A Syntactic Approach to Type Soundness

Andrew K. Wright* Matthias Felleisen*

Department of Computer Science
Rice University
Houston, TX 77251-1892

June 18, 1992

Rice Technical Report TR91-160
To appear in: *Information and Computation*

Abstract

We present a new approach to proving type soundness for Hindley/Milner-style polymorphic type systems. The keys to our approach are (1) an adaptation of subject reduction theorems from combinatory logic to programming languages, and (2) the use of rewriting techniques for the specification of the language semantics. The approach easily extends from polymorphic functional languages to imperative languages that provide references, exceptions, continuations, and similar features. We illustrate the technique with a type soundness theorem for the core of Standard ML, which includes the first type soundness proof for polymorphic exceptions and continuations.

1 Type Soundness

Static type systems for programming languages attempt to prevent the occurrence of *type errors* during execution. A definition of type error depends on a specific language and type system, but always includes the use of a function on arguments for which it is not defined, and the attempted application of a non-function. A static type system is *sound* if well-typed programs cannot cause type errors; a programming language with a sound static type system is *strongly typed* [7].

While it is easy to design a sound type system for an explicitly typed monomorphic language, the formulation of a sound type system for a language based on Hindley/Milner-style polymorphism and type inference is delicate [6, 8, 15, 17, 20, 37]. Although the treatment of purely functional languages is relatively well understood in this framework, the incorporation of imperative features such as references and exceptions requires extreme care, and the soundness of such systems is not obvious. A formal proof of soundness is

*This research was supported in part by the United States Department of Defense under a National Defense Science and Engineering Graduate Fellowship, by NSF grant CCR 89–17022, and by Texas Grant ARP/ATP-91 #003604-014.

required to lend credibility to the claim that such a system prevents type errors, and may be a crucial tool in the design of the type system.

We view a static type system for an implicitly typed language as a filter that selects well-typed programs from a larger universe of untyped programs. A partial function $\mathit{eval} : \mathit{Programs} \rightharpoonup \mathit{Answers} \cup \{\textsc{wrong}\}$ defines the semantics of untyped programs. The result WRONG is returned for programs whose evaluation causes a type error; *eval* is undefined for programs whose evaluation does not terminate. Let $\rhd e : \tau$ mean that the type system assigns program e the type τ, *i.e.*, e is well-typed. The simplest soundness property states that well-typed programs do not yield WRONG [20].

Definition (Weak Soundness) *If* $\rhd e : \tau$ *then* $\mathit{eval}(e) \neq \textsc{wrong}$.

While weak soundness establishes that a static type system achieves its primary goal of preventing type errors, it is often possible to demonstrate a stronger property that relates the answer produced to the type of the program. If we view each type τ as denoting different subsets V^τ of the set of all answers V, then strong soundness states that an answer v produced by a terminating program of type τ is an element of the subset V^τ.

Definition (Strong Soundness) *If* $\rhd e : \tau$ *and* $\mathit{eval}(e) = v$ *then* $v \in V^\tau$.

Strong soundness permits an implementation of the language to associate the representation of a value with its type, and thereby omit the representation tags required by dynamically typed languages. Weak soundness follows from strong soundness since WRONG is not a member of V^τ for any type τ.

Significant effort has been invested in proving type soundness for Hindley/Milner-style type systems, and their practical realization in the programming language STANDARD ML [22, 23]. Soundness proofs exist for the functional fragment [20, 36], for extensions including references [6, 17, 35, 37], and for a monomorphic language including first-class continuations [8]. However, there are several drawbacks to the existing proofs. Proofs of type soundness are sensitive to the precise formulation of the semantics of the language—different techniques are required for denotational versus operational formulations of the semantics, and even for different languages within the same semantic framework. The proofs for two different languages are difficult to reconcile to prove soundness for a language including features of both. The existing techniques are complicated, and the resulting proofs are lengthy and error-prone. For an illustration of the difficulties with such proofs, we refer the reader to Tofte's [37] discussion of Damas's [6] faulty proof of a type soundness theorem for a polymorphic language with references.

We present a simple approach to the proof of soundness for Hindley/Milner-style polymorphic type systems. Our approach is based on *subject reduction*, a classical result from combinatory logic [4], and on *rewriting* as a means to specifying operational semantics [11, 12, 13, 14]. To demonstrate the approach, we develop a proof of soundness for the core[1] of STANDARD ML, which extends a functional polymorphic language with references and exceptions. We also show soundness for an extension to STANDARD ML with first-class continuations. In principle, our approach is uniformly applicable to any language;

[1]Mitchell and Harper refer to the functional polymorphic sublanguage as the *essence* of ML [24]. However, the difficulties of typing references and exceptions [6, 17, 35, 37] and the fact that they cannot be *expressed* by facilities of the functional core [10] indicate that they are equally important.

in practice, the resulting proofs are lengthy but simple, requiring only ordinary inductive techniques.

In the next section, we describe a prototypical functional language with a polymorphic type system, and discuss the various approaches that have previously been used to prove type soundness. Section 3 presents the essence of our approach to proving type soundness; the remaining sections develop illustrative proofs of type soundness for a specific functional language and various extensions.

2 Previous Approaches to Proving Type Soundness

Any proof of type soundness is intimately tied to the formulation of the semantics of the language. The earlier proofs relied on denotational semantics; later proofs used structural operational semantics. The following two subsections briefly present these approaches from a historical perspective. The third subsection discusses the problems with them. We begin with a brief introduction to the formulation of Hindley/Milner-style type systems, in order to introduce our notation.

Our prototypical functional language has the following (abstract) syntax:

$$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2$$

where $x \in \mathit{Var}$ and $c \in \mathit{Const}$. Constants (Const) include both data (integer, real, boolean, etc.) and operations ($+$, $\div$, $\wedge$, $\vee$, etc.). Variables (Var) are lexically scoped. Juxtaposition denotes application and is left associative; λ-expressions construct call-by-value procedural abstractions. Semantically the **let**-expression behaves like $((\lambda x.e_2)\ e_1)$, binding the value of e_1 to x in e_2; however, the type system allows x to be *polymorphic*: different occurrences of x in e_2 may be assigned different types.

A polymorphic type system for this language has types of the form:

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \tau_1 \rightarrow \tau_2$$

where $\iota_1, \ldots, \iota_n$ are ground types like *int* and *bool*. The type system is formulated as a deductive proof system that assigns types to expressions. The proof system produces conclusions, or *type judgments*, of the form $\Gamma \rhd e : \tau$, meaning that expression e has type τ in type environment Γ. Type environments are finite maps from variables to types, and are used to give types to open expressions. The typing rules are:

(var) $$\Gamma \rhd x : \tau \ \text{ if } \ \Gamma(x) = \tau$$

(const) $$\Gamma \rhd c : \tau \ \text{ if } \ \mathit{TypeOf}(c) = \tau$$

(abs) $$\frac{\Gamma[x \mapsto \tau_1] \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \tau_1 \rightarrow \tau_2}$$

(app) $$\frac{\Gamma \rhd e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1\ e_2 : \tau_2}$$

(let) $$\frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma \rhd e_2[x \mapsto e_1] : \tau_2}{\Gamma \rhd \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 : \tau_2}$$

The function *TypeOf* assigns types to constants. $\Gamma[x \mapsto \tau]$ denotes the functional extension or update of map Γ at x to τ; $e[x \mapsto e']$ means the capture-avoiding substitution of e' for x in e. *Programs* are closed expressions. A program e is *well-typed* if it has a type τ in the empty type environment; we then write $\rhd\, e : \tau$.

2.1 Proofs based on Denotational Semantics

Milner [20]: Milner formulated and proved a type soundness theorem for a functional language like the above based on a denotational semantics for the language of untyped expressions. The semantic domain is the solution of the following reflexive domain equation [32]:

$$\begin{aligned} \mathbb{V} &= \mathbb{B}_0 \oplus \ldots \oplus \mathbb{B}_n \oplus \mathbb{F} \oplus \mathbb{W} \\ \mathbb{F} &= \mathbb{V} \rightarrow \mathbb{V} \end{aligned}$$

$\mathbb{B}_0, \ldots, \mathbb{B}_n$ are basic domains (with bottom), such as integers and booleans; $\mathbb{F}$ is the function domain; $\mathbb{W}$ is a domain consisting of the single element WRONG; $\oplus$ is the separated sum; and $\rightarrow$ builds continuous functions. The meaning function $\mathcal{E}$ is constructed so as to assign the denotation WRONG to programs that result in type errors. To assign meanings to expressions with free variables, the meaning function takes an environment $\rho : \mathit{Var} \xrightarrow{\circ} \mathbb{V}$, a finite map from variables to denotational values.

To establish soundness, Milner introduced a semantic relation $\models$. This relation identifies each type τ with an *ideal* $\mathbb{V}^\tau$ of the domain $\mathbb{V}$ [18]. An ideal is simply a subset of the domain that is closed under certain operations (subset and least upper bounds of finite consistent subsets); each of the basic domains forms an ideal. A denotational value $v \in \mathbb{V}$ *possesses* type τ, written $\models v : \tau$, if v is a member of the ideal corresponding to τ, *i.e.*,

$$v \in \mathbb{V}^\tau \text{ iff } \models v : \tau.$$

An environment ρ *respects* a type environment Γ, written $\models \rho : \Gamma$, if the domain of ρ is at least as large as the domain of Γ, and for every variable x in the domain of Γ, $\models \rho(x) : \Gamma(x)$. A *semantic soundness* theorem states that if an expression e has type τ in type environment Γ, and if environment ρ respects Γ, then the denotation of e in ρ possesses type τ.

Theorem (Semantic Soundness) *If* $\Gamma \rhd e : \tau$ *and* $\models \rho : \Gamma$ *then* $\models \mathcal{E}[\![e]\!]\rho : \tau$.

This theorem is proved by induction on the structure of e. Strong soundness follows by restricting the theorem to closed expressions.

Theorem (Strong Soundness) *If* $\rhd\, e : \tau$ *then* $\models \mathcal{E}[\![e]\!]\emptyset : \tau$.

Weak soundness is a consequence of the fact that WRONG does not possess any type.

Theorem (Weak Soundness) *If* $\rhd\, e : \tau$ *then* $\mathcal{E}[\![e]\!]\emptyset \neq$ WRONG.

Damas [5, 6]**:** Damas extended Milner's results to a language with reference cells and destructive assignment. The proof technique is derived from Milner's technique; however, the proof is significantly more complicated. The denotational semantics uses the following domain construction:

$$\begin{array}{rcl} \mathbb{V} & = & \mathbb{B}_0 \oplus \ldots \oplus \mathbb{B}_n \oplus \mathbb{F} \oplus \mathbb{L} \oplus \mathbb{W} \\ \mathbb{F} & = & \mathbb{V} \rightarrow \mathbb{S} \rightarrow \mathbb{V} \otimes \mathbb{S} \\ \mathbb{S} & = & \mathbb{V}^* \\ \mathbb{L} & = & \mathbb{N}_\perp \end{array}$$

where the domain of locations $\mathbb{L}$ is the flat domain of natural numbers, $\mathbb{S}$ is the domain of stores (sequences of values), and $\otimes$ is the smash product. Procedures take both an input value and a store, and produce a result and a new store. The meaning function is parameterized over both a store and an environment. Due to the presence of stores, types are no longer ideals in $\mathbb{V}$, but are finite maps from store typings to subsets of $\mathbb{V}$.[2] A store typing is a finite map from locations to the types of values stored there. Since the semantic relation ($\models$) must involve the types of values in the store, it can no longer be defined by induction on types; its existence must be established by a category theoretical argument that generalizes the technique of inclusive predicates [31]. The complexity of the semantic relation complicates the proof; indeed, Tofte found a mistake in Damas's proof, although the theorem is not thought to be false [37: page 2].

Abadi, Cardelli, Pierce, and Plotkin [1]**:** Abadi *et al.* demonstrate type soundness for a functional language with a *dynamic* type and related operations. Their proof with respect to a denotational semantics is similar to Milner's proof, although it is complicated by the presence of *dynamic* values. A *dynamic* value is a pair consisting of a value v and a tag, drawn from a set *TypeCode*, that encodes the type of v. The domain equation includes a domain of *dynamic* values $\mathbb{D}$:

$$\begin{array}{rcl} \mathbb{V} & = & \mathbb{B}_0 \oplus \ldots \oplus \mathbb{B}_n \oplus \mathbb{F} \oplus \mathbb{D} \oplus \mathbb{W} \\ \mathbb{F} & = & \mathbb{V} \rightarrow \mathbb{V} \\ \mathbb{D} & = & \mathit{TypeCode} \times \mathbb{V} \end{array}$$

As type codes correspond to types, the denotation of a type code is a type. Types, in turn, denote ideals over $\mathbb{V}$, but due to the unusual element $\mathbb{D}$ of the domain equation, establishing that types denote ideals requires extending the ideal model for recursive types [18]. The proof that the required fixed points exist in the new model involves a metric space argument, but is a straightforward extension of the original. Although the type system Abadi *et al.* use is not polymorphic, it should be possible to extend the type system and the soundness proof to include polymorphism.

Duba, Harper, and MacQueen [8]**:** Duba *et al.* present several languages extending a monomorphic functional core with first-class continuations in the spirit of Scheme [28]. They describe several approaches to proving type soundness for the languages. One is with

[2]This is an over-simplification; the reader interested in the precise definition is referred to Damas's thesis [6].

respect to a continuation-passing denotational semantics, based on the following domain equations:

$$\begin{array}{rcl} \mathbb{V} & = & \mathbb{B}_0 \oplus \ldots \oplus \mathbb{B}_n \oplus \mathbb{F} \oplus \mathbb{K} \\ \mathbb{F} & = & \mathbb{V} \to \mathbb{K} \to \mathbb{A} \\ \mathbb{K} & = & \mathbb{V} \to \mathbb{A} \\ \mathbb{A} & = & \mathbb{V} \oplus \mathbb{W} \end{array}$$

where $\mathbb{K}$ is the domain of continuations, and $\mathbb{A}$ is the domain of answers. The meaning function $\mathcal{E}$ takes both an environment and a continuation. Two semantic relations state what it means for a value v to possess a type τ, written $\models v : \tau$, and for a continuation κ to accept a value of type τ, written $\models \kappa :: \tau$. These relations are defined simultaneously by induction on types. The semantic soundness theorem states that if an expression e has type τ in type environment Γ, and if ρ respects Γ, and if continuation κ accepts values of type τ, then the meaning of e in environment ρ and continuation κ is not WRONG.

Theorem (Semantic Soundness) *If* $\Gamma \triangleright e : \tau$ *and* $\models \rho : \Gamma$ *and* $\models \kappa :: \tau$ *then* $\mathcal{E}[\![e]\!]\rho\kappa \neq$ WRONG.

Unlike in the purely functional setting, the semantic soundness theorem only states that well-typed programs do not go WRONG (weak soundness), not that they produce answers of the expected type. To approach strong soundness, Duba *et al.* give an argument based on a continuation-passing-style (CPS) translation into the simply typed λ-calculus. For programs of ground type, the denotation of an expression is the same as the denotation of its CPS transform with the identity function as an initial continuation, *i.e.*,

$$\mathcal{E}[\![e]\!]\emptyset = \mathcal{E}[\![\mathit{CPS}(e)\ (\lambda x.x)]\!]\emptyset.$$

Since strong soundness holds for the simply typed λ-calculus, and $(\mathit{CPS}(e)\ (\lambda x.x))$ has the same type as e for programs of ground type, strong soundness holds for programs of ground type. However, the argument does not extend to higher types. For further details, we refer the reader to their paper [8: page 169].

2.2 Proofs based on Structural Operational Semantics

Tofte [36, 37]**:** Tofte reformulated Milner's functional language with a structural operational semantics [27]. The semantics is specified as a deductive proof system; a conclusion $E \vdash e \Rightarrow v$ of a deduction states that expression e evaluates to v in value environment E. A value environment is a finite map from variables to operational values. Values are either basic constants or closures, which result from the evaluation of λ-expressions. The soundness theorem again requires the definition of a semantic relation ($\models$) between operational values and types. For basic constants, the relation is explicitly specified by the function *TypeOf*; for closures, the relation is defined by induction on types, and by evaluation at correctly typed argument values. The semantic soundness theorem states that if an expression e has type τ in type environment Γ, and if environment E respects Γ, and if e evaluates to a value v in environment E, then v possesses type τ.

Theorem (Semantic Soundness) *If* $\Gamma \triangleright e : \tau$ *and* $\models E : \Gamma$ *and* $E \vdash e \Rightarrow v$ *then* $\models v : \tau$.

The proof is straightforward, and proceeds by induction on the depth of the deduction of $E \vdash e \Rightarrow v$. Strong and weak soundness follow as before.

Tofte turned to structural operational semantics in order to consider references. To this end, he reformulated the semantics with conclusions of the form $s, E \vdash e \Rightarrow v, s'$, meaning that in environment E and store s, expression e evaluates to value v and the new store s'. The semantic relation now involves the contents of the store and the store typing (ST), a map from locations to the types of values stored there. The semantic relation $s : ST \models v : \tau$ may be read as "given the ST-typed store s, value v possesses type τ." Because of the possibility of cycles amongst references in the store, this relation is not definable by induction, but must be defined as the maximal fixed-point of a monotonic operator on the appropriate space. The semantic soundness theorem states that if an expression e has type τ in type environment Γ, and given store s of type ST, environment E respecting Γ, and that e evaluates to v and store s', then a typing ST' can be found for s' such that v possesses type τ.

Theorem (Semantic Soundness) *If* $\Gamma \rhd e : \tau$ *and* $s : ST \models E : \Gamma$ *and* $s, E \vdash e \Rightarrow v, s'$ *then there exists* ST' *such that* $s' : ST' \models v : \tau$.

The theorem is proved by induction on the depth of the deduction of $s, E \vdash e \Rightarrow v, s'$. Lemmas involving the semantic relation are proved using the technique of co-induction that Milner and Tofte developed for this purpose [21]. Leroy and Weis have successfully applied Tofte's technique to show soundness for a different type system for polymorphic references [17]. Talpin and Jouvelot have also applied the technique to demonstrate soundness for a system that infers types, effects, and regions for references [35].

Duba, Harper, and MacQueen [8]**:** Duba *et al.* also present a proof of type soundness with respect to a structural operational semantics for a monomorphic language with continuations. The proof is an adaptation of Tofte's technique; however, the semantics is significantly restructured, as the technique of defunctionalization [30] is used to represent the flow of control explicitly. The semantics has two judgment forms. The first kind of conclusion, $E; K \vdash e \Rightarrow v$, indicates that expression e evaluates to answer v in environment E and continuation K. The second kind of conclusion, $v \vdash K \Rightarrow v'$, indicates that continuation K evaluated at value v produces answer v'. As in the denotational case, two semantic relations are defined, indicating what it means for a value to possess a type ($\models v : \tau$), and for a continuation to accept a type ($\models K :: \tau$). Like the denotational case, the theorem only establishes weak soundness.

Theorem (Semantic Soundness) *If* $\Gamma \rhd e : \tau$ *and* $\models E : \Gamma$ *and* $\models K :: \tau$ *and* $E; K \vdash e \Rightarrow v$ *then* $v \neq \text{WRONG}$.

It is not clear that the proof goes through in the presence of fixed-point operators [8: page 171], nor how to obtain strong soundness in this framework.

2.3 Discussion

In each case above, the proof of soundness for a language or for a different formulation of a language's semantics involves the use of a different proof technique. The techniques are often unrelated, so they provide no guidance in proving soundness for new languages or language features. A seemingly minor extension to a language may require a complete restructuring of its denotational or structural operational semantics, and may therefore require a completely new approach to re-establish soundness. For example, in introducing references to Milner's functional language, Damas changed the domain equations for the denotational semantics to accommodate a store component, necessitating a completely different strategy for induction. In introducing type *dynamic*, Abadi *et al.* again changed the domain equations, and extended the ideal model of types to match. To accommodate continuations, Duba *et al.* changed the domain equations yet again, also foregoing the tools developed by Milner and Damas. They used a separate argument to establish strong soundness. In introducing references to an operational formulation of the functional language, Tofte changed the form of judgments obtained by the semantics, and used the technique of co-induction to establish a proof. For continuations in a structural operational semantics, Duba *et al.* completely altered the semantics,[3] again changing the structure of the proof dramatically.

Each of the above proofs considers a language with a single extension to a functional core. It is natural to ask whether two extensions can be merged, such that a natural union of the two type systems is sound. However, since each of the proofs involves a different technique, it is generally impossible to merge the proofs. Tofte's proof for references and Duba *et al.*'s proof for continuations give no direct assurance that STANDARD ML's type system is sound, even if the other features of STANDARD ML are ignored.

3 A Syntactic Approach to Proving Type Soundness

Our approach to type soundness is based upon an operational formulation of a language's semantics by *rewriting*. Each intermediate state of an evaluation of a program is itself a program, and the evaluation of a program is performed by successive *reductions* into a new state: $e_1 \longmapsto e_2 \longmapsto \ldots$ Reduction may either continue forever ($e\Uparrow$), or may reach a *final* state where no further evaluation is possible: $e_1 \longmapsto\!\!\!\!\rightarrow e_k$ and $e_k \not\longmapsto e_{k+1}$. Such a final state represents either an answer or a type error. Thus, proving type soundness reduces to proving that well-typed programs yield only well-typed answers.

Programming language calculi, like the λ-calculus, are the natural choice to specify the semantics of a language such that each intermediate step of evaluation is a program. Plotkin [26] shows how the semantics of a prototypical functional language relates to the λ-calculus. The λ-calculi extensions for state [12, 14] extend this strategy to languages with references and similar constructs. Likewise, the control calculi [13, 14] can be adapted to address non-local control facilities such as exceptions and first-class continuations.

Since the intermediate states of evaluation are programs, we may apply the type system to deduce a type for each state. Our strategy for proving type soundness rests upon two fundamental observations about the types of states. The first observation is that each intermediate state may be assigned the same type as the original program, *i.e.*, reductions preserve type:

[3]Furthermore, they take this as "evidence that the addition of [continuations] to Standard ML would be a substantial change, rather than an incremental modification, to the language" [8: page 170]. However, we believe that this only indicates a problem with their semantic framework.

$$\textit{if } \rhd e_1 : \tau \textit{ and } e_1 \longmapsto e_2 \textit{ then } \rhd e_2 : \tau.$$

In combinatory logic, this property is known as *subject reduction* [4].

The second observation is that sound type systems do not assign a type to irreducible expressions that are the sources of type errors, such as $\mathbf{1} + \mathsf{true}$. Furthermore, any expression containing such a subexpression is untypable; we call expressions containing such irreducible subexpressions *faulty*:

$$\textit{if } e \textit{ is faulty, there is no } \tau \textit{ such that } \rhd e : \tau.$$

Together, these two observations imply that all final states that can be reached from a well-typed program are well-typed answers. With a definition of *eval* that takes faulty expressions to WRONG, the type system is weakly sound.

As all answers produced are well-typed programs, subject reduction also establishes strong soundness. To state that an answer v lies within the correct subset V^τ of values, we employ the type system itself:

$$v \in V^\tau \text{ iff } \rhd v : \tau.$$

This yields a soundness theorem based on a *syntactic* connection between answers and types:

$$\textit{if } \rhd e : \tau \textit{ then either } e \Uparrow \textit{ or } e \longmapsto\!\!\!\!\rightarrow v \textit{ and } \rhd v : \tau.$$

Again, strong soundness follows with a definition of *eval* that takes faulty expressions to WRONG.

In summary, proving a syntactic soundness theorem involves:

- demonstrating subject reduction,
- characterizing answers and faulty expressions, and
- showing that the faulty expressions are untypable.

The structure of the proof always remains the same, whether or not the language includes some combination of references, exceptions, continuations, or other features.

3.1 Related Work

Curry and Feys [4] introduced the notion of subject reduction for combinatory logic. In the language of combinatory logic, the CL-term e of a deduction concluding $\rhd e : \tau$ is called the *subject*, and the type τ is called the *predicate*. Subject reduction states that reduction of the subject of a deduction preserves the predicate. Subject reduction holds for terms in CL and the λ-calculus [33].

Mitchell and Plotkin [25] present a type preservation theorem, *i.e.*, subject reduction, for a variant of the second order polymorphic λ-calculus. This language has explicitly typed declarations; the proof of type preservation is significantly simpler in the presence of such declarations. They do not develop a type soundness theorem.

Abadi *et al.* [1] give a proof of soundness for type *dynamic* with respect to a structural operational semantics, in addition to their denotationally based proof. In their structural operational semantics, substitutions are carried out immediately: λ-expressions evaluate to themselves, rather than closures, and judgments do not contain an environment component. Since answers are syntactic values, the type system can be applied to answers to show that evaluation preserves typing. Although this proof is similar to our proof of subject reduction, its extension to references or continuations apparently involves the same difficulties as Tofte's and Duba *et al.*'s proofs.

3.2 Road Map

The next three sections illustrate our approach by proving soundness for several languages. Section 4 considers a functional language, similar to that considered by Milner [20]. Section 5 considers an extension to references, an extension to exceptions, and shows how the results may be merged to consider a language with both. To our knowledge, this is the first proof of soundness for exceptions. Section 6 illustrates soundness for an extension providing first-class continuations, and is the first proof of strong type soundness for continuations in a polymorphic language. We conclude with a discussion of our technique, and suggestions for its application to other languages.

4 Functional ML

Functional ML is an applicative language with constants, call-by-value higher-order functions, and a Hindley/Milner-style polymorphic type system. A natural basis for a calculus of Functional ML is Plotkin's untyped λ_v-calculus [26].

Let *Var* be a denumerable set of *variables* and *Const* be a set of *constants*. The *expressions* and *values* of Functional ML are:

(*Expressions*)	$e ::= v \mid e_1\ e_2 \mid \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2$
(*Values*)	$v ::= c \mid x \mid \mathsf{Y} \mid \lambda x.e$

where $x \in \mathit{Var}$ and $c \in \mathit{Const}$. The free variables, $FV(e)$, and bound variables of an expression are defined as usual, with λ- and let-expressions binding their variables. The let-expression binds x in e_2 but not e_1, *i.e.*, let bindings are not recursive. The fixed-point combinator Y provides recursion. Following Barendregt [2], we adopt the convention that bound variables are always distinct from free variables in distinct expressions, and we identify expressions that differ only by a consistent renaming of the bound variables. *ClosedVal* is the set of values with no free variables.

4.1 Semantics

The calculus for Functional ML is based upon four relations, called *notions of reduction*:

(δ)	$c\ v \longrightarrow \delta(c,v)$ if $\delta(c,v)$ is defined
(β_v)	$(\lambda x.e)\ v \longrightarrow e[x \mapsto v]$
(*let*)	$\mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]$
(Y)	$\mathsf{Y}\ v \longrightarrow v\ (\lambda x.(\mathsf{Y}\ v)\ x)$

To abstract from the precise set of constants, we only assume the existence of a partial function:

$$\delta : \mathit{Const} \times \mathit{ClosedVal} \rightharpoonup \mathit{ClosedVal}$$

that interprets the application of functional constants to closed values and yields closed values. The β_{v} and *let* reductions use substitution: the notation $e[x \mapsto v]$ means the substitution of v for free occurrences of x in e, renaming bound variables of v as necessary to avoid capture. The Y reduction introduces an abstraction around $(\mathsf{Y}\ v)$ as v must be applied to a value (x does not appear free in v by the variable conventions).

We refer to the union of these relations as $\mathbf{v}$, or simply $\longrightarrow$ when there is no danger of confusion. The notion of reduction $\mathbf{v}$ gives rise to a system of reductions, $\longrightarrow\!\!\!\!\!\rightarrow$. The relation $\longrightarrow\!\!\!\!\!\rightarrow$ is the reflexive, transitive, and compatible closure of $\longrightarrow$:

$$(\longrightarrow\!\!\!\!\!\rightarrow) \qquad \frac{e_1 \longrightarrow e_2}{e_1 \longrightarrow\!\!\!\!\!\rightarrow e_2} \qquad e_1 \longrightarrow\!\!\!\!\!\rightarrow e_1 \qquad \frac{e_1 \longrightarrow\!\!\!\!\!\rightarrow e_2 \quad e_2 \longrightarrow\!\!\!\!\!\rightarrow e_3}{e_1 \longrightarrow\!\!\!\!\!\rightarrow e_3} \qquad \frac{e_1 \longrightarrow\!\!\!\!\!\rightarrow e_2}{C[e_1] \longrightarrow\!\!\!\!\!\rightarrow C[e_2]}$$

where

$$C ::= [\,] \mid C\ e \mid e\ C \mid \mathsf{let}\ x\ \mathsf{be}\ C\ \mathsf{in}\ e \mid \mathsf{let}\ x\ \mathsf{be}\ e\ \mathsf{in}\ C \mid \lambda x.C.$$

A *context* C is an expression with one subexpression replaced by a hole, denoted $[\,]$. The expression $C[e]$ results from placing an expression e in the hole of C; this operation may involve capture of the free variables of e by C. An equational system may be constructed as the congruence closure of $\mathbf{v}$, but plays no rôle for our work.

With an appropriate choice of δ, the calculus satisfies Church-Rosser and Standardization properties; the proofs are variants of Plotkin's proofs for the λ_{v}-calculus [26]. By Standardization, we know that an expression reduces to a value precisely if a reduction of leftmost-outermost redexes outside of abstractions leads to a value. Based on this idea, we can use the calculus to define an evaluation function [11]. Let $\longmapsto$ be the relation:

$$(\longmapsto) \qquad E[e] \longmapsto E[e'] \quad \text{iff} \quad e \longrightarrow e'$$

where

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ e.$$

The special contexts E are *evaluation contexts*: they ensure that the leftmost-outermost reduction is the only applicable reduction in the entire program. Evaluating from left to right in an application is consistent with STANDARD ML. Unlike with C contexts, there is no possibility of capture when placing an expression in the hole of an evaluation context, because the hole cannot appear inside a binding construct. Let $\longmapsto\!\!\!\!\!\rightarrow$ be the reflexive and transitive closure of $\longmapsto$. Then the partial function *eval* is defined for closed expressions e as:

$$(\mathit{eval}) \qquad \mathit{eval}(e) = v \quad \text{iff} \quad e \longmapsto\!\!\!\!\!\rightarrow v.$$

It is easy to see that the decomposition of a program into a $\mathbf{v}$-redex and an evaluation context is unique. It follows that the stepping relation $(\longmapsto)$ is a function, and therefore that *eval* is a function, too.

4.2 Typing

The type system is a deductive proof system that assigns types to expressions. The types of Functional ML are:

$$(\textit{Types}) \qquad \tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \to \tau_2$$

where $\iota_1, \ldots, \iota_n$ are ground types for basic constants (*int*, *bool*, *real*, etc.), and α ranges over a denumerable set of *type variables*. Type variables stand for some fixed but unknown type. Function types, $\tau_1 \to \tau_2$, are right associative.

The polymorphic type system allows certain variables to be used with a set of different types. For example, if the identity function $\lambda x.x$ is **let**-bound to the variable f, then f may used as a function of type $\mathit{int} \to \mathit{int}$ and applied to values of type *int*, or f may be used as a function of type $\mathit{bool} \to \mathit{bool}$ and applied to values of type *bool*. *Type schemes* represent such sets of types (α^* means zero or more distinct type variables):

$$(\textit{TypeSchemes}) \qquad \sigma ::= \forall \alpha^*.\tau$$

The free type variables $FTV(\sigma)$ of a type or type scheme are defined as usual, where $\forall \alpha_1 \ldots \alpha_n.\tau$ binds α_1 through α_n in τ. We identify type schemes that differ only by a consistent renaming of bound type variables, or by a reordering of the binding occurrences. We also identify $\forall.\tau$ with τ, so types may be regarded as type schemes with no bound type variables.

A type scheme $\forall \alpha_1 \ldots \alpha_n.\tau$ represents the set of types that may be obtained from τ by substituting for α_1 through α_n. This is formalized by the notion of *generalization*: a type scheme σ generalizes a type τ, written $\sigma \succ \tau$, if there is a substitution for the bound variables of σ yielding τ. A substitution S is a (partial) function from type variables to types. A substitution may be applied to a type in the obvious way; we write $S\tau$. Substitution on type schemes respects bound type variables and is capture-avoiding. Generalization of a type is defined as:

$$(\succ) \qquad \forall \alpha_1 \ldots \alpha_n.\tau' \succ \tau \;\text{ iff }\; S\tau' = \tau \text{ and } \mathrm{Dom}(S) = \{\alpha_1, \ldots, \alpha_n\} \text{ for some } S.$$

Generalization is extended to type schemes:

$$(\succ) \qquad \sigma \succ \sigma' \;\text{ iff }\; \text{whenever } \sigma' \succ \tau \text{ then } \sigma \succ \tau.$$

Some examples are:

$$\begin{array}{rcl}
\tau & \succ & \tau \\
\forall \alpha.\alpha \to \alpha & \succ & \mathit{int} \to \mathit{int} \\
\forall \alpha_1\alpha_2.\alpha_1 \to \alpha_2 & \succ & \mathit{int} \to \mathit{bool} \\
\forall \alpha.\alpha \to \alpha & \succ & \alpha_2 \to \alpha_2 \\
\forall \alpha_1\alpha_2.\alpha_1 \to \alpha_2 & \succ & \forall \alpha_1.\alpha_1 \to \alpha_2
\end{array}$$

Generalization is a partial order on type schemes.

In order to assign types to open expressions, assumptions about the types of free variables are provided by *type environments*, denoted Γ. Type environments are finite maps from variables to type schemes. The free type variables $FTV(\Gamma)$ of a type environment

(var) $$\Gamma \rhd x : \tau \;\text{ if }\; \Gamma(x) \succ \tau$$

(const) $$\Gamma \rhd c : \tau \;\text{ if }\; \mathit{TypeOf}(c) \succ \tau$$

(Y) $$\Gamma \rhd \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

(abs) $$\frac{\Gamma[x \mapsto \tau_1] \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \tau_1 \to \tau_2}$$

(app) $$\frac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1\ e_2 : \tau_2}$$

(let) $$\frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \rhd e_2 : \tau_2}{\Gamma \rhd \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\mathit{Close}(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma)$$

Figure 1: Typing rules for Functional ML

are the free type variables of the type schemes in its range. For stating and proving various lemmas, it is convenient to define substitution on type environments pointwise, *i.e.*, $S(\{\langle x_1, \sigma_1\rangle, \ldots, \langle x_n, \sigma_n\rangle\}) = \{\langle x_1, S\sigma_1\rangle, \ldots, \langle x_n, S\sigma_n\rangle\}$.

The function $\mathit{TypeOf} : \mathit{Const} \to \mathit{TypeSchemes}$ associates a type scheme with every constant. Since we have not precisely specified the set of constants, Functional ML is really a class of languages parameterized by the set *Const* and the functions δ and *TypeOf*. Just as the Standardization and Church-Rosser properties restrict the admissible set of δ functions, a particular choice of *Const*, δ, and *TypeOf* must satisfy a *typability* condition for type soundness to hold. For every c, τ, τ', and v,

(δ-typability) $$\begin{array}{c}\text{if } \mathit{TypeOf}(c) \succ \tau' \to \tau \text{ and } \rhd v : \tau', \\ \text{then } \delta(c, v) \text{ is defined and } \rhd \delta(c, v) : \tau.\end{array}$$

This condition requires that δ be defined for all constants of functional type and arguments of matching type, and restricts the set of results that δ may produce. This rules out, for the moment, functional constants such as division that are not defined on all values of their input type.

A *type judgment* $\Gamma \rhd e : \tau$ for e an expression of Functional ML is the conclusion of a deduction constructed according to the inference rules in Figure 1. The judgment $\Gamma \rhd e : \tau$ is read "in type environment Γ, expression e has type τ". If the type environment in a judgment is empty, we write $\rhd e : \tau$. An expression e is a *well-typed program* if it is closed and there is a judgment $\rhd e : \tau$.

In this formulation of the typing rules, only one rule is applicable to an expression. Hence if there is a deduction assigning an expression a specific type, that deduction is unique. This permits proofs by induction on the structure of a deduction for a type judgment to proceed by case analysis on the structure of the expression of the type judgment.

This type system has the important property that there is an algorithm $\mathcal{W}$ to determine whether an expression has a type [20]. Given a type environment and an expression, the algorithm computes a substitution and a type. The algorithm is sound [20] with respect to the type system, meaning that it infers only valid typings.

Theorem (Soundness of $\mathcal{W}$) [Milner] *If $(S, \tau) = \mathcal{W}(\Gamma, e)$ succeeds then $S\Gamma \triangleright e : \tau$.*

The algorithm is also complete [5], meaning that if an expression has a valid typing, then the algorithm will find a typing. Furthermore, the algorithm always terminates.

Theorem (Completeness of $\mathcal{W}$) [Damas, Milner] *If $S_1\Gamma \triangleright e : \tau_1$ then $(S, \tau) = \mathcal{W}(\Gamma, e)$ succeeds and there exists S' such that $S'(S\Gamma) = S_1\Gamma$ and $S'\mathit{Close}(\tau, S\Gamma) \succ \tau_1$. If $\mathcal{W}(\Gamma, e)$ does not succeed, then it stops with* **fail**.

Proving completeness involves demonstrating that if an expression has a type, then it has a *principal* type, of which all others are substitution instances.

Theorem (Principal Typing) *If $\Gamma \triangleright e : \tau_1$ then there exists τ such that $\Gamma \triangleright e : \tau$ and whenever $\Gamma \triangleright e : \tau_2$ then $\mathit{Close}(\tau, \Gamma) \succ \tau_2$. The type τ is* principal *for e in Γ.*

The existence of a sound and complete algorithm establishes the decidability of the type system. However, this is solely a property of the type system, and says nothing about the relation of the type system to the semantics of the language.

4.3 Type Soundness

Our proof of type soundness rests upon the notion of subject reduction [4]. The subject reduction property states that reductions preserve the type of expressions. Lemma 4.3 below extends subject reduction to Functional ML.

Subject reduction by itself is not sufficient for type soundness. In addition, we must prove that programs containing type errors are not typable. Put differently, if an expression e is irreducible due to some type error, say $(/\ \mathbf{1}\ \mathbf{0})$, then the type system should not be able to assign a type to the expression. Otherwise, a well-typed program could reduce to such an expression without violating subject reduction, and still cause a type error to occur. We call such expressions with type errors *faulty expressions* and prove that faulty expressions cannot be typed.

Some obvious facts about deductions that we use with no more ado are:

(i) if $\Gamma \triangleright C[e] : \tau$ then there exist Γ' and τ' such that $\Gamma' \triangleright e : \tau'$;

(ii) if there are no Γ', τ' such that $\Gamma' \triangleright e : \tau'$, then there are no Γ, τ such that $\Gamma \triangleright C[e] : \tau$.

These follow from the facts that (1) there is exactly one inference rule for each expression form, and (2) each inference rule requires a proof for each subexpression of the expression in its conclusion. Several facts about generalization follow from the definition of *Close*:

(i) if $\sigma' \succ \sigma$ then $\mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma']) \succ \mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma])$;

(ii) if $x \notin \mathrm{Dom}(\Gamma)$ then $\mathit{Close}(\tau_1, \Gamma) \succ \mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma])$.

The following lemma states that extra variables in the type environment Γ of a judgment $\Gamma \triangleright e : \tau$ that are not free in the expression e may be ignored.

Lemma 4.1 *If* $\Gamma(x) = \Gamma'(x)$ *for all* $x \in FV(e)$*, then* $\Gamma \rhd e : \tau$ *iff* $\Gamma' \rhd e : \tau$.

A key lemma that we use in the proof of Subject Reduction for Functional ML and its extensions is the Replacement Lemma, adapted from Hindley and Seldin [16: page 181]. This allows the replacement of one of the subexpressions of a typable expression with another subexpression of the same type, without disturbing the type of the overall expression.

Lemma 4.2 (Replacement) *If*:

(*i*) $\mathcal{D}$ *is a deduction concluding* $\Gamma \rhd C[e_1] : \tau$,

(*ii*) $\mathcal{D}_1$ *is a subdeduction of* $\mathcal{D}$ *concluding* $\Gamma' \rhd e_1 : \tau'$,

(*iii*) $\mathcal{D}_1$ *occurs in* $\mathcal{D}$ *in the position corresponding to the hole* ($[]$) *in* C*, and*

(*iv*) $\Gamma' \rhd e_2 : \tau'$,

then $\Gamma \rhd C[e_2] : \tau$.

Proof. We think of deductions as trees with the conclusion at the root. Let $\mathcal{D}_2$ be the deduction concluding $\Gamma' \rhd e_2 : \tau'$. Cut the subtree $\mathcal{D}_1$ out of $\mathcal{D}$ and replace it with $\mathcal{D}_2$. Replace all relevant occurrences of e_1 in the resulting tree with e_2. Then the resulting tree is a valid deduction concluding $\Gamma \rhd C[e_2] : \tau$, as may be shown by induction on the height of the tree [16: page 181]. ∎

Subject reduction for Functional ML states that the reductions of **v** preserve type.

Main Lemma 4.3 (Subject Reduction) *If* $\Gamma \rhd e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $\Gamma \rhd e_2 : \tau$.

Proof. The proof proceeds by case analysis according to the reduction $e_1 \longrightarrow e_2$.

Case $c\ v \longrightarrow \delta(c, v)$. Then $\Gamma \rhd c : \tau' \to \tau$ and $\Gamma \rhd v : \tau'$ follow from $\Gamma \rhd c\ v : \tau$ by **app**. Hence $\Gamma \rhd \delta(c, v) : \tau$ by the δ-typability condition.

Case $(\lambda x.e)\ v \longrightarrow e[x \mapsto v]$. From $\Gamma \rhd (\lambda x.e)\ v : \tau$ we have $\Gamma \rhd v : \tau'$ and $\Gamma \rhd \lambda x.e : \tau' \to \tau$ by **app**. From the latter, $\Gamma[x \mapsto \tau'] \rhd e : \tau$ follows by **abs**. Hence $\Gamma \rhd e[x \mapsto v] : \tau$ by Lemma 4.4.

Case $\mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]$. From $\Gamma \rhd \mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e : \tau$ we have $\Gamma \rhd v : \tau'$ and $\Gamma[x \mapsto \mathit{Close}(\tau', \Gamma)] \rhd e : \tau$ by **let**. As $\mathit{Close}(\tau', \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau'$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathit{FTV}(\tau') \setminus \mathit{FTV}(\Gamma)$, we have $\Gamma \rhd e[x \mapsto v] : \tau$ by Lemma 4.4.

Case $\mathsf{Y}\ v \longrightarrow v\ (\lambda x.(\mathsf{Y}\ v)\ x)$. From $\Gamma \rhd \mathsf{Y}\ v : \tau$, where $\tau = \tau_1 \to \tau_2$, by **Y** and **app**

(1) $$\Gamma \rhd v : (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2.$$

Then from $\Gamma \rhd \mathsf{Y}\ v : \tau$,

$$\begin{array}{ll}
\Gamma[x \mapsto \tau_1] \rhd \mathsf{Y}\ v : \tau_1 \to \tau_2 & \text{by Lemma 4.1,} \\
\Gamma[x \mapsto \tau_1] \rhd (\mathsf{Y}\ v)\ x : \tau_2 & \text{by } \textbf{app}, \\
\Gamma \rhd \lambda x.(\mathsf{Y}\ v)\ x : \tau_1 \to \tau_2 & \text{by } \textbf{abs}, \\
\Gamma \rhd v\ (\lambda x.(\mathsf{Y}\ v)\ x) : \tau_1 \to \tau_2 & \text{by } \textbf{app} \text{ with (1).}
\end{array}$$

This completes the essence of the proof. It remains to establish several lemmas. ∎

A Substitution Lemma is the key to showing type preservation for reductions involving substitution.

Lemma 4.4 (Substitution) *If* $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e : \tau'$ *and* $x \notin \mathrm{Dom}(\Gamma)$ *and* $\Gamma \rhd v : \tau$ *and* $\{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\Gamma) = \emptyset$ *then* $\Gamma \rhd e[x \mapsto v] : \tau'$.

Proof. We proceed by induction on the length of the proof of $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e : \tau'$, and case analysis on the last step.

Case $e = c$. Then $\mathit{TypeOf}(c) \succ \tau'$ by **const**, and thus $\Gamma \rhd c : \tau'$ by **const**. Then $\Gamma \rhd c[x \mapsto v] : \tau'$ since $c[x \mapsto v] = c$.

Case $e = x'$. If $x' \neq x$, then $\Gamma x' \succ \tau'$ by **var**, and $\Gamma \rhd x' : \tau'$ again by **var**. Hence $\Gamma \rhd x'[x \mapsto v] : \tau'$ since $x'[x \mapsto v] = x'$.

If $x' = x$, then by **var**

$$\begin{aligned} &\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau](x) \succ \tau' \\ \textit{i.e. } &\forall\alpha_1 \ldots \alpha_n.\tau \succ \tau', \end{aligned}$$

i.e. we can find a substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S\tau = \tau'$. Next, we have $S\Gamma \rhd v : S\tau$ by Lemma 4.5 since $\Gamma \rhd v : \tau$, and therefore $S\Gamma \rhd v : \tau'$. But $\mathrm{Dom}(S) \cap \mathit{FTV}(\Gamma) = \emptyset$, so $S\Gamma = \Gamma$. Also $x[x \mapsto v] = v$, hence $\Gamma \rhd x[x \mapsto v] : \tau'$.

Case $e = \lambda x'.e_1$. By **abs**, it follows from the assumption $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd \lambda x'.e_1 : \tau'$ that $\tau' = \tau_1 \to \tau_2$, and

$$\begin{aligned} &\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau][x' \mapsto \tau_1] \rhd e_1 : \tau_2 \\ \textit{i.e. } &\Gamma[x' \mapsto \tau_1][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_1 : \tau_2. \end{aligned}$$

Choose a substitution $S : \{\alpha_1, \ldots, \alpha_n\} \to \{\alpha'_1, \ldots, \alpha'_n\}$ such that $\alpha'_1, \ldots, \alpha'_n$, $\alpha_1, \ldots, \alpha_n$, and $\mathit{FTV}(\Gamma)$ are all distinct. Then

$$\begin{array}{lll} (2) & \Gamma[x' \mapsto S\tau_1][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_1 : S\tau_2 & \text{by Lemma 4.5,} \\ (3) & \Gamma[x' \mapsto S\tau_1] \rhd v : \tau & \text{by Lemma 4.1, and} \\ (4) & \mathit{FTV}(\Gamma[x' \mapsto S\tau_1]) \cap \{\alpha_1, \ldots, \alpha_n\} = \emptyset & \text{by the choice of } S. \end{array}$$

Thus, $\Gamma[x' \mapsto S\tau_1] \rhd e_1[x \mapsto v] : S\tau_2$ by the inductive hypothesis with (2), (3), and (4). Since S is a bijection, S^{-1} exists, hence

$$\begin{aligned} &S^{-1}(\Gamma[x' \mapsto S\tau_1]) \rhd e_1[x \mapsto v] : S^{-1}(S\tau_2) \quad \text{by Lemma 4.5,} \\ \textit{i.e. } &\Gamma[x' \mapsto \tau_1] \rhd e_1[x \mapsto v] : \tau_2. \end{aligned}$$

But then

$$\begin{aligned} &\Gamma \rhd \lambda x'.e_1[x \mapsto v] : \tau_1 \to \tau_2 \quad \text{by } \textbf{abs}, \\ \textit{i.e. } &\Gamma \rhd (\lambda x'.e_1)[x \mapsto v] : \tau_1 \to \tau_2. \end{aligned}$$

Case $e = (e_1\ e_2)$. From $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd (e_1\ e_2) : \tau'$ by the first premise of **app**

$$\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_1 : \tau_1 \to \tau'$$

(5) $\quad \Gamma \rhd e_1[x \mapsto v] : \tau_1 \to \tau'$ by ind. hyp.

By the second premise of **app**

$$\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_2 : \tau_1$$

(6) $\quad \Gamma \rhd e_2[x \mapsto v] : \tau_1$ by ind. hyp.

Then by **app** with (5) and (6)

$$\Gamma \rhd (e_1[x \mapsto v]\ e_2[x \mapsto v]) : \tau'$$
$$\textit{i.e.}\ \Gamma \rhd (e_1\ e_2)[x \mapsto v] : \tau'.$$

Case $e = \textsf{let}\ x'\ \textsf{be}\ e_1\ \textsf{in}\ e_2$. By the first premise of **let**

$$\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_1 : \tau_1$$

(7) $\quad \Gamma \rhd e_1[x \mapsto v] : \tau_1$ by ind. hyp.

By the second premise of **let**

$$\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau][x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd e_2 : \tau'$$

(8) $\quad \textit{i.e.}\ \Gamma[x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_2 : \tau'.$

Since $\Gamma \rhd v : \tau$

(9) $\quad \Gamma[x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd v : \tau$ by Lemma 4.1.

Now

$$\begin{aligned}
& \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\Gamma[x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])]) \\
\subseteq\ & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(\mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau]))) \\
=\ & \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])) \\
=\ & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(\tau_1) \setminus (\mathit{FTV}(\tau_1) \setminus \mathit{FTV}(\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau]))) \\
=\ & \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\tau_1) \cap \mathit{FTV}(\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau]) \\
\subseteq\ & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(\forall\alpha_1 \ldots \alpha_n.\tau)) \\
=\ & \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\forall\alpha_1 \ldots \alpha_n.\tau) \\
=\ & \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}) \\
=\ & \emptyset
\end{aligned}$$

so by the inductive hypothesis with (8) and (9)

$$\Gamma[x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd e_2[x \mapsto v] : \tau'$$

$\Gamma[x' \mapsto \mathit{Close}(\tau_1, \Gamma)] \rhd e_2[x \mapsto v] : \tau'$ by Lemma 4.6,

$\Gamma \rhd \textsf{let}\ x'\ \textsf{be}\ v_1[x \mapsto v]\ \textsf{in}\ e_2[x \mapsto v] : \tau'$ by **let** with (7),

$\textit{i.e.}\ \Gamma \rhd (\textsf{let}\ x'\ \textsf{be}\ v_1\ \textsf{in}\ e_2)[x \mapsto v] : \tau'.$ ∎

Lemma 4.5 establishes that typing is stable under substitution.

Lemma 4.5 *If* $\Gamma \rhd e : \tau$ *and* S *is a substitution then* $S\Gamma \rhd e : S\tau$.

Proof. The proof proceeds by induction on the length of the proof of $\Gamma \rhd e : \tau$ and case analysis on the last step. The proof is an adaptation of Tofte's proof of a similar lemma [37: lemma 4.2]. ∎

Finally, Lemma 4.6 shows that generalizing the type of a variable in the type environment has no impact on the conclusion of a deduction.

Lemma 4.6 *If* $\Gamma[x \mapsto \sigma] \rhd e : \tau$ *and* $\sigma' \succ \sigma$ *then* $\Gamma[x \mapsto \sigma'] \rhd e : \tau$.

Proof. The proof proceeds by induction on the length of the proof of $\Gamma[x \mapsto \sigma] \rhd e : \tau$ and case analysis on the last step. The proof is an adaptation of Damas and Milner's proof of a similar lemma [5: lemma 1]. The only interesting case is the case for **let**-expressions.

Case $e = \textsf{let}\ x'\ \textsf{be}\ e_1\ \textsf{in}\ e_2$. By the first premise of **let**

$$\begin{array}{lll} & \Gamma[x \mapsto \sigma] \rhd e_1 : \tau_1 & \\ (10) & \Gamma[x \mapsto \sigma'] \rhd e_1 : \tau_1 & \text{by ind. hyp.} \end{array}$$

By the second premise of **let**

$$\begin{array}{lll} & \Gamma[x \mapsto \sigma][x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma])] \rhd e_2 : \tau & \\ & \Gamma[x \mapsto \sigma'][x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma])] \rhd e_2 : \tau & \text{by ind. hyp.} \\ (11) & \Gamma[x \mapsto \sigma'][x' \mapsto \mathit{Close}(\tau_1, \Gamma[x \mapsto \sigma'])] \rhd e_2 : \tau & \text{by ind. hyp.} \end{array}$$

Then $\Gamma[x \mapsto \sigma'] \rhd \textsf{let}\ x'\ \textsf{be}\ e_1\ \textsf{in}\ e_2 : \tau$ by **let** with (10) and (11). ∎

A corollary of Subject Reduction is that standard reduction steps preserve type.

Corollary 4.7 *If* $\Gamma \rhd e_1 : \tau$ *and* $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$ *then* $\Gamma \rhd e_2 : \tau$.

Proof. First, if $e_1 \longmapsto e_2$ then $e_1 = E[e]$ and $e_2 = E[e']$ and $e \longrightarrow e'$, so $\Gamma \rhd e_2 : \tau$ by the Replacement Lemma. Then the result follows by induction on the length of the reduction sequence $e_1 \longmapsto\!\!\!\!\!\rightarrow e_2$. ∎

Subject reduction ensures that if we start with a typable expression, we cannot reach an untypable expression through any sequence of reductions. This by itself, however, does not yield type soundness. Subject reduction simply ensures that any properties implied by typability are preserved by reduction. The critical property we seek is that evaluation of a typable expression cannot get *stuck*.

Definition 4.8 (Stuck Expressions) *The evaluation of an expression* e *is* stuck *if* e *is not a value and there is no* e' *such that* $e \longmapsto e'$.

Of course, whether an expression eventually reduces to a stuck expression is not a decidable property. We approximate the set of expressions that become stuck with a set of *faulty* expressions.

Definition 4.9 (Faulty Expressions) *The* faulty *expressions of Functional ML are the expressions containing a subexpression* $(c\ v)$ *where* $\delta(c, v)$ *is undefined.*

The idea is that any faulty expression *may* become stuck, *i.e.*, the property "reduces to a faulty expression" is a conservative approximation to "reduces to an expression that is stuck". Thus the faulty expressions are a superset of the stuck expressions. For example, the expression $((\lambda \mathsf{y}.2)\ (\lambda \mathsf{x}.+\ 1\ \mathsf{true}))$ is faulty because of the subexpression $(+\ 1\ \mathsf{true})$, but is not stuck because it reduces to 2.

The behavior of evaluation is summarized by the following lemma. Let $e\Uparrow$ mean that e diverges, *i.e.*, $e \longmapsto e'$ for some e', and for all e' such that $e \longmapsto\!\!\!\!\rightarrow e'$, there exists e'' such that $e' \longmapsto e''$.

Lemma 4.10 (Uniform Evaluation) *For closed* e*, if there is no* e' *such that* $e \longmapsto\!\!\!\!\rightarrow e'$ *and* e' *is faulty, then either* $e\Uparrow$ *or* $e \longmapsto\!\!\!\!\rightarrow v$.

Proof. By induction on the length of the reduction sequence, we need only show that either e is faulty, $e \longmapsto e'$ and e' is closed, or e is a value. From the definition of $\longmapsto$, $e \longmapsto e'$ iff $e = E[e_1]$, $e' = E[e_1']$, and $e_1 \longrightarrow e_1'$. (Recall that $E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ e$.) We proceed by induction on the structure of e.

Case $e = c$, Y, or $\lambda x.e$. Then e is a value.

Case $e = x$. Since e is closed, this case cannot occur.

Case $e = \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2$. By the inductive hypothesis with e_1, there are three cases to consider. If e_1 is faulty, then $\mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2$ is faulty. If $e_1 = E_1[e']$ and $e' \longrightarrow e''$ then $e = E[e']$ where $E = \mathsf{let}\ x\ \mathsf{be}\ E_1\ \mathsf{in}\ e_2$; thus $e \longmapsto E[e'']$. Otherwise, if e_1 is a value, then $\mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 \longmapsto e_2[x \mapsto e_1]$.

Case $e = (e_1\ e_2)$. By the inductive hypothesis with e_1, there are three cases to consider. If e_1 is faulty, then $(e_1\ e_2)$ is faulty. If $e_1 = E_1[e']$ and $e' \longrightarrow e''$ then $e = E[e']$ where $E = (E_1\ e_2)$; thus $e \longmapsto E[e'']$. Otherwise, e_1 is a value. By the inductive hypothesis with e_2, there are three subcases to consider. If e_2 is faulty, then $(e_1\ e_2)$ is faulty. If $e_2 = E_2[e']$ and $e' \longrightarrow e''$ then $e = E[e']$ where $E = (e_1\ E_2)$; thus $e \longmapsto E[e'']$. Otherwise, both e_1 and e_2 are values, and the possibilities can be summarized as follows:

${e_1}^{e_2}$	c	$\lambda x.e$	Y	x
c	$\longmapsto$ *or* faulty	$\longmapsto$ *or* faulty	$\longmapsto$ *or* faulty	$\times$
$\lambda x.e$	$\longmapsto$	$\longmapsto$	$\longmapsto$	$\times$
Y	$\longmapsto$	$\longmapsto$	$\longmapsto$	$\times$
x	$\times$	$\times$	$\times$	$\times$

Cases marked with $\times$ cannot occur because the expression is not closed. Cases marked with $\longmapsto$ indicate that the combination reduces with $E = [\,]$. ∎

To relate faulty expressions and typability, we show that no faulty expression is typable.

Main Lemma 4.11 (Faulty Expressions are Untypable) *If e is faulty, there are no Γ, τ such that $\Gamma \rhd e : \tau$.*

Proof. It suffices to show that the subexpressions of e that cause e to be faulty are untypable. In general, we proceed by case analysis according to the form of the subexpression, but for Functional ML there is only one case.

Suppose $(c\ v)$ is faulty and $\Gamma \rhd c\ v : \tau$. By **app**, $\Gamma \rhd c : \tau' \to \tau$ and $\Gamma \rhd v : \tau'$. Then by the δ-typability condition, $\delta(c, v)$ is defined, but this contradicts the assumption that $(c\ v)$ is faulty. ∎

Since the faulty expressions are untypable and we have type preservation, the property "does not reduce to a faulty expression" is implied by typability.[4]

Theorem 4.12 (Syntactic Soundness) *If $\rhd e : \tau$ then either $e \Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\rhd v : \tau$.*

Proof. By Uniform Evaluation, either $e \longmapsto\!\!\!\!\!\rightarrow e'$ and e' is faulty, or $e \Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$. Since $\rhd e : \tau$, Subject Reduction implies $\rhd v : \tau$ and $\rhd e' : \tau$. Suppose $e \longmapsto\!\!\!\!\!\rightarrow e'$ and e' is faulty. Since faulty expressions are untypable, $\rhd e' : \tau$ is a contradiction, therefore this case cannot occur. Hence either $e \Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$ and $\rhd v : \tau$. ∎

To state strong and weak soundness theorems, we must have a definition of evaluation that differentiates between programs that diverge and those that cause type errors. Let *answers* (a) be values or the special result WRONG, and define *eval'* as:

$$(eval') \qquad eval'(e) = \begin{cases} \text{WRONG} & \text{if } e \longmapsto\!\!\!\!\!\rightarrow e' \text{ and } e' \text{ is faulty;} \\ v & \text{if } e \longmapsto\!\!\!\!\!\rightarrow v. \end{cases}$$

Programs that cause a type error return the special answer WRONG. Strong and weak soundness are now corollaries of Syntactic Soundness.

Theorem 4.13 (Strong Soundness) *If $\rhd e : \tau$ and $eval'(e) = a$ then $\rhd a : \tau$.*

Theorem 4.14 (Weak soundness) *If $\rhd e : \tau$ then $eval'(e) \neq$ WRONG.*

5 References and Exceptions

Besides functional facilities, first-class references and exceptions are the most important core programming facilities of STANDARD ML and similar languages. In this section we consider several extensions to Functional ML. The first subsection deals with Reference ML, an extension of Functional ML with references. It uses our previous work on a calculus of state, λ_v-S [12, 14], and in particular its cell-oriented variant [3]. The second subsection addresses Exception ML, an extension of Functional ML with exceptions. It uses a modified version of our control calculus, λ_v-C [13, 14], especially its fragment with prompts [9]. The final subsection presents Core ML, which includes both references and exceptions.

[4]Type preservation by itself does not guarantee type soundness. Consider the (trivial) type system where every type is a subtype of every other type. Type preservation holds since *every* expression can be assigned every type. However, the system does not prevent type errors, because it rejects no expressions.

5.1 Reference ML

Extending the λ_v-calculus to a calculus of functions and references requires extending the syntax with a new kind of expression and several new values (underlined):

(*Expressions*)	$e ::= v \mid e_1\ e_2 \mid \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 \mid \underline{\rho\theta.e}$
(*Values*)	$v ::= c \mid x \mid \textsf{Y} \mid \lambda x.e \mid \underline{\textsf{ref}} \mid \underline{!} \mid \underline{:=} \mid \underline{:=\ v}$
	$\underline{\theta} ::= \underline{\{\,\langle x, v\rangle\,\}^*}$

The expression $\rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ binds $x_1, \ldots, x_n$ in $v_1, \ldots, v_n$ and e. Above, θ represents a finite function from variables to values, *i.e.*, we treat θ as a set of pairs whose first components are distinct. We also identify all ρ-expressions that differ only by a consistent renaming of bound variables.

The values ref, !, and := are the familiar operations of ML.[5] The application of ref to a value creates a reference cell containing that value. The application of ! to a cell returns the value contained in that cell. The binary assignment operator := evaluates both its operands, the first of which must evaluate to a cell, and assigns the value of the second operand to that cell. Since all operations are curried, the application of the assignment operator to a variable *is* a value. Specifically, the expression $(:=\ c)$ may be thought of as a *capability* to assign to the cell c. The ρ-expression is semantically an abbreviation of a let-expression:

$$\begin{array}{rl} & \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e \\ \equiv & \textsf{let}\ x_1\ \textsf{be ref}\ u_1\ \textsf{in} \\ & \quad \ldots \\ & \quad \textsf{let}\ x_n\ \textsf{be ref}\ u_n\ \textsf{in} \\ & \qquad (\lambda y_1 \ldots y_n y_{n+1}.\, y_{n+1})\,(:=\ x_1\ v_1)\ldots(:=\ x_n\ v_n)\ e \end{array}$$

where $u_1, \ldots, u_n$ are arbitrary values (of the right type). In order to simplify the semantics, we do not treat it as such [14]. Intuitively, a ρ-expression plays the rôle of a store fragment during the reduction of a program with imperative assignment statements. A reference cell is represented as a variable; the pair $\langle x, v\rangle$ in a ρ-expression indicates that reference cell x contains value v.

5.1.1 Semantics

In addition to the reductions for $\mathbf{v}$, the reductions for references are:[6]

(*ref*)	$\textsf{ref}\ v \longrightarrow \rho\langle x, v\rangle.x$
(*deref*)	$\rho\theta\langle x, v\rangle.R[!\ x] \longrightarrow \rho\theta\langle x, v\rangle.R[v]$
(*assign*)	$\rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] \longrightarrow \rho\theta\langle x, v_2\rangle.R[v_2]$
(ρ_{merge})	$\rho\theta_1.\rho\theta_2.e \longrightarrow \rho\theta_1\theta_2.e$
(ρ_{lift})	$R[\rho\theta.e] \longrightarrow \rho\theta.R[e]$ if $R \neq []$

[5]Introducing ref, !, and := as values rather than special forms like (ref e), (! e), and $(:=\ e_1\ e_2)$ simplifies the proofs of several lemmas that proceed by induction on the structure of expressions. The latter approach requires duplicating much of the work of the case for applications.

[6]This semantics is due to Crank and Felleisen [3], which was derived from Felleisen and Hieb's work [14]. A similar definition of evaluation for dealing with state using rewriting techniques also appears in the work of Mason and Talcott [19].

By the variable conventions, x is not free in v in the *ref* reduction; the domains of θ_1 and θ_2 are disjoint in ρ_{merge}; and the free variables of R are disjoint from the domain of θ in ρ_{lift}. We refer to the union of these five reductions as $\mathbf{r}$. The notions of reduction $\mathbf{r}$ and $\mathbf{v}$ (adapted *mutatis mutandis* to the full syntax) form the basis for a calculus of functions and references. We refer to the union as $\mathbf{vr}$, and write $\xrightarrow{\mathbf{vr}}$ for a reduction in $\mathbf{vr}$. The extended notion of reduction gives rise to a system of reductions and equations with the extension of contexts to:

$$C ::= [\,] \mid C\ e \mid e\ C \mid \mathsf{let}\ x\ \mathsf{be}\ C\ \mathsf{in}\ e \mid \mathsf{let}\ x\ \mathsf{be}\ e\ \mathsf{in}\ C \mid \lambda x.C \mid \rho\theta.C \mid \rho\theta\langle x, C\rangle.e.$$

The calculus satisfies the same basic properties as the λ_v-calculus [3].

The definition of $\mathbf{r}$ relies on a set of *R contexts*:

$$R ::= [\,] \mid R\ e \mid v\ R \mid \mathsf{let}\ x\ \mathsf{be}\ R\ \mathsf{in}\ e.$$

The use of R contexts in the new reductions reflects the additional sequencing restrictions that the introduction of side-effects in a programming language requires for the semantics to be deterministic. In particular, the creation, the dereferencing, and the updating operations on a reference cell must be ordered in a linear fashion, which implies some further ordering among the operations on distinct cells. The R contexts build this minimal order of evaluation into those reductions that refer to reference cells and their operations; following ML, we choose to perform operations from left to right.

For $\mathbf{r}$ reductions, a cell is represented by an ordinary variable appearing in the domain of a store fragment θ. When a program creates a new cell via ref, the *ref* reduction introduces a ρ-expression that records the cell's current value. The dereference operator ! selects a cell's value from the *closest* ρ-expression (relative to R contexts). If the appropriate cell is bound in a ρ-expression that is not the closest one, the intervening ρ-expressions must first be merged with the outer one. This is accomplished with ρ_{lift} and ρ_{merge} reductions, which lift a partial store out of R contexts, and merge it with outer stores (by the variable convention, cells are renamed appropriately to avoid collisions). An assignment replaces a cell's current value in the closest ρ-expression, after performing all necessary lift and merge steps, and returns the assigned value.

For example, consider the following expression:

$$\begin{array}{l}((\mathsf{let\ x\ be\ ref}\ (\lambda \mathsf{x}.+\ !\mathsf{x}\ 2)\ \mathsf{in}\ :=\ \mathsf{x}\ (\lambda \mathsf{x}.+\ !\mathsf{x}\ 3))\\ \ (\mathsf{let\ x\ be\ ref}\ 2\ \mathsf{in}\ (\lambda \mathsf{z.x})(:=\ \mathsf{x}\ 4)))\end{array}$$

In this expression, the two assignments can happen in an arbitrary order, and the system of reductions admits both possibilities. Both assignments, however, must happen before the outermost application is reduced. Figure 2 gives one possible reduction sequence.

The above calculus only constrains the relative ordering of *ref*, *deref*, and *assign* reductions. By defining extended evaluation contexts:

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ e \mid \rho\theta.E$$

we fix the order in which reductions in $\mathbf{v}$ and *ref*, *deref*, and *assign* may be applied. The stepping function $\overset{\mathbf{vr}}{\longmapsto}$ is defined as before. To show that $\overset{\mathbf{vr}}{\longmapsto}$ is a function requires a simple

$$\begin{array}{lll}
 & ((\mathsf{let\ x\ be\ ref\ } (\lambda \mathsf{x}.+\ !\mathsf{x}\ 2)\ \mathsf{in}\ :=\ \mathsf{x}\ (\lambda \mathsf{x}.+\ !\mathsf{x}\ 3)) & \\
 & \ (\mathsf{let\ x\ be\ ref\ } 2\ \mathsf{in}\ (\lambda \mathsf{z.x})(:=\ \mathsf{x}\ 4))) & \\
\twoheadrightarrow & (\rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 2\rangle.:=\ \mathsf{x}\ (\lambda \mathsf{x}.+\ !\mathsf{x}\ 3))\ \ (\rho\langle \mathsf{x}, 2\rangle.(\lambda \mathsf{z.x})(:=\ \mathsf{x}\ 4)) & 2 \times \mathit{ref}, \rho_{lift}, \mathit{let} \\
\twoheadrightarrow & (\rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle.\lambda \mathsf{x}.+\ !\mathsf{x}\ 3)\ \ (\rho\langle \mathsf{x}, 2\rangle.(\lambda \mathsf{z.x})(:=\ \mathsf{x}\ 4)) & \mathit{assign} \\
\twoheadrightarrow & (\rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle.\lambda \mathsf{x}.+\ !\mathsf{x}\ 3)\ \ (\rho\langle \mathsf{x}, 4\rangle.\mathsf{x}) & \mathit{assign}, \beta_{\mathsf{v}} \\
\twoheadrightarrow & \rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle\langle \mathsf{y}, 4\rangle.(\lambda \mathsf{x}.+\ !\mathsf{x}\ 3)\ \mathsf{y} & 2 \times \rho_{lift}, \rho_{merge} \\
\twoheadrightarrow & \rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle\langle \mathsf{y}, 4\rangle.+\ !\mathsf{y}\ 3 & \beta_{\mathsf{v}} \\
\twoheadrightarrow & \rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle\langle \mathsf{y}, 4\rangle.+\ 4\ 3 & \mathit{deref} \\
\twoheadrightarrow & \rho\langle \mathsf{x}, \lambda \mathsf{x}.+\ !\mathsf{x}\ 3\rangle\langle \mathsf{y}, 4\rangle.7 & \delta
\end{array}$$

Figure 2: Reducing a program with side-effects

diamond theorem, since the order in which ρ_{lift} and ρ_{merge} reductions happen is not fixed, and the context R in the ρ_{lift} rule is selected in a nondeterministic fashion. The answers returned by evaluation of a program are no longer simply closed values, since an answer may be a reference cell. Answers are values or ρ-expressions enclosing values:

$$(\mathit{answers}) \qquad\qquad a ::= v \mid \rho\theta.v.$$

Based on these definitions, evaluation may be defined like before:

$$(\mathit{eval}_{\mathbf{vr}}) \qquad\qquad \mathit{eval}_{\mathbf{vr}}(e) = a \quad \text{iff} \quad e \overset{\mathbf{vr}}{\longmapsto\!\!\!\!\rightarrow} a.$$

5.1.2 Typing

Typing reference cells in the presence of polymorphism is not straightforward, as the obvious solution is unsound [6, 36, 37]. To assign types to reference cells, we extend the set of types with an additional constructor:

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \underline{\tau\ \mathit{ref}}$$

The type τ *ref* is the type of cells containing a value of type τ.

The obvious types for the three operators are indicated by their semantics:

$$\begin{array}{l}
\Gamma \rhd \mathsf{ref} : \tau \rightarrow \tau\ \mathit{ref} \\
\Gamma \rhd\ !\ \ \ : \tau\ \mathit{ref} \rightarrow \tau \\
\Gamma \rhd := : \tau\ \mathit{ref} \rightarrow \tau \rightarrow \tau
\end{array}$$

For example, the first argument of the assignment operator (:=) must be a reference cell. The second argument must be a value matching the type of the cell. The result is of the same type as the second argument since our assignment operator returns the value assigned. The following expression illustrates why this typing is unsound:

$$\begin{array}{l}
\mathsf{let\ f\ be\ ref}\ (\lambda \mathsf{y.y})\ \mathsf{in} \\
\quad (\lambda \mathsf{z}.\ !\ \mathsf{f\ true})\ (:=\ \mathsf{f}\ (\lambda \mathsf{n}.+\ 1\ \mathsf{n}))
\end{array}$$

If the type of the reference bound to x is generalized to $\forall\alpha.(\alpha \rightarrow \alpha)$ *ref*, then the reference cell can be treated as having type (*int* $\rightarrow$ *int*) *ref* for the assignment, and as having type (*bool* $\rightarrow$ *bool*) *ref* for the dereference, hence this expression is typable as *bool*. But evaluating this expression leads to a type error:

$$\begin{array}{ll}
& \textsf{let f be ref } (\lambda \textsf{y.y}) \textsf{ in} \\
& \quad (\lambda \textsf{z. ! f true}) \; (\textsf{:= f } (\lambda \textsf{n.+ 1 n})) \\
\twoheadrightarrow & \textsf{let f be } \rho\langle \textsf{x}, \lambda \textsf{y.y}\rangle.\ \textsf{x in} \\
& \quad (\lambda \textsf{z. ! f true}) \; (\textsf{:= f } (\lambda \textsf{n.+ 1 n})) \\
\twoheadrightarrow & \rho\langle \textsf{x}, \lambda \textsf{y.y}\rangle. \\
& \quad (\lambda \textsf{z. ! x true}) \; (\textsf{:= x } (\lambda \textsf{n.+ 1 n})) \\
\twoheadrightarrow & \rho\langle \textsf{x}, \lambda \textsf{n.+ 1 n}\rangle.\ \textsf{! x true} \\
\twoheadrightarrow & \rho\langle \textsf{x}, \lambda \textsf{n.+ 1 n}\rangle.\ \textsf{+ 1 true}
\end{array}$$

As Tofte [36, 37] points out, the problem with the obvious typing is the generalization of type variables that appear free in the type of a value in the store.

The central idea of Tofte's solution is to ensure that the only storable values are those that will not be used polymorphically at run-time. To this end, type variables are partitioned into *imperative* and *applicative* variables: the former set of type variables is named *ImpTypeVar*; the latter *AppTypeVar*. Types are classified accordingly: an *imperative type* cannot contain any applicative type variables. Only values of imperative type can be stored. When the type of a value is generalized in a let-expression, type variables that might appear in the types of values in the store are required to be imperative, and are not generalized.

The notion of generalization requires appropriate adaptation. We still have:

$$(\succ) \qquad \forall\alpha_1 \ldots \alpha_n.\tau' \succ \tau \;\text{ if }\; S\tau' = \tau \text{ and } \mathrm{Dom}(S) = \{\alpha_1, \ldots, \alpha_n\} \text{ for some } S,$$

but the substitution S is required to map imperative type variables to imperative types. Thus, if $\underline{\alpha}$ is an imperative type variable, we have:

$$\begin{array}{rcl}
\forall\underline{\alpha}.\underline{\alpha} \rightarrow \underline{\alpha} & \succ & \mathit{int} \rightarrow \mathit{int} \\
\text{and } \forall\underline{\alpha}.\underline{\alpha} \rightarrow \underline{\alpha} & \succ & \underline{\alpha} \rightarrow \underline{\alpha} \\
\text{but not } \forall\underline{\alpha}.\underline{\alpha} \rightarrow \underline{\alpha} & \succ & \alpha \rightarrow \alpha
\end{array}$$

since α is potentially applicative (the meta-variable α still represents *any* type variable—imperative or applicative).

Like all other techniques for typing references, Tofte's system requires modifying the typing rule for let-expressions. The let rule is split into two rules, according to whether or not the right-hand side of the declaration is a value. Figure 3 gives the new rules. The second rule does not generalize over imperative type variables, and thus will not generalize the type of a value in the store. However, if the expression bound by a let-expression *is* a value, as in the first rule, then its evaluation cannot create a new cell, so generalization of imperative type variables in its type cannot generalize the type of a value in the store. Tofte uses the terminology *expansive* and *non-expansive* to denote this syntactic classification of expressions into those that may create new references, and those that definitely do not create new references. Like Tofte, we classify only values as non-expansive, but a stronger type system is possible by classifying more expressions as non-expansive [36, 37].

$$(\mathbf{let}_v)\qquad \frac{\Gamma \triangleright v : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \triangleright e : \tau_2}{\Gamma \triangleright \mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e : \tau_2}$$

$$(\mathbf{let}_e)\qquad \frac{\Gamma \triangleright e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{AppClose}(\tau_1, \Gamma)] \triangleright e_2 : \tau_2 \quad e_1 \notin \mathit{Values}}{\Gamma \triangleright \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\mathit{Close}(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma)$$
$$\mathit{AppClose}(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1, \ldots, \alpha_n\} = (\mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma)) \cap \mathit{AppTypeVar}.$$

Figure 3: Modified **let**-expression typing rules

$$(\mathbf{ref})\qquad \Gamma \triangleright \mathsf{ref} : \tau \to \tau\ \mathit{ref} \ \text{ if } \ \tau \text{ is imperative}$$
$$(\mathbf{deref})\qquad \Gamma \triangleright\ ! : \tau\ \mathit{ref} \to \tau$$
$$(\mathbf{assign})\qquad \Gamma \triangleright := : (\tau\ \mathit{ref} \to \tau \to \tau)$$

$$(\mathbf{rho})\qquad \frac{\begin{array}{l}\Gamma[x_1 \mapsto \tau_1\ \mathit{ref}] \ldots [x_n \mapsto \tau_n\ \mathit{ref}] \triangleright e : \tau \\ \Gamma[x_1 \mapsto \tau_1\ \mathit{ref}] \ldots [x_n \mapsto \tau_n\ \mathit{ref}] \triangleright v_i : \tau_i \qquad \tau_i \text{ is imperative} \quad 1 \le i \le n\end{array}}{\Gamma \triangleright \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau}$$

Figure 4: Additional typing rules for references

Figure 4 gives four additional typing rules for reference cells. The typing rules for **ref** and ρ-expressions ensure that any value placed in the store has an imperative type. The typing rules for ! and := do not need to be explicitly constrained, since their use is implicitly constrained by the type of a value already in the store.

In addition to the restrictions placed on the typing of constants by Functional ML, we also require that there be no constants of reference type ($\tau\ \mathit{ref}$). This ensures that all values of reference type are in fact reference cells, and can be assigned or dereferenced. This restriction is used in the proof of Theorem 5.6.

The new system assigns the same types as the old one to Functional ML expressions. Let $\overset{f}{\triangleright}$ indicate provability in the old, functional system, and let $\overset{r}{\triangleright}$ indicate provability in the system with references.

Proposition 5.1 *If e is an expression in Functional ML, and Γ and τ are purely applicative, then $\Gamma \overset{f}{\triangleright} e : \tau$ iff $\Gamma \overset{r}{\triangleright} e : \tau$.*

Since Functional ML expressions do not contain **ref** or ρ-expressions, no imperative type variables need be chosen to type an expression, and *Close* and *AppClose* yield the same type scheme for applicative types.

5.1.3 Type Soundness for References

To prove type preservation for the extended language, we must show that the notion of reduction **v** extended to the new syntax preserves types, and that the reductions of **r** preserve types. Showing that the extended **v** preserves types amounts to re-proving the various lemmas used in section 4.3 for the extended system. In general, we refer to the extension of Lemma 4.n as Lemma 4.n^+. Showing that **r** preserves types involves adding a case for each reduction to the proof of Subject Reduction. The cases involving R contexts use the fact that:

$$\text{if } \Gamma \triangleright R[e] : \tau \text{ then } \Gamma \triangleright e : \tau'.$$

The type environment Γ is the same in both the antecedent and consequent since R contexts do not bind variables.

Main Lemma 5.2 (Subject Reduction for vr) *If* $\Gamma \overset{\tau}{\triangleright} e_1 : \tau$ *and* $e_1 \xrightarrow{\mathbf{vr}} e_2$ *then* $\Gamma \overset{\tau}{\triangleright} e_2 : \tau$.

Proof. The cases for the reductions of **v** are the same as before, using an appropriately extended Replacement Lemma and a similarly extended Lemma 4.1.

Case $\mathsf{ref}\ v \longrightarrow \rho\langle x, v\rangle.x$. From $\Gamma \triangleright \mathsf{ref}\ v : \tau$, where $\tau = \tau'\ \mathit{ref}$ and τ' is imperative, $\Gamma \triangleright v : \tau'$ by **ref** and **app**, and $\Gamma[x \mapsto \tau'\ \mathit{ref}] \triangleright v : \tau'$ by Lemma 4.1$^+$. Since $\Gamma[x \mapsto \tau'\ \mathit{ref}] \triangleright x : \tau'\ \mathit{ref}$ by **var**, $\Gamma \triangleright \rho\langle x, v\rangle.x : \tau'\ \mathit{ref}$ by **rho**.

In order to satisfy the imperative restriction on the typing of ρ-expressions, ref applications must yield values of imperative type, as ensured by **ref**.

Case $\rho\theta\langle x, v\rangle.R[!\ x] \longrightarrow \rho\theta\langle x, v\rangle.R[v]$. From $\Gamma \triangleright \rho\theta\langle x, v\rangle.R[!\ x] : \tau$ by **rho**

$$(12) \qquad \Gamma' \triangleright v : \tau'$$

where τ' is imperative and $\Gamma' x = \tau' \mathit{ref}$. Thus $\Gamma' \triangleright x : \tau' \mathit{ref}$, hence $\Gamma' \triangleright !\ x : \tau'$ by **deref** and **app**. With (12) by the Replacement$^+$ Lemma, we have $\Gamma \triangleright \rho\theta\langle x, v\rangle.R[v] : \tau$.

Case $\rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] \longrightarrow \rho\theta\langle x, v_2\rangle.R[v_2]$. From $\Gamma \triangleright \rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] : \tau$ by **rho**

$$(13) \qquad \Gamma' \triangleright v_1 : \tau'$$

where τ' is imperative and $\Gamma' x = \tau'\ \mathit{ref}$. Thus $\Gamma' \triangleright x : \tau'\ \mathit{ref}$, and by **assign** and **app**

$$\begin{aligned} &\Gamma' \triangleright :=\ x : \tau' \to \tau' \\ &\Gamma' \triangleright :=\ x\ v_2 : \tau' \quad \text{and} \quad \Gamma' \triangleright v_2 : \tau' \qquad \text{by } \mathbf{app}. \end{aligned}$$

With (13) and using the Replacement$^+$ Lemma twice, we have $\Gamma \triangleright \rho\theta\langle x, v_2\rangle.R[v_2] : \tau$.

Case $\rho\theta_1.\rho\theta_2.e \longrightarrow \rho\theta_1\theta_2.e$. From $\Gamma \triangleright \rho\theta_1.\rho\theta_2.e : \tau$ by **rho**

$$(14) \qquad \Gamma' \triangleright \rho\theta_2.e : \tau \quad \text{and} \quad \Gamma' \triangleright v_i : \tau_i$$

where $\theta_1 = \langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle$, $\Gamma' = \Gamma[x_1 \mapsto \tau_1\ \mathit{ref}] \ldots [x_n \mapsto \tau_n\ \mathit{ref}]$, and $1 \le i \le n$. Again by **rho**

$$(15) \qquad \Gamma'' \triangleright e : \tau \quad \text{and} \quad \Gamma'' \triangleright v'_j : \tau'_j$$

where $\theta_2 = \langle x'_1, v'_1\rangle \ldots \langle x'_n, v'_m\rangle$, $\Gamma'' = \Gamma'[x'_1 \mapsto \tau'_1 \mathit{ref}] \ldots [x'_m \mapsto \tau'_m \mathit{ref}]$, and $1 \le j \le m$. Since $\mathrm{Dom}(\theta_1) \cap \mathrm{Dom}(\theta_2) = \emptyset$ (by the variable conventions),

$$\begin{aligned} &\Gamma'' \triangleright v_i : \tau_i \quad \text{for } 1 \le i \le n && \text{by (14)}, \\ &\Gamma \triangleright \rho\theta_1\theta_2.e : \tau && \text{by } \mathbf{rho} \text{ with (15)}. \end{aligned}$$

Case $R[\rho\theta.e] \longrightarrow \rho\theta.R[e]$. We have $\Gamma \triangleright R[\rho\theta.e] : \tau$, and proceed by induction on the structure of R to show $\Gamma \triangleright \rho\theta.R[e] : \tau$.

Case $R = []$. Then $R[\rho\theta.e] = \rho\theta.R[e]$.

Case $R = (R'\ e')$. Then $\Gamma \triangleright (R'[\rho\theta.e]\ e') : \tau$, and by **app**

$$\Gamma \triangleright R'[\rho\theta.e] : \tau' \to \tau \tag{16}$$
$$\text{and } \Gamma \triangleright e' : \tau'. \tag{17}$$

Then $\Gamma \triangleright \rho\theta.R'[e] : \tau' \to \tau$ by ind. hyp. with (16), and by **rho**

$$\Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright v_i : \tau_i \tag{18}$$
$$\text{and } \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright R'[e] : \tau' \to \tau \tag{19}$$

where $\theta = \langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle$ and $1 \le i \le n$. Since $x_1, \ldots, x_n \notin FV(e')$,

$$\begin{array}{ll} \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright e' : \tau' & \text{by Lemma 4.1}^+ \text{ with (17),} \\ \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright R'[e]\ e' : \tau & \text{by } \textbf{app} \text{ with (19),} \\ \Gamma \triangleright \rho\theta.R'[e]\ e' : \tau & \text{by } \textbf{rho} \text{ with (18).} \end{array}$$

Case $R = (v\ R')$. Similar to the previous case.

Case $R = \mathsf{let}\ x\ \mathsf{be}\ R'\ \mathsf{in}\ e'$. Then $\Gamma \triangleright \mathsf{let}\ x\ \mathsf{be}\ R'[\rho\theta.e]\ \mathsf{in}\ e' : \tau$, and by $\mathbf{let}_e$

$$\Gamma \triangleright R'[\rho\theta.e] : \tau' \tag{20}$$
$$\text{and } \Gamma[x \mapsto \mathit{AppClose}(\tau', \Gamma)] \triangleright e' : \tau. \tag{21}$$

Then $\Gamma \triangleright \rho\theta.R'[e] : \tau'$ by ind. hyp. with (20), and by **rho**

$$\Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright v_i : \tau_i \tag{22}$$
$$\text{and } \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright R'[e] : \tau' \tag{23}$$

where $\theta = \langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle$ and each τ_i is imperative for $1 \le i \le n$. Since $x_1, \ldots, x_n \notin FV(e')$,

$$\begin{array}{rll} & \Gamma[x \mapsto \mathit{AppClose}(\tau', \Gamma)][x_i \mapsto \tau_i\ \mathit{ref}] \triangleright e' : \tau & \text{by Lemma 4.1}^+ \text{ with (21)} \\ \textit{i.e.} & \Gamma[x_i \mapsto \tau_i\ \mathit{ref}][x \mapsto \mathit{AppClose}(\tau', \Gamma)] \triangleright e' : \tau. & \end{array}$$

But each τ_i is imperative, due to the restriction on the typing of ρ-expressions, hence

$$\mathit{AppClose}(\tau', \Gamma) = \mathit{AppClose}(\tau', \Gamma[x_i \mapsto \tau_i\ \mathit{ref}]).$$

If $R'[e]$ is expansive, then

$$\begin{array}{ll} \Gamma[x_i \mapsto \tau_i\ \mathit{ref}][x \mapsto \mathit{AppClose}(\tau', \Gamma[x_i \mapsto \tau_i\ \mathit{ref}])] \triangleright e' : \tau & \\ \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright \mathsf{let}\ x\ \mathsf{be}\ R'[e]\ \mathsf{in}\ e' : \tau & \text{by } \mathbf{let}_e \text{ with (23)} \\ \Gamma \triangleright \rho\theta.\mathsf{let}\ x\ \mathsf{be}\ R'[e]\ \mathsf{in}\ e' : \tau & \text{by } \textbf{rho} \text{ with (22).} \end{array}$$

Otherwise, $R'[e]$ is non-expansive, and

$$\begin{array}{ll} \Gamma[x_i \mapsto \tau_i\ \mathit{ref}][x \mapsto \mathit{Close}(\tau', \Gamma[x_i \mapsto \tau_i\ \mathit{ref}])] \triangleright e' : \tau & \\ \Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright \mathsf{let}\ x\ \mathsf{be}\ R'[e]\ \mathsf{in}\ e' : \tau & \text{by } \mathbf{let}_v \text{ with (23)} \\ \Gamma \triangleright \rho\theta.\mathsf{let}\ x\ \mathsf{be}\ R'[e]\ \mathsf{in}\ e' : \tau & \text{by } \textbf{rho} \text{ with (22).} \end{array}$$

Thus $\Gamma \triangleright \rho\theta.R[e] : \tau$ by induction. ∎

It remains to extend the proofs of the various lemmas. The only one that is not straightforward is the Substitution Lemma. As in the definition of generalization, substitutions S in the following proof are required to map imperative type variables to imperative types.

Lemma 5.3 (Substitution) *If* $\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau] \overset{\tau}{\triangleright} e : \tau'$ *and* $x \notin \mathrm{Dom}(\Gamma)$ *and* $\Gamma \overset{\tau}{\triangleright} v : \tau$ *and* $\{\alpha_1, \dots, \alpha_n\} \cap FTV(\Gamma) = \emptyset$ *then* $\Gamma \overset{\tau}{\triangleright} e[x \mapsto v] : \tau'$.

Proof. The cases from the proof for Functional ML are unchanged, with the exception that the **let**-expression case applies only when the bound expression is a value. There is one new case for expansive **let**-expressions, and new cases for the additional syntax.

Case $e =$ **ref**, !, or :=. In each case, $\Gamma' \triangleright e : \tau'$ for any Γ', hence $\Gamma \triangleright e : \tau'$. Also in each case, $e[x \mapsto v] = e$, so $\Gamma \triangleright e[x \mapsto v] : \tau'$.

Case $e = \rho\theta.e_1$. From $\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau] \triangleright \rho\theta.e_1 : \tau'$ by **rho**

$$\begin{aligned} &\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto \tau_i\ \mathit{ref}] \triangleright v_i : \tau_i \\ \text{and } &\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto \tau_i\ \mathit{ref}] \triangleright e_1 : \tau' \end{aligned}$$

where $\theta = \langle x_1, v_1\rangle \dots \langle x_m, v_m\rangle$, $x_1, \dots, x_m, x$ are distinct, and each τ_i is imperative for $1 \leq i \leq m$. Choose a substitution $S : \{\alpha_1, \dots, \alpha_n\} \to \{\alpha'_1, \dots, \alpha'_n\}$ such that $\alpha'_1, \dots, \alpha'_n$ are distinct type variables, $S\alpha_i = \alpha'_i$, and $\{\alpha'_1, \dots, \alpha'_n\} \cap (\{\alpha_1, \dots, \alpha_n\} \cup FTV(\Gamma) \cup FTV(\tau') \cup \bigcup_i FTV(\tau_i)) = \emptyset$. Then by Lemma 4.5$^+$

$$\begin{aligned} &S(\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto \tau_i\ \mathit{ref}]) \triangleright v_i : S\tau_i \\ \textit{i.e.}\ &\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto S\tau_i\ \mathit{ref}] \triangleright v_i : S\tau_i \\ \textit{i.e.}\ &\Gamma[x_i \mapsto S\tau_i\ \mathit{ref}][x \mapsto \forall\alpha_1 \dots \alpha_n.\tau] \triangleright v_i : S\tau_i. \end{aligned}$$

Also by Lemma 4.5$^+$,

$$\begin{aligned} &S(\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto \tau_i\ \mathit{ref}]) \triangleright e_1 : S\tau' \\ \textit{i.e.}\ &\Gamma[x \mapsto \forall\alpha_1 \dots \alpha_n.\tau][x_i \mapsto S\tau_i\ \mathit{ref}] \triangleright e_1 : S\tau' \\ \textit{i.e.}\ &\Gamma[x_i \mapsto S\tau_i\ \mathit{ref}][x \mapsto \forall\alpha_1 \dots \alpha_n.\tau] \triangleright e_1 : S\tau'. \end{aligned}$$

By applying the inductive hypothesis to each,

$$\begin{aligned} &\Gamma[x_i \mapsto S\tau_i\ \mathit{ref}] \triangleright v_i[x \mapsto v] : S\tau_i \\ \text{and } &\Gamma[x_i \mapsto S\tau_i\ \mathit{ref}] \triangleright e_1[x \mapsto v] : S\tau'. \end{aligned}$$

Since S is a bijection, S^{-1} exists; hence

$$\begin{aligned} &\Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright v_i[x \mapsto v] : \tau_i \\ \text{and } &\Gamma[x_i \mapsto \tau_i\ \mathit{ref}] \triangleright e_1[x \mapsto v] : \tau'. \end{aligned}$$

Then $\Gamma \triangleright \rho\theta[x \mapsto v].e_1[x \mapsto v] : \tau'$ by **rho**, thus $\Gamma \triangleright (\rho\theta.e_1)[x \mapsto v] : \tau'$.

Case $e = \textsf{let } x' \textsf{ be } e_1 \textsf{ in } e_2$ where $e_1 \notin \mathit{Values}$. By the first premise of $\mathbf{let}_e$

$$\begin{array}{lll} & \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_1 : \tau_1 & \\ (24) & \Gamma \rhd e_1[x \mapsto v] : \tau_1 & \text{by ind. hyp.} \end{array}$$

By the second premise of $\mathbf{let}_e$

$$\begin{array}{l} \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau][x' \mapsto \mathit{AppClose}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd e_2 : \tau' \\ \textit{i.e. } \Gamma[x' \mapsto \mathit{AppClose}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_2 : \tau'. \end{array}$$

Choose a substitution $S : \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{ImpTypeVar} \rightarrow \{\alpha'_1, \ldots, \alpha'_m\}$ such that $\alpha'_1, \ldots, \alpha'_m$ are distinct imperative type variables, S is a bijection, and $\{\alpha'_1, \ldots, \alpha'_m\} \cap (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(\tau) \cup \{\alpha_1, \ldots, \alpha_n\}) = \emptyset$. Then by Lemma 4.5$^+$

$$\begin{array}{ll} & S(\Gamma[x' \mapsto \mathit{AppClose}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau]) \rhd e_2 : S\tau' \\ & \textit{i.e. } \Gamma[x' \mapsto S\mathit{AppClose}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_2 : S\tau' \\ (25) & \textit{i.e. } \Gamma[x' \mapsto \mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])][x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \rhd e_2 : S\tau' \end{array}$$

since the domain of S contains only imperative type variables. From $\Gamma \rhd v : \tau$ since x' is not free in v

$$(26) \qquad \Gamma[x' \mapsto \mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd v : \tau.$$

Now,

$$\begin{array}{l} \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\Gamma[x' \mapsto \mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])]) \\ \subseteq \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(\mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau]))) \\ = \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])) \\ = \{\alpha_1, \ldots, \alpha_n\} \cap \mathit{FTV}(\forall\alpha''_1 \ldots \alpha''_m.S\tau_1) \\ = \{\alpha_1, \ldots, \alpha_n\} \cap (\mathit{FTV}(S\tau_1) \setminus \{\alpha''_1, \ldots, \alpha''_m\}) \\ = \emptyset \end{array}$$

where $\{\alpha''_1, \ldots, \alpha''_m\} = (\mathit{FTV}(S\tau_1) \setminus \mathit{FTV}(\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])) \cap \mathit{AppTypeVar}$. By the inductive hypothesis with (25) and (26), it follows that

$$\Gamma[x' \mapsto \mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd e_2[x \mapsto v] : S\tau'.$$

Since S is a bijection, S^{-1} exists, so by Lemma 4.5$^+$

$$\begin{array}{l} S^{-1}(\Gamma[x' \mapsto \mathit{AppClose}(S\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])]) \rhd e_2[x \mapsto v] : S^{-1}(S\tau') \\ \textit{i.e. } \Gamma[x' \mapsto \mathit{AppClose}(\tau_1, \Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau])] \rhd e_2[x \mapsto v] : \tau'. \end{array}$$

Then by Lemma 4.6$^+$

$$\Gamma[x' \mapsto \mathit{AppClose}(\tau_1, \Gamma)] \rhd e_2[x \mapsto v] : \tau'$$

and by $\mathbf{let}_e$ with (24)

$$\begin{array}{l} \Gamma \rhd \textsf{let } x \textsf{ be } e_1[x \mapsto v] \textsf{ in } e_2[x \mapsto v] : \tau' \\ \textit{i.e. } \Gamma \rhd (\textsf{let } x \textsf{ be } e_1 \textsf{ in } e_2)[x \mapsto v] : \tau'. \end{array}$$

∎

The evaluation of an expression in Reference ML can become stuck for several new reasons. The expanded definition of faulty expressions reflects this fact.

Definition 5.4 (Faulty Expressions) *The* faulty *expressions of Reference ML are those expressions containing a subexpression of the form:*

$(c\ v)$ *where* $\delta(c, v)$ *is undefined;*
$(!\ v)$ *where* $v \notin \mathit{Var}$*;*
$(:=\ v)$ *where* $v \notin \mathit{Var}$*; or*
$\rho\theta\langle x, v_2\rangle.C[x\ v_1]$.

As before, we have a Uniform Evaluation Lemma, stating that programs either yield an answer, diverge, or reduce to a faulty expression.

Lemma 5.5 (Uniform Evaluation) *For closed* e*, if there is no* e' *such that* $e \overset{\mathbf{vr}}{\longmapsto\!\!\!\!\rightarrow} e'$ *and* e' *is faulty, then either* $e \Uparrow$ *or* $e \overset{\mathbf{vr}}{\longmapsto\!\!\!\!\rightarrow} a$.

Proof. The proof proceeds by induction on the length of the reduction sequence, and is similar to the proof for Functional ML. ∎

All of the expressions in the expanded definition are untypable.

Main Lemma 5.6 (Faulty Expressions are Untypable) *If* e *is faulty, there are no* Γ *and* τ *such that* $\Gamma \overset{\tau}{\triangleright} e : \tau$.

Proof. Again, it suffices to show that the subexpressions of e that cause e to be faulty are untypable. We proceed by case analysis according to the form of the subexpression, assuming for each case that the expression can be typed, and deriving a contradiction.

Case $(c\ v)$ where $\delta(c, v)$ is undefined. As for Functional ML.

Case $(!\ v)$ where $v \notin \mathit{Var}$. Assume that $\Gamma \triangleright\ !\ v : \tau$. Then $\Gamma \triangleright v : \tau\ \mathit{ref}$ by **app** and **deref**, which implies v is a variable (since no other value can have type $\tau\ \mathit{ref}$), contrary to the assumption.

Case $(:=\ v)$ where $v \notin \mathit{Var}$. Assume that $\Gamma \triangleright := v : \tau$. Then $\Gamma \triangleright v : \tau'\ \mathit{ref}$ where $\tau = \tau' \rightarrow \tau'$ by **app** and **assign**, which implies v is a variable, contrary to the assumption.

Case $\rho\theta\langle x, v'\rangle.C[x\ v]$. Assume that $\Gamma \triangleright \rho\theta\langle x, v'\rangle.C[x\ v] : \tau$. Then $\Gamma' \triangleright C[x\ v] : \tau$ by **rho**, where $\Gamma'(x) = \tau'\ \mathit{ref}$. Since $\Gamma'' \triangleright x\ v : \tau_2$ where $\Gamma'(x) = \Gamma''(x)$, $\Gamma'' \triangleright x : \tau_1 \rightarrow \tau_2$ by **app**, hence $\Gamma''(x) \succ \tau_1 \rightarrow \tau_2$. This contradicts $\Gamma''(x) = \tau'\ \mathit{ref}$. ∎

Type soundness now follows as before.

Theorem 5.7 (Syntactic Soundness for Reference ML) *If* $\overset{\tau}{\triangleright} e : \tau$ *then either* $e \Uparrow$ *or* $e \overset{\mathbf{vr}}{\longmapsto\!\!\!\!\rightarrow} a$ *and* $\overset{\tau}{\triangleright} a : \tau$.

Proof. Exactly as for Theorem 4.12, using the appropriately extended lemmas. ∎

5.2 Exception ML

Due to δ-typability, constant functions must be defined for *all* values of their input type. This precludes constants, such as integer or real division, that are defined on all but a few recognizable input values of correct type. STANDARD ML solves this problem by introducing exceptions.

Our exception extension provides named exceptions with parameters, a means of raising exceptions, and a means of handling exceptions. Our extension thus includes three new phrases:

$$\textsf{exception}\ x_1 \ldots x_n\ \textsf{in}\ e, \qquad \textsf{raise}\ e_1\ e_2, \qquad e_3\ \textsf{handle}\ e_1\ e_2.$$

The first phrase declares $x_1, \ldots, x_n$ to be exception names lexically bound in e. The second phrase requires e_1 to evaluate to an exception name, and raises that exception with a parameter, the value of e_2. The parameter propagates along with the exception, and is used at the site where an exception is handled. The third phrase evaluates its subexpressions e_1 and e_2 *first*, and installs the value of e_2 as an exception handler for e_1 exceptions that are raised during the evaluation of e_3. The subexpression e_1 must evaluate to an exception name, and e_2 must evaluate to a function that accepts parameters of e_1 exceptions. When an exception is raised, control transfers to the *dynamically* closest handler for that kind of exception. The handler function is applied to the exception parameter, and the result of the application is returned as the result of the **handle**-phrase. If there is no active handler for a raised exception, evaluation terminates with an "unhandled exception" answer.

To extend Functional ML with exceptions, we add several new phrases to the syntax:

$$\begin{array}{lll}
(\textit{Expressions}) & e ::= v \mid e_1\ e_2 \mid \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 \mid \underline{\textsf{exception}\ \chi\ \textsf{in}\ e} \\
(\textit{Values}) & v ::= c \mid x \mid \mathsf{Y} \mid \lambda x.e \mid \underline{\textsf{raise}} \mid \underline{e\ \textsf{handle}} \mid \underline{\textsf{raise}\ v} \mid \underline{e\ \textsf{handle}\ v} \\
 & \underline{\chi} ::= \underline{x^*}
\end{array}$$

In the expression **exception** χ **in** e, the variables in χ are bound in e. We treat χ as a set of variables, and call these *exception names.* As **raise** is a curried binary operator, the application of **raise** to a value is a value, and is included in the syntactic class of values (like := in Reference ML). Likewise, the expression e **handle** is a value, and receives an exception name and handler function by ordinary application. Hence the application of e **handle** to a value is also a value.

5.2.1 Semantics

Raising and handling exceptions requires several new reductions:

$$\begin{array}{lrcll}
(\textit{raise}) & U[\textsf{raise}\ x\ v] & \longrightarrow & \textsf{raise}\ x\ v & \text{if } U \neq [\,] \\
(\textit{handle}) & (\textsf{raise}\ x\ v_1)\ \textsf{handle}\ x\ v_2 & \longrightarrow & v_2\ v_1 & \\
(\textit{reraise}) & \begin{array}{l}\textsf{exception}\ \chi x_1 x_2\ \textsf{in} \\ \quad X[(\textsf{raise}\ x_1\ v_1)\ \textsf{handle}\ x_2\ v_2]\end{array} & \longrightarrow & \begin{array}{l}\textsf{exception}\ \chi x_1 x_2\ \textsf{in} \\ \quad X[\textsf{raise}\ x_1\ v_1]\end{array} & x_1 \neq x_2 \\
(\textit{unhandle}) & v_1\ \textsf{handle}\ x\ v_2 & \longrightarrow & v_1 &
\end{array}$$

These new notions of reduction again rely on (two different kinds of) contexts for the enforcement of a specific order of evaluation for applications in connection with the raising and handling of exceptions:

$(raising)$ $\quad U ::= [] \mid U\ e \mid v\ U \mid \textsf{let}\ x\ \textsf{be}\ U\ \textsf{in}\ e$
$(handling)$ $\quad X ::= [] \mid X\ e \mid v\ X \mid \textsf{let}\ x\ \textsf{be}\ X\ \textsf{in}\ e \mid X\ \textsf{handle}\ x\ v.$

When an exception is raised, *raise* reductions propagate the exception outwards, eliminating pending computations. As U contexts do not include the phrase U **handle** $x\ v$, *raise* reductions propagate the exception only to the closest exception handler. If the closest handler matches the raised exception, a *handle* reduction applies the handler function to the exception parameter. If the handler does not match the exception, a *reraise* reduction discards the handler, and the exception may continue to propagate by *raise* reductions. Should an expression protected by a handler evaluate to a value without raising an exception, the *unhandle* reduction discards the handler.

The **exception**-expression requires two additional reductions, similar to those for ρ-expressions:[7]

(exn_{merge}) $\quad \textsf{exception}\ \chi_1\ \textsf{in exception}\ \chi_2\ \textsf{in}\ e \longrightarrow \textsf{exception}\ \chi_1\chi_2\ \textsf{in}\ e$
(exn_{lift}) $\quad X[\textsf{exception}\ \chi\ \textsf{in}\ e] \longrightarrow \textsf{exception}\ \chi\ \textsf{in}\ X[e] \quad \text{if } X \neq []$

Indeed, the only difference is that the **exception**-expression does not bind any values to its variables. The purpose of an **exception**-expression is to distinguish different exceptions in the *reraise* reduction, in order that evaluation be deterministic. If x_1 and x_2 are different but only λ- or **let**-bound, subsequent reductions may substitute them to the same **exception**-bound variable, making the *raise* reduction applicable. For example:

$$\begin{array}{ll} & \textsf{exception x in } (\lambda \textsf{y}.\lambda \textsf{z}.(\textsf{raise y}\ v_1)\ \textsf{handle z}\ v_2)\ \textsf{x x} \\ \twoheadrightarrow & \textsf{exception x in } (\textsf{raise x}\ v_1)\ \textsf{handle x}\ v_2 \\ \twoheadrightarrow & \textsf{exception x in } v_2\ v_1. \end{array}$$

Insisting that x_1 and x_2 be **exception**-bound in *reraise* ensures that they refer to distinct exceptions. The reduction *handle* does not require its variables to be **exception**-bound, because substitution necessarily replaces both instances of x in the *handle* redex with the same exception name.

We use **x** to refer to the six reductions for exceptions introduced above. Taking the union of **v** (extended to the full syntax) and **x** yields **vx**, the notion of reduction for Exception ML. With evaluation contexts extended to:

$$E ::= [] \mid E\ e \mid v\ E \mid \textsf{let}\ x\ \textsf{be}\ E\ \textsf{in}\ e \mid \textsf{exception}\ \chi\ \textsf{in}\ E \mid E\ \textsf{handle}\ x\ v$$

the stepping function $\overset{\mathbf{vx}}{\longmapsto}$ may be defined along the same lines as before. Answers for $eval_{\mathbf{vx}}$ are:

$(answers)$ $\quad a ::= v \mid \textsf{exception}\ \chi\ \textsf{in}\ v \mid \textsf{exception}\ \chi x\ \textsf{in raise}\ x\ v.$

[7]Due to our use of the variable conventions for exn_{merge}, our exceptions are *generative* as in STANDARD ML [22, 23].

Answers of the third form are *unhandled exceptions*: since handlers are dynamically bound, it is possible for an exception to be raised when no handler for it is installed. Evaluation is defined as:

$$(eval_{\mathbf{vx}}) \qquad\qquad eval_{\mathbf{vx}}(e) = a \quad \text{iff} \quad e \overset{\mathbf{vx}}{\longmapsto\!\!\!\!\rightarrow} a.$$

We can now extend the domain of δ to admit functions such as division, by allowing δ to return an expression that raises an exception:[8]

$$\delta : \mathit{Const} \times \mathit{ClosedVal} \rightharpoonup \mathit{ClosedVal} \cup \{\textsf{exception x in raise x } v \mid v \text{ is closed}\}.$$

Functions such as division can now be defined on every element of their input type, by returning an exception when their application does not make sense.

5.2.2 Typing

We extend the set of types with an additional type constructor:

$$\tau ::= \iota_1 \mid \ldots \mid \iota_n \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \underline{\tau\ \mathit{exn}}$$

The type τ *exn* is the type of exceptions with a parameter of type τ.

In typing exceptions in the presence of polymorphism, one encounters similar difficulties as in typing references. The phrase **raise** e_1 e_2 requires e_1 to be an exception of type τ *exn*, and e_2 to be a matching parameter of type τ. If e_3 has type τ', the phrase e_3 **handle** e_1 e_2 requires e_1 to be an exception of type τ *exn*, and e_2 to be a handler for such exceptions. The handler must be a function of type $\tau \rightarrow \tau'$, since it takes the exception parameter as input, and returns a value to be returned in place of the value of e_3. The obvious typing for exceptions allows the parameter types of exceptions to be fully polymorphic, permitting the following expression to be typed:[9]

$$\begin{array}{ll} & \textsf{let rh be exception x in pair (raise x) } (\lambda\textsf{a}.\lambda\textsf{b}.(\textsf{a 3) handle x b}) \\ & \textsf{in (snd rh) } (\lambda\textsf{d}.(\textsf{fst rh) true}) \ (\lambda\textsf{y}.+ \textsf{ 1 y}) \\ \longrightarrow\!\!\!\!\rightarrow & \textsf{exception x in + 1 true} \end{array}$$

Here, functions to raise and handle the same x-exception are returned as a pair, bound by a **let**-expression to **rh**, and then used. In the body of the **let**-expression, the expression (**raise x**) has type $\alpha_1 \rightarrow \alpha_2$, and ($\lambda$a.$\lambda$b.(a 3) **handle** x b) has type $(\mathit{int} \rightarrow \alpha_3) \rightarrow (\alpha_1 \rightarrow \alpha_3) \rightarrow \alpha_3$. If the exception parameter type α_1 is generalized, then in the body of the **let**-expression an x-exception may be raised with an argument of type *bool* by (**fst rh**), and caught by (**snd rh**), which is expecting an argument of type *int*.

Since the problems of typing exceptions and references are similar, it is not surprising that the same modifications to the typing rules for functional expressions apply: types and type variables are classified as imperative or applicative, and the **let** rule is split according to whether the bound expression is expansive (see Figure 3). Figure 5 presents the additional typing rules for exceptions. Again, as with Reference ML, we require the additional restriction that there be no constants of exception type (τ *exn*).

[8] Usually a set of exceptions that may be raised by constant functions is defined in an initial environment; we take an alternative approach to simplify the presentation.

[9] The constants pair, fst, and snd provide pairing and projection operations.

(raise) $\Gamma \rhd \textsf{raise} : \tau_1\ exn \rightarrow \tau_1 \rightarrow \tau_2$

(handle) $$\frac{\Gamma \rhd e : \tau_2}{\Gamma \rhd e\ \textsf{handle} : \tau_1\ exn \rightarrow (\tau_1 \rightarrow \tau_2) \rightarrow \tau_2}$$

(exn) $$\frac{\Gamma[x_1 \mapsto \tau_1\ exn] \ldots [x_n \mapsto \tau_n\ exn] \rhd e : \tau \quad \tau_i \text{ is imperative} \quad 1 \leq i \leq n}{\Gamma \rhd \textsf{exception}\ x_1 \ldots x_n\ \textsf{in}\ e : \tau}$$

Figure 5: Additional typing rules for exceptions

5.2.3 Type Soundness for Exceptions

To establish soundness for the exception typing, we proceed as for references: we establish type preservation for **vx**, and show that the faulty expressions of Exception ML are untypable.

Main Lemma 5.8 (Subject Reduction for vx) *If* $\Gamma \overset{x}{\rhd} e_1 : \tau$ *and* $e_1 \xrightarrow{\mathbf{vx}} e_2$ *then* $\Gamma \overset{x}{\rhd} e_2 : \tau$.

Proof. The cases for the reductions of **v** are the same as before.

Case $U[\textsf{raise}\ x\ v] \longrightarrow \textsf{raise}\ x\ v$. From $\Gamma \rhd U[\textsf{raise}\ x\ v] : \tau$

$$\Gamma \rhd \textsf{raise}\ x\ v : \tau'$$

by **app**. Hence

(27) $\Gamma \rhd \textsf{raise}\ x : \tau_1 \rightarrow \tau_2$

(28) and $\Gamma \rhd v : \tau_1$

by **app**. From (27)

$\Gamma \rhd x : \tau_1\ exn$ by **app** and **raise**,

$\Gamma \rhd \textsf{raise}\ x : \tau_1 \rightarrow \tau$ by **app** and **raise**,

$\Gamma \rhd \textsf{raise}\ x\ v : \tau$ by **app** with (28).

Case $\textsf{exception}\ \chi_1\ \textsf{in exception}\ \chi_2\ \textsf{in}\ e \longrightarrow \textsf{exception}\ \chi_1\chi_2\ \textsf{in}\ e$. Similar to ρ_{merge} in Lemma 5.2.

Case $X[\textsf{exception}\ \chi\ \textsf{in}\ e] \longrightarrow \textsf{exception}\ \chi\ \textsf{in}\ X[e]$. Similar to ρ_{lift} in Lemma 5.2.

Case $v_1\ \textsf{handle}\ x\ v_2 \longrightarrow v_1$. $\Gamma \rhd v_1 : \tau$ from $\Gamma \rhd v_1\ \textsf{handle}\ x\ v_2 : \tau$ by **handle** and **app**.

Case $(\textsf{raise}\ x\ v_1)\ \textsf{handle}\ x\ v_2 \longrightarrow v_2\ v_1$. From $\Gamma \rhd (\textsf{raise}\ x\ v_1)\ \textsf{handle}\ x\ v_2 : \tau$ by **handle**, **app**, and **raise**

$$\Gamma \rhd v_1 : \tau' \quad \text{and} \quad \Gamma \rhd v_2 : \tau' \rightarrow \tau$$

where $\Gamma x = \tau'\ exn$. Then $\Gamma \rhd v_2\ v_1 : \tau$ by **app**.

Case $\begin{array}{c}\textsf{exception } \chi x_1 x_2 \textsf{ in}\\ X[(\textsf{raise } x_1\ v_1) \textsf{ handle } x_2\ v_2]\end{array} \longrightarrow \begin{array}{c}\textsf{exception } \chi x_1 x_2 \textsf{ in}\\ X[\textsf{raise } x_1\ v_1]\end{array}$. Then

$$\begin{array}{ll}
\Gamma' \rhd (\textsf{raise } x_1\ v_1) \textsf{ handle } x_2\ v_2 : \tau' & \\
\Gamma' \rhd (\textsf{raise } x_1\ v_1) \textsf{ handle} : \tau_2\ \mathit{exn} \rightarrow (\tau_2 \rightarrow \tau') \rightarrow \tau' & \text{by } \textbf{app}, \\
\Gamma' \rhd \textsf{raise } x_1\ v_1 : \tau' & \text{by } \textbf{handle}, \\
\Gamma \rhd \textsf{exception } \chi x_1 x_2 \textsf{ in } X[\textsf{raise } x_1\ v_1] : \tau & \text{by Replacement}^+.
\end{array}$$

With the extension of the various lemmas, the proof of type preservation for Exception ML is complete. The lemmas go through with simple adaptations to the new syntax. ∎

There are several new kinds of faulty expressions for Exception ML.

Definition 5.9 (Faulty expressions) *The* faulty *expressions of Exception ML are those expressions containing a subexpression of the form:*

$$\begin{array}{l}
(c\ v) \textit{ where } \delta(c,v) \textit{ is undefined,}\\
(\textsf{raise } v) \textit{ where } v \notin \mathit{Var},\\
(e \textsf{ handle } v) \textit{ where } v \notin \mathit{Var}, \textit{ or}\\
\textsf{exception } \chi x \textsf{ in } C[x\ v],
\end{array}$$

where

$$\begin{array}{rl}
C ::= & [\,] \mid C\ e \mid e\ C \mid \textsf{let } x \textsf{ be } C \textsf{ in } e \mid \textsf{let } x \textsf{ be } e \textsf{ in } C \mid \lambda x.C\\
& \mid \textsf{exception } \chi \textsf{ in } C \mid C \textsf{ handle}.
\end{array}$$

Again, a uniform evaluation lemma can be shown, and the faulty expressions can be proven untypable. Type soundness for Exception ML follows as before.

Theorem 5.10 (Syntactic Soundness for Exception ML) *If* $\overset{x}{\rhd}\ e : \tau$ *then either* $e \Uparrow$ *or* $e \overset{\mathbf{vx}}{\longmapsto\!\!\!\!\rightarrow} a$ *and* $\overset{x}{\rhd}\ a : \tau$.

5.3 Core ML

We can combine references and exceptions in one language, Core ML, that has all the essential features of Standard ML. In combining the reference and exception extensions, we must ensure that they interact appropriately.

5.3.1 Semantics

To combine the calculi for the two extensions, the R contexts of the reference fragment must be extended to include phrases from the exception fragment; likewise, X contexts must be extended to include phrases from the reference fragment:

$$\begin{array}{l}
R ::= \ldots \mid R \textsf{ handle } v_1\ v_2\\
X ::= \ldots \mid \rho\theta.X
\end{array}$$

The phrase $\mathsf{exception}\ \chi\ \mathsf{in}\ R$ is not included in R in order that exceptions in the store do not escape their bindings, *i.e.*, $\mathsf{exception}$-expressions can move through ρ-expressions by the exn_{lift} reduction, with appropriate variable renaming:

$$\rho\theta.\mathsf{exception}\ \chi x\ \mathsf{in}\ e \longrightarrow \mathsf{exception}\ \chi x\ \mathsf{in}\ \rho\theta.e$$

but not vice versa:

$$\mathsf{exception}\ \chi x\ \mathsf{in}\ \rho\theta\langle r, x\rangle.e \not\longrightarrow \rho\theta\langle r, x\rangle.\mathsf{exception}\ \chi x\ \mathsf{in}\ e.$$

If the second were a permissible reduction, the occurrence of x in the store would escape its binding.

After this extension of the R and X contexts, the resulting notion of reduction for the full syntax is **vrx**. The E evaluation contexts used in defining $\overset{\mathbf{vrx}}{\longmapsto}$ and $eval_{\mathbf{vrx}}$ must also be extended:

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ e \mid E\ \mathsf{handle}\ v_1\ v_2 \mid \rho\theta.E \mid \mathsf{exception}\ \chi\ \mathsf{in}\ E.$$

Answers are:

$$(answers) \qquad a ::= \{\mathsf{exception}\ \chi\ \mathsf{in}\}\ \{\rho\theta.\}\ v \mid \{\mathsf{exception}\ \chi x\ \mathsf{in}\}\ \{\rho\theta.\}\ \mathsf{raise}\ x\ v$$

(where $\{phrase\}$ means *phrase* may be omitted). The complete calculus for Core ML may be found in the appendix.

5.3.2 Typing

To combine the typing rules for the two fragments is easy, since they both rely on the same classification of types as imperative or applicative, and do not interfere. The complete typing rules for Core ML may be found in the appendix.

5.3.3 Type Soundness for Core ML

The type soundness of the resulting calculus requires re-establishing the various lemmas and theorems for the extended syntax and extended evaluation contexts. These proofs are all straightforward. There is a combinatorial increase in the kinds of faulty expressions because of the interaction between the two fragments: references cannot be raised or handled; exceptions cannot be dereferenced nor assigned. The structure of the proof, however, stays the same.

Definition 5.11 (Faulty Expressions) *The* faulty *expressions of Core ML are those expressions containing a subexpression of the form:*

$$(c\ v)\ \textit{where}\ \delta(c, v)\ \textit{is undefined;}$$

$$\begin{array}{ll} (!\ v)\ \textit{where}\ v \notin \textit{Var}, & (\mathsf{raise}\ v)\ \textit{where}\ v \notin \textit{Var}, \\ (:=\ v)\ \textit{where}\ v \notin \textit{Var}, & (e\ \mathsf{handle}\ v)\ \textit{where}\ v \notin \textit{Var}, \\ \rho\theta\langle x, v_2\rangle.C[x\ v_1], & \mathsf{exception}\ \chi x\ \mathsf{in}\ C[x\ v], \\ \mathsf{exception}\ \chi x\ \mathsf{in}\ C[!\ x], & \rho\theta\langle x, v\rangle.C[\mathsf{raise}\ x], \\ \mathsf{exception}\ \chi x\ \mathsf{in}\ C[:=\ x], & \rho\theta\langle x, v\rangle.C[e\ \mathsf{handle}\ x], \end{array}$$

where

$$\begin{array}{l} C ::= [\,] \mid C\ e \mid e\ C \mid \textsf{let}\ x\ \textsf{be}\ C\ \textsf{in}\ e \mid \textsf{let}\ x\ \textsf{be}\ e\ \textsf{in}\ C \mid \lambda x.C \\ \quad \mid \rho\theta.C \mid \rho\theta\langle x, C\rangle.e \mid \textsf{exception}\ \chi\ \textsf{in}\ C \mid C\ \textsf{handle}. \end{array}$$

Type soundness follows from subject reduction, uniform evaluation, and faulty expressions being untypable, as before.

Theorem 5.12 (Syntactic Soundness for Core ML) *If* $\overset{\tau x}{\triangleright}\ e : \tau$ *then either* $e \Uparrow$ *or* $e \overset{\mathbf{vrx}}{\longmapsto\!\!\!\!\rightarrow} a$ *and* $\overset{\tau x}{\triangleright}\ a : \tau$.

6 Control ML

In this section we present an extension to Functional ML providing first-class continuations. The typing of our extension is similar to that described by Duba *et al.* [8], and implemented in STANDARD ML OF NEW JERSEY [34]. It is a simple matter to merge this extension with Core ML.

Control ML extends Functional ML's syntax with two new constructs:

(*Expressions*)	$e ::= v \mid e_1\ e_2 \mid \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 \mid \underline{\textsf{abort}\ e}$
(*Values*)	$v ::= c \mid x \mid \textsf{Y} \mid \lambda x.e \mid \underline{\textsf{callcc}}$

An abort-expression evaluates its subexpression to a value, and returns this value as the result of the program. When the callcc operator (*call-with-current-continuation*) is applied to a function f, it captures a representation of the current continuation (or program control stack), encapsulates this continuation in an abstraction, and applies f to this abstraction. If this abstraction is later applied to a value, control transfers to the captured continuation.

6.1 Semantics

While defining calculus reductions for first-class continuations is possible [13, 14], the resulting calculus is inappropriate for our problem. Since we are only interested in evaluation, we take a simpler approach [11]. We give top-level evaluation rules, *program reductions*, for the control operators by extending the stepping relation ($\longmapsto$) *directly*:

($\delta, \beta_{\mathrm{v}}$, *let*, Y)	$E[e] \longmapsto E[e']$ iff $e \xrightarrow{\mathbf{v}} e'$
(*callcc*)	$E[\textsf{callcc}\ v] \longmapsto E[v\ (\lambda x.\textsf{abort}\ E[x])]$
(*abort*)	$E[\textsf{abort}\ e] \longmapsto e$

where

$$E ::= [\,] \mid E\ e \mid v\ E \mid \textsf{let}\ x\ \textsf{be}\ E\ \textsf{in}\ e.$$

In this style of semantics, the evaluation context E represents the current continuation. An *abort* reduction simply discards the current continuation, continuing evaluation with its subexpression. The *callcc* reduction captures the current continuation, encapsulates it in a function, and applies callcc's argument to this function. The continuations created by callcc are *abortive*: when invoked, they discard the continuation surrounding their application

```
let product be λx. callcc
                    (λk. letrec p be λz. if nil? z then 1
                                          else if = 0 (car z) then k 0
                                          else * (car z) (p (cdr z))
                         in p x)
in product (cons 2 (cons 3 (cons 0 (cons 5 nil))))
```

Figure 6: A program illustrating a simple use of callcc

(via an *abort* reduction), installing instead the captured continuation. Answers are simply values, and *eval* is defined as for Functional ML.

Figure 6 illustrates a simple use of **callcc**. This program computes the product of a list of numbers, performing no multiplications if the answer is zero. The constants **nil**, **cons**, **nil?**, **car**, and **cdr** are the usual constants and operations for lists. The expressions **if**. . .**then**. . .**else**. . . and **letrec**. . .**be**. . .**in**. . . are syntactic sugar for appropriate constant applications. This program computes the product of a list of numbers, performing no multiplications if the answer is zero. The program escapes from pending multiplications by jumping out of the recursion to the continuation of the application of **product**.

6.2 Typing

The obvious approach to typing **callcc** leads to the following rule:

(naïve-callcc) $\Gamma \rhd \mathsf{callcc} : ((\tau_1 \to \tau_2) \to \tau_1) \to \tau_1$

As indicated by the *callcc* reduction, **callcc** takes a function whose argument is a procedural abstraction of the evaluation context, the *continuation*. If the evaluation context expects a value of type τ_1, then the continuation has type $\tau_1 \to \tau_2$ for any type τ_2. If **callcc**'s argument ignores this continuation, it must produce a value of type τ_1, which implies that **callcc**'s argument must have type $(\tau_1 \to \tau_2) \to \tau_1$. The result type of the continuation, τ_2, is arbitrary, because the application of k built into the continuation ensures that it never returns.

However, this obvious typing is unsound, as illustrated by the following example:[10]

```
let x be callcc (λk. pair (λx.x) (λf. k (pair f (λd.5))))
in  (λz. snd x (λx.+ x 1)) (fst x true)
```

The **callcc** application returns a pair of type $(\alpha \to \alpha) \times ((\alpha \to \alpha) \to int)$, consisting of the identity function and a function that applies a continuation. Since α is not free in the type environment, it is closed over, and **x** has type scheme $\forall\alpha.(\alpha \to \alpha) \times ((\alpha \to \alpha) \to int)$ in the body of the **let**-expression. In evaluating the body, **fst x** is the polymorphic identity function, so it may validly be applied to **true**. Then the second function of the pair is applied

[10]A variation of this example was discovered by Robert Harper and Mark Lillibridge [`sml` electronic mailing list, July 8, 1991]. Despite widespread use of a naïvely typed callcc extension in Standard ML of New Jersey, it took two years until this problem was discovered.

to (λx.+ x 1), and the continuation so invoked rebinds x to the pair consisting of (λx.+ x 1) and (λd.5). The second evaluation of fst x true results in an attempt to add 1 to true.

As with references and exceptions, a correct typing for continuations builds upon the classification of types as imperative or applicative, and requires the let typing rules of Figure 3. The correct typing rules are:

(callcc) $\Gamma \triangleright \mathsf{callcc} : ((\tau_1 \rightarrow \tau_2) \rightarrow \tau_1) \rightarrow \tau_1$ if τ_1 is imperative.

(abort) $$\frac{\Gamma \triangleright e : \tau_1}{\Gamma \triangleright \mathsf{abort}\ e : \tau_2}$$

An abort-expression may have any type, regardless of the type of its subexpression, because when evaluated it never returns.

6.3 Weak Type Soundness

Because abort-expressions can return a value of any type, proving type soundness for the continuation extension is not as simple as for the previous extensions. For example, if e is an expression of type *bool*, the program:

$$\mathsf{if}\ e\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{abort\ true})$$

is typable according to the above typing rules, but returns either *int* or *bool* according to whether e is true or false. Thus subject reduction in the usual sense does not hold; however, a weaker lemma does hold, stating that *typability* is preserved.

Main Lemma 6.1 (Typability Preservation) *If* $\triangleright e_1 : \tau$ *and* $e_1 \longmapsto e_2$ *then* $\triangleright e_2 : \tau'$.

Proof. We need only consider the additional cases for the *callcc* and *abort* reductions. The others are simply adapted from Lemma 4.3 (Subject Reduction for Functional ML).

Case $E[\mathsf{abort}\ e] \longmapsto e$. Then $\Gamma \triangleright e : \tau'$ for some τ'. But since $E[\mathsf{abort}\ e]$ is closed and E does not bind variables, e is closed, hence $\triangleright e : \tau'$.

Case $E[\mathsf{callcc}\ v] \longmapsto E[v\ (\lambda x.\mathsf{abort}\ E[x])]$. From $\triangleright E[\mathsf{callcc}\ v] : \tau$ we have $\triangleright \mathsf{callcc}\ v : \tau_1$, and by **callcc** and **app**

(29) $$\triangleright v : (\tau_1 \rightarrow \tau_2) \rightarrow \tau_1$$

where τ_1 is imperative.

Since E is closed, x does not appear free in E, and we claim that $[x \mapsto \tau_1] \triangleright E[x] : \tau$. The proof of this claim proceeds by induction on the structure of E.

Case $E = []$. Then $\tau = \tau_1$, and $[x \mapsto \tau_1] \triangleright x : \tau$ by **var**.

Case $E = (E'\ e')$. Then $\triangleright (E'[\mathsf{callcc}\ v]\ e') : \tau$, and by **app**

(30) $$\triangleright E'[\mathsf{callcc}\ v] : \tau_3 \rightarrow \tau$$

(31) and $\triangleright e' : \tau_3$

for some τ_3. Then $[x \mapsto \tau_1] \triangleright E'[x] : \tau_3 \rightarrow \tau$ by ind. hyp. with (30). Also $[x \mapsto \tau_1] \triangleright e' : \tau_3$ by Lemma 4.1$^+$ with (31) since $x \notin FV(E)$. Hence

$$[x \mapsto \tau_1] \triangleright E'[x]\ e' : \tau \qquad \text{by } \mathbf{app}.$$

Case $E = (v\ E')$. Similar to the previous case.

Case $E = \textsf{let } x' \textsf{ be } E' \textsf{ in } e'$. Then $\rhd \textsf{let } x' \textsf{ be } E'[\textsf{callcc } v] \textsf{ in } e' : \tau$. Since $(\textsf{callcc } v)$ is expansive, $E'[\textsf{callcc } v]$ is expansive. Thus by $\textbf{let}_e$

$$
\begin{array}{lrl}
(32) & & \rhd E'[\textsf{callcc } v] : \tau_3 \\
(33) & \text{and} & [x' \mapsto \mathit{AppClose}(\tau_3, \emptyset)] \rhd e' : \tau
\end{array}
$$

for some τ_3. Then

$$
\begin{array}{lll}
(34) & [x \mapsto \tau_1] \rhd E'[x] : \tau_3 & \text{by ind. hyp. with (32),}
\end{array}
$$

and

$$
\begin{array}{rl}
[x' \mapsto \mathit{AppClose}(\tau_3, \emptyset)][x \mapsto \tau_1] \rhd e' : \tau & \text{by Lemma } 4.1^{+} \text{ with (33)} \\
i.e.\ [x \mapsto \tau_1][x' \mapsto \mathit{AppClose}(\tau_3, \emptyset)] \rhd e' : \tau. &
\end{array}
$$

By the critical restriction that τ_1 is imperative,

$$\mathit{AppClose}(\tau_3, \emptyset) = \mathit{AppClose}(\tau_3, [x \mapsto \tau_1]),$$

hence

$$
\begin{array}{ll}
[x \mapsto \tau_1][x' \mapsto \mathit{AppClose}(\tau_3, [x \mapsto \tau_1])] \rhd e' : \tau & \\
[x \mapsto \tau_1] \rhd \textsf{let } x' \textsf{ be } E'[x] \textsf{ in } e' : \tau & \text{by } \textbf{let}_e \text{ with (34).}
\end{array}
$$

This completes the proof of the claim. Hence

$$
\begin{array}{ll}
[x \mapsto \tau_1] \rhd \textsf{abort } E[x] : \tau_2 & \text{by } \textbf{abort}, \\
\rhd (\lambda x.\textsf{abort } E[x]) : \tau_1 \to \tau_2 & \text{by } \textbf{abs}, \\
\rhd v\ (\lambda x.\textsf{abort } E[x]) : \tau_1 & \text{by } \textbf{app} \text{ with (29)} \\
\rhd E[v\ (\lambda x.\textsf{abort } E[x])] : \tau & \text{by Replacement}^{+}. \ \blacksquare
\end{array}
$$

The faulty expressions for Control ML are the same as those of Functional ML, extended to the new syntax.

Definition 6.2 (Faulty Expressions) *The* faulty *expressions of Control ML are the expressions containing a subexpression $(c\ v)$ where $\delta(c, v)$ is undefined.*

As with Functional ML, the faulty expressions are untypable and Uniform Evaluation holds. However, since we only have typability preservation, we obtain a weaker version of syntactic soundness that does not indicate the type of answers.

Theorem 6.3 (Weak Syntactic Soundness) *If $\rhd e : \tau$ then either $e\Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$.*

Proof. By Uniform Evaluation, either $e \longmapsto\!\!\!\!\!\rightarrow e'$ and e' is faulty, or $e\Uparrow$, or $e \longmapsto\!\!\!\!\!\rightarrow v$. Since $\rhd e : \tau$, Typability Preservation implies $\rhd v : \tau'$ and $\rhd e' : \tau'$. Suppose $e \longmapsto\!\!\!\!\!\rightarrow e'$ and e' is faulty. Since faulty expressions are untypable, $\rhd e' : \tau'$ is a contradiction, therefore this case cannot occur. Hence either $e\Uparrow$ or $e \longmapsto\!\!\!\!\!\rightarrow v$. ∎

This weaker theorem establishes weak soundness, but does not imply strong soundness. In fact, if the use of **abort** is unrestricted, it is not possible to predict the type of an answer, as the example at the beginning of this section illustrates.

6.4 Strong Type Soundness

Examining the proof of Typability Preservation reveals that every reduction with the exception of *abort* preserves type. In particular, the *callcc* reduction preserves type, which suggests that callcc by itself is strongly sound. To obtain a proof of strong soundness for callcc, we must eliminate abort from the surface language available to programmers, but retain it in the underlying language of evaluation for callcc reductions. Because the abort expressions introduced by callcc applications have a restricted shape, they never return values of other than the top-level type of the program, and do not compromise strong soundness.

To establish a subject reduction lemma, we define an augmented type system that infers both the ordinary type of an expression and a set of *abort types*. The abort types of an expression e are the types of the immediate subexpressions of abort-expressions in e. The augmented system does not permit abort types to be generalized, thereby ensuring that each syntactic occurrence of abort can produce only one type of answer. This restriction eliminates expressions such as the following:

```
let a be λx. abort x
in  if e then  a 1
    else  a true
```

Hence if a well-typed program aborts, the answer produced is one of the program's abort types. Type judgments for this augmented system have the form $\Gamma \overset{a}{\triangleright} e : \tau, T$, meaning that in type environment Γ, expression e has ordinary type τ and abort types T, where T is a set of types. Figure 7 presents the typing rules of the augmented system.

The augmented type system corresponds closely to our original type system for Control ML. The rule for typing abort-expressions is the only rule that connects the ordinary type and the abort types of an expression. In each axiom (**var**, **const**, **Y**, **callcc**) the abort set is completely unconstrained; the inference rules simply propagate the abort set. The let-expression rules do not generalize type variables in the set of abort types, hence the augmented system does not permit polymorphic uses of abort, and accepts a subset of the expressions accepted by the ordinary system. A Correspondence Lemma establishes this connection between the ordinary system ($\triangleright$) and the augmented system ($\overset{a}{\triangleright}$).

Lemma 6.4 (Correspondence)

(*i*) *If* $\Gamma \triangleright e : \tau$ *and* e *contains no* abort*-expressions then* $\Gamma \overset{a}{\triangleright} e : \tau, T$ *for any* T*;*

(*ii*) *If* $\Gamma \overset{a}{\triangleright} e : \tau, T$ *then* $\Gamma \triangleright e : \tau$.

Proof. The proof of each direction shows how to construct a deduction for the consequent from a deduction of the antecedent. Both proofs are straightforward. ∎

Subject Reduction for the augmented system establishes that reduction preserves the set of abort types. As the callcc reduction introduces an abort-expression with the top-level type of the program, the set of abort types preserved by Subject reduction includes the top-level type.

(var) $\Gamma \overset{a}{\triangleright} x : \tau, T \ \text{ if } \ \Gamma(x) \succ \tau$

(const) $\Gamma \overset{a}{\triangleright} c : \tau, T \ \text{ if } \ \mathit{TypeOf}(c) \succ \tau$

(Y) $\Gamma \overset{a}{\triangleright} \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2, T$

(abs) $$\frac{\Gamma[x \mapsto \tau_1] \overset{a}{\triangleright} e : \tau_2, T}{\Gamma \overset{a}{\triangleright} \lambda x.e : \tau_1 \to \tau_2, T}$$

(app) $$\frac{\Gamma \overset{a}{\triangleright} e_1 : \tau_1 \to \tau_2, T \quad \Gamma \overset{a}{\triangleright} e_2 : \tau_1, T}{\Gamma \overset{a}{\triangleright} e_1\ e_2 : \tau_2, T}$$

(let$_v$) $$\frac{\Gamma \overset{a}{\triangleright} v : \tau_1, T \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma, T)] \overset{a}{\triangleright} e : \tau_2, T}{\Gamma \overset{a}{\triangleright} \mathsf{let}\ x\ \mathsf{be}\ v\ \mathsf{in}\ e : \tau_2, T}$$

(let$_e$) $$\frac{\Gamma \overset{a}{\triangleright} e_1 : \tau_1, T \quad \Gamma[x \mapsto \mathit{AppClose}(\tau_1, \Gamma, T)] \overset{a}{\triangleright} e_2 : \tau_2, T \quad e_1 \notin \mathit{Values}}{\Gamma \overset{a}{\triangleright} \mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2 : \tau_2, T}$$

(callcc) $\Gamma \overset{a}{\triangleright} \mathsf{callcc} : ((\tau_1 \to \tau_2) \to \tau_1) \to \tau_1, T \ \text{ if } \ \tau_1 \text{ is imperative}$

(abort) $$\frac{\Gamma \overset{a}{\triangleright} e : \tau_1, T \quad \tau_1 \in T}{\Gamma \overset{a}{\triangleright} \mathsf{abort}\ e : \tau_2, T}$$

$$\begin{aligned} \mathit{Close}(\tau, \Gamma, T) &= \forall \alpha_1 \ldots \alpha_n.\tau \\ \text{where } \{\alpha_1, \ldots, \alpha_n\} &= \mathit{FTV}(\tau) \setminus (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(T)) \\ \mathit{AppClose}(\tau, \Gamma, T) &= \forall \alpha_1 \ldots \alpha_n.\tau \\ \text{where } \{\alpha_1, \ldots, \alpha_n\} &= (\mathit{FTV}(\tau) \setminus (\mathit{FTV}(\Gamma) \cup \mathit{FTV}(T))) \cap \mathit{AppTypeVar} \end{aligned}$$

Figure 7: The augmented type system for Control ML

Main Lemma 6.5 (Subject Reduction) *If* $\overset{a}{\triangleright} e_1 : \tau, T$ *and* $\tau \in T$ *and* $e_1 \longmapsto e_2$ *then* $\overset{a}{\triangleright} e_2 : \tau', T$ *and* $\tau' \in T$.

Proof. The proof proceeds by case analysis according to the reduction $e_1 \longmapsto e_2$. Each case is easily adapted from the corresponding case in the proof of Typability Preservation with straightforward adaptations of the necessary lemmas. ∎

With Uniform Evaluation and the fact that faulty expressions are untypable (in the ordinary system), we obtain a syntactic soundness theorem for **abort**-free expressions.

Theorem 6.6 (Syntactic Soundness) *If* $\triangleright\ e : \tau$ *and* e *contains no* **abort**-*expressions then either* $e \Uparrow$ *or* $e \longmapsto\!\!\!\!\rightarrow v$ *and* $\triangleright\ v : \tau$.

Proof. By Uniform Evaluation, either $e \longmapsto\!\!\!\!\rightarrow e'$ and e' is faulty, or $e \Uparrow$, or $e \longmapsto\!\!\!\!\rightarrow v$. Since $\triangleright\ e : \tau$ and e contains no **abort**-expressions, by Correspondence $\overset{a}{\triangleright} e : \tau, \{\tau\}$. Subject Reduction then implies $\overset{a}{\triangleright} e' : \tau, \{\tau\}$. Similarly $\overset{a}{\triangleright} v : \tau, \{\tau\}$, and by Correspondence $\triangleright\ e' : \tau$ and $\triangleright\ v : \tau$. Suppose $e \longmapsto\!\!\!\!\rightarrow e'$ and e' is faulty. Since faulty expressions are untypable, $\triangleright\ e' : \tau$ is a contradiction, therefore this case cannot occur. Hence either $e \Uparrow$ or $e \longmapsto\!\!\!\!\rightarrow v$ and $\triangleright\ v : \tau$. ∎

7 Discussion

Subject reduction is the key lemma in our approach to proving type soundness. In order to demonstrate subject reduction, it must be possible to assign a type to each intermediate evaluation state of a program. This is most easily accomplished by specifying evaluation as rewriting. Rewriting may be specified as local reductions, as our calculus for Core ML, or as program (or top-level) rewriting, as in Control ML.

Through most of this paper, we present the semantics of the various languages with calculi that permit local reductions. It is also possible to use our proof technique with a semantics that specifies only program reductions ($\longmapsto$), as in Control ML. Such a semantics for Reference ML is slightly simpler, as the *ref*, ρ_{merge}, and ρ_{lift} reductions coalesce into one program reduction, but the structure of the proof is essentially the same. Our work on an alternative type system for references uses this approach [38]. We chose to use calculi in this paper as the resulting proofs are more regular in structure.

In specifying the semantics of references as a calculus, we use an additional expression form, the ρ-expression, that is not present in ML. As noted earlier, ρ-expressions may be regarded as abbreviations, both from a semantic perspective and a typing perspective (the typing rule for ρ-expressions can be derived from the abbreviation). In principle, it is possible to eliminate ρ-expressions from the syntax, however the specification of redexes in the reduction rules becomes complicated. Alternatively, ρ-expressions may be considered as belonging to the state space of evaluation, and not to the language of programs that may be written by a programmer. In a presentation of the typing rules for consumption by programmers, the typing rule for ρ-expressions may be deleted. However, unlike ρ-expressions, the abort-expressions of the continuation fragment cannot be considered as abbreviations [10]. To obtain a strong soundness theorem, abort-expressions must be considered as belonging only to the evaluation space, and not to the syntax of programs. The exception fragment has no such additional expressions.

Other operational or denotational techniques for specifying semantics introduce additional semantic objects, such as closures and stores, rather than additional expression forms. Since the type system does not apply to such objects, stating and proving strong type soundness requires introducing a semantic relation ($\models$) between semantic objects and types (see Section 2). We believe that introducing additional expression forms and typing rules is the simpler choice, and offer the simpler proofs produced by our method as evidence.

While in this paper we have concentrated on the essential aspects of ML, there are other features of static type systems that we believe our technique can address. STANDARD ML contains a datatype specification mechanism that allows the definition of new types and associated data constructors. This mechanism is indispensable when writing non-trivial ML programs. Subtyping, inheritance, and type inference for records are a strong focus of recent research efforts to explain object-oriented languages, as many popular object-oriented languages have unsound static type systems. Reppy [29] has successfully addressed concurrency with our technique; it should also be possible to treat nondeterminism and distributed computing. We have used our technique to prove an alternative type system for references sound [38]. Finally, it may be possible to adapt our technique to demonstrate the consistency of module systems like that of STANDARD ML.

Acknowledgements

We would like to thank Bob Harper for pointing out that type preservation is known as subject reduction in combinatory logic, and Hans Boehm and the anonymous referees for comments on earlier drafts of this paper.

Appendix: The Core of Standard ML

Syntax

$$
\begin{aligned}
e &::= v \mid e_1\ e_2 \mid \textsf{let}\ x\ \textsf{be}\ e_1\ \textsf{in}\ e_2 \mid \rho\theta.e \mid \textsf{exception}\ \chi\ \textsf{in}\ e\\
v &::= c \mid x \mid \textsf{Y} \mid \lambda x.e \mid \textsf{ref} \mid\ ! \mid\ := \mid \textsf{raise} \mid e\ \textsf{handle} \mid\ :=\ v \mid \textsf{raise}\ v \mid e\ \textsf{handle}\ v\\
\theta &::= \{\ \langle x, v\rangle\ \}^*\\
\chi &::= x^*\\
a &::= \{\textsf{exception}\ \chi\ \textsf{in}\}\ \{\rho\theta.\}\ v \mid \{\textsf{exception}\ \chi x\ \textsf{in}\}\ \{\rho\theta.\}\ \textsf{raise}\ x\ v
\end{aligned}
$$

Semantics

$$
\begin{aligned}
&eval(e) = a \text{ iff } e \longmapsto\!\!\!\!\!\rightarrow a\\
&E[e] \longmapsto E[e'] \text{ iff } e \longrightarrow e'\\
&E ::= [\,] \mid E\ e \mid v\ E \mid \textsf{let}\ x\ \textsf{be}\ E\ \textsf{in}\ e \mid E\ \textsf{handle}\ v_1\ v_2 \mid \rho\theta.E \mid \textsf{exception}\ \chi\ \textsf{in}\ E
\end{aligned}
$$

Functional ML

$$
\begin{array}{lrcll}
(\delta) & c\ v & \longrightarrow & \delta(c, v) & \text{if } \delta(c, v) \text{ is defined}\\
(\beta_{\text{v}}) & (\lambda x.e)\ v & \longrightarrow & e[x \mapsto v] & \\
(let) & \textsf{let}\ x\ \textsf{be}\ v\ \textsf{in}\ e & \longrightarrow & e[x \mapsto v] & \\
(Y) & \textsf{Y}\ v & \longrightarrow & v\ (\lambda x.(\textsf{Y}\ v)\ x) &
\end{array}
$$

References

$$
\begin{array}{lrcll}
(ref) & \textsf{ref}\ v & \longrightarrow & \rho\langle x, v\rangle.x & \\
(deref) & \rho\theta\langle x, v\rangle.R[!\ x] & \longrightarrow & \rho\theta\langle x, v\rangle.R[v] & \\
(assign) & \rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] & \longrightarrow & \rho\theta\langle x, v_2\rangle.R[v_2] & \\
(\rho_{merge}) & \rho\theta_1.\rho\theta_2.e & \longrightarrow & \rho\theta_1\theta_2.e & \\
(\rho_{lift}) & R[\rho\theta.e] & \longrightarrow & \rho\theta.R[e] & \text{if } R \neq [\,]
\end{array}
$$

$$
R ::= [\,] \mid R\ e \mid v\ R \mid \textsf{let}\ x\ \textsf{be}\ R\ \textsf{in}\ e \mid R\ \textsf{handle}\ v_1\ v_2
$$

Exceptions

$$
\begin{array}{lrcll}
(raise) & U[\textsf{raise}\ x\ v] & \longrightarrow & \textsf{raise}\ x\ v & \text{if } U \neq [\,]\\
(handle) & (\textsf{raise}\ x\ v_1)\ \textsf{handle}\ x\ v_2 & \longrightarrow & v_2\ v_1 & \\
(reraise) & \begin{array}{r}\textsf{exception}\ \chi x_1 x_2\ \textsf{in}\\ X[(\textsf{raise}\ x_1\ v_1)\ \textsf{handle}\ x_2\ v_2]\end{array} & \longrightarrow & \begin{array}{l}\textsf{exception}\ \chi x_1 x_2\ \textsf{in}\\ \quad X[\textsf{raise}\ x_1\ v_1]\end{array} & x_1 \neq x_2\\
(unhandle) & v_1\ \textsf{handle}\ x\ v_2 & \longrightarrow & v_1 & \\
(exn_{merge}) & \textsf{exception}\ \chi_1\ \textsf{in exception}\ \chi_2\ \textsf{in}\ e & \longrightarrow & \textsf{exception}\ \chi_1\chi_2\ \textsf{in}\ e & \\
(exn_{lift}) & X[\textsf{exception}\ \chi\ \textsf{in}\ e] & \longrightarrow & \textsf{exception}\ \chi\ \textsf{in}\ X[e] & \text{if } X \neq [\,]
\end{array}
$$

$$
\begin{aligned}
U &::= [\,] \mid U\ e \mid v\ U \mid \textsf{let}\ x\ \textsf{be}\ U\ \textsf{in}\ e\\
X &::= [\,] \mid X\ e \mid v\ X \mid \textsf{let}\ x\ \textsf{be}\ X\ \textsf{in}\ e \mid X\ \textsf{handle}\ v_1\ v_2 \mid \rho\theta.X
\end{aligned}
$$

Typing

Functional ML

(var) $$\Gamma \triangleright x : \tau \ \text{ if } \ \Gamma(x) \succ \tau$$

(const) $$\Gamma \triangleright c : \tau \ \text{ if } \ \mathit{TypeOf}(c) \succ \tau$$

(Y) $$\Gamma \triangleright \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2$$

(abs) $$\frac{\Gamma[x \mapsto \tau_1] \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \to \tau_2}$$

(app) $$\frac{\Gamma \triangleright e_1 : \tau_1 \to \tau_2 \quad \Gamma \triangleright e_2 : \tau_1}{\Gamma \triangleright e_1\ e_2 : \tau_2}$$

(let$_v$) $$\frac{\Gamma \triangleright v : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \triangleright e : \tau_2}{\Gamma \triangleright \textsf{let } x \textsf{ be } v \textsf{ in } e : \tau_2}$$

(let$_e$) $$\frac{\Gamma \triangleright e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{AppClose}(\tau_1, \Gamma)] \triangleright e_2 : \tau_2 \quad e_1 \notin \mathit{Values}}{\Gamma \triangleright \textsf{let } x \textsf{ be } e_1 \textsf{ in } e_2 : \tau_2}$$

$$\mathit{Close}(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1, \ldots, \alpha_n\} = \mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma)$$
$$\mathit{AppClose}(\tau, \Gamma) = \forall \alpha_1 \ldots \alpha_n.\tau \text{ where } \{\alpha_1, \ldots, \alpha_n\} = (\mathit{FTV}(\tau) \setminus \mathit{FTV}(\Gamma)) \cap \mathit{AppTypeVar}$$

References

(ref) $$\Gamma \triangleright \textsf{ref} : \tau \to \tau\ \mathit{ref} \ \text{ if } \ \tau \text{ is imperative}$$

(deref) $$\Gamma \triangleright\ ! : \tau\ \mathit{ref} \to \tau$$

(assign) $$\Gamma \triangleright := : (\tau\ \mathit{ref} \to \tau \to \tau)$$

(rho) $$\frac{\begin{array}{l}\Gamma[x_1 \mapsto \tau_1\ \mathit{ref}] \ldots [x_n \mapsto \tau_n\ \mathit{ref}] \triangleright e : \tau \\ \Gamma[x_1 \mapsto \tau_1\ \mathit{ref}] \ldots [x_n \mapsto \tau_n\ \mathit{ref}] \triangleright v_i : \tau_i \quad \tau_i \text{ is imperative} \quad 1 \le i \le n\end{array}}{\Gamma \triangleright \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau}$$

Exceptions

(raise) $$\Gamma \triangleright \textsf{raise} : \tau_1\ \mathit{exn} \to \tau_1 \to \tau_2$$

(handle) $$\frac{\Gamma \triangleright e : \tau_2}{\Gamma \triangleright e \textsf{ handle} : \tau_1\ \mathit{exn} \to (\tau_1 \to \tau_2) \to \tau_2}$$

(exn) $$\frac{\Gamma[x_1 \mapsto \tau_1\ \mathit{exn}] \ldots [x_n \mapsto \tau_n\ \mathit{exn}] \triangleright e : \tau \quad \tau_i \text{ is imperative} \quad 1 \le i \le n}{\Gamma \triangleright \textsf{exception } x_1 \ldots x_n \textsf{ in } e : \tau}$$

References

[1] Abadi, M., Cardelli, L., Pierce, B., and Plotkin, G. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems 13*, 2 (April 1991), 237–268. Previously appeared in: *Proceedings of the 16th Annual Symposium on Principles of Programming Languages* (January 1989), 213–227.

[2] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[3] Crank, E., and Felleisen, M. Parameter-passing and the lambda calculus. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), 233–244.

[4] Curry, H. B., and Feys, R. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1958.

[5] Damas, L., and Milner, R. Principal type schemes for functional programs. *Proceedings of the 9th Annual Symposium on Principles of Programming Languages* (January 1982), 207–212.

[6] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[7] Donahue, J., and Demers, A. Data types are values. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 426–445.

[8] Duba, B. F., Harper, R., and MacQueen, D. Typing first-class continuations in ML. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), 163–173.

[9] Felleisen, M. The theory and practice of first-class prompts. *Proceedings of the 15th Annual Symposium on Principles of Programming Languages* (1988), 180–190.

[10] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming 17* (1991), 35–75. Preliminary version in: *Proceedings of the European Symposium on Programming, LNCS 432* (1990), 134–151.

[11] Felleisen, M., and Friedman, D. P. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, M. Wirsing, Ed. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986, pp. 193–217.

[12] Felleisen, M., and Friedman, D. P. A syntactic theory of sequential state. *Theoretical Computer Science 69*, 3 (1989), 243–287. Preliminary version in: *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, 1987, 314-325.

[13] Felleisen, M., Friedman, D. P., Kohlbecker, E. E., and Duba, B. A syntactic theory of sequential control. *Theoretical Computer Science 52*, 3 (1987), 205–237. Preliminary version in: *Proceedings of the Symposium on Logic in Computer Science*, 1986, 131–141.

[14] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. TR-100, Rice University, June 1989. To appear in: *Theoretical Computer Science*, 1992.

[15] Hindley, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (December 1969), 29–60.

[16] Hindley, R. J., and Seldin, J. P. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[17] Leroy, X., and Weis, P. Polymorphic type inference and assignment. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), 291–302.

[18] MacQueen, D. B., Plotkin, G., and Sethi, R. An ideal model for recursive polymorphic types. *Proceedings of the 11th Annual Symposium on Principles of Programming Languages* (January 1984), 165–174.

[19] Mason, I., and Talcott, C. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the International Conference on Automata, Languages, and Programming, LNCS 372* (1989), Springer-Verlag, pp. 574–588.

[20] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[21] Milner, R., and Tofte, M. Co-induction in relational semantics. *Theoretical Computer Science 87* (1991), 209–220.

[22] Milner, R., and Tofte, M. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[23] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[24] Mitchell, J. C., and Harper, R. The essence of ML. *Proceedings of the 15th Annual Symposium on Principles of Programming Languages* (January 1988), 28–46.

[25] Mitchell, J. C., and Plotkin, G. D. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems 10*, 3 (July 1988), 470–502. Also appeared in: *Proceedings of the 11th Annual Symposium on Principles of Programming Languages*, 1984, 37-51.

[26] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[27] Plotkin, G. D. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Århus University, September 1981.

[28] Rees, J., and Clinger, W. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices 21*, 12 (December 1986), 37–79.

[29] Reppy, J. H. *Higher-order Concurrency.* PhD thesis, Cornell University, 1991.

[30] Reynolds, J. Definitional interpreters for higher order programming languages. *ACM Conference Proceedings* (1972), 717–740.

[31] Reynolds, J. C. On the relation between direct and continuation semantics. *Proceedings of the International Conference on Automata, Languages, and Programming* (1974), 141–156.

[32] Scott, D. Data types as lattices. *SIAM Journal of Computing 3*, 5 (1976), 522–586.

[33] Seldin, J. P. A sequent calculus for type assignment. *Journal of Symbolic Logic 42* (1977), 11–28.

[34] Standard ML of New Jersey release notes (version 0.75). AT&T Bell Laboratories, November 1991.

[35] Talpin, J.-P., and Jouvelot, P. The type and effect discipline. Tech. Rep. EMP-CRI A/206, Ecole des Mines de Paris, July 1991.

[36] Tofte, M. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1987.

[37] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[38] Wright, A. K. Typing references by effect inference. In *Proceedings of the European Symposium on Programming, LNCS 582* (1992), Springer-Verlag, pp. 473–491.