between inferable and checkable terms (expressions).

The difference between the two judgment classes lies in whether we regard the type refinement on the right side of the judgment as an input or as an ouput of the refinement-checking algorithm. The inference judgments regard the type refinement as an output. The checking judgments regard the type refinement as an input to be checked against the term (or expression) specified in the judgment. Therefore, while there is only one refinement that can be inferred for a given inferable term (expression), there are possibly many that can be checked against a checkable term (expression).

Another notable feature of the above judgments is the addition of the output Δ' to the expression judgments. This output represents the portion of the ephemeral context that is unused in inferring E_I to have type refinement η or in checking E_I against η.

The annotation savings of the bi-directional system are two-fold. First, any refinement that can be inferred is, indeed, inferred. Second, an annotation on an expression can be applied to subexpressions without requiring the programmer to retype the annotation. For example, in the case of the (NSC-T-If) rule, an annotation applied to the entire **if** term can be applied to the **then** and **else** clauses of the term without any additional programmer effort.

To elmininate the second source of non-determinism, that is, to make the subsumption rules syntax-directed, we allow subsumption only when checking the type refinement of an inferable term or expression, in rules (NSC-T-ItoC) and (NSC-E-ItoC). As the inferable term (expression) in the premise provides the sub type-refinement and the checkable term (expression) in the conclusion provides the super type-refinement, all elements of the rule are well specified. Additionally, the restriction of the rule to inferable terms used in a checking judgment ensures that the subsumption rule is syntax-directed.

Figures 19 and 20 give the cut-free and subsumption-free rules for term and expression refinement checking, respectively. The rules for both terms and expressions have been ordered according to our reformulated grammar, with inference rules grouped seperately from checking rules. Figure 19 starts with rules for variables, constants and booleans. As the type refinements of the terms are immediately obvious they are inferred. Notice that the inference judgment assigns the exact type refinement of the term and not a super type-refinement. In rule (NSC-T-TApp), function refinements are now universally quantified and so the value of $\vec{c}$ must be properly "guessed" in order to successfully infer the refinement of the application. This guess may be resolved by standard first-order unification.

Rule (NSC-T-CtoI) provides an intuitive transition between the inference and checking judgments. We can infer that term M has type refinement ϕ, specified by the programmer, if we can check M against ϕ. Furthermore, this rule is central to the bi-directional system as it allows the programmer to specify an annotation at a point where the compiler cannot automatically infer one. Similarly, rule (NSC-T-ItoC) provides a transition between the checking and inference judgments, as a checking rule for inferable terms. A simple such rule might specify that any term M with an inferred type refinement ϕ checks only against ϕ. However, we integrate subsumption into this rule, with the result that M can successfully be checked against any super type-refinement of ϕ. This rule provides a natural place to add subsumption because of the rule's syntax-directedness and because subsumption includes an entailment judgment requiring two inputs, which the inference judgment in the premise and checking judgment of the conclusion conveniently provide. We further note that, as any checkable term can be converted to an inferable one through annotation, subsumption can, in effect, be applied to any term through the insertion of annotations.

In rules (NSC-T-Lam) and (NSC-T-Fun), when checking M_C (E_C), the variables in $\vec{b}$ are considered bound in M_C (E_C) itself, as if the function were written $\Lambda[\vec{b}].\lambda(x{:}A).M_C$ (and correspondingly for **fun**). The binding is implicit to avoid unnecessary programmer effort.

The first rule of Figure 20, (NSC-E-ITerm), demonstrates a simple form of context threading. As all terms are pure - no ephemeral state is changed during their execution - we can infer that the entire ephemeral context is unused. Rule (NSC-E-Op) is similar to term application, except that the operator's type refinement is known from the interface and therefore need not be inferred. Additionally, in both (NSC-E-Op) and (NSC-E-PApp), the emphemeral context is split into a portion used to entail the precondition of the operator and an unused portion, which is inferred as output. In rule (NSC-E-ItoC), the context threading is more complicated, as it happens in two steps. First, the entire context is threaded through the inference judgment. Then, the remaining portion of that context is split, with one portion passed to the entailment judgment and the other inferred to be unused.

A second term rule, (NSC-E-CTerm), is included due to the existence of two classes of terms, M_I and M_C. Note that the instantion of $\vec{b}$ needs to be guessed here as is the case for the operator and function application rules. In rule (NSC-E-Let), the first expression of the **let** must be inferable, because η_1 and Δ_1 are not specified by the type refinement, η_2, in the conclusion of the rule.

Definition 12 (Cleanly Annotated Terms and Expressions)
We define a term M to be cleanly annotated *if $\forall\, M', \alpha,\ M \neq (M' \triangleleft \alpha)$. That is, a cleanly annotated term is one that is unnannotated at its top level. Similarly, we define an expression E to be* cleanly annotated *if $\forall\, E', \alpha,\ E \neq (E' \triangleleft \alpha)$.*

Definition 13 (Simply Annotated Terms and Expressions)
We define a term M to be simply annotated *if it is cleanly annotated or $M = (M' \triangleleft \phi)$ and M' is simply annotated. That is, a simply annotated term is one that has only term annotations at its top level. Similarly, we define an expression E to be* simply annotated *if it is cleanly annotated or $E = (E' \triangleleft \eta)$ and E' is simply annotated.*

Definition 14 (List-Simply Annotated Terms and Expressions)
We define a term M to be list-simply annotated *if it is cleanly annotated or $M = (M' \triangleleft \{\phi_i\}_m)$ and M' is list-simply annotated. That is, a list-simply annotated term is one that is annotated with only lists of refinements, at its top level. Similarly, we define an expression E to be* list-simply annotated *if it is cleanly annotated or $E = (E' \triangleleft \{\eta_i\}_m)$ and E' is list-simply annotated.*

To relate the cut-free syntax to the cut-and-subsumption free syntax, we define an *erasure* function $|E|$ on terms and expressions.

Definition 15 (Erasure)
The erasure function $|E_C|$ removes all annotations from the term or expression E_C. When the top-level expression in E_C does not carry an annotation, the erasure function is applied recursively to the subcomponents of E_C with no other modifications. Otherwise, the erasure function is defined as follows:

$$|E_C \triangleleft \alpha| = |E_C|$$
$$|\mathbf{let}\,\beta, X\,\mathbf{be}\,E_I\,\mathbf{in}\,E_C\,\mathbf{end}| = \mathbf{let}\,X\,\mathbf{be}\,|E_I|\,\mathbf{in}\,|E_C|\,\mathbf{end}$$

To enable the checking of terms and expressions with annotation lists, we define the *slice* function.

Definition 16 (Slice)
The slice function $/E_C/_i$ on terms and expressions recursively extracts a copy of E_C with each annotation list replaced by the i^{th} element of that list. Most notably,

$$\begin{array}{lcll}
/E_C \triangleleft \{\ldots,\alpha_{i-1},\bullet,\alpha_{i+1},\ldots\}/_i & = & /E_C/_i & \\
/E_C \triangleleft \{\alpha_j\}_m/_i & = & /E_C/_i \triangleleft \alpha_i & (i \le m) \\
/\mathbf{let}\,\{\beta_j\}_m, X\,\mathbf{be}\,E_I\,\mathbf{in}\,E_C\,\mathbf{end}/_i & = & \mathbf{let}\,\beta_i, X\,\mathbf{be}\,/E_I/_i\,\mathbf{in}\,/E_C/_i\,\mathbf{end} & (i \le m)
\end{array}$$

We also define an inverse of the slice function: the *join* function $\langle\langle E_1, \ldots, E_n\rangle\rangle$.

Definition 17 (Join)
For a set of expressions $\{E_1, \ldots, E_n\}$, *with* $|E_i| = |E_1|$ *for all* i *from* 1 *to* n, *the join function uses annotation lists to create a unified expression from the individual* E_i. *If none of the* E_i *carry annotations on the top-level expression, then join is defined recursively on the syntax of the expression. If some, or all, of the* E_i *are annotated – that is,* $E_1, \ldots, E_i$ *are annotated and* $E_{i+1}, \ldots, E_n$ *are not and* $1 \leq i \leq n$ *– then join is defined as follows:*

$$\langle\langle E_1 \lhd \alpha_1, \ldots, E_i \lhd \alpha_i, E_{i+1}, \ldots, E_n \rangle\rangle = \langle\langle E_1, \ldots, E_n \rangle\rangle \lhd \{\alpha_1, \ldots, \alpha_i, \overbrace{\bullet, \ldots, \bullet}^{n-i}\} \quad (i \geq 1)$$

Lemma 18 (Properties of Slice and Join)

1. $/\langle\langle E_1, \ldots, E_n \rangle\rangle/_i = E_i$
2. $\forall i,\ 1 \leq i \leq n,\ |\langle\langle E_1, \ldots, E_n \rangle\rangle| = |E_i|$
3. $\forall i,\ 1 \leq i \leq n,\ |/E_C/_i| = |E_C|$
4. *If,* $\forall i,\ 1 \leq i \leq n$, E_i *is cleanly annotated, then* $\langle\langle E_1, \ldots, E_n \rangle\rangle$ *is cleanly annotated.*
5. *If,* $\forall i,\ 1 \leq i \leq n$, E_i *is simply annotated, then* $\langle\langle E_1, \ldots, E_n \rangle\rangle$ *is list-simply annotated.*
6. *If* E_C *is cleanly annotated then* $/E_C/_i$ *is cleanly annotated.*
7. *If* E_C *is list-simply annotated then* $/E_C/_i$ *is simply annotated.*

Proof: Items one and two are proven by induction on the structure of expressions. Item three follows from one and two, items four and five from the definition of the join function, and item six from the definition of the slice function. Item seven is proven by induction on the definition of list-simply annotated terms and expressions. ■

As mentioned in Section 3.3, the rule (R-E-Sub) is the key to local reasoning. In order to eliminate this rule without sacrificing completeness, our system needs to similarly support local reasoning. We show that it does in Lemma 19, below. It states that unneeded ephemeral context is inferred to be unused in both the inference and checking judgments.

Together with the local reasoning lemma, we present a lemma expressing an extensibility property of the context reduction-closure judgment. It is needed in the proof of the local reasoning lemma.

Lemma 19 (Local Reasoning)

- *If* $\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_m$ *then* $\Omega; \Delta', \Delta \rightsquigarrow^* (\Omega_i; \Delta', \Delta_i)_m$
- *If* $\Omega; \Delta \gg^{nsc}_E E \uparrow \eta; \Delta'$ *then* $\Omega; \Delta, \Delta'' \gg^{nsc}_E E \uparrow \eta; \Delta', \Delta''$. *Similarly, if* $\Omega; \Delta \gg^{nsc}_E E \downarrow \eta; \Delta'$ *then* $\Omega; \Delta, \Delta'' \gg^{nsc}_E E \downarrow \eta; \Delta', \Delta''$.

Proof: The proof of item 1 is by inspection of context-reduction and induction on context-reduction closure. The proof of item 2 is by induction on refinement-checking derivations. In the case of rule (NSC-E-Context), we apply item 1 to the context reduction-closure judgment in the premise. ■

The following lemma is needed in the proof of completeness of the NSC system.

Lemma 20
If $\Omega; \Delta \gg^{nsc}_E E_C \lhd m \downarrow \eta'; \Delta', \Delta''$ *and* $\Omega; \Delta''; \eta' \Longrightarrow^{nc}_E \eta$ *then* $\Omega; \Delta \gg^{nsc}_E E_C \lhd \{\overbrace{\eta', \ldots, \eta'}^{m}\} \lhd m \downarrow \eta; \Delta'$.

Proof: By inversion of rule (NSC-E-CONTEXT) we have

$$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_m$$
$$\Omega_i; \Delta_i \gg_E^{nsc} /E_C/_i \downarrow \eta; \Delta', \Delta'' \quad (\text{for } 1 \leq i \leq m).$$

For each i, we apply rule (NSC-E-CTOI) to $/E_C/_i \triangleleft \eta'$. Then, by rule (NSC-E-ITOC) and weakening of our second assumption to $\Omega_i; \Delta''; \eta' \Longrightarrow_E^{nc} \eta$ (based on Lemma 30), we have

$$\Omega_i; \Delta_i \gg_E^{nsc} /E_C/_i \triangleleft \eta' \downarrow \eta; \Delta' \quad (\text{for } 1 \leq i \leq m).$$

Finally, setting

$$E'_C = \langle\langle /E_C/_1 \triangleleft \eta', \ldots, /E_C/_m \triangleleft \eta' \rangle\rangle = E_C \triangleleft \overbrace{\{\eta', \ldots, \eta'\}}^{m}$$

and applying rule (NSC-E-CONTEXT) to E'_C, we obtain our result. ■

We now state and prove the equivalence of the NSC system to the NC system.

Lemma 21 (Soundness of Cut-Free, Subsumption-Free R. C.)

- *If* $\Omega \gg_M^{nsc} M \updownarrow \phi$ *then* $\Omega \gg_M^{nc} |M| : \phi$.
- *If* $\Omega; \Delta \gg_E^{nsc} E \updownarrow \eta'; \Delta'$ *and* $\Omega; \Delta'; \eta' \Longrightarrow_E^{nc} \eta$ *then* $\Omega; \Delta \gg_E^{nc} |E| : \eta$.

Proof: By simultaneous induction on refinement-checking derivations. We detail the more difficult cases below.

- Case NSC-E-ITOC: Suppose $\Omega; \Delta'; \eta \Longrightarrow_E^{nc} \eta''$. From the derivation, we have $\Omega; \Delta''; \eta' \Longrightarrow_E^{nc} \eta$. By Lemma 9 (transitivity), $\Omega; \Delta', \Delta''; \eta' \Longrightarrow_E^{nc} \eta''$. By induction on the derivation of $\Omega; \Delta \gg_E^{nsc} E_I \uparrow \eta'; \Delta', \Delta''$, we obtain our result that $\Omega; \Delta \gg_E^{nc} |E_I| : \eta''$.

- Case NSC-E-CTERM: By induction, $\Omega \gg_M^{nc} |M_C| : [\vec{c}/\vec{b}]\phi$. By rule (NC-E-TERM), $\Omega; \cdot \gg_E^{nc} |M_C| : ([\vec{c}/\vec{b}]\phi, \mathbf{1})$. From the derivation and expression entailment rules, we derive that $\Omega; \Delta; ([\vec{c}/\vec{b}]\phi, \mathbf{1}) \Longrightarrow_E^{nc} \eta$. By rule (NC-E-SUB), $\Omega; \Delta \gg_E^{nc} |M_C| : \eta$. Now, supposing $\Omega; \Delta'; \eta \Longrightarrow_E^{nc} \eta'$ and applying rule (NC-E-SUB) again, we obtain our result that $\Omega; \Delta, \Delta' \gg_E^{nc} |M_C| : \eta'$.

- Case NSC-E-LET: By expression entailment rules we can derive that $\Omega; \Delta_1; \exists[\vec{b}](\phi_1, \psi_1) \Longrightarrow_E^{nc} \exists[\vec{b}](\phi_1, \bigotimes(\Delta_1) \otimes \psi_1)$, where $\bigotimes(\psi_1, \psi_2, \ldots, \psi_n) = \psi_1 \otimes \psi_2 \otimes \cdots \otimes \psi_n$. Then, by induction on the first derivation in the Let rule, we get $\Omega; \Delta \gg_E^{nc} |E_{C1}| : \exists[\vec{b}](\phi_1, \bigotimes(\Delta_1) \otimes \psi_1)$.

  Now, supposing that $\Omega; \Delta_2; \eta_2 \Longrightarrow_E^{nc} \eta$, by induction on the second derivation in the Let rule, we get $\Omega, \vec{b}, x{:}\phi_1; \Delta_1, \psi_1 \gg_E^{nc} |E_{C2}| : \eta$. Next, notice that $\bigotimes(\Delta_1) \otimes \psi_1 \rightsquigarrow^* \Delta_1, \psi_1$. So, by rule (NC-E-CONTEXT), we have $\Omega, \vec{b}, x{:}\phi_1; \bigotimes(\Delta_1) \otimes \psi_1 \gg_E^{nc} |E_{C2}| : \eta$

  Finally, from the above results and rule (NC-E-LET) (omitting the proof of premise $\vec{b} \notin \mathsf{FV_c}(\eta)$) we have $\Omega; \Delta \gg_E^{nc} \mathbf{let}\, x\, \mathbf{be}\, E_1\, \mathbf{in}\, E_2\, \mathbf{end} : \eta$.

■

Lemma 22 (Completeness of Cut-Free, Subsumption-Free R. C.)

- *If* $\Omega \gg_M^{nc} M : \phi$ *then* $\exists\, M_C$ *such that* $|M_C| = M$, M_C *is simply annotated and* $\Omega \gg_M^{nsc} M_C \downarrow \phi$.
- *If* $\Omega; \Delta \gg_E^{nc} E : \eta$ *then* $\exists\, E_C, m$ *such that* $|E_C| = E$, E_C *is list-simply annotated and* $\Omega; \Delta \gg_E^{nsc} E_C \triangleleft m \downarrow \eta; \cdot$.

- *If $\Omega \gg^{nc}_{M} M : \phi$ then $\exists\, M_I, \phi'$ such that $|M_I| = M$ and $\Omega \gg^{nsc}_{M} M_I \uparrow \phi$.*
- *If $\Omega; \Delta \gg^{nc}_{E} E : \eta$ then $\exists\, E_I$ such that $|E_I| = E$ and $\Omega; \Delta \gg^{nsc}_{E} E_I \uparrow \eta; \cdot$.*

Proof: The proof of completeness with respect to the checking judgments (the first two items above) is by simultaneous induction on typing derivations. The most difficult cases are (NC-E-Sub) and (NC-E-Context), which we detail below. The proof of item three follows directly from item one and rule (NSC-T-CtoI). The proof of item four follows directly from item two and rule (NSC-E-CtoI).

- Case NC-E-Context:
$$\frac{\begin{array}{c}\Omega; \Delta \leadsto (\Omega_i; \Delta_i)_n \quad (A) \\ \Omega_i; \Delta_i \gg^{nc}_{E} E : \eta \quad (\text{for } 1 \le i \le n) \quad (B)\end{array}}{\Omega; \Delta \gg^{nc}_{E} E : \eta}$$

  By the induction hypothesis and derivation B, for $1 \le i \le n, \exists\, E_i, m_i$ such that

$$|E_i| = E \tag{1}$$
$$E_i \text{ is list-simply annotated} \tag{2}$$
$$\Omega_i; \Delta_i \gg^{nsc}_{E} E_i \lhd m_i \downarrow \eta; \cdot \tag{3}$$

  By inversion of NSC-E-Context and (3),

$$\Omega_i; \Delta_i \leadsto^* (\Omega_{i,j}; \Delta_{i,j})_{m_i} \tag{4}$$
$$\Omega_{i,j}; \Delta_{i,j} \gg^{nsc}_{E} /E_i/_j \downarrow \eta; \cdot \quad (\text{for } 1 \le j \le m_i) \tag{5}$$

  By CR*-Trans, derivation A and (4),

$$\Omega; \Delta \leadsto^* (\Omega_{1,j}; \Delta_{1,j})_{m_1}, \cdots, (\Omega_{n,j}; \Delta_{n,j})_{m_n} \tag{6}$$

  By NSC-E-Context, (6) and (5),

$$\Omega; \Delta \gg^{nsc}_{E} E_{\text{all}} \lhd m_{\text{all}} \downarrow \eta; \cdot \tag{7}$$
$$\text{where } E_{\text{all}} = \left\langle\!\!\left\langle \begin{matrix} /E_1/_1, & \ldots, & /E_1/_{m_1}, \\ \vdots & & \vdots \\ /E_n/_1, & \ldots, & /E_n/_{m_n} \end{matrix} \right\rangle\!\!\right\rangle$$
$$\text{and } m_{\text{all}} = (m_1 + \cdots + m_n)$$

  By Lemma 18.3,

$$|/E_i/_j| = E \tag{8}$$

  By Lemma 18.2 and (8),

$$|E_{\text{all}}| = E \tag{9}$$

  By Lemma 18.7 and (2),

$$/E_i/_j \text{ is simply annotated} \tag{10}$$

By Lemma 18.5 and (10),

$$E_{\text{all}} \neq (E_C \triangleleft \alpha) \tag{11}$$

Note that items (7), (9) and (11) are the conclusions of this case of the proof.

- Case NC-E-Sub:

$$\frac{\overset{A}{\Omega; \Delta_1 \gg^{nc}_E E : \eta'} \quad \overset{B}{\Omega; \Delta_2; \eta' \Longrightarrow^{nc}_E \eta}}{\Omega; \Delta_1, \Delta_2 \gg^{nc}_E E : \eta}$$

By the induction hypothesis and derivation A, $\exists\, E_C, m$ such that,

$$|E_C| = E \tag{12}$$

$$E_C \text{ is list-simply annotated} \tag{13}$$

$$\Omega; \Delta_1 \gg^{nsc}_E E_C \triangleleft m \downarrow \eta'; \cdot \tag{14}$$

By Lemma 19 and (14),

$$\Omega; \Delta_1, \Delta_2 \gg^{nsc}_E E_C \triangleleft m \downarrow \eta'; \Delta_2 \tag{15}$$

By Lemma 20, (15) and derivation B,

$$\Omega; \Delta_1, \Delta_2 \gg^{nsc}_E E'_C \triangleleft m \downarrow \eta; \cdot \tag{16}$$

$$\text{where } E'_C = E_C \triangleleft \{\eta', \ldots, \eta'\}$$

By the definition of erasure and (12),

$$|E'_C| = |E_C| = E \tag{17}$$

By (13) and the definition of a list-simply annotated expression,

$$E'_C \text{ is list-simply annotated} \tag{18}$$

Note that items (17), (18) and (16) are the conclusions of this case of the proof.

■

We conclude this section with two lemmas about the NSC system that will be useful in proving the refinement inversion lemmas of Section 3.7.

Lemma 23

- *If M_C is simply annotated and $\Omega \gg^{nsc}_M M_C \downarrow \phi$ then $\exists\, M'_C$ such that $|M'_C| = |M_C|$, M'_C is cleanly annotated, $\Omega \gg^{nsc}_M M'_C \downarrow \phi'$ and $\Omega; \phi' \Longrightarrow^{nc}_M \phi$.*
- *If E_C is simply annotated and $\Omega; \Delta \gg^{nsc}_E E_C \downarrow \eta; \cdot$ then $\exists\, E'_C$ such that $|E'_C| = |E_C|$, E'_C is cleanly annotated, $\Omega; \Delta \gg^{nsc}_E E'_C \downarrow \eta'; \Delta'$ and $\Omega; \Delta'; \eta' \Longrightarrow^{nc}_E \eta$.*

Proof: The proof is by simultaneous induction on refinement-checking derivations. If a derivation ends with rule (NSC-T-ItoC) then we argue that either M_I is a cleanly annotated term, in which case the result follows immediately, or $M = (M' \triangleleft \phi)$, in which case, by inversion of rule (NSC-T-CtoI), $\Omega \gg^{nsc}_M M_C \downarrow \phi$. Then, we obtain our result by induction and transitivity of term entailment. We argue similarly for rule (NSC-E-ItoC). ■

Lemma 24 (NSC Substitution)

- *If $\Omega, x{:}\phi' \gg_M^{nsc} M \uparrow \phi$ and $\Omega \gg_M^{nsc} V \uparrow \phi'$ then $\Omega \gg_M^{nsc} [V/x]M \uparrow \phi$.*

  Similarly, if $\Omega, x{:}\phi'; \Delta \gg_E^{nsc} E \uparrow \eta; \Delta'$ and $\Omega \gg_M^{nsc} V \uparrow \phi'$, then $\Omega; \Delta \gg_E^{nsc} [V/x]E \uparrow \eta; \Delta'$.

- *If $\Omega, x{:}\phi' \gg_M^{nsc} M \downarrow \phi$ and $\Omega \gg_M^{nsc} V \uparrow \phi'$ then $\Omega \gg_M^{nsc} [V/x]M \downarrow \phi$.*

  Similarly, if $\Omega, x{:}\phi'; \Delta \gg_E^{nsc} E \downarrow \eta; \Delta'$ and $\Omega \gg_M^{nsc} V \uparrow \phi'$, then $\Omega; \Delta \gg_E^{nsc} [V/x]E \downarrow \eta; \Delta'$.

Proof: The proof is by simultaneous induction on the first refinement-checking derivation of each case in the lemma. ■

3.6 Decidability

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.

To prove our system decidable, we must show that these sources of nondeterminism do not cause the system to be undecidable. Fortunately, 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 (NSC-E-PApp). There are several known approaches to efficient resource management in linear logic [CHP96]. Third, theorem proving in the multiplicative-additive fragment of linear logic (MALL) has been proven decidable [LS94]. However, solving all three of the above problems in the context of our system will be challenging. We believe that further investigation should be done in the setting of a pratical implementation.

3.7 Soundness

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

Condition 25 (Soundness of Primitives)
Suppose

$$\Sigma_\phi(\mathsf{o}) = \forall \vec{b_1} \cdot ((\phi_{1,1}, \ldots, \phi_{1,n}, \psi_1) \to \exists[\vec{b_2}](\phi_2, \psi_2))$$

If $w \vDash \Omega; [\vec{c_1}/\vec{b_1}]\psi_1$, and for $1 \le i \le n$, $\Omega \gg_M c_i' : [\vec{c_1}/\vec{b_1}]\phi_{1,i}$, and $\mathcal{T}(\mathsf{o})(c_1', \ldots, c_n', w + u) = c', w'$ then there exist $\vec{c_2}$ and Ω' 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 $w + u$.

Lemma 26 (Inversion of Term Entailment)
If $\Omega; \phi_1 \Longrightarrow_M \phi_2$ *then*

- $\phi_1 = \textbf{\textit{Bool}}$ *iff* $\phi_2 = \textbf{\textit{Bool}}$
- *if* $\phi_1 = \textbf{\textit{a}}$ *then* $\phi_2 = \textbf{\textit{a}}$
- *if* $\phi_2 = \textbf{\textit{Its}}(c)$ *then* $\phi_1 = \textbf{\textit{Its}}(c)$
- *if* $\phi_1 = \textbf{\textit{Its}}(c)$ *then either*
  - $\phi_2 = \textbf{\textit{Its}}(c)$*, or*
  - $\phi_2 = \textbf{\textit{a}}$ *and either* $\Sigma_A(c) = \mathbf{a}$ *or* $\Omega(c) = \mathbf{a}$.
- $\phi_1 = \forall \vec{b}_1 \cdot \phi_1' \to \phi_1''$ *iff* $\phi_2 = \forall \vec{b}_2 \cdot \phi_2' \to \phi_2''$
- *if* $\phi_1 = \forall \vec{b}_1 \cdot \phi_1' \to \phi_1''$ *or* $\phi_2 = \forall \vec{b}_2 \cdot \phi_2' \to \phi_2''$ *then all of the following hold:*
  - $\Omega, \vec{b}_2; \phi_2' \Longrightarrow_M [\vec{c}_1/\vec{b}_1]\phi_1'$
  - $\Omega, \vec{b}_2; [\vec{c}_1/\vec{b}_1]\phi_1'' \Longrightarrow_M \phi_2''$
  - $\Omega_b, \vec{b}_2 \vdash \phi_2' \ \mathsf{ok}$
- $\phi_1 = \forall \vec{b}_1 \cdot (\phi, \psi) \rightharpoonup \eta$ *iff* $\phi_2 = \forall \vec{b}_2 \cdot (\phi', \psi') \rightharpoonup \eta'$
- *if* $\phi_1 = \forall \vec{b}_1 \cdot (\phi, \psi) \rightharpoonup \eta$ *or* $\phi_2 = \forall \vec{b}_2 \cdot (\phi', \psi') \rightharpoonup \eta'$ *then all of the following hold:*
  - $\Omega, \vec{b}_2; \phi' \Longrightarrow_M [\vec{c}_1/\vec{b}_1]\phi$
  - $\Omega, \vec{b}_2; \psi' \Longrightarrow_W [\vec{c}_1/\vec{b}_1]\psi$
  - $\Omega, \vec{b}_2; \cdot; [\vec{c}_1/\vec{b}_1]\eta \Longrightarrow_E \eta'$
  - $\Omega_b, \vec{b}_2 \vdash \phi' \ \mathsf{ok}$
  - $\Omega_b, \vec{b}_2 \vdash \psi' \ \mathsf{ok}$

Proof: By induction on the term entailment rules. ■

Lemma 27 (Inversion of Expression Entailment)
If $\Omega; \Delta; \eta_1 \Longrightarrow_E \eta_2$ *then*

- $\eta_1 = \exists[\vec{b}_1](\phi_1, \psi_1)$
- $\eta_2 = \exists[\vec{b}_2](\phi_2, \psi_2)$
- $\Omega, \vec{b}_1; \phi_1 \Longrightarrow_M [\vec{c}_2/\vec{b}_2]\phi_2$
- $\Omega, \vec{b}_1; \Delta, \psi \Longrightarrow_W [\vec{c}_2/\vec{b}_2]\psi_2$

Proof: By inspection of expression entailment rules. ■

Lemma 28 (Inversion of Term Refinement)
If $\Omega \gg_M M : \phi$ *then*

- *if* $M = x$ *then*
  - $\Omega(x) = \phi'$ *and*
  - $\Omega; \phi' \Longrightarrow_M \phi$.
- *if* $M = c$ *then*
  - $\phi = \boldsymbol{Its}(c)$ *or*
  - $\phi = \boldsymbol{a}$ *and either* $\Sigma_A(c) = \mathbf{a}$ *or* $\Omega(c) = \mathbf{a}$.
- *if* $M =$ ***true*** *then* $\phi = \boldsymbol{Bool}$.
- *if* $M =$ ***false*** *then* $\phi = \boldsymbol{Bool}$.
- *if* $M = \lambda(x{:}A).M$ *then*
  - $\phi = \forall\vec{b} \cdot \phi_1 \rightarrow \phi_2$,
  - $\Omega_b, \vec{b} \vdash \phi_1$ ok,
  - $\Omega_b, \vec{b} \vdash \phi_1 \sqsubseteq A$ *and*
  - $\Omega, \vec{b}, x{:}\phi_1 \gg_M M : \phi_2$.
- *if* $M =$ **fun** $x\ (x_1{:}A_1) : A$ **is** E *then*
  - $\phi' = \forall\vec{b}' \cdot (\phi'_1, \psi'_1) \rightharpoonup \eta'$,
  - $\Omega_b \vdash \phi' \sqsubseteq A_1 \rightharpoonup A$,
  - $\Omega, x{:}\phi', \vec{b}', x_1{:}\phi'_1; \psi'_1 \gg_E E : \eta'$,
  - $\Omega_b \vdash \phi'$ ok,
  - $\phi = \forall\vec{b} \cdot (\phi_1, \psi_1) \rightharpoonup \eta$,
  - $\Omega, x{:}\phi', \vec{b}, x_1{:}\phi_1; \psi_1 \gg_E E : \eta$ *and*
  - $\Omega; \phi' \Longrightarrow_M \phi$
- *if* $M =$ ***if*** M' ***then*** M_1 ***else*** M_2 *then*
  - $\Omega \gg_M M_1 : \phi$ *and*
  - $\Omega \gg_M M_2 : \phi$.
- *if* $M = M_1\,(M_2)$ *then*
  - $\Omega \gg_M M_1 : \phi' \rightarrow \phi$ *and*
  - $\Omega \gg_M M_2 : \phi'$.

Proof: The proof follows from the soundness and completeness of the bi-directional refinement checking rules, using lemmas 23 and 24 as needed. ∎

Lemma 29 (Inversion of Expression Refinement)

- *If* $\Omega; \Delta \gg_E M : \eta$, *then*
  - $\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n$

- *for all* $1 \leq i \leq n$,
  - $\Omega_i \gg_M M : \phi$
  - $\Omega_i ; \Delta_i ; (\phi, 1) \Longrightarrow_E \eta$

- *If* $\Omega ; \Delta \gg_E \mathsf{o}(M_1, \ldots, M_k) : \eta$, *then*
  - $\Omega ; \Delta \rightsquigarrow^* (\Omega_i ; \Delta_i)_n$
  - $\Sigma_\phi(\mathsf{o}) = \forall \vec{b} \cdot (\phi_1, \ldots, \phi_k, \psi) \rightharpoonup \eta'$
  - *for all* $1 \leq i \leq n$, $\exists\, \vec{c}_i$, *such that*
    - $\Delta_i = \Delta_{i,1}, \Delta_{i,2}$
    - $\Omega_i ; \Delta_{i,1} \Longrightarrow_W [\vec{c}_i / \vec{b}]\psi$
    - *for each* $1 \leq j \leq k$, $\Omega_i \gg_M M_j : [\vec{c}_i / \vec{b}]\phi_j$
    - $\Omega_i ; \Delta_{i,2} ; [\vec{c}_i / \vec{b}]\eta' \Longrightarrow_E \eta$

- *If* $\Omega ; \Delta \gg_E \mathbf{let}\, x\, \mathbf{be}\, E_1\, \mathbf{in}\, E_2\, \mathbf{end} : \eta$, *then*
  - $\Omega ; \Delta \rightsquigarrow^* (\Omega_i ; \Delta_i)_n$
  - *for all* $1 \leq i \leq n$,
    - $\Omega_i ; \Delta_i \gg_E E_1 : \exists [\vec{b}_i](\phi_i, \psi_i)$
    - $\Omega_i, \vec{b}_i, x{:}\phi_i ; \psi_i \gg_E E_2 : \eta$

- *If* $\Omega ; \Delta \gg_E \mathbf{app}(M, M_1) : \eta$, *then*
  - $\Omega ; \Delta \rightsquigarrow^* (\Omega_i ; \Delta_i)_n$
  - *for all* $1 \leq i \leq n$, $\exists\, \vec{c}_i$, *such that*
    - $\Delta_i = \Delta_{i,1}, \Delta_{i,2}$
    - $\Omega_i \gg_M M : \forall \vec{b}_{1,i} \cdot (\phi_{1,i}, \psi_{1,i}) \rightharpoonup \eta_i$
    - $\Omega_i ; \Delta_{i,1} \Longrightarrow_W [\vec{c}_i / \vec{b}_i]\psi_{1,i}$
    - $\Omega_i \gg_M M_1 : [\vec{c}_i / \vec{b}_i]\phi_{1,i}$
    - $\Omega_i ; \Delta_{i,2} ; [\vec{c}_i / \vec{b}_i]\eta_i \Longrightarrow_E \eta$

- *If* $\Omega ; \Delta \gg_E \boldsymbol{if} M\, \boldsymbol{then}\, E_1\, \boldsymbol{else}\, E_2 : \eta$, *then*
  - $\Omega ; \Delta \gg_E E_1 : \eta$
  - $\Omega ; \Delta \gg_E E_2 : \eta$

Proof: The proof follows from the soundness and completeness of the bi-directional refinement checking rules, using lemmas 23 and 24 as needed. ■

Lemma 30
If $\Omega ; \Delta \rightsquigarrow^* (\Omega_i ; \Delta_i)_n$ *then* $\forall i, 1 \leq i \leq n, \Omega \subseteq \Omega_i$.

Proof: By induction on the $\rightsquigarrow^*$ relation. For the base case, notice that only rule (CR-!) changes Ω, growing it by one predicate. ■

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

Theorem 31 (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 $\vec{c}$ and Ω' such that $\Omega' \gg_M V : [\vec{c}/\vec{b}]\phi$, $w' = w'' + u$, $\Omega \subseteq \Omega'$ and $w'' \vDash \Omega'; [\vec{c}/\vec{b}]\psi$.*

Proof: The proof is by simultaneous induction on evaluation. We consider below those cases covered by the second induction hypothesis.

1. Case Term:

   Suppose that:

$$\Omega; \Delta \gg_E M : \eta \tag{19}$$
$$w \vDash \Omega; \Delta \tag{20}$$
$$M \,@\, w + u \Downarrow V \,@\, w' \tag{21}$$
$$\text{where } \eta = \exists[\vec{b}](\phi, \psi)$$

   By inversion of expression r.c.-checking (Lemma 29) and (19),

$$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n \tag{22}$$
$$\text{for all i, } 1 \leq i \leq n,$$
$$\Omega_i \gg_M M : \phi' \tag{23}$$
$$\Omega_i; \Delta_i; (\phi', \mathbf{1}) \Longrightarrow_E \eta \tag{24}$$

   By inversion of evaluation and (21),

$$M \Downarrow V \tag{25}$$
$$w' = w + u \tag{26}$$

   By soundness of logical judgments (Lemma 7.1), (20) and (22),

$$\exists\, i.\, w \vDash \Omega_i; \Delta_i \tag{27}$$

   By definition of $\vDash$ and (27),

$$w \vDash_\Omega \Omega_i \tag{28}$$
$$w \vDash_\Delta \Delta_i, \mathbf{1} \tag{29}$$

   By inversion of expression entailment (Lemma 27) and (24),

$$\Omega_i; \phi' \Longrightarrow_M [\vec{c}/\vec{b}]\phi \tag{30}$$
$$\Omega_i; \Delta_i, \mathbf{1} \Longrightarrow_W [\vec{c}/\vec{b}]\psi \tag{31}$$

   By induction hypothesis 1, (23) and (25),

$$\Omega_i \gg_M V : \phi' \tag{32}$$

By (R-T-SUB), (32) and (30),

$$\Omega_i \gg_M V : [\vec{c}/\vec{b}]\phi \tag{33}$$

By Lemma 7.2, (29) and (31),

$$w \vDash [\vec{c}/\vec{b}]\psi \tag{34}$$

By definition of $\vDash$, (28) and (34),

$$w \vDash \Omega_i; [\vec{c}/\vec{b}]\psi \tag{35}$$

By Lemma 30 and (22),

$$\Omega \subseteq \Omega_i \tag{36}$$

Note that items (26), (33), (35) and (36) are the conclusions of this case of the proof.

2. Case Operator:

Suppose that:

$$\Omega; \Delta \gg_E \mathsf{o}(M_1, \ldots, M_k) : \eta \tag{37}$$
$$w \vDash \Omega; \Delta \tag{38}$$
$$\mathsf{o}(M_1, \ldots, M_k) \mathbin{@} w + u \Downarrow V \mathbin{@} w' \tag{39}$$
$$\text{where } \eta = \exists[\vec{b}](\phi, \psi)$$

By inversion of evaluation,

$$M_j \Downarrow c_j, \forall\, j, 1 \leq j \leq k \tag{40}$$
$$\mathcal{T}(\mathsf{o})(c_1, \ldots, c_k, w + u) = c, w' \tag{41}$$
$$V = c \tag{42}$$

By inversion of expression r.c.-checking (Lemma 29) and (37),

$$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n \tag{43}$$
$$\Sigma_\phi(\mathsf{o}) = \forall \vec{b}_1 \cdot (\phi_{1,1}, \ldots, \phi_{1,k}, \psi_1) \rightharpoonup \eta' \tag{44}$$
$$\text{for all i, } 1 \leq i \leq n,\ \exists\, \vec{c}_{1,i}, \text{such that:}$$
$$\Delta_i = \Delta_{i,1}, \Delta_{i,2} \tag{45}$$
$$\Omega_i; \Delta_{i,1} \Longrightarrow_W [\vec{c}_{1,i}/\vec{b}_1]\psi_1 \tag{46}$$
$$\Omega_i \gg_M M_j : [\vec{c}_{1,i}/\vec{b}_1]\phi_{1,j}, \forall\, j, 1 \leq j \leq k \tag{47}$$
$$\Omega_i; \Delta_{i,2}; [\vec{c}_{1,i}/\vec{b}_1]\eta' \Longrightarrow_E \eta \tag{48}$$
$$\text{where } \eta' = \exists[\vec{b}_2](\phi_2, \psi_2)$$

By soundness of logical judgments (Lemma 7.1), (38) and (43),

$$\exists\, i.\, w \vDash \Omega_i; \Delta_i \tag{49}$$

By definition of $\vDash$ and (49),

$$\begin{aligned}
&w \vDash_{\Omega} \Omega_i && (50)\\
&w \vDash_{\Delta} \Delta_{i,1}, \Delta_{i,2} && (51)\\
&\exists\, w_{i,1}, w_{i.2}\,\text{such that:} && (52)\\
&\quad w = w_{i,1} + w_{i,2} && (53)\\
&\quad w_{i,1} \vDash \Delta_{i,1} &&\\
&\quad w_{i,2} \vDash \Delta_{i,2} &&\\
&\text{and, WLOG:} &&\\
&\quad w_{i,1} \vDash \Omega_i &&\\
&\quad w_{i,2} \vDash \Omega_i &&
\end{aligned}$$

By above and definition of $\vDash$,

$$\begin{aligned}
&w_{i,1} \vDash \Omega_i; \Delta_{i,1} && (54)\\
&w_{i,2} \vDash \Omega_i; \Delta_{i,2} && (55)
\end{aligned}$$

By soundness of logical judgments (Lemma 7.2), (54) and (46),

$$w_{i,1} \vDash [\vec{c}_{1,i}/\vec{b}_1]\psi_1 \qquad (56)$$

By induction hypothesis 1, (47) and (40),

$$\Omega_i \gg_M c_j : [\vec{c}_{1,i}/\vec{b}_1]\phi_{1,j}, \forall\, j, 1 \leq j \leq k \qquad (57)$$

By (53) and associativity of the + operation,

$$w + u = w_{i,1} + w_{i,2} + u = w_{i,1} + (w_{i,2} + u) \qquad (58)$$

By the soundness of primitives (Condition 25),(56), (57), (58), and (41),

$$\begin{aligned}
&\exists\, \vec{c}_{2,i},\ \Omega',\ \text{such that:} &&\\
&\quad w' = w'' + (w_{i,2} + u) && (59)\\
&\quad \Omega' \gg_M c : [\vec{c}_{2,i}/\vec{b}_2][\vec{c}_{1,i}/\vec{b}_1]\phi_2 && (60)\\
&\quad w'' \vDash \Omega'; [\vec{c}_{2,i}/\vec{b}_2][\vec{c}_{1,i}/\vec{b}_1]\psi_2 && (61)\\
&\quad \Omega_i \subseteq \Omega' && (62)
\end{aligned}$$

By inversion of expression entailment (Lemma 27), and (48),

$$\begin{aligned}
&\Omega_i, \vec{b}_2; [\vec{c}_{1,i}/\vec{b}_1]\phi_2 \Longrightarrow_M [\vec{c}/\vec{b}]\phi\\
&\Omega_i, \vec{b}_2; \Delta_{i,2}; [\vec{c}_{1,i}/\vec{b}_1]\psi_2 \Longrightarrow_W [\vec{c}/\vec{b}]\psi
\end{aligned}$$

By substitution (Lemma 8), weakening, (62) and above,

$$\begin{aligned}
&\Omega'; [\vec{c}_{2,i}/\vec{b}_2][\vec{c}_{1,i}/\vec{b}_1]\phi_2 \Longrightarrow_M [\vec{c}/\vec{b}]\phi && (63)\\
&\Omega'; \Delta_{i,2}; [\vec{c}_{2,i}/\vec{b}_2][\vec{c}_{1,i}/\vec{b}_1]\psi_2 \Longrightarrow_W [\vec{c}/\vec{b}]\psi && (64)
\end{aligned}$$

By (R-T-Sub), (60) and (63),

$$\Omega' \gg_M c : [\vec{c}/\vec{b}]\phi \tag{65}$$

By (59) and associativity of the + operation,

$$w' = (w'' + w_{i,2}) + u \tag{66}$$

By Lemma 30 and (43),

$$\Omega \subseteq \Omega_i \tag{67}$$

By above and (62),

$$\Omega \subseteq \Omega' \tag{68}$$

By definition of $\vDash$, (55) and (61),

$$w'' + w_{i,2} \vDash \Omega'; \Delta_{i,2}, [\vec{c}_{2,i}/\vec{b}_2][\vec{c}_{1,i}/\vec{b}_1]\psi_2 \tag{69}$$

By soundness of logical judgment (Lemma 7.2), (69), (64),

$$w'' + w_{i,2} \vDash \Omega'; [\vec{c}/\vec{b}]\psi \tag{70}$$

Note that items (65), (66), (68) and (70) are the conclusions of this case of the proof.

3. Case App:

Suppose that:

$$\Omega; \Delta \gg_E \mathbf{app}(M_1, M_2) : \eta \tag{71}$$
$$w \vDash \Omega; \Delta \tag{72}$$
$$\mathbf{app}(M_1, M_2) \mathbin{@} w + u \Downarrow V \mathbin{@} w' \tag{73}$$
$$\text{where } \eta = \exists[\vec{b}](\phi, \psi)$$

By inversion of evaluation,

$$M_1 \Downarrow V_1 \tag{74}$$
$$M_2 \Downarrow V_2 \tag{75}$$
$$[V_1/x][V_2/x_1]E \mathbin{@} w + u \Downarrow V' \mathbin{@} w' \tag{76}$$
$$\text{where } V_1 = \mathbf{fun}\ x\ (x_1{:}A_1) : A\ \mathbf{is}\ E$$

By inversion of expression r.c.-checking (Lemma 29) and (71),

$$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n \tag{77}$$
$$\text{for all i, } 1 \le i \le n,\ \exists\, \vec{c}_i, \text{such that:}$$
$$\Delta_i = \Delta_{i,1}, \Delta_{i,2} \tag{78}$$
$$\Omega_i \gg_M M_1 : \forall \vec{b}_i \cdot (\phi_{1,i}, \psi_{1,i}) \rightharpoonup \eta_i \tag{79}$$
$$\Omega_i; \Delta_{i,1} \Longrightarrow_W [\vec{c}_i/\vec{b}_i]\psi_{1,i} \tag{80}$$
$$\Omega_i \gg_M M_2 : [\vec{c}_i/\vec{b}_i]\phi_{1,i} \tag{81}$$
$$\Omega_i; \Delta_{i,2}; [\vec{c}_i/\vec{b}_i]\eta_i \Longrightarrow_E \eta \tag{82}$$
$$\text{where } \eta_i = \exists[\vec{b}_{2,i}](\phi_{2,i}, \psi_{2,i})$$

By induction hypothesis 1, (79) and (74),

$$\Omega_i \gg_M V_1 : \forall\vec{b}_i \cdot (\phi_{1,i}, \psi_{1,i}) \rightharpoonup \eta_i \tag{83}$$

By induction hypothesis 1, (81) and (75),

$$\Omega_i \gg_M V_2 : [\vec{c}_i/\vec{b}_i]\phi_{1,i} \tag{84}$$

By inversion of term refinement (Lemma 28) and (83),

$$\phi'_i = \forall\vec{b}'_i \cdot (\phi'_{1,i}, \psi'_{1,i}) \rightharpoonup \eta'_i \tag{85}$$
$$\Omega_{ib} \vdash \phi'_i \sqsubseteq A_1 \rightharpoonup A \tag{86}$$
$$\Omega_{ib} \vdash \phi'_i \ \mathsf{ok} \tag{87}$$
$$\Omega_i, \vec{b}'_i, x{:}\phi'_i, x_1{:}\phi'_{1,i}; \psi'_{1,i} \gg_E E : \eta'_i \tag{88}$$
$$\Omega_i; \phi'_i \Longrightarrow_M \forall\vec{b}_i \cdot (\phi_{1,i}, \psi_{1,i}) \rightharpoonup \eta_i \tag{89}$$
$$\Omega_i, \vec{b}_i, x : \phi'_i, x_1 : \phi_{1,i}; \psi_{1,i} \gg_E E : \eta_i \tag{90}$$

By substitution lemma and (90),

$$\Omega_i, x{:}\phi'_i, x_1{:}[\vec{c}_i/\vec{b}_i]\phi_{1,i}; [\vec{c}_i/\vec{b}_i]\psi_{1,i} \gg_E E : [\vec{c}_i/\vec{b}_i]\eta_i \tag{91}$$

By (R-T-Fun) and (85)-(88),

$$\Omega_i \gg_M V_1 : \phi'_i \tag{92}$$

By substitution lemma, (92), (84) and (91),

$$\Omega_i; [\vec{c}_i/\vec{b}_i]\psi_{1,i} \gg_E [V_1/x][V_2/x_1]E : [\vec{c}_i/\vec{b}_i]\eta_i \tag{93}$$

By soundness of logical judgments (Lemma 7.1), (72) and (77),

$$\exists\, i.\, w \vDash \Omega_i; \Delta_i \tag{94}$$

By definition of $\vDash$, (78) and (94),

$$w \vDash_\Omega \Omega_i \tag{95}$$
$$w \vDash_\Delta \Delta_{i,1}, \Delta_{i,2} \tag{96}$$
$$\exists\, w_{i,1}, w_{i.2} \text{ such that:} \tag{97}$$
$$\quad w = w_{i,1} + w_{i,2} \tag{98}$$
$$\quad w_{i,1} \vDash \Delta_{i,1}$$
$$\quad w_{i,2} \vDash \Delta_{i,2}$$
and, WLOG:
$$\quad w_{i,1} \vDash \Omega_i$$
$$\quad w_{i,2} \vDash \Omega_i$$

By above and definition of $\vDash$,

$$w_{i,1} \vDash \Omega_i; \Delta_{i,1} \tag{99}$$
$$w_{i,2} \vDash \Omega_i; \Delta_{i,2} \tag{100}$$

By soundness of logical judgments lemma (Lemma 7.2), (99) and (80),

$$w_{i,1} \vDash \Omega_i; [\vec{c}_i/\vec{b}_i]\psi_1 \tag{101}$$

By (98) and associativity of the + operation,

$$w + u = w_{i,1} + w_{i,2} + u = w_{i,1} + (w_{i,2} + u) \tag{102}$$

By induction hypothesis 2, (93),(101), (102), and (76), $\exists\, \vec{c}_i$, Ω', such that:,

$$w' = w'' + (w_{i,2} + u) \tag{103}$$
$$\Omega' \gg_M V' : [\vec{c}_{2,i}/\vec{b}_{2,i}][\vec{c}_i/\vec{b}_i]\phi_{2,i} \tag{104}$$
$$w'' \vDash \Omega'; [\vec{c}_{2,i}/\vec{b}_{2,i}][\vec{c}_i/\vec{b}_i]\psi_2 \tag{105}$$
$$\Omega_i \subseteq \Omega' \tag{106}$$

By inversion of expression entailment, Lemma 27, and (82),

$$\Omega_i, \vec{b}_{2,i}; [\vec{c}_i/\vec{b}_i]\phi_{2,i} \Longrightarrow_M [\vec{c}/\vec{b}]\phi$$
$$\Omega_i, \vec{b}_{2,i}; \Delta_{i,2}; [\vec{c}_i/\vec{b}_i]\psi_{2,i} \Longrightarrow_W [\vec{c}/\vec{b}]\psi$$

By substitution, Lemma 8, weakening, and above,

$$\Omega'; [\vec{c}_{2,i}/\vec{b}_{2,i}][\vec{c}_i/\vec{b}_i]\phi_{2,i} \Longrightarrow_M [\vec{c}/\vec{b}]\phi \tag{107}$$
$$\Omega'; \Delta_{i,2}; [\vec{c}_{2,i}/\vec{b}_{2,i}][\vec{c}_i/\vec{b}_i]\psi_{2,i} \Longrightarrow_W [\vec{c}/\vec{b}]\psi \tag{108}$$

By (R-T-Sub), (104) and (107),

$$\Omega' \gg_M V' : [\vec{c}/\vec{b}]\phi \tag{109}$$

By (103) and associativity of the + operation,

$$w' = (w'' + w_{i,2}) + u \tag{110}$$

By Lemma 30 and (77),

$$\Omega \subseteq \Omega_i \tag{111}$$

By above and (106),

$$\Omega \subseteq \Omega' \tag{112}$$

By definition of $\vDash$, (100) and (105),

$$w'' + w_{i,2} \vDash \Omega'; \Delta_{i,2}, [\vec{c}_{2,i}/\vec{b}_{2,i}][\vec{c}_i/\vec{b}_i]\psi_{2,i} \tag{113}$$

By soundness of logical judgments lemma (Lemma 7.2), (113), (108),

$$w'' + w_{i,2} \vDash \Omega'; [\vec{c}/\vec{b}]\psi \tag{114}$$

Note that items (109), (110), (112) and (114) are the conclusions of this case of the proof.

4. Case Let:

   Suppose that:

$$\Omega; \Delta \gg_E \textbf{let}\, x\, \textbf{be}\, E_1\, \textbf{in}\, E_2\, \textbf{end} : \eta \tag{115}$$
$$w \vDash \Omega; \Delta \tag{116}$$
$$\textbf{let}\, x\, \textbf{be}\, E_1\, \textbf{in}\, E_2\, \textbf{end} \,@\, w + u \Downarrow V \,@\, w' \tag{117}$$
$$\text{where } \eta = \exists[\vec{b}](\phi, \psi)$$

   By inversion of expression refinement-checking (Lemma 29) and (115),

$$\Omega; \Delta \rightsquigarrow^* (\Omega_i; \Delta_i)_n \tag{118}$$
$$\text{for all i, } 1 \leq i \leq n,$$
$$\Omega_i; \Delta_i \gg_E E_1 : \exists[\vec{b}_i](\phi_i, \psi_i) \tag{119}$$
$$\Omega_i, \vec{b}_i, x{:}\phi_i; \psi_i \gg_E E_2 : \eta \tag{120}$$
$$\vec{b}_i \notin \mathsf{FV_c}(\eta) \tag{121}$$

   By inversion of evaluation and (117),

$$E_1 \,@\, w + u \Downarrow V_1 \,@\, w_1' \tag{122}$$
$$[V_1/x]E_2 \,@\, w_1' \Downarrow V \,@\, w' \tag{123}$$

   By soundness of logical judgments (Lemma 7.1), (116) and (118),

$$\exists\, i.\, w \vDash \Omega_i; \Delta_i \tag{124}$$

   By induction hypothesis 2, (119), (124) and (122), $\exists\, \vec{c}_i$, Ω', such that:

$$w_1' = w_1'' + u \tag{125}$$
$$\Omega' \gg_M V_1 : [\vec{c}_i/\vec{b}_i]\phi_i \tag{126}$$
$$w_1'' \vDash \Omega'; [\vec{c}_i/\vec{b}_i]\psi_i \tag{127}$$
$$\Omega_i \subseteq \Omega' \tag{128}$$

   By weakening and (120),

$$\Omega', \vec{b}_i, x{:}\phi_i; \psi_i \gg_E E_2 : \eta \tag{129}$$

   By refinement substitution (Lemma 8),(129), and (121),

$$\Omega', x{:}[\vec{c}_i/\vec{b}_i]\phi_i; [\vec{c}_i/\vec{b}_i]\psi_i \gg_E E_2 : \eta$$
$$\Omega'; [\vec{c}_i/\vec{b}_i]\psi_i \gg_E [V_1/x]E_2 : \eta \tag{130}$$

By (123) and (125),

$$[V_1/x]E_2 \mathbin{@} w''_1 + u \Downarrow V \mathbin{@} w' \tag{131}$$

By induction hypothesis 2, (130), (127) and (131), $\exists\, \vec{c}$, Ω'', such that:

$$w' = w'' + u \tag{132}$$
$$\Omega'' \gg_M V : [\vec{c}/\vec{b}]\phi \tag{133}$$
$$w'' \vDash \Omega''; [\vec{c}/\vec{b}]\psi \tag{134}$$
$$\Omega' \subseteq \Omega'' \tag{135}$$

By Lemma 30, (118), (128) and (135),

$$\Omega \subseteq \Omega'' \tag{136}$$

Note that items (132), (133), (134) and (136) are the conclusions of this case of the proof.

5. Case If:

   Suppose that:

$$\Omega; \Delta \gg_E \textbf{if}\, M\, \textbf{then}\, E_1\, \textbf{else}\, E_2 : \eta \tag{137}$$
$$w \vDash \Omega; \Delta \tag{138}$$
$$\textbf{if}\, M\, \textbf{then}\, E_1\, \textbf{else}\, E_2 \mathbin{@} w + u \Downarrow V \mathbin{@} w' \tag{139}$$
$$\text{where } \eta = \exists[\vec{b}](\phi, \psi)$$

   By inversion of expression r.c.-checking (Lemma 29) and (137),

$$\Omega; \Delta \gg_E E_1 : \eta \tag{140}$$
$$\Omega; \Delta \gg_E E_2 : \eta \tag{141}$$

   By inversion of evaluation we have either,

$$M \Downarrow \textbf{true} \tag{142}$$
$$E_1 \mathbin{@} w + u \Downarrow V \mathbin{@} w' \tag{143}$$

   or

$$M \Downarrow \textbf{false} \tag{144}$$
$$E_2 \mathbin{@} w + u \Downarrow V \mathbin{@} w' \tag{145}$$

   In either case of evaluation the result follows immediately by the induction hypothesis.

■

The following canonical refinement forms theorem expresses the properties of values that refinement checking provides *in addition to* ordinary type checking. We do not state properties of values implied by the conventional canonical forms lemma, except where necessary.

Theorem 32 (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. $\phi = \boldsymbol{Bool}$
2. $\phi = \boldsymbol{Its}(c)$ *and* $V = c$
3. $\phi = \boldsymbol{a}$
4. $\phi = \forall\vec{b} \cdot \phi_1 \rightarrow \phi_2$, $V = \lambda(x_1{:}A_1).M$ *and* $\Omega, \vec{b}, x_1{:}\phi_1 \gg_M M : \phi_2$.
5. $\phi = \forall\vec{b} \cdot (\phi_1, \psi_1) \rightharpoonup \eta$, $V =$ **fun** $x\ (x_1{:}A_1) : A_2$ **is** E, *and* $\Omega, x{:}\phi', \vec{b}, x_1{:}\phi_1; \psi_1 \gg_E E : \eta$, *where* $\Omega; \phi' \Longrightarrow_M \phi$.

3.8 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 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 $x{:}A \in \mathsf{type}(\Omega)$.

Theorem 33
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 refinement associated with A. That is, any well-typed term (expression) can also be shown to be well-refined with a trivial refinement. 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 34
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 $\mathsf{triv}_\Gamma(\Gamma); \top \gg_E E : (\mathsf{triv}(A), \top)$.

Proof: The proof is by induction on the typing derivation. ■

3.9 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.

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

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 35 (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 $E \mathbin{@} w \hat{\Downarrow} V \mathbin{@} w'$.

Proof: ($\Leftarrow$) The proof in this direction is trivial. The reduction relation on optimized programs is a subset of the relation on unoptimized programs.

($\Rightarrow$) The proof in this direction is by induction on the evaluation relation. Only the case for operators is difficult. It uses the inversion of refinement checking and the Preservation Theorem. ■

4 Examples

In this section, we provide a number of examples that demonstrate the expressive power of our language.

4.1 ML-style integer references

In Section 2.2 we discussed how to extend the semantics of our language with mutable integer references. Our first example demonstrates how to simulate the (limited) information one gets from the ML type system for reasoning about references.

Recall that in our basic model of integer references, there are two predicates $\mathsf{alloc}(\ell)$ and $\mathsf{ctns}(\ell, i)$. For each location that has been allocated, the world's persistent facts include a predicate $\mathsf{alloc}(\ell)$ and the world's ephemeral facts include the predicate $\mathsf{ctns}(\ell, i)$.

In ML, the only information we have about references is that a reference that appears in our program has been allocated at some point in the past (and once allocated, references remain allocated forever since there is no free operation in ML). We know nothing about the contents of the reference. In order for access to a reference to be safe, we require proof that the reference in question has been allocated. In ML, such proof is implicit in the way the language is structured. Constants with reference type may not explicitly appear in user-defined programs. However, such facts can only be established through meta-theoretic reasoning about ML's semantics.

Here, we make the proof that a reference has been allocated explicit within the language by refining the type of the allocation primitive.

$$\mathbf{new} \quad : \quad (\mathbf{int}, \top) \rightharpoonup \exists[\ell{:}\mathbf{int\,ref}](\boldsymbol{Its}(\ell), !\mathsf{alloc}(\ell) \otimes \top)$$

The **new** operation has no particular requirements before it can be invoked. It operates on the whole world w, choosing a new location ℓ that does not appear in $\mathsf{Per}(w)$ and returns the extended world $w + (\{\mathsf{alloc}(\ell)\}, \{\mathsf{ctns}(\ell, i)\})$. Since we do not care about the specifics of w, we allowed the precondition for **new** to be $\top$. Since we later want to establish explicitly the fact that any reference that we read or update is allocated, the postcondition includes the formula $!\mathsf{alloc}(\ell)$. The modality

"!" indicates that this is a persistent fact that may be used as many times as necessary (or not at all). The second part of the postcondition is again $\top$: We say nothing about the contents of the location or any other part of the world.

The **get** and **set** operations only care that the location that they access has been allocated. They make no use of the contents of the reference or any other part of the world. As a result, the precondition for the operations specify $!\mathsf{alloc}(\ell) \otimes \top$.

$$\begin{array}{lll}\textbf{get} & : & \forall \ell{:}\textbf{int ref} \cdot (\textbf{\textit{Its}}(\ell), !\mathsf{alloc}(\ell) \otimes \top) \rightharpoonup (\textbf{int}, \top) \\ \textbf{set} & : & \forall \ell{:}\textbf{int ref} \cdot (\textbf{\textit{Its}}(\ell), \textbf{int}, !\mathsf{alloc}(\ell) \otimes \top) \rightharpoonup (\textbf{unit}, \top)\end{array}$$

4.2 Alias Types

Here we demonstrate how our system of type refinements is able to capture simple aliasing constraints, as in previous work on alias types [SWM00, WM00]. These constraints allow us to deallocate memory explicitly, yet safely, using the **free** function. The refinement signature for this application appears below.

$$\begin{array}{ll} () & : \textbf{unit} \\ n & : \textbf{int} \qquad \text{(for any integer } n\text{)} \\ \ell & : \textbf{int ref} \qquad \text{(for any location } \ell\text{)} \\ \textbf{new} & : \forall w{:}\textbf{int} \cdot (\textbf{\textit{Its}}(w), \mathbf{1}) \rightharpoonup \exists[\ell{:}\textbf{int ref}](\textbf{\textit{Its}}(\ell), \mathsf{ctns}(\ell, w)) \\ \textbf{get} & : \forall \ell{:}\textbf{int ref} \cdot \forall w{:}\textbf{int} \cdot (\textbf{\textit{Its}}(\ell), \mathsf{ctns}(\ell, w)) \rightharpoonup (\textbf{\textit{Its}}(w), \mathsf{ctns}(\ell, w)) \\ \textbf{set} & : \forall \ell{:}\textbf{int ref} \cdot \forall w{:}\textbf{int} \cdot \forall w'{:}\textbf{int} \cdot \\ & \qquad (\textbf{\textit{Its}}(\ell), \textbf{\textit{Its}}(w), \mathsf{ctns}(\ell, w')) \rightharpoonup (\textbf{unit}, \mathsf{ctns}(\ell, w)) \\ \textbf{free} & : \forall \ell{:}\textbf{int ref} \cdot \forall w{:}\textbf{int} \cdot (\textbf{\textit{Its}}(\ell), \mathsf{ctns}(\ell, w)) \rightharpoonup (\textbf{unit}, \mathbf{1}) \end{array}$$

A single predicate $\mathsf{ctns}(\ell, w)$ appears in the signature. It indicates that the location ℓ holds the integer w. The new operation places no requirements on the world in which it operates and therefore its precondition is simply $\mathbf{1}$. The postcondition specifies that exactly one new location has been allocated. The other three functions require that the world refinement $\mathsf{ctns}(\ell, w)$ be satisfied before the function is called.

This example can easily be extended to accommodate region-based memory management [TT94]. We would need to augment the signature with a collection of region constants r and a pair of predicates, $\mathsf{allocreg}(r)$ to indicate that the region r is allocated, and $\mathsf{inreg}(\ell, r)$ to link the location to its region. Refinements can then be written for region allocation, object allocation, get, set and region deallocation operations.

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 [DF01]. 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	:	$\textbf{level} \rightarrow \textbf{prop}$	Level Predicate

Next we consider a variety of kernel functions and their type refinements. First, the `KeSet-PriorityThread` 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.

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

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 $\mathsf{L}(\mathsf{pass}) \oplus \mathsf{L}(\mathsf{apc}) \oplus \mathsf{L}(\mathsf{dis})$.

$$\begin{array}{l}\texttt{KeReleaseSemaphore} : \\ \quad \forall l{:}\mathbf{level} \cdot (\mathbf{sem}, \mathbf{pr}, \mathbf{long}, \mathsf{L}(l) \otimes (\mathsf{L}(l) \multimap \mathsf{less}(\mathsf{dis}))) \rightharpoonup (\mathbf{pr}, \mathsf{L}(l))\end{array}$$

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.

$$\begin{array}{l}\texttt{KeAcquireSpinLock} : \\ \quad \forall l{:}\mathbf{level} \cdot (\mathbf{sem}, \mathbf{pr}, \mathbf{long}, \mathsf{L}(l) \otimes (\mathsf{L}(l) \multimap \mathsf{less}(\mathsf{dis}))) \rightharpoonup (\boldsymbol{Its}(l), \mathsf{L}(\mathsf{dis}))\end{array}$$

4.4 Recursion Counts

Worlds have no persistent properties and the ephemeral properties are the singleton sets of predicates $\mathsf{count}(i)$ for all integers i. Before refinement checking occurs, we assume that programs have been instrumented with a call to the constant **inc** at the entry point of every recursive function. We would require a slight extension of our language with function symbols such as "+" in order to encode the necessary invariants in this example. Below, we provide an appropriate interface.

- $\Sigma_p = \{\mathsf{count}{:}(\mathbf{int}) \rightharpoonup (\mathbf{prop}), +{:}(\mathbf{int}, \mathbf{int}) \rightharpoonup (\mathbf{int})\}$
- $\Sigma_A = \{(){:}\mathbf{unit}, n{:}\mathbf{int}, \mathbf{inc}\ {:}(\mathbf{unit}) \rightharpoonup (\mathbf{unit})\}$
- $\Sigma_\phi = \{\mathbf{inc}\ {:}\forall c{:}\mathbf{int} \cdot (\mathbf{unit}, \mathsf{count}(c)) \rightharpoonup (\mathbf{unit}, \mathsf{count}(c+1))\}$

5 Discussion

5.1 Variants and Further Extensions

We have provided semantics for an expressive core set of program refinements. In this section, we informally discuss a number of possible extensions to and variations of our language.

Term Refinements Previous work by Davies, Pfenning and Xi [DP00, XP99] has considered refinements for terms including conjunctive refinements (intersection types) and implication (for stating preconditions on functions in classical logic). We do not expect to encounter difficulties when extending our system with these additional sorts of refinements.

Linear Data Structures In practice, one often associates invariants about state with each cell in a recursive data structure. For example, one might like to represent a list of files, each of which is in the "open" state (as opposed to the "closed" state). One way to accomplish this task is to add linear, or more generally, single-threaded data structures as has been considered elsewhere [WM00, DF01, WW01]. To extend our system, we need only augment our linear context Δ with linear term variables and add the appropriate introduction and elimination forms. A more general approach would be to add arbitrary inductive definitions to our world refinements as in work by Ishtiaq, O'Hearn and Reynolds [IO01, Rey00].

Other Substructural Logics We have worked hard to define our core refinement checking rules for expressions and terms independently of the specifics of any particular logic. As a result, we believe it is possible to consider using a variety of different substructural logics in place of the multiplicative-additive linear logic that we chose to explore in this paper. Whichever logic is chosen, it needs to satisfy the following requirements.

- The logic must contain a linear hypothetical judgment that can control contraction and weakening of hypotheses.
- The logic must satisfy cut-elimination.
- There must be an algorithm for deciding entailment or else programmers must be satisfied with an incomplete type checker or the task of placing additional annotations on programs to guide proof search.

For example, it seems likely that we could use O'Hearn and Pym's bunched logic with its shared implication if we were to interpret "," in our expression refinement judgments as the multiplicative separator in a bunch. It would also be interesting to explore applications involving ordered logic [Pol01]. In this case, we would replace our linear context Δ with a pair of contexts $(\Delta'; \Theta)$ where Δ' is linear as before and Θ is an ordered context. An ordered context does not even allow the structural exchange principle and an ordered type system ensures that assumptions are used in the order they appear. Such an extension may be quite effective for reasoning about resources allocated on the stack [AW03].

5.2 Current and Future Work

There are many directions for future work. We have begun to investigate the following three further issues.

Encoding Type-and-Effect Systems We believe that our language provides a general framework in which to encode many type-and-effect systems. 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 [FA99], 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 [DF01].

Implementation One of the authors (Mandelbaum) has developed a preliminary implementation for small core subset of Java. The current implementation is built using Polyglot [NCM02], an extensible compiler infrastructure for Java. We are very grateful to the Polyglot implementers for giving us access to their software. The current version allows programmers to reason with a minimalist subset of the logic that includes $\mathbf{1}$, $\otimes$ and $\top$. We would like to extend this implementation to include the additive connectives & and $\oplus$ as well as second-order quantifiers. However, the combination of the features will require new algorithms and heuristics for inferring instantiation of second-order quantifiers, unless we require programmers to explicitly instantiate second-order quantifiers, which we believe is likely to be too burdensome in practice. We also need to investigate mechanisms for handling some of Java's advanced features, including exceptions.

Semantics We are interested in extending our semantics for world refinements in several directions. First, as mentioned in section 3.2, linear logic is incomplete with respect to our resource semantics. Although this incompleteness may not cause too much trouble in practice, it is unsatisfying in theory. We plan to attempt to find a substructural logic that is both expressive and complete with respect to this semantics. Second, we have been experimenting with more general uses of the unrestricted modality ! and we are interested in extending our semantics to incorporate $!\psi$ for arbitrary formulas ψ.

Modularity 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.

Concurrency There are a number of different strategies for reasoning about effects in concurrent systems. For example, Gordon and Jefferies [GJ01, GJ02] have used a type and effect system to check the correctness of security policies written in the pi calculus. We believe our logical approach to reasoning about program effects can be extended to a similar sort of concurrent setting. We are eager to discover which substructural logics are best suited for reasoning in concurrent domains.

5.3 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[DP00] and Denney [Den98] and the practical dependent types proposed by Xi and Pfenning [XP98, XP99]. Each of these authors propose 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 consider the ephemeral properties that we are able to specify and check.

Safe Languages CCured [NMW02], CQual [FTA02], Cyclone [GMJ$^+$02], ESC [Det96, fJ02], and Vault [DF01, FD02] 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 [WCM00] and alias types [SWM00, WM00], 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. Our logic is more general as we may take advantage of linear implication and the additive connectives, and yet our type system remains decidable.

Proof-Carrying Type Systems Shao et al. [SSTP02] and Crary and Vanderwaart [CV02] 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 [NL96, Nec97] 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. On the other hand, neither of these logics contain linear logic's left-asynchronous connectives ($\mathbf{1}$, $\otimes$, $\mathbf{0}$, $\oplus$, $\exists$) which we find very convenient in many of our applications.

Hoare Logic Recent efforts by Ishtiaq, O'Hearn and Reynolds [IO01, Rey00] on the reasoning about pointers in Hoare logic 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. 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-refinement checking algorithm.

Model Checking Slightly further removed is work on model checking programs. Model checkers normally attempt to verify programs with little or no programmer annotations; in contrast, our type system requires type refinement annotations and hence requires more work from programmers. On the other hand, type systems usually scale much more effectively to large programs than model checkers since type checking may be done on a module by module basis.

5.4 Conclusions

We have developed a theory of type refinements for effectful computations. This theory includes a semantics for refinements and a sound and decidable syntactic refinement checking system. We have demonstrated the usefulness of our refinements by showing how to encode a number of different kinds of invariants concerning the state of a computation.

Acknowledgments

We would like to thank the implementers of Polyglot [NCM02] 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. This work began at Carnegie Mellon University when Walker was a postdoc there. Walker would like to thank the CMU Pop group and particularly Frank Pfenning for teaching him about the structure of linear logic.

References

[Amb91] Simon Ambler. *First-order Linear Logic in Symmetric Monoidal Closed Categories*. PhD thesis, University of Edinburgh, 1991.

[Aug99] Leonart Augustsson. Cayenne — a language with dependent types. In *ACM International Conference on Functional Programming*, pages 239–250, Baltimore, September 1999. ACM Press.

[AW03] Amal Ahmed and David Walker. The logical approach to stack typing. In *TLDI*, New Orleans, January 2003.

[CHP96] 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.

[CV02] 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.

[CW99] Karl Crary and Stephanie Weirich. Flexible type analysis. In *ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[Den98] Ewen Denney. *A Theory of Program Refinement.* PhD thesis, University of Edinburgh, Edinburgh, 1998.

[Det96] 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.

[DF01] 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.

[DP00] 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.

[FA99] 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.

[FD02] 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.

[fJ02] Extended Static Checking for Java. Cormac Flanagan and Rustan Leino and Mark Lillibridge and Greg Nelson and James Saxes and Raymie Stata. In *ACM Conference on Programming Language Design and Implementation*, Berlin, June 2002. ACM Press.

[FTA02] 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.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ01] Andrew Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *Computer Security Foundations Workshop*, Cape Breton, June 2001. IEEE Computer Society.

[GJ02] Andrew Gordon and Alan Jeffrey. Types and effects for asymmetric cryptographic protocols. In *Computer Security Foundations Workshop*, Cape Breton, June 2002. IEEE Computer Society.

[GMJ$^+$02] 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.

[IO01] 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.

[LS94] Patrick Lincoln and Andre Scedrov. First-order linear logic without modalities is NEXPTIME-hard. *Theoretical Computer Science*, 135:139–154, 1994.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[NCM02] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: A compiler front-end framework for building Java language extensions. Available at `http://www.cs.cornell.edu/Projects/ polyglot/`, 2002.

[Nec97] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[NL96] 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.

[NMW02] 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.

[OP99] Peter O'Hearn and David Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Pol01] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, 2001. Available As Technical Report CMU-CS-01-152.

[Rey00] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial perspectives in computer science*, Palgrove, 2000.

[SSTP02] 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.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381, Berlin, March 2000.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[WCM00] 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.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Montreal, September 2000.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *ACM International Conference on Functional Programming*, Florence, September 2001. ACM Press.

[XP98] 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.

[XP99] 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.