1990?

A Polymorphic λ-Calculus for Imperative ML

Andrew Wright Matthias Felleisen

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

Summary

Theoretical research on typed programming languages ignores the presence of imperative programming facilities in practical languages. Although it is true that modified versions of the results carry over to imperative extensions, the adaptation process is often difficult. In practically all instances, the imperative extensions demand different semantic models and proof techniques than the functional language.

In this paper we investigate the connection between λ-calculi and statically typed languages. We show that modeling typed programming languages with λ-calculi allows the natural and straightforward extension of type soundness results to imperative features, such as assignment, exceptions, and non-local control.

- Our *primary* contribution is the clarification of the connection between a calculus for a statically typed programming language and the type system of the language in the form of Type Preservation and Type Soundness Theorems. The technique is powerful enough to encompass polymorphism and is flexible enough to deal with imperative features.
- The *secondary* contribution is a series of Type Preservation and Type Soundness theorems for specific fragments of Imperative ML, in particular, ML with reference cells, exceptions, and a newly proposed control mechanism.

1 The Essence and the Core of ML

In a recent paper Mitchell and Harper [15] presented a polymorphic λ-calculus for the essence of STANDARD ML [14]. The core of STANDARD ML is an expressive, statically typed, polymorphic programming language. It is basically a λ-calculus expression language, extended with facilities for declaring, raising, and handling exceptions, and for creating, dereferencing, and updating reference cells. What sets STANDARD ML apart from other strongly typed programming languages is the requirement that there be a decidable type inference algorithm for programs *without* type declarations. In other words, STANDARD ML is basically an untyped programming language that only evaluates typable expressions.

The work by Mitchell and Harper is important because it provides a calculus for a *useful* and *practical* programming language. However, as has become customary in the study of language calculi, they "omit exceptions and references [from their treatment]: what is left is still quite interesting," justifying this decision with the desire "to simplify the presentation" [15: page 29]. Thus, Mitchell and Harper concentrate on the functional parts of the core (plus the module language) but ignore all imperative features based on the belief that a functional sublanguage represents the *essential* parts of a language.

While we agree that at some high level of abstraction, the omission of exceptions and references affects only the presentation and not the essence of results on programming language calculi, we also claim that the omission concerns *practical* core facilities, and that it hides the tremendous difficulties of extending, say, type soundness results to the full core of STANDARD ML. A cursory reading of the papers on this topic reveals that every imperative extension appears to require a restructuring of the semantics for the functional sublanguage and a new set of proof techniques for the establishment of type soundness.

Damas [5] was the first to propose and study an ML-like polymorphic type system for first-class references. He used a denotational store model to study the semantics of the language and to prove the soundness of the type inference mechanism. The proof technique for the type soundness result relied on inclusive predicates and sophisticated domain constructions. Later Tofte [19] found a mistake in Damas's complicated proof (but not in the type system) and proceeded to design and explore his own type system for references. He used structural operational semantics with stores instead of denotational models, and relied on a newly invented principle of co-induction to prove type soundness [13]. He claimed, without proof, that his technique would carry over to the exception handling facilities of STANDARD ML. Recently, Duba *et al.* [6] showed how to add Scheme's continuation facilities [18]—a general control mechanism—to STANDARD ML. Like Tofte, they used a structural operational semantics but based on continuation codes and continuation-passing style as opposed to stores and store-passing style.

In all three cases, the semantic model for the extended language is not an extension of the model for the functional sublanguage, and the proof of type soundness is not an

extension of the proof for the functional case. Indeed, the proof techniques all differ, and are all complicated. Finally, even though both Tofte and Duba *et al.* use structural operational semantics, a combination of the two results is again non-trivial because they use two completely different approaches.

As an alternative, we develop an untyped calculus for the *entire* core of STANDARD ML. The approach draws from our earlier work on the λ-v-{C,S,CS}-calculi [3, 8, 9, 10], which are adequate and conservative extensions of Plotkin's [16] λ-value calculus for imperative programming languages. The calculi for Core ML and its major fragments provide a simple equational theory based on reductions and, together with an evaluation strategy, define an operational semantics. For each of these calculi, we prove a Type Preservation Theorem showing that the reduction of a typed term results in a term of the same type. The Type Preservation Theorem almost immediately implies a Strong and a Weak Type Soundness Theorem à la Milner [12] for the respective operational semantics. Unlike previous work, all of these results rely on the *same* proof techniques.

The following section illustrates the basic idea with the example of Functional ML. Section 3 is dedicated to Core ML, the full imperative core of STANDARD ML. The fourth section indicates how to apply our technique to the newly proposed control structure for STANDARD ML [6]. The fifth section addresses related work. The last section summarizes our results.

2 Functional ML

Functional ML is a call-by-value applicative language with constants and higher-order functions. A natural basis for a calculus of Functional ML is Plotkin's untyped λ-value-calculus [16]. It captures the semantics of the untyped syntax of ML programs and thus permits a simple proof of type soundness.

Let $x \in \mathit{Var}$ be a set of variables and $c \in \mathit{Const}$ be a set of constants. Then the expressions $e \in \mathit{Exp}$ and values $v \in \mathit{Val}$ of Functional ML are:

$$e ::= v \mid e_1\ e_2 \mid \texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2$$
$$v ::= c \mid x \mid \mathrm{Y} \mid \lambda x.e$$

Free and bound variables are defined as usual; λ and `let` are binding constructs. The `let`-expression binds x in e_2. Following Barendregt [2], we assume that bound variables are always distinct from free variables in distinct meta-variables ranging over expressions; we identify expressions that differ only by a renaming of the bound variables.

The set of constants is the disjoint union of a set of *basic* constants, *BConst*, and a set of *functional* constants, *FConst*. Basic constants are flat data, such as numbers and booleans. Functional constants are primitive operations, such as $+, -, *$, and $/$. Constants do not include structured data, such as pairs, lists, arrays, etc. To abstract from the precise set of

constants, we only assume the existence of a partial function $\delta : FConst \times BConst \rightharpoonup Val^{\circ}$ that interprets the application of functional constants to basic constants and yields closed values.

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

$$c_1\ c_2 \longrightarrow \delta(c_1, c_2) \quad \text{if } \delta(c_1, c_2) \text{ is defined} \qquad (\delta)$$

$$(\lambda x.e)\ v \longrightarrow e[x \mapsto v] \qquad (\beta_v)$$

$$\texttt{let}\ x\ \texttt{be}\ v\ \texttt{in}\ e \longrightarrow e[x \mapsto v] \qquad (let)$$

$$\mathrm{Y}\ v \longrightarrow v\ (\lambda x.(\mathrm{Y}\ v)\ x) \qquad (Y)$$

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 and equations in the usual manner. The relation $\twoheadrightarrow$ is the reflexive, transitive, and compatible closure of $\mathbf{v}$. An equational system, $\boldsymbol{\lambda}_{\mathbf{v}} \vdash e = e'$, may be constructed as the congruence closure of $\mathbf{v}$.

The calculus satisfies Church-Rosser and Standardization properties; the proofs are variants of Plotkin's proofs for the λ-value-calculus [16]. By Standardization, we know that a program 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 for *programs* (closed expressions). Let $\longmapsto$ be a function such that:

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

where $E ::= [] \mid E\ e \mid v\ E \mid \texttt{let}\ x\ \texttt{be}\ E\ \texttt{in}\ e$. The special contexts E are *evaluation contexts*: they ensure that the leftmost-outermost reduction is the only applicable reduction. Let $\longmapsto\!\!\!\!\twoheadrightarrow$ be the reflexive and transitive closure of $\longmapsto$. Then the partial function *eval* is:

$$eval(e) = v \quad \text{iff} \quad e \longmapsto\!\!\!\!\twoheadrightarrow v$$

for closed expressions e. It follows from the Standardization Theorem that the stepping relation $\longmapsto$ is a function, and therefore *eval* is a function, too. Evaluating from left to right is consistent with STANDARD ML.

For an arbitrary program, *eval* may yield a value or may be undefined. A closer look at the stepping function reveals that the latter case can occur for two different reasons. First, there may be an infinite sequence of reduction steps that does not end in a value, *i.e.*, the program represents an infinite computation. Or, after some number of steps, there may be no applicable $\longmapsto$-reduction, yet the last member of the sequence is not a value, *i.e.*, the evaluation is stuck. *Stuck states* arise when a basic constant is applied to arbitrary values, when a functional constant is applied to abstractions, or when a functional constant is applied to a value outside the domain of the function. In all three cases, stuck states are a manifestation of *type errors*.

Formally, the stuck states of Functional ML are closed expressions s such that:

$$s = E[c\ v] \text{ where } c \in \mathit{BConst},\ v \notin \mathit{BConst}, \text{ or } \delta(c, v) \text{ is undefined.}$$

A Uniform Evaluation Lemma summarizes the behavior of evaluation.

Lemma 2.1 (Uniform evaluation) *For e closed, either $e \longmapsto\!\!\!\!\!\rightarrow v$, or $e \longmapsto\!\!\!\!\!\rightarrow s$, or for all e' such that $e \longmapsto\!\!\!\!\!\rightarrow e'$, there exists e'' such that $e' \longmapsto e''$.*

A desire to eliminate the possibility of type errors motivates the introduction of a static type system. A static type system imposes additional context-sensitive restrictions on the domain of *eval* to filter out expressions that might get stuck. The type system is *sound* if it filters out all expressions that get stuck. As the set of expressions that evaluate to stuck states is not recursive, any sound, decidable, static type system must also eliminate some good expressions. The problem is to design a type system that rejects as few good expressions as possible, while remaining sound. ML's polymorphic type system is both sound and practical [12].

The type system is formulated as a proof system that assigns types to expressions. A subset of the typing rules assign *type schemes* to variables. *Types* τ and *type schemes* σ are defined as:

$$\tau ::= \iota \mid \alpha \mid \tau_1 \to \tau_2 \qquad \sigma ::= \tau \mid \forall\alpha_1 \ldots \alpha_n.\tau$$

where ι ranges over a set of ground types for basic constants, and α ranges over a set of *type variables*. Free and bound variables are defined as usual, where $\forall$ is a binding construct. We identify $\forall.\tau$ with τ. A type scheme σ *generalizes* a type τ, written $\sigma \succ \tau$, if $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau'$ and there exist types $\vec{\tau_i}$ such that $\tau'[\vec{\alpha_i} \mapsto \vec{\tau_i}] = \tau$. A *type environment* $\Gamma : \mathit{Var} \multimap \mathit{TypeScheme}$ is an assignment of type schemes to variables. We define

$$\mathit{Close}_\Gamma\tau = \forall\alpha_1 \ldots \alpha_n.\tau \quad \textit{where } \{\alpha_1 \ldots \alpha_n\} = FTV(\tau) \setminus FTV(\Gamma)$$

where FTV gives the free type variables of a type or type environment. The relation *TypeOf* gives types for constants. A type judgement $\Gamma \triangleright e : \tau$ states that the expression e has type τ in type environment Γ. The typing rules for Functional ML are:

$$\Gamma \triangleright x : \tau \text{ if } \Gamma x \succ \tau \ \ (\tau\text{-}x) \qquad \Gamma \triangleright c : \tau \text{ if } \mathit{TypeOf}\, c \succ \tau \ \ (\tau\text{-}c)$$

$$\Gamma \triangleright \mathrm{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \ \ (\tau\text{-Y})$$

$$\frac{\Gamma[x \mapsto \tau_1] \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \to \tau_2} \ (\tau\text{-}\lambda) \qquad \frac{\Gamma \triangleright e_1 : \tau_1 \to \tau_2 \quad \Gamma \triangleright e_2 : \tau_1}{\Gamma \triangleright e_1\ e_2 : \tau_2} \ (\tau\text{-app})$$

$$\frac{\Gamma \triangleright e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}_\Gamma\tau_1] \triangleright e_2 : \tau_2}{\Gamma \triangleright \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 : \tau_2} \ (\tau\text{-let})$$

We refer to the collection of τ-x, τ-c, τ-Y, and τ-λ (but not τ-let) as τ-Λ. Finally, for type soundness to make sense for an unspecified set of constants, we require that

$$\delta(c, v) \text{ is defined iff } \Gamma \triangleright c : \tau_1 \to \tau_2,\ \Gamma \triangleright v : \tau_1, \text{ and } \Gamma \triangleright \delta(c, v) : \tau_2.$$

This restriction rules out functional constants such as division that are not defined on all values of their input type.

The type system has two important properties. First, there is an algorithm to determine whether an expression has a type [12]. Second, if an expression has a type, there is an algorithm to determine its *principal* type [4]. All other types an expression may have can be derived as substitution instances of its principal type.

The key observation that underlies our proof technique for type soundness is that reductions preserve type.

Theorem 2.2 (Type preservation) *If* $\Gamma \triangleright e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $\Gamma \triangleright e_2 : \tau$.

Proof Idea. The proof proceeds by cases according to the reduction $e_1 \longrightarrow e_2$. The case for δ relies on the restriction on δ; the case for β_v requires Lemma 2.3. ∎

Lemma 2.3 *If* $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \triangleright e : \tau'$ *and* $\Gamma \triangleright v : \tau$ *and* $\{\alpha_1, \ldots, \alpha_n\} \cap FTV(\Gamma) = \emptyset$ *then* $\Gamma \triangleright e[x \mapsto v] : \tau'$.

Proof Idea. The proof proceeds by induction on the proof of $\Gamma[x \mapsto \forall\alpha_1 \ldots \alpha_n.\tau] \triangleright e : \tau'$, and case analysis on the last step. ∎

Given type preservation and uniform evaluation, we can prove a general type soundness theorem for programs (closed expressions).

Theorem 2.4 (Type soundness) *If* $\triangleright e : \tau$ *then either* $e \longmapsto\!\!\!\!\rightarrow v$ *and* $\triangleright v : \tau$ *or for all* e' *such that* $e \longmapsto\!\!\!\!\rightarrow e'$, *there exists* e'' *such that* $e' \longrightarrow e''$.

Proof. We require another lemma regarding the non-typability of stuck states. Then, by uniform evaluation, either e diverges, $e \longmapsto\!\!\!\!\rightarrow v$, or $e \longmapsto\!\!\!\!\rightarrow s$. Since $\triangleright e : \tau$ by assumption, type preservation implies $\triangleright v : \tau$ and $\triangleright s : \tau$. But the latter contradicts the non-typability of stuck states, and therefore, $e \longmapsto\!\!\!\!\rightarrow s$ cannot occur. ∎

Lemma 2.5 (Non-typability of stuck states) *There is no* τ *such that* $\triangleright s : \tau$.

Proof Idea. Let $s = E[c\ v]$ and assume $\triangleright E[c\ v] : \tau$. Then $\Gamma \triangleright c\ v : \tau_2$. By τ-app, $\Gamma \triangleright c : \tau_1 \rightarrow \tau_2$ and $\Gamma \triangleright v : \tau_1$, which with the conditions on δ imply $\delta(c, v)$ is defined, leading to a contradiction. Since τ is arbitrary, there is no τ such that $\triangleright E[c\ v] : \tau$. ∎

We can also show that the classical Milner-style strong and weak type soundness theorems hold for evaluation. Strong soundness follows directly from Theorem 2.2.

Theorem 2.6 (Strong soundness) *If* $\triangleright e : \tau$ *and* $eval(e) = v$ *then* $\triangleright v : \tau$.

To state weak soundness, *i.e.* that "typed programs do not go wrong", we must make explicit the notion of "wrong" by extending the definition of *eval* with an additional clause:

$$eval(e) = \text{WRONG} \quad \textit{if } e \longmapsto\!\!\!\!\!\rightarrow s.$$

Now weak soundness follows from Theorem 2.4.

Theorem 2.7 (Weak soundness) *If* $\triangleright e : \tau$ *then* $eval(e) \neq \text{WRONG}$.

3 Core ML: Adding References and Exceptions

Next to functional facilities, first-class references and exceptions are the most important core programming facilities of STANDARD ML. The first two subsections present calculi for extensions of Functional ML with references and exceptions, respectively. The first extension uses our previous work [8, 9] on a calculus of state, λ-v-S, and in particular its cell-oriented variant [3]. The second subsection is a modified version of our control calculus λ-v-c [9, 10], especially its fragment with prompts [7]. The third subsection briefly indicates how to combine the two calculi into a calculus for Core ML.

3.1 References

The extension of the λ-value-calculus to a calculus of functions and references requires the extension of the syntax with a new kind of expression and several new values:

$$\begin{array}{l} e ::= \ldots \mid \rho\theta.e \quad \text{where} \quad \theta ::= \epsilon \mid \theta\langle x, v\rangle \quad \text{such that } \langle x, \cdot\rangle \text{ does not occur in } \theta \\ v ::= \ldots \mid \texttt{ref} \mid \texttt{!} \mid \texttt{:=} \mid \texttt{:=}\, x \end{array}$$

Above, θ represents a finite function from variables to values, *i.e.* we identify all rearrangements of θ; $\rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ binds $x_1, \ldots, x_n$ in $v_1, \ldots, v_n$ and e.

The expressions `ref`, `!`, and `:=` are the familiar operations of ML. 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 assignment operation `:=` 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 `:=` to a variable *is* a value—it may be thought of as a *capability* to assign to a cell. The ρ-expression is an abbreviation of a `let`-expression:

$$\begin{array}{rcl} \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e & \equiv & \texttt{let } x_1 \texttt{ be ref } u_1 \ \ldots \ x_n \texttt{ be ref } u_n \texttt{ in} \\ & & (\lambda x_1 \ldots x_n x_{n+1}.x_{n+1})(\texttt{:=}\ x_1\ v_1) \ldots (\texttt{:=}\ x_n\ v_n)\ e \end{array}$$

where $u_1, \ldots, u_n$ are arbitrary values (of the right type); for technical reasons, we do not treat it as such. A ρ-expression plays the role of a store fragment during the reduction of a program with imperative assignment statements.

The set of reductions is extended with several new rules:

$$\begin{aligned}
\texttt{ref}\ v &\longrightarrow \rho\langle x, v\rangle.x && (\mathit{ref})\\
\rho\theta\langle x, v\rangle.R[\texttt{!}\ x] &\longrightarrow \rho\theta\langle x, v\rangle.R[v] && (\mathit{deref})\\
\rho\theta\langle x, v_1\rangle.R[\texttt{:=}\ x\ v_2] &\longrightarrow \rho\theta\langle x, v_2\rangle.R[v_2] && (\mathit{assign})\\
\rho\theta_1.\rho\theta_2.e &\longrightarrow \rho\theta_1\theta_2.e && (\rho_\cup)\\
R[\rho\theta.e] &\longrightarrow \rho\theta.R[e] && (\rho_{\mathit{lift}})
\end{aligned}$$

We refer to the union of these five reductions as **r**. The definition of **r** relies on a set of *evaluation contexts* R:

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

The use of evaluation contexts in the new notions of reduction reflects the additional sequencing restrictions that the introduction of side-effects in a programming language requires for applications. 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 evaluation contexts build this minimal order of evaluation into those reductions that refer to reference cells and their operations; again following ML, we choose to evaluate from left to right.

For 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, which contains the cell's current value. The dereference operator `!` selects a cell's value from the *closest* ρ-expression (relative to evaluation 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 $\rho_\cup$ reductions, which lift a partial store out of evaluation contexts, and merge it with outer stores (by the variable convention cells are renamed appropriately to avoid collision). An assignment replaces a cell's current value in the closest ρ-expression, after performing all necessary lift and merge steps, and returns the value assigned.

See Appendix A *for an example reduction sequence.*

The notions of reduction **r** and **v** (adapted to the full syntax) form the basis for a calculus of functions and references. We refer to the union as **vr**. The extended notion of reduction gives rise to a system of reductions and equations as before. The calculus satisfies the same basic properties that Plotkin proved for the λ-value-calculus [3]. In particular, the calculus's standard reduction can emulate the evaluation function of an extended SECD-machine [8].

Designing a polymorphic type system for a functional language is relatively straightforward [12]. Incorporating reference cells requires more machinery [5, 11, 19]. We adopt Tofte's technique [19], which is one of the simplest known, but still complicates the type system significantly. It is the method used in the definition of STANDARD ML [14].

The central idea 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 classified as either imperative or applicative; the latter set of type variables is named *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 which might appear in the types of values in the store are required to be imperative, and are not generalized.

Like all other techniques for typing references, Tofte's system requires modifying the typing rule for `let`-expressions. First, we define a secondary notion of closing:

$$AppClose_\Gamma \tau = \forall \alpha_1 \ldots \alpha_n.\tau \quad \textit{where } \{\alpha_1 \ldots \alpha_n\} = (FTV(\tau) \setminus FTV(\Gamma)) \cap AppTypeVar.$$

Given this, the typing rule for `let`-expressions is split into two rules, according to whether or not the right-hand side of the declaration is a value:

$$\frac{\Gamma \rhd v : \tau_1 \quad \Gamma[x \mapsto Close_\Gamma \tau_1] \rhd e : \tau_2}{\Gamma \rhd \texttt{let } x \texttt{ be } v \texttt{ in } e : \tau_2} \ (\tau\text{-let-}v)$$

$$\frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma[x \mapsto AppClose_\Gamma \tau_1] \rhd e_2 : \tau_2 \quad e_1 \neq v}{\Gamma \rhd \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 : \tau_2} \ (\tau\text{-let-}e)$$

We use τ-*imp* for the collection of τ-Λ, τ-let-v, and τ-let-e. If an expression e has type τ according to τ-*imp*, we write $\Gamma \rhd_i e : \tau$. The new system assigns the same types as the old one to Functional ML expressions.

There are four additional typing rules for reference cells:

$$\Gamma \rhd \texttt{ref} : \tau \to \tau \ ref \quad \text{if } \tau \text{ is imperative} \ (\tau\text{-ref})$$

$$\Gamma \rhd \texttt{!} : \tau \ ref \to \tau \ (\tau\text{-!}) \qquad \Gamma \rhd \texttt{:=} : (\tau \ ref \to \tau \to \tau) \ (\tau\text{-:=})$$

$$\frac{\Gamma[x_1 \mapsto \tau_1 \ ref] \ldots [x_n \mapsto \tau_n \ ref] \rhd v_i : \tau_i, \ e : \tau \quad \tau_i \text{ is imperative} \quad 1 \leq i \leq n}{\Gamma \rhd \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau} \ (\tau\text{-}\rho)$$

The typing rules for `!` and `:=` do not need to be explicitly constrained, since when they are applied they will be constrained by the type of a value already in the store. See Tofte [19] for a detailed discussion of this system. If an expression e is assigned a type τ by the complete set of typing rules, we write $\Gamma \rhd_r e : \tau$.

To prove type soundness for the extended language, we must show that the extended notion of reduction **v** preserves the types of expressions in the new syntax, and that the notion of reduction **r** preserves types. From this, it easily follows that the reductions and equations based on **vr** are type-preserving.

Theorem 3.1 (Type preservation for references) *If* $\Gamma \rhd_r e_1 : \tau$ *and* $e_1 \longrightarrow_{vr} e_2$ *then* $\Gamma \rhd_r e_2 : \tau$.

Based on the Type Preservation Theorem for the reference calculus, we can now easily show Strong and Weak Type Soundness theorems for an appropriate evaluation function. The steps are adapted from the previous section. The adaptation requires the definition of an (extended leftmost-outermost) evaluation strategy, of stuck states (for reference operations, too), and of possible results (values plus ρ-declarations over values).

3.2 Exceptions

In the functional core, constant functions must be defined for *all* values of their input *type*. This precludes constants such as division that are defined on all but a few recognizable input values of correct type. STANDARD ML solves this problem by introducing exceptions.

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

$e ::= \ldots \mid \texttt{exception}\ \chi\ \texttt{in}\ e$ where $\chi ::= \epsilon \mid \chi\, x$ such that x does not occur in χ
$v ::= \ldots \mid \texttt{raise} \mid \texttt{raise}\ x$

In the expression $\texttt{exception}\ x_1 \ldots x_n\ \texttt{in}\ e$, the variables $x_1, \ldots, x_n$ are bound in e. The expression $\texttt{exception}\ \chi\ \texttt{in}\ e$ declares a number of exceptions whose lexical scope is e. When `raise` is applied to two arguments, an exception is raised, terminating execution of the program. The first argument to `raise` must be an exception; the second argument is a parameter of the exception.

The declaration and raising of exceptions requires three new reductions:

$$U[\texttt{raise}\ x\ v] \longrightarrow \texttt{raise}\ x\ v \qquad (\textit{raise})$$

$$\texttt{exception}\ \chi_1\ \texttt{in exception}\ \chi_2\ \texttt{in}\ e \longrightarrow \texttt{exception}\ \chi_1\chi_2\ \texttt{in}\ e \qquad (\texttt{exception}_{\cup})$$

$$X[\texttt{exception}\ \chi\ \texttt{in}\ e] \longrightarrow \texttt{exception}\ \chi\ \texttt{in}\ X[e] \qquad (\texttt{exception}_{\textit{lift}})$$

These new notions of reduction again rely on (two kinds of) evaluation contexts for the enforcement of a specific order of evaluation for applications:

$$X ::= [] \mid X\ e \mid v\ X \mid \texttt{let}\ x\ \texttt{be}\ X\ \texttt{in}\ e$$
$$U ::= [] \mid U\ e \mid v\ U \mid \texttt{let}\ x\ \texttt{be}\ U\ \texttt{in}\ e$$

A `exception`-expression behaves in the same way as a ρ-expression. Indeed, the only difference is that the `exception`-expression does not bind any values to its variables. When an exception is raised, it propagates outwards through pending computations towards the top of the program.

We can now extend the domain of δ to admit functions such as division:[1]

$$\delta : \mathit{FConst} \times \mathit{BConst} \rightharpoonup \mathit{Val}^{\circ} \cup \{\texttt{exception x in raise x}\ v^{\circ}\}.$$

[1] Usually a set of exceptions are defined in an initial environment for constant function errors; we prefer to avoid discussing initial environments and initial typing environments.

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

To provide a modest amount of control, STANDARD ML also offers exception handlers:

$$e ::= \ldots \mid e_1 \texttt{ handle } x\ e_2$$

The **handle**-expression handles lexically bound x-exceptions during the evaluation of the expression e_1. If e_1 returns a value, that value is returned as the result of the **handle**-expression. If e_1 raises an x-exception, e_2 is applied to the argument of the **raise**-expression, and the result of the application is returned from the **handle**-expression. If e_1 raises any other exception, the exception propagates out, possibly to an enclosing handler.

Exception handlers require an extension of the set of X-evaluation contexts:

$$X ::= [] \mid X\ e \mid v\ X \mid \texttt{let } x \texttt{ be } X \texttt{ in } e \mid X \texttt{ handle } x\ e$$

The reductions for handling exceptions are:

$$v \texttt{ handle } x\ e \longrightarrow v \qquad (\textit{handle})$$

$$(\texttt{raise } x\ v) \texttt{ handle } x\ e \longrightarrow e\ v \qquad (\textit{catch})$$

$$\texttt{exception } \chi x_1 x_2 \texttt{ in } X[(\texttt{raise } x_1\ v) \texttt{ handle } x_2\ e] \longrightarrow \texttt{exception } \chi x_1 x_2 \texttt{ in } X[\texttt{raise } x_1\ v] \qquad (\textit{reraise})$$

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 basic notion of reduction for the calculus of functions and exceptions.

In typing exceptions one encounters the same difficulties as for polymorphic references. Thus the typing for exceptions builds on the imperative-core typing rules. The typing rules for exceptions are as follows:

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

$$\Gamma \rhd \texttt{raise} : \tau_1\ exn \to \tau_1 \to \tau_2 \ \ (\tau\text{-raise})$$

$$\frac{\Gamma[x \mapsto \tau_1\ exn] \rhd e_1 : \tau_2 \quad \Gamma[x \mapsto \tau_1\ exn] \rhd e_2 : \tau_1 \to \tau_2}{\Gamma[x \mapsto \tau_1\ exn] \rhd e_1 \texttt{ handle } x\ e_2 : \tau_2} \ (\tau\text{-handle})$$

If an expression e is typable with type τ according to τ-*imp* plus τ-exn, τ-raise, or τ-handle, we write $\Gamma \rhd_x e : \tau$.

To establish soundness for the exception typing, we need to proceed as in the preceding subsection: we extend the type preservation theorem for **v** to the full syntax, prove it for **x**, and combine the results. The main result is the preservation of types across reductions in the calculus of functions and exceptions.

Theorem 3.2 (Type preservation for exceptions) *If* $\Gamma \rhd_x e_1 : \tau$ *and* $e_1 \longrightarrow_{vx} e_2$ *then* $\Gamma \rhd_x e_2 : \tau$.

The rest of Section 2 is adapted mutatis mutandis. *The definition of an evaluation function based on* vx *must account for exceptional results, but is otherwise a simple extension of the functional case.*

3.3 Core ML

We can combine the imperative core with the reference extension and the exception extension to obtain a language with all of the core features of STANDARD ML. In combining the reference and exception extensions, we must ensure that they interact appropriately.

A naïve union of the two extensions does not quite work. The evaluation contexts R of the reference fragment do not contain contexts for the expressions of the exception fragment, and likewise the evaluation contexts X and U of the exception fragment do not contain contexts for the expressions of the reference fragment. The complete calculus and typing rules for Core ML may be found in Appendix B.

The type soundness of the resulting calculus is an immediate extension of the type soundness of the fragments because their typing rules do not interfere.

4 Continuations

Duba *et al.* [6] recently added a typed version of Scheme's *call-with-current-continuation* operator, abbreviated *callcc*, to a variant of Functional ML. To prove the type soundness of this extension, they designed a structural operational semantics for continuations and a new proof technique for the proof of type soundness. We can extend the calculus for Functional ML to include *callcc*, and prove type soundness for the resulting calculus along the same lines as for the above extensions. It is also easy to combine the above results for Core ML with this new extension. *See* Appendix C *for more details.*

5 Related Work

Abadi *et al.* [1] formulate a structural operational semantics for a simple extension of Functional ML. Their semantics uses substitution and is therefore closely related to our reduction system of Section 2. They prove a Type Soundness Theorem with a Type Preservation Theorem based on their semantic function. This is essentially the same theorem as our Theorem 2.2. However, beyond this specific use, they do not explore any further ramifications or applications of the technique.

Reddy and Swarup [17] recently proposed an ML-like functional language with a modicum of imperativeness. Their language contains reference cells and first-class functions but the type system restricts their use such that cells cannot escape from their lexical scope, function closures cannot contain cells, etc. Although Reddy and Swarup show that a reduc-

tion system for the language preserves types, they do not use this idea for a type soundness theorem nor do they offer it as a general proof technique.

6 Conclusions

We have presented calculi for ML-like languages that include imperative features. The calculi are adequate to evaluate programs, and admit simple and extensible proofs of type soundness. There are no published proofs of type soundness for references, exceptions, or control that are as simple or as extensible as our proofs.

Our work clarifies the connection among the operational semantics of typed languages, their λ-calculi, and their type soundness theorems. We believe that there are further applications of our technique to languages with different types of polymorphism.

References

[1] ABADI, M., CARDELLI, L., PIERCE, B., AND PLOTKIN, G. Dynamic typing in a statically-typed language. *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), to appear.

[4] 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.

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

[6] 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), to appear.

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

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

[9] 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.

[10] 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: *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

[11] LEROY, X., AND WEIS, P. Polymorphic type inference and assignment. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), to appear.

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

[13] MILNER, R., AND TOFTE, M. Co-induction in relational semantics. Tech. Rep. ECS-LFCS-88-65, Edinburgh University, October 1988.

[14] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, 1990. Preliminary versions in: Tech. Rep. ECS-LFCS-87-36, ECS-LFCS-88-62, ECS-LFCS-89-81, Edinburgh University.

[15] 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.

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

[17] REDDY, U. S., AND SWARUP, V. Assignments for applicative languages. Unpublished manuscript, University of Illinois, 1990.

[18] REES, J., AND CLINGER, W. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices 21*, 12 (December 1986), 37–79.

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

Appendices

The appendices are provided for referees with an interest in more details. They are not essential for an understanding of the results of the paper.

A Reducing Programs with Side-Effects

For an example reduction of a program with side-effects, consider the expression

```
((let x be ref (λx.+ (!x) 2) in := x (λx.+ (!x) 3))
 (let x be ref 2 in (λz.x)(:= x 4))).
```

In this example, 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. Here is one possible reduction sequence:

```
    ((let x be ref (λx.+ (!x) 2) in := x (λx.+ (!x) 3))
     (let x be ref 2 in (λz.x)(:= x 4)))
⟶  ((ρ{⟨x, λx.+ (!x) 2⟩}.:= x (λx.+ (!x) 3)) (ρ{⟨x,2⟩}.(λz.x)(:= x 4)))
⟶  ((ρ{⟨x, λx.+ (!x) 3⟩}.λx.+ (!x) 3) (ρ{⟨x,2⟩}.(λz.x)(:= x 4)))
⟶  ((ρ{⟨x, λx.+ (!x) 3⟩}.λx.+ (!x) 3) (ρ{⟨x,4⟩}.x))
⟶  ρ{⟨x, λx.+ (!x) 3⟩, ⟨y,4⟩}.(λx.+ (!x) 3) y
⟶  ρ{⟨x, λx.+ (!x) 3⟩, ⟨y,4⟩}.+ (!y) 3
⟶  ρ{⟨x, λx.+ (!x) 3⟩, ⟨y,4⟩}.+ 4 3
⟶  ρ{⟨x, λx.+ (!x) 3⟩, ⟨y,4⟩}.7
```

B The Core of STANDARD ML

Syntax

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

Typing

Functional ML

$$\Gamma \rhd x : \tau \ \text{ if } \Gamma x \succ \tau \quad (\tau\text{-}x) \qquad \Gamma \rhd c : \tau \ \text{ if } \mathit{TypeOf}\, c \succ \tau \quad (\tau\text{-}c)$$

$$\Gamma \rhd \mathrm{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \quad (\tau\text{-Y})$$

$$\frac{\Gamma[x \mapsto \tau_1] \rhd e : \tau_2}{\Gamma \rhd \lambda x.e : \tau_1 \to \tau_2} \ (\tau\text{-}\lambda) \qquad \frac{\Gamma \rhd e_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd e_1\ e_2 : \tau_2} \ (\tau\text{-app})$$

$$\frac{\Gamma \rhd v : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}_\Gamma \tau_1] \rhd e : \tau_2}{\Gamma \rhd \texttt{let}\ x\ \texttt{be}\ v\ \texttt{in}\ e : \tau_2} \ (\tau\text{-let-}v)$$

$$\frac{\Gamma \rhd e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{AppClose}_\Gamma \tau_1] \rhd e_2 : \tau_2 \quad e_1 \neq v}{\Gamma \rhd \texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 : \tau_2} \ (\tau\text{-let-}e)$$

References

$$\Gamma \triangleright \texttt{ref} : \tau \to \tau\ ref \quad \text{if } \tau \text{ is imperative} \quad (\tau\text{-ref})$$

$$\Gamma \triangleright\ ! : \tau\ ref \to \tau \quad (\tau\text{-}!) \qquad \Gamma \triangleright\ := : (\tau\ ref \to \tau \to \tau) \quad (\tau\text{-}:=)$$

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

Exceptions

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

$$\Gamma \triangleright \texttt{raise} : \tau_1\ exn \to \tau_1 \to \tau_2 \quad (\tau\text{-raise})$$

$$\frac{\Gamma[x \mapsto \tau_1\ exn] \triangleright e_1 : \tau_2 \quad \Gamma[x \mapsto \tau_1\ exn] \triangleright e_2 : \tau_1 \to \tau_2}{\Gamma[x \mapsto \tau_1\ exn] \triangleright e_1\ \texttt{handle}\ x\ e_2 : \tau_2} \quad (\tau\text{-handle})$$

Semantics

Functional ML

$$c_1\ c_2 \longrightarrow \delta(c_1, c_2) \quad \text{if } \delta(c_1, c_2) \text{ is defined} \qquad (\delta)$$

$$(\lambda x.e)\ v \longrightarrow e[x \mapsto v] \qquad (\beta_v)$$

$$\texttt{let}\ x\ \texttt{be}\ v\ \texttt{in}\ e \longrightarrow e[x \mapsto v] \qquad (let)$$

$$\mathrm{Y}\ v \longrightarrow v\ (\lambda x.(\mathrm{Y}\ v)\ x) \qquad (Y)$$

References

$$\texttt{ref}\ v \longrightarrow \rho\langle x, v\rangle.x \qquad (ref)$$

$$\rho\theta\langle x, v\rangle.R[!\ x] \longrightarrow \rho\theta\langle x, v\rangle.R[v] \qquad (deref)$$

$$\rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] \longrightarrow \rho\theta\langle x, v_2\rangle.R[v_2] \qquad (assign)$$

$$\rho\theta_1.\rho\theta_2.e \longrightarrow \rho\theta_1\theta_2.e \qquad (\rho_\cup)$$

$$R[\rho\theta.e] \longrightarrow \rho\theta.R[e] \qquad (\rho_{lift})$$

Exceptions

$$U[\texttt{raise}\ x\ v] \longrightarrow \texttt{raise}\ x\ v \qquad (raise)$$

$$\texttt{exception}\ \chi_1\ \texttt{in exception}\ \chi_2\ \texttt{in}\ e \longrightarrow \texttt{exception}\ \chi_1\chi_2\ \texttt{in}\ e \qquad (\texttt{exception}_\cup)$$

$$X[\texttt{exception}\ \chi\ \texttt{in}\ e] \longrightarrow \texttt{exception}\ \chi\ \texttt{in}\ X[e] \qquad (\texttt{exception}_{lift})$$

$$v\ \texttt{handle}\ x\ e \longrightarrow v \qquad (handle)$$

$$(\texttt{raise}\ x\ v)\ \texttt{handle}\ x\ e \longrightarrow e\ v \qquad (catch)$$

$$\texttt{exception}\ \chi x_1 x_2\ \texttt{in}\ X[(\texttt{raise}\ x_1\ v)\ \texttt{handle}\ x_2\ e] \longrightarrow \texttt{exception}\ \chi x_1 x_2\ \texttt{in}\ X[\texttt{raise}\ x_1\ v] \qquad (reraise)$$

Evaluation Contexts

$$R ::= [] \mid R\ e \mid v\ R \mid \texttt{let}\ x\ \texttt{be}\ R\ \texttt{in}\ e \mid R\ \texttt{handle}\ x\ e$$
$$X ::= [] \mid X\ e \mid v\ X \mid \texttt{let}\ x\ \texttt{be}\ X\ \texttt{in}\ e \mid X\ \texttt{handle}\ x\ e \mid \rho\theta.X$$
$$U ::= [] \mid U\ e \mid v\ U \mid \texttt{let}\ x\ \texttt{be}\ U\ \texttt{in}\ e$$

C Adding Continuations to STANDARD ML

We extend the syntax by adding the primitive $\mathcal{K}$ (for creating continuations), a throw operator $\mathcal{T}$ (for using continuations), and the expression $\kappa x.e$ to represent continuations. We also introduce a type constructor for continuations:

$$\begin{aligned} v &::= \ldots \mid \mathcal{K} \mid \mathcal{T} \mid \kappa x.e \\ \tau &::= \ldots \mid \tau \ \mathit{cont} \end{aligned}$$

The reductions of the control fragment are:

$$\begin{aligned} G[\mathcal{K}\ e] &\longrightarrow \mathcal{K}\ (\lambda x_1.G[e\ (\kappa x_2.\mathcal{T}\ x_1\ G[x_2])]) && (\mathcal{K}) \\ \mathcal{T}\ (\kappa x.e) &\longrightarrow \lambda x.e && (\mathcal{T}) \\ \mathcal{K}\ (\lambda x.G[\mathcal{T}\ x\ e]) &\longrightarrow \mathcal{K}\ (\lambda x.e) && (\mathit{throw}) \\ \mathcal{K}\ (\lambda x.e) &\longrightarrow e \ \text{ if } x \notin FV(e) && (\mathcal{K}_{\mathit{elim}}) \\ \mathcal{K}\ (\lambda x.\mathcal{K}\ e) &\longrightarrow \mathcal{K}\ (\lambda x.e\ x) && (\mathcal{K}_{\mathit{idem}}) \end{aligned}$$

where the evaluation contexts G are:

$$G ::= [] \mid G\ e \mid v\ G \mid \texttt{let}\ x\ \texttt{be}\ G\ \texttt{in}\ e \mid G\ \texttt{handle}\ x\ e$$

The typing rules are as follows [6]:

$$\Gamma \triangleright \mathcal{K} : (\tau\ \mathit{cont} \rightarrow \tau) \rightarrow \tau \quad (\tau\text{-}\mathcal{K}) \qquad \Gamma \triangleright \mathcal{T} : \tau_1\ \mathit{cont} \rightarrow \tau_1 \rightarrow \tau_2 \quad (\tau\text{-}\mathcal{T})$$

$$\frac{\Gamma[x \mapsto \tau_1] \triangleright e : \tau_2 \ \text{ for all } \tau_2}{\Gamma \triangleright \kappa x.e : \tau_1\ \mathit{cont}} \quad (\tau\text{-}\kappa)$$

The calculus is adequate to evaluate programs. The proof is similar to Felleisen's proof for the λ-v-C calculus [9]. The proof of type soundness for the *callcc* extension follows the usual strategy.