Standard ML-NJ weak polymorphism and imperative constructs

*John Mitchell** *Ramesh Viswanathan**

Department of Computer Science
Stanford University, Stanford, CA 94305
{mitchell, vramesh}@cs.stanford.edu

July 15, 1994

Abstract

Standard ML of New Jersey (SML-NJ) uses "weak type variables" to restrict the polymorphic use of functions that may allocate reference cells, manipulate continuations, or use exceptions. However, the type system used in the SML-NJ compiler has not previously been presented in a form other than source code nor proved correct. We present a set of typing rules, based on a careful analysis of the concepts underlying "weak polymorphism", that appears to subsume the implemented algorithm and uses type variables of only a slightly more general nature than the compiler. One insight in the analysis is that allowing a variable to occur both "ordinarily" and "weakly" in a type permits a simpler and more flexible formulation of the typing rules. In particular, we are able to treat applications of polymorphic functions to imperative arguments with greater flexibility than SML-NJ. The soundness of the type system is proved for imperative code using operational semantics, by showing that evaluation preserves typability. By incorporating assumptions about memory addresses in the type system, we avoid proofs by co-induction.

1 Introduction

In the absence of side effects and so-called control effects (continuations and exceptions), the type inference methods developed by Curry, Hindley and Milner [CF58, Hin69, Mil78] provide an effective and useful method for detecting type errors at compile time. For any typable untyped expression, the type inference algorithm computes a single *most general type*, from which all other types of the expression may be derived easily. Although the worst-case complexity is high [KMM91, KTU90], the algorithm has been found efficient in practice. More importantly, experience suggests that it is possible for a programmer to predict, with relative ease, the type that the algorithm will produce. In addition, type inference allows a form of error detection that is unfamiliar to explicitly-typed programming. Even when a declaration is typable, the programmer may detect an error by noticing that the type inferred by the compiler differs from the type expected by the programmer (*c.f.* [Koe92]).

Unfortunately, the straightforward, predictable polymorphic typing algorithm fails for programs with side-effects or control effects. To illustrate the problems involved, we use a lambda calculus extended with let-expressions (for polymorphism) and imperative operators `ref`, `!` and `:=`. Briefly, `ref` x allocates a new cell, initialized to x, $r := x$ updates the contents of reference r to the

*Supported in part by an NSF PYI Award, with matching funds from AT&T, Digital Equipment Corporation, the Powell Foundation and Xerox Corporation; and by the Wallace F. and Lucille M. Davis Faculty Scholarship.

value of x, and $!r$ returns the current contents of reference r. The incorrectness of the naive polymorphic typing of side-effects is illustrated by the program

$$\texttt{let } r = \texttt{ref } (\lambda x.\, x) \texttt{ in } r := \lambda x.\, x + 1\, ;\; !r\, \texttt{true}$$

where ; is the usual sequencing operator (definable by lambda abstraction under call-by-value). If we apply the Milner typing algorithm, we give r type $\forall\alpha.[(\alpha \rightarrow \alpha)\ \mathit{ref}]$, where $\tau\ \mathit{ref}$ is the type of references to values of type τ. Intuitively, this means that for any type σ, α can be instantiated to σ to give r the type $(\sigma \rightarrow \sigma)\ \mathit{ref}$. Both expressions in the body of the declaration are well-typed, under this assumption, since we can take the instances $(\mathit{int} \rightarrow \mathit{int})\ \mathit{ref}$ and $(\mathit{bool} \rightarrow \mathit{bool})\ \mathit{ref}$ for the two occurrences of r, respectively. However, since the assignment changes the value of r from a polymorphic function to an integer function, the application of $!r$ to `true` results in a run-time type error. Thus, a naive polymorphic treatment of side-effects does not prevent run-time type errors in the presence of assignment.

We can view solutions to this problematic interaction of polymorphism and side-effects from a general perspective, by understanding the way polymorphic type inference works. In typing an expression of the form `let` $x = e_1$ `in` e_2, the most general type τ of e_1 is calculated. Certain type variables in τ are determined to be "generic" or "polymorphic". If $\alpha_1, \ldots, \alpha_n$ is a list of all the generic variables in τ, we write the type of e_1 as $\forall\alpha_1 \ldots \forall\alpha_n.\tau$. In typing e_2, all the generic/polymorphic type variables may then be replaced by types, independently chosen for each occurrence of x. For example, if e_1 has type $\forall\alpha.\alpha \rightarrow \mathit{int}$, and e_2 is the expression $x\ x$, then e_2 may be typed by replacing α by $\mathit{int} \rightarrow \mathit{int}$ in the first occurrence of x and int in the second occurrence. If a type variable is determined to be non-polymorphic, it can only be replaced by the same type in all occurrences of the let-bound variable x. Thus a critical aspect of polymorphic type inference is the method for identifying the type variables that may be safely be treated as polymorphic. In the absence of side-effects or control effects, there is a simple condition depending on the types of free variables of e_1. The main topic of this paper, and previous studies of polymorphism and imperative constructs, is the safe and flexible extension of the condition for pure terms to terms with side-effects. Over the past decade, a number of restrictions on the polymorphic uses of references have been proposed. Some of these have been implemented and others have been analyzed and proved correct, as described below. Our main contribution is a precise definition of a type system that appears to subsume the algorithm used in SML-NJ and a correctness proof for it. Although similar problems occur with continuations and exceptions, as illustrated in [HL91, WF92], we restrict our attention to side effects, in this paper.

In Standard ML of New Jersey [AM87, AM91], so-called "weak type variables" are used to characterize functions that may create reference cells after some number of applications. Since the type of a polymorphic function may become more specialized with each application (as, for example, when a polymorphic function is applied to a monomorphic argument), many functions that create references could create polymorphic references for some actual parameters, and monomorphic references for others. For example, the function

$$\lambda f.\, \lambda x.\, \texttt{ref}(f x)$$

creates a reference to the result of applying the first actual parameter to the second. This function is unproblematic for creating references to integers or booleans, but we cannot allow the result of application to two arguments to be used polymorphically because of the phenomenon described above. The most restrictive solution to the problem of typing references would be to require some fixed, non-polymorphic type for this function. A more flexible approach might be to force every argument to this function to be non-polymorphic. However, this too is overly restrictive since the reference cell is not created until the second actual parameter is supplied. For example, we may safely obtain a polymorphic function by applying to the polymorphic identity, provided any eventual second argument is non-polymorphic. The main innovation of SML-NJ weak typing, over previous approaches, is to use numeric indices on each type variable to count the number of function applications required to produce a reference cell of that type. Although this system has seen widespread and largely successful use in recent years, the SML-NJ algorithm has never been presented in a form other than relatively complicated source code, and consequently has never been analyzed or proved sound.

We present our type system as a set of typing rules, using a mild extension of the weak type variables of SML-NJ. These rules were derived from experiments with the algorithm implemented in Standard ML of New Jersey, Version 75 (November 11, 1991) and perusal of the source code. Although our original intention was to formalize the implemented algorithm exactly, we found it easier and more informative to define a more general type system with simpler typing rules and fewer apparent special cases. One insight came as a result of examining the way that the type of an expression depends on the types of its non-polymorphic free variables. In SML-NJ, every weak type variable appears throughout the type of an expression with the same strength. For example, the expression `ref` has the type $\alpha^1 \rightarrow \alpha^1 \ \mathit{ref}$. Such a restriction forces the numeric superscripts in the type of a free variable to characterize two sources of storage allocation. They must simultaneously give a lower bound on the number of applications of the variable itself that may produce a reference, and a lower bound on the number of applications of the expression containing the variable. Thus, in typing an expression, the type assumption of a free variable needs to be modified according to the sub-expression that one is typing. We avoide this is by allowing both "ordinary" and "weak" occurrences of a type variable in a type. This leads to the formulation of a stronger typing rule for application than the compiler output indicates. Consequently, our typing rules appear to subsume the implemented algorithm, particularly for applications of purely functional expressions to potentially imperative arguments. We show that our typing rules are sound, using an operational semantics of ML with references, implying the semantic soundness of any algorithm that decides a conservative approximation to these rules.

More expressive type systems have been developed by Leroy and Weis [LW91], and, independently, by Talpin, Jouvelot and others [TJ92, Wri92], partly based on the framework of Gifford's FX system [JG91]. In contrast to the SML-NJ algorithm and preceding ML compilers, which keep track of the types of all values in the store, these approaches approximately identify the values in the store that are accessible in a particular context. This additional accesibility information leads to type systems that are capable of giving functions which use references only locally the same types as their corresponding applicative versions. However, to be able to maintain this information, more complicated type expressions are required. While the more complex systems are promising, it is more difficult to predict the types of certain functions. Since we believe that the success of ML typing lies not only in its correctness, but the ease with which it may be understood in programming, any added complication to the type system deserves careful scrutiny. Consequently, it seems worthwhile to explore systems using only weak type variables and determine the limits of this approach.

2 Weak Polymorphism

In the approach we refer to as *weak polymorphism,* each type variable has an associated *degree of strength,* or *degree* for short. In the SML-NJ implementation, a degree may be a nonnegative integer, or ∞. Variables of infinite degree are printed without any degree designation. Intuitively, the degree of a type variable α, occurring in the typing of some expression e, indicates how many applications of e are required to produce a reference to a value whose type contains α. Functions that do not produce references at all are typed using variables of infinite degree. For example, the type of `ref` itself is

$$\texttt{ref} : \alpha^1 \to \alpha^1\, ref$$

where the superscript indicates that α^1 is a weak type variable of degree 1. This type implies that when the function `ref` is applied, α may appear in the type of a stored value, and should not be treated polymorphically. However, the α in the type of `ref` is polymorphic, as is any type variable of non-zero degree. This means that `ref` may be applied to an argument of any type, but the result cannot be used polymorphically. Another example is the list-producing function $\lambda x.\, \texttt{ref}[x]$, which has type $\alpha^1 \to \alpha^1$ *list ref* since it takes one application of this function to produce a reference to a value whose type contains α, or some type substituted for α. Since each application weakens the type of a function, each surrounding lambda abstraction strengthens every type variable in the type. The reader may enjoy giving the example used in the introduction, $\lambda f.\, \lambda x.\, \texttt{ref}(f x)$, to the SML-NJ compiler and seeing where the strength 2 appears.

We may put the SML-NJ type system in perspective by reviewing the history of polymorphic type systems for imperative constructs. Early implementations of ML only allowed references to values of monomorphic, statically-known types. This approach guarantees soundness, but makes it impossible to write polymorphic functions that create or use references. In his thesis, Damas introduced the first type system to allow polymorphic use of functions that create references [Dam85]. Unfortunately, although his system has not been shown unsound, his claimed soundness proof is incorrect. Tofte was the first to give a type discipline for polymorphic references together with a correct soundness proof [Tof88, Tof90]. The central idea of Tofte's solution is to partition the types into "imperative" and "applicative" types, requiring all type variables that occur in an imperative type to be imperative. Values of imperative type can be stored, but only applicative type variables are treated polymorphically, successfully avoiding the creation of references to polymorphic values. MacQueen's system, implemented in the Standard ML of New Jersey ML compiler, is largely derived from Tofte's, using the numeric "degree" of a weak type variable to count how many function applications it takes for a type variable to become "imperative."

The power of weak polymorphism may be understood by comparison with Tofte's system. In Tofte's system, any type variable that appears in the type of a value that could potentially be stored has to be imperative and hence cannot be used polymorphically, even if the value does not actually appear in the store yet. Technically, imperative type variables are treated non-polymorphically only in the types of so-called "expansive" expressions, which include `let` expressions. Therefore, we illustrate the limitations of Tofte's system using the following expression, *rid*, which is a version of the identity function using side-effects.

$$rid \quad \equiv \quad \texttt{let}\ z = dummy\ \texttt{in}\ \lambda x.\, !(\texttt{ref}\, x)$$

Since x is stored, its type must be imperative. Consequently, in Tofte's system, the expression e is given a type $u \to u$, where u is an imperative type variable that cannot be generalized. With weak polymorphism, however, we can take into account the fact that *rid* must be applied once before the reference to x is created. This allows us to give *rid* the type $\alpha^1 \to \alpha^1$, where the type variable α^1 can now be generalized. Since *rid* may be used polymorphically using weak polymorphism, but not in Tofte's system, the expression

$$\texttt{let}\ f = rid\ \texttt{in}\ f\, 1\ ;\ f\, \texttt{true}$$

is typable in SML-NJ but not in Tofte's system.

In our typing rules, we generalize the SML-NJ type expressions slightly to associate numeric indices with ordinary type variables. In addition, a type variable can have both "ordinary" and "weak" occurrences, as well as occur with different strengths in a type. Aside from these relatively minor changes, which allow us to express the relationship between degrees of various type variables more clearly, our type expressions are identical to the those used in Standard ML of New Jersey. It seems that significantly more complicated typing rules are needed if only the SML-NJ type expressions are used. In fact, we were not able to devise a sound system with SML-NJ type expressions using ordinary proof rules without excessively complicated side conditions.

3 Language

3.1 Syntax

In describing our typing rules and proving them correct, we use an untyped call-by-value language with ML-style imperative features. In writing the expressions of our language we allow arbitrary term constructors of specified arity. More formally, the language is parameterized by a set C of pairs, $\langle f, n\rangle$, where $n \geq 0$ gives the arity of the term constructor f. Having term constructors, as opposed to term constants, allows one to represent non-strict operators in the call-by-value language. The expressions are given by the grammar

$$\begin{array}{rcl} e & ::= & x \mid f(e_1, \ldots, e_n) \mid e_1\, e_2 \mid \lambda x.\, e \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\ & & l \mid \texttt{ref} \mid !e \mid e_1 := e_2 \end{array}$$

where $\langle f, n\rangle \in C$, $x \in Var$, $l \in Loc$, *Var* is a denumerable set of variables, and *Loc* is a denumerable set of location constants. The location constants do not appear in user programs (this would be tantamount to allowing the user to pick addresses), but help in defining the operational semantics. We thus define programs to be closed expressions not containing any location constants.

In defining the operational semantics, it is useful to distinguish a subset of closed expressions that cannot be further evaluated. These are called *values*. Besides some expressions that depend on the particular choice of term constructors, the set of values always includes λ-abstractions, location constants, and the constant `ref`.

$$v ::= \cdots \mid \lambda x.\, e \mid l \mid \texttt{ref}$$

The additional values that depend on the term constructors are specified in a set V. Each element of V is of the form $f(d_1, \ldots, d_n)$ where $\langle f, n\rangle \in C$ and each d_i is an expression or a value.

3.2 Operational Semantics

We use a rewriting semantics to specify operational behaviour. In defining the operational semantics, we use a syntactic form `wrong` to represent run-time errors.

We begin by defining a relation, $\leadsto$, with $e \leadsto r$ indicating that e is a purely functional redex reducing directly to r, *i.e.*, without any side-effects. The result, r, of the reduction is either another expression or `wrong`. The behaviour of the functional term constructors is given by a set F of pairs of the form $\langle e, r\rangle$, indicating that e reduces to r, where e is an expression of the form $f(e_1, \ldots, e_n)$ and r is another expression or `wrong`. The purely *functional* reduction relation, $\leadsto$,

$$
\begin{array}{rcll}
(E[e], s) & \longmapsto & (E[e'], s) & \text{if } e \leadsto e', e' \neq \texttt{wrong} \\
(E[\texttt{ref}\ v], s) & \longmapsto & (E[l], s[l \mapsto v]) & \text{where } l \notin dom(s) \\
(E[!l], s) & \longmapsto & (E[s(l)], s) & \\
(E[l := v], s) & \longmapsto & (E[v], s[l \mapsto v]) & \\
(E[e], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } e \leadsto \texttt{wrong} \\
(E[!v], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } v \notin \mathit{Loc} \\
(E[v_1 := v_2], s) & \longmapsto & (\texttt{wrong}, s) & \text{if } v_1 \notin \mathit{Loc}
\end{array}
$$

Table 1: One-step evaluation relation.

is defined by

$$
\begin{array}{rcll}
(\lambda x.\, e)\, v & \leadsto & [v/x]e & \\
\texttt{let}\ x = v\ \texttt{in}\ e & \leadsto & [v/x]e & \\
v_1\, v_2 & \leadsto & \texttt{wrong} & \text{if } v_1 \text{ not a } \lambda\text{-abstraction, } \texttt{ref} \\
e & \leadsto & r & \text{if } \langle e, r\rangle \in F
\end{array}
$$

To impose an order on the evaluation of subexpressions, we use the notion of an *evaluation context* [WF92]. The evaluation contexts $E[\,]$ are defined by the following grammar

$$
E[\,] \ ::= \ [\,] \mid E\ e \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \cdots \mid \texttt{ref}\ E \mid !E \mid E := e \mid v := E
$$

where the unspecified productions are used to reflect the order of evaluation of the arguments of term constructors and are of the form $f(d_1, \ldots, d_n)$, where exactly one of the d_i's is an evaluation context E and the others are values v or expressions e. Intuitively, the hole in an evaluation context specifies the next subexpression to be evaluated.

To handle imperative features, our operational rules will manipulate "stores" in addition to terms, where a store is a finite partial map from location constants to values. The basic one-step evaluation relation $\longmapsto$ is defined in Table 1, with $(e, s) \longmapsto (r, s')$ indicating that the expression e in store s evaluates in one step to the result r resulting in new store s'. The result r is either another expression or the run-time error `wrong`.

The result of execution of any program, obtained by evaluating it in the initial empty store, is captured by a partial function *Eval*. Writing $\longmapsto\!\!\!\!\rightarrow$ for the reflexive and transitive closure of $\longmapsto$, $\mathit{Eval}(e) = a$ iff $(e, \mathbf{init}) \longmapsto\!\!\!\!\rightarrow (a, s')$ for some store s', where the answer a is either a value or `wrong` and **init** is the empty store that is undefined on all locations. A program e may diverge, in which case $\mathit{Eval}(e)$ is undefined. We say that a program e *does not lead to a run-time error* if $\mathit{Eval}(e) \neq \texttt{wrong}$.

4 Type System

4.1 Type Expressions and Instances

Our type expressions are built from type variables using the constructor $\rightarrow$ for function types, and *ref* for reference cells, in addition to type constructors for other datatypes. To incorporate weak polymorphism, every "weak" type variable will have an index which is its degree of strength. However, unlike ML, an ordinary type variable will also have an associated index in our type system.

We start with a set B of type constructors for datatypes such as integers, lists etc. Each element of B is a pair of the form $\langle b, n\rangle$, where $n \geq 0$ gives the arity of the type constructor b.

Let TVars be an infinite set of type variables whose elements will be indicated by metavariables $\alpha, \alpha_1, \ldots, \beta, \beta_1, \ldots$. A "weak" occurrence of a type variable will be signified by underlining it. As mentioned before, every "weak" and "ordinary" occurrence of a type variable has an associated index, which will be a natural number. We thus arrive at the following grammar for generating pre-types, some of which will be deemed to be well-formed types:

$$\rho \ ::= \ \alpha^j \mid \underline{\alpha}^j \mid \rho_1 \rightarrow \rho_2 \mid \rho_1 \ \mathit{ref} \mid b(\rho_1, \ldots, \rho_n)$$

In this grammar we assume that $\langle b, n\rangle \in B$ and $j \in \mathcal{N}$. Intuitively, a pre-type of the form $\rho_1 \rightarrow \rho_2$ can be used as the type of some expression only if any type variable occurring "imperatively" in ρ_1 does not occur "applicatively" in ρ_2 and vice-versa. We now elaborate the precise condition distinguishing types from pre-types.

For any pre-type ρ, we define its free type variables, $\mathrm{FTV}(\rho)$, to be the set of variable names appearing in it, without regard to their index or whether they are underlined. More precisely,

$$\begin{aligned} \mathrm{FTV}(\alpha^j) &= \{\alpha\} \\ \mathrm{FTV}(\underline{\alpha}^j) &= \{\alpha\} \\ \mathrm{FTV}(\rho_1 \rightarrow \rho_2) &= \mathrm{FTV}(\rho_1) \cup \mathrm{FTV}(\rho_2) \\ \mathrm{FTV}(\rho_1 \ \mathit{ref}) &= \mathrm{FTV}(\rho_1) \\ \mathrm{FTV}(b(\rho_1, \ldots, \rho_n)) &= \mathrm{FTV}(\rho_1) \cup \cdots \cup \mathrm{FTV}(\rho_n) \end{aligned}$$

Intuitively, a weak occurrence of a type variable α indicates that α appears in the type of a value to which a reference may potentially be created. The index j in a weak occurrence $\underline{\alpha}^j$ indicates that at least j applications of the relevant expression are needed before the reference to the type containing α is *actually* created. Thus, only $\underline{\alpha}^0$ is deemed to be an *imperative occurrence* and indicates that α is imperative, *i.e.*, appears in the type of a stored value. On the other hand ordinary type variables never appear in the type of a stored value. Thus the *applicative* variables, $\mathrm{ATV}(\rho)$, *i.e.*, those variables that do not appear in the store typing can be defined by

$$\begin{aligned} \mathrm{ATV}(\alpha^j) &= \{\alpha\} \\ \mathrm{ATV}(\underline{\alpha}^j) &= \{\alpha\} \text{ if } j > 0 \\ \mathrm{ATV}(\underline{\alpha}^0) &= \phi \\ \mathrm{ATV}(\rho_1 \rightarrow \rho_2) &= \mathrm{ATV}(\rho_1) \cup \mathrm{ATV}(\rho_2) \\ \mathrm{ATV}(\rho_1 \ \mathit{ref}) &= \mathrm{ATV}(\rho_1) \\ \mathrm{ATV}(b(\rho_1, \ldots, \rho_n)) &= \mathrm{ATV}(\rho_1) \cup \cdots \cup \mathrm{ATV}(\rho_n) \end{aligned}$$

The imperative type variables are the free type variables that do not occur applicatively:

$$\mathrm{ITV}(\rho) = \mathrm{FTV}(\rho) \setminus \mathrm{ATV}(\rho)$$

We say that a pre-type ρ is imperative if all its free variables occur imperatively, *i.e.*, $\mathrm{ITV}(\rho) = \mathrm{FTV}(\rho)$.

Since the index in a weak occurrence indicates the number of applications needed before it becomes imperative, we would intuitively expect every application to decrement the strength in every occurrence by 1. We will also decrement the index in ordinary occurrences after an application. All this is formally captured by the *weakening map* W defined on indexed variables as

$$\begin{aligned} W(\alpha^j) &= \begin{cases} \alpha^{j-1} & \text{if } j > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\ W(\underline{\alpha}^j) &= \begin{cases} \underline{\alpha}^{j-1} & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{cases} \end{aligned}$$

and extended to all pre-types in the obvious way using the type constructors, $\rightarrow, ref, b$. If ordinary occurrences of type variables were allowed to have negative indices then we can avoid having $W(\alpha^0)$ undefined. However, we prefer the present framework as it allows us to define substitutions more elegantly.

We can now give some intuition for the indices in ordinary occurrences of a type variable. These are used to keep track of any applications and abstractions occurring in the body of an expression. The weakening map W is used in typing an application to decrease the index in every ordinary occurrence by 1. Analogously, in typing an abstraction, we will increase the indices in ordinary occurrences by 1. Thus if an there is an ordinary occurrence of a type variable α in the type ρ of an expression e, its index relative to the indices of other occurrences of α in ρ gives us information about any applications and abstractions within the body of e. This allows a more uniform treatment when a type containing weak occurrences of type variables is substituted for an ordinary occurrence of a type variable. This arises, for example, when a purely functional expression is applied to an expression using imperative features, as is further discussed in Section 5.

The main property guaranteed by the type system is that when a type variable α appears in the type of a stored value after evaluation of an expression e, all occurrences of α in the type of e must be imperative. This requires some restrictions on the pre-types that can be used as types of expressions. The well-formed monotypes, indicated by meta variables $\tau, \tau', \tau'', \ldots, \tau_1, \tau_2, \ldots$ are defined as follows:

- For any $\alpha \in \mathrm{TVars}, j \in \mathcal{N}$, $\alpha^j, \underline{\alpha}^j$ are monotypes.
- If τ_1, τ_2 are well-formed monotypes then $\tau_1 \rightarrow \tau_2$ is a well-formed monotype, provided
  - (i) For all $\alpha \in \mathrm{FTV}(\tau_1) \cap \mathrm{FTV}(\tau_2)$, $\alpha \in \mathrm{ITV}(\tau_1)$ iff $\alpha \in \mathrm{ITV}(\tau_2)$.
  - (ii) $W(\tau_2)$ is defined and is a well-formed monotype.
- $\tau_1\ ref$ is a well-formed monotype, if τ_1 is a well-formed monotype.
- $b(\tau_1, \ldots, \tau_n)$ is a well-formed monotype, if $\tau_1, \ldots, \tau_n$ are well-formed monotypes.

Condition (i) on a type $\tau_1 \rightarrow \tau_2$ forces any imperative type variable of τ_1 to also occur only imperatively in τ_2 and vice-versa. When an expression of type $\tau_1 \rightarrow \tau_2$ is applied to an expression of type τ_1, the type of the resulting expression is $W(\tau_2)$, which is why we require Condition (ii).

The polymorphic types, indicated by $\sigma, \sigma_1, \ldots$, are given by the grammar

$$\sigma \ ::= \ \tau \mid \forall \alpha . \sigma$$

where τ is a monotype and $\alpha \in \mathrm{TVars}$. The variables $\alpha_1, \ldots, \alpha_n$ are bound in the polymorphic type $\forall \alpha_1 \ldots \alpha_n .\ \tau$. We consider polymorphic types which differ only by a consistent renaming of bound variables equivalent. We define the set of free type variables, $\mathrm{FTV}(\sigma)$, of a polytype as $\mathrm{FTV}(\forall \alpha_1 \ldots \alpha_n .\ \tau) = \mathrm{FTV}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\}$ and similarly $\mathrm{ITV}(\sigma)$ as the set of free variables in σ that occur imperatively.

The instances of a polytype are defined using substitutions. A substitution S with domain $T \subseteq \mathrm{TVars}$ is a map from T to the well-formed monotypes. Intuitively, the substitution S specifies the type to replace ordinary occurrences with index 0, with the types for other occurrences determined systematically from this. To state the types to use for other occurrences of a type variable, we will define maps $Set_i, i \geq 0$ and Inc on monotypes. We will specify the behaviour of these maps on indexed variables, with the obvious extension to monotypes. The map Set_i sets all occurrences of a type variable, except those that are imperative, to a weak occurrence with strength i.

$$\begin{array}{rcl} Set_i(\alpha^j) & = & \underline{\alpha}^i \\ Set_i(\underline{\alpha}^j) & = & \left\{ \begin{array}{ll} \underline{\alpha}^i & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{array} \right. \end{array}$$

The following map, Inc, increments the indices in all occurrences of type variables, except the imperative ones.

$$\begin{array}{rcl} Inc(\alpha^j) & = & \alpha^{j+1} \\ Inc(\underline{\alpha}^j) & = & \left\{ \begin{array}{ll} \underline{\alpha}^{j+1} & \text{if } j > 0 \\ \underline{\alpha}^0 & \text{otherwise} \end{array} \right. \end{array}$$

The effect of a substitution S with domain T on an indexed type variable is given by $S(\alpha^j) = Inc^j(S(\alpha))$ and $S(\underline{\alpha}^j) = Set_i(S(\alpha))$, for $\alpha \in T$, where Inc^j denotes the n-fold composition of Inc. If S has domain T, then S leaves any variable not in T unchanged. The type $S(\tau)$ and $S(\sigma)$ for monotype τ and polytype σ can then be defined in the usual way, with possible renaming of bound variables. We often use $[S(\alpha_1)/\alpha_1, \ldots, S(\alpha_n)/\alpha_n]$ to denote the substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$. Note that for a monotype τ, $S(\tau)$ may not necessarily be a well-formed monotype.

We say that a well-formed monotype τ' is an instance of a polymorphic type $\sigma = \forall\alpha_1 \ldots \alpha_n . \tau$, written $\forall\alpha_1 \ldots \alpha_n . \tau \succ \tau'$, if there exists a finite substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $S\,\tau = \tau'$.

4.2 Typing Rules

The type system is given by a set of axioms and inference rules. The formulas that are derivable in this proof system are judgements of the form $\Gamma, \mathcal{R} \triangleright e\colon \tau$, where Γ is a variable context, $\mathcal{R}$ is a location context, e and expression, and τ a monotype. The variable context Γ states assumptions about the free variables of e and is a set of pairs $x\colon \sigma$, where σ is a polytype, and no variable x appears twice in Γ. The location context $\mathcal{R}$ gives the types of values stored in the location constants that may appear in e. It is a set of pairs $l\colon \tau$, where τ is an imperative monotype, with no location constant l appearing twice. A provable judgement $\Gamma, \mathcal{R} \triangleright e\colon \tau$ can thus be read as "the expression e has type τ in context $\Gamma, \mathcal{R}$". Since expressions that appear in ML prorams never contain location constants, we never need assumptions about location constants in typing them. However, since expressions containing location constants could be the result of an evaluation step, and our proof of soundness proceeds by showing that evaluation preserves typability, we include location contexts, $\mathcal{R}$, in our judgements. A type system for ML program expressions can be obtained by letting $\mathcal{R}$ be empty in all the axioms and inference rules.

Just as for pre-types, we will place some restrictions on valid typing judgements for expressions. Define the imperative type variables of a context to be the set of type variables that are imperative in any type appearing in the context, *i.e.*,

$$\mathrm{ITV}(\Gamma, \mathcal{R}) = \bigcup_{x:\sigma \in \Gamma} \mathrm{ITV}(\sigma) \cup \bigcup_{l:\tau \in \mathcal{R}} \mathrm{ITV}(\tau)$$

To motivate the restrictions that we will place on typing judgements, consider a store s with domain $\{l_1, \ldots, l_m\}$ such that the value, $s(l_i)$, stored in location l_i has type τ_i. Let $e_1, \ldots, e_n$ be closed expressions with e_i having type σ_i. Informally, a judgement, $\{x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\}, \{l_1\colon \tau_1, \ldots, l_m\colon \tau_m\} \triangleright$

$e{:}\,\tau$, asserts that if, starting in store s, we evaluate expressions $e_1, \ldots, e_n$ and substitute the resulting values for $x_1, \ldots, x_n$ in e, the resulting expression evaluates to a value of type τ. In this case, we want any type variable free in the type of a value in the final store to be imperative in τ. Any variable that is imperative in one of the σ_i's or τ_i's may be free in the type of a stored value after evaluation of the e_i's or in the initial store, and will therefore be free in the type of a stored value in the final store obtained after evaluation of e. We therefore say that a judgement $\Gamma, \mathcal{R} \triangleright e{:}\,\tau$ is a *well-formed typing judgment* iff

- τ is well-formed monotype, and
- For all $\alpha \in \mathrm{ITV}(\Gamma, \mathcal{R}) \cap \mathrm{FTV}(\tau)$, $\alpha \in \mathrm{ITV}(\tau)$.

The type system depends on a set of typings for term constructors. These are given by a function *TypeOf* defined on all term constructors f, where if $\langle f, n\rangle \in C$, then $\mathit{TypeOf}(f)$ is an expression of the form $(\tau_1, \ldots, \tau_n \Rightarrow \tau)$. We use the following notational conventions in the presentation of the type system. We write $\Gamma(x)$ (and similarly $\mathcal{R}(l)$) for the unique σ with $x{:}\,\sigma$ in Γ, if it exists, and $\Gamma, x{:}\,\sigma$ for $\Gamma \cup \{x{:}\,\sigma\}$, assuming $x \notin \Gamma$. For any set of type variables, T, we define the substitution Imp_T with domain T by $\mathit{Imp}_T(\alpha) = \underline{\alpha}^0$. The *closure of* τ *with respect to* Γ, written $\mathit{Clos}_\Gamma \tau$ is the polytype $\forall \alpha_1, \ldots, \alpha_n . \tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau) \setminus \mathrm{FTV}(\Gamma)$ and $\mathrm{FTV}(\Gamma)$ is the set of all free type variables in the types appearing in Γ. The restriction on only binding applicative type variables keeps variables that might appear in the types of stored values from being used polymorphically.

(*var*) $$\Gamma, \mathcal{R} \triangleright x{:}\, \mathit{Imp}_T(\tau) \qquad \text{if } \Gamma(x) \succ \tau$$

(*loc*) $$\Gamma, \mathcal{R} \triangleright l{:}\, \mathcal{R}(l)\ \mathit{ref}$$

(*const*) $$\frac{\Gamma, \mathcal{R} \triangleright e_1{:}\, S\tau_1 \quad \cdots \quad \Gamma, \mathcal{R} \triangleright e_n{:}\, S\tau_n}{\Gamma, \mathcal{R} \triangleright f(e_1, \ldots, e_n){:}\, S\tau} \quad \text{if} \quad \begin{array}{l} \mathit{TypeOf}(f) = (\tau_1, \ldots, \tau_n \Rightarrow \tau), \\ \mathrm{ITV}(S\tau_i) \cap \mathrm{ATV}(S\tau) = \phi \end{array}$$

(*abs*) $$\frac{\Gamma, x{:}\,\tau_1, \mathcal{R} \triangleright e{:}\, W(\tau_2)}{\Gamma, \mathcal{R} \triangleright \lambda x.\, e{:}\, \tau_1 \rightarrow \tau_2}$$

(*app*) $$\frac{\Gamma, \mathcal{R} \triangleright e_1{:}\, \tau_1 \rightarrow \tau_2 \qquad \Gamma, \mathcal{R} \triangleright e_2{:}\, \tau_1}{\Gamma, \mathcal{R} \triangleright e_1\ e_2{:}\, W(\tau_2)}$$

(*let*) $$\frac{\Gamma, \mathcal{R} \triangleright e_1{:}\, \tau_1 \qquad \Gamma, x{:}\, \mathit{Clos}_\Gamma \tau_1, \mathcal{R} \triangleright e_2{:}\, \tau}{\Gamma, \mathcal{R} \triangleright \texttt{let}\ x = e_1\ \texttt{in}\ e_2{:}\, \tau}$$

(*ref*) $$\Gamma, \mathcal{R} \triangleright \texttt{ref}{:}\, \tau \rightarrow \mathit{Set}_1(\tau)\ \mathit{ref}$$

(*deref*) $$\frac{\Gamma, \mathcal{R} \triangleright e{:}\, \tau\ \mathit{ref}}{\Gamma, \mathcal{R} \triangleright !e{:}\, \tau}$$

$$(assign) \qquad \frac{\Gamma, \mathcal{R} \rhd e_1 : \tau\ ref \qquad \Gamma, \mathcal{R} \rhd e_2 : \tau}{\Gamma, \mathcal{R} \rhd e_1 := e_2 : \tau}$$

An implicit side-condition on all the axioms and typing rules is that the type judgement in the consequent is a well-formed typing judgment. This condition is non-trivial only for (var), $(const)$, (ref), and (abs). In other words, for all the remaining rules, if the premises are well-formed typing judgments, so is the consequent.

The typing rule for `ref` enforces the restriction that any value placed in the store must have a type in which all type variables are imperative. Thus, while `ref` can be applied to an expression of unrestricted type τ, the resulting reference cell would have an imperative type, since $W(Set_1(\tau))$ is an imperative type for any type τ. Since applicative variables do not appear in the type of a stored value, the (let) rule generalizes all applicative type variables that are not free in the variable-context. Further, by our restriction on well-formed typing judgements, these applicative variables also cannot occur free in $\mathcal{R}$, since all the free variables of $\mathcal{R}$ are imperative.

To understand the (abs) rule, we can give an alternative but equivalent formulation. For any set T of type variables, we define the map $Up_{\backslash T}$ on monotypes, increasing the strength of all variables except imperative occurrences of variables from T, by

$$\begin{aligned} Up_{\backslash T}(\underline{\alpha}^0) &= \underline{\alpha}^0 \quad \text{if } \alpha \in T \\ Up_{\backslash T}(\underline{\beta}^i) &= \underline{\beta}^{i+1} \quad \text{if } \beta \notin T \text{ or } i > 0 \\ Up_{\backslash T}(\alpha^i) &= \alpha^{i+1} \end{aligned}$$

Then the (abs) rule, together with the side-condition that the resulting judgement is well-formed, is equivalent to the rule

$$\frac{\Gamma, x{:}\, \tau_1, \mathcal{R} \rhd e : \tau_2}{\Gamma, \mathcal{R} \rhd \lambda x.\, e : \tau_1 \to Up_{\backslash T}(\tau_2)}$$

with $T \supseteq \mathrm{ITV}(\Gamma, x{:}\,\tau_1, \mathcal{R})$, $T \subseteq \mathrm{ATV}(\tau_1)$ and for any $\alpha \in \mathrm{ITV}(\Gamma, \mathcal{R}) \cap \mathrm{FTV}(\tau_1), \alpha \in \mathrm{ITV}(\tau_1)$. Focussing only on the condition that $T \supseteq \mathrm{ITV}(\Gamma, x{:}\,\tau_1)$, this rule has the effect of permitting any of the imperative variables of τ_2, except those occurring imperatively in $\Gamma, x{:}\,\tau_1, \mathcal{R}$, to be made applicative. The variables that are imperative in τ_2 but not in $\Gamma, x{:}\,\tau_1, \mathcal{R}$ are those that appear in a stored value as a result of the evaluation of e. Since the evaluation of $\lambda x.\, e$ does not evaluate e, the references that e creates would not result from the evaluation of $\lambda x.\, e$ and hence can be safely made applicative.

Since stores are defined using values, which are syntactic expressions, we can use the type system to define store typings for stores. We say that a store s has store typing $\mathcal{R}$, written $s{:}\,\mathcal{R}$, if $\mathrm{Dom}(s) = \mathrm{Dom}(\mathcal{R})$ and for all $l \in \mathrm{Dom}(s)$ we can derive the judgement $\phi, \mathcal{R} \rhd s(l) : \mathcal{R}(l)$.

5 Relation to SML-NJ

In the absence of a simple presentation of the algorithm used by the New Jersey implementation of ML, it is difficult to prove that all the expressions typable in ML are typable in our type system. Such a result coupled with our soundness proof would establish the soundness of SML-NJ. However, by allowing mixed "ordinary" and "weak" occurrences of a type variable and associating indices with "ordinary" occurrences as well, and a more careful accounting of the strengths during application and abstraction we are able to type-check many expressions which SML-NJ rejects. In this section, we give examples of such expressions. Along with demonstrating the expressive power of our type system, these examples help understand how the type system works.

In [LW91], Leroy and Weis give examples of various benchmark expressions to type-check. Their expressions fall into two categories. One class of expressions is those which use references only locally and which are given the same types in their type system as the corresponding applicative expressions. The other class of expressions are those which exercise the compatibility between functions that create mutable data and higher-order functions, all of which are rejected by SML-NJ. Leroy and Weis take this to be strong evidence that SML-NJ does not handle full functionality correctly. However, our type system gives all such expressions in [LW91, TJ92] the correct types. We illustrate this by considering just one example. Consider the expression $(\lambda x.\, x)\, \mathtt{ref}$. With our abstraction and application rules it can be seen that $\phi \rhd \lambda x.\, x\colon (\alpha^0 \to \underline{\alpha}^1\ \mathit{ref}) \to (\alpha^1 \to \underline{\alpha}^2\ \mathit{ref})$ and hence $\phi \rhd (\lambda x.\, x)\, \mathtt{ref}\colon \alpha^0 \to \underline{\alpha}^1\ \mathit{ref}$ which is the same type as $\mathtt{ref}$. However, SML-NJ rejects this expression. Thus our type system demonstrates that weak polymorphism, if formulated carefully, can indeed work well when applicative and imperative functions are mingled.

The other concept in our system that is different from SML-NJ is the association of indices with ordinary type variables, which allows us to infer any applications in the body of an expression. Thus $\lambda x.\, x$ has the type $\alpha^0 \to \alpha^1$, while $\lambda x.\, x\, \mathtt{nil}$ has the type $(\alpha^0\ \mathit{list} \to \beta^1) \to \beta^1$. The increase in the index of α in the result type of $\lambda x.\, x$ relative to its index in the argument type indicates the lack of any applications , while the index of β remaining the same in both argument and result types of $\lambda x.\, x\, \mathtt{nil}$ indicates the application inside its body. (Since the occurrences of β are all ordinary, we also have $\lambda x.\, x\, \mathtt{nil}\colon (\alpha^0\ \mathit{list} \to \beta^6) \to \beta^6$, for example.) On the other hand SML-NJ gives these expressions the types $\alpha \to \alpha$ and $(\alpha\ \mathit{list} \to \beta) \to \beta$, with α and β implicitly considered to be of strength ∞. However, since such a type gives no information about applications inside the body, SML-NJ is forced to be conservative when such expressions are applied to say, $\mathtt{ref}$. Hence, SML-NJ rejects $(\lambda x.\, x\, \mathtt{nil})\, \mathtt{ref}$ correctly, but is also forced to reject $(\lambda x.\, x)\, \mathtt{ref}$. Using indices on ordinary type variables, we correctly deal with both these expressions as shown below:

$$(\lambda x.\, x\, \mathtt{nil}) : \forall \alpha\beta.(\alpha^0\ \mathit{list} \to \beta^1) \to \beta^1 \ \succ\ (\alpha^0\ \mathit{list} \to \underline{\alpha}^1\ \mathit{list}\ \mathit{ref}) \to \underline{\alpha}^1\ \mathit{list}\ \mathit{ref}$$

Hence, $(\lambda x.\, x\, \mathtt{nil})\, \mathtt{ref} : \underline{\alpha}^0\ \mathit{list}\ \mathit{ref}$.
On the other hand,

$$(\lambda x.\, x) : \forall \alpha.\alpha^0 \to \alpha^1 \ \succ\ (\alpha^0 \to \underline{\alpha}^1\ \mathit{ref}) \to (\alpha^1 \to \underline{\alpha}^2\ \mathit{ref})$$

and thus, $(\lambda x.\, x)\, \mathtt{ref} : \alpha^0 \to \underline{\alpha}^1\ \mathit{ref}$.

6 Soundness

In this section, we prove the soundness of the type system. The main lemma is a weaker form of the subject reduction property which shows the preservation of typability, for closed expressions, under evaluation. Since $\mathtt{wrong}$, the run-time error, is not typable and only closed expressions are compiled, this is sufficient to prove the correctness of our type system. Since store typings were defined in Section 4.2 using derivability in the type system, rather than as maximal fixed points, we do not have to use co-induction in our proofs. This makes our soundness proof simpler than the one given in [Tof90].

We begin by proving some syntactic properties of the type system. In the following we use $\vdash \Gamma, \mathcal{R} \rhd e\colon \tau$ to indicate that the judgement $\Gamma, \mathcal{R} \rhd e\colon \tau$ is derivable. Our first lemma is a generalization of the standard property of closure of typing derivations under substitution. It shows that by substituting for variables in a provable typing judgement, adding assumptions to the context, and making any number of variables in the type of the expression imperative, we still obtain provable typing judgements. To make the lemma easier to state, we first define the notion of an "instance" of a typing judgement.

Definition 6.1 *Let* $\Gamma, \mathcal{R} \triangleright e\colon \tau$ *and* $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ *be well-formed typing judgements. We say that* $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ *is an* **instance** *of* $\Gamma, \mathcal{R} \triangleright e\colon \tau$ *if there is a substitution* S *such that* $\forall x \in Dom(\Gamma)$, $\Gamma'(x) = Imp_{T_x}(S\,\Gamma(x))$ *for some set* T_x, $\forall l \in Dom(\mathcal{R})$, $\mathcal{R}'(l) = S\,\mathcal{R}(l)$, *and* $\tau' = Imp_T(S\,\tau)$ *for some set* T.

Lemma 6.2 *If* $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ *is an instance of* $\Gamma, \mathcal{R} \triangleright e\colon \tau$ *and* $\vdash \Gamma, \mathcal{R} \triangleright e\colon \tau$, *then* $\vdash \Gamma', \mathcal{R}' \triangleright e\colon \tau'$.

Proof The proof proceeds by induction on the proof of $\Gamma, \mathcal{R} \triangleright e\colon \tau$ and case analysis on the last axiom or proof rule used. We describe the representative cases of (var) and (abs). The latter case is particularly illustrative since it uses the restrictions on well-formed types and well-formed typing judgements. In each case, assume that $\Gamma', \mathcal{R}' \triangleright e\colon \tau'$ is an instance of $\Gamma, \mathcal{R} \triangleright e\colon \tau$ via substitution S and sets T_x, for each $x \in \mathrm{Dom}(\Gamma)$, and set T.

(var) The axiom has the form

$$\Gamma, \mathcal{R} \triangleright x\colon Imp_{T_1}(\tau_1)$$

with $\tau = Imp_{T_1}(\tau_1)$. Let $\Gamma(x) = \forall\alpha_1 \ldots \alpha_n.\ \rho$ and let S_1 be the substitution with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $\tau_1 = S_1\,\rho$. We need to find substitution S' and variable-set T', so that τ' is the result of making the variables of T' imperative in an instance of $\Gamma'(x)$ via substitution S'. Let $\Gamma'(x) = \forall\beta_1 \ldots \beta_n.\ \rho'$. Consider the substitution S' with domain $\{\beta_1, \ldots, \beta_n\}$ given by $S'(\beta_i) = S(S_1(\alpha_i))$. Then, $\tau' = Imp_{\mathrm{ITV}(\tau')}(S'\,\Gamma'(x))$ and hence $\Gamma', \mathcal{R} \triangleright x\colon \tau'$ is a valid application of the (var) axiom.

(abs) The last step is of the form

$$\frac{\Gamma, x\colon \tau_1, \mathcal{R} \triangleright e'\colon W(\tau_2)}{\Gamma, \mathcal{R} \triangleright \lambda x.\, e'\colon \tau_1 \to \tau_2}$$

with $\tau = \tau_1 \to \tau_2$. Consider $\Gamma'' = \Gamma', x\colon Imp_T(S\,\tau_1), \tau'' = W(Imp_T(S\,\tau_2))$. We prove that $\vdash \Gamma'', \mathcal{R}' \triangleright e'\colon \tau''$, from which it follows using the (abs) rule that $\vdash \Gamma', \mathcal{R}' \triangleright \lambda x.\, e'\colon \tau'$. We first see that $\Gamma'', \mathcal{R}' \triangleright e'\colon \tau''$ is a well-formed judgement. Since, $\tau' = Imp_T(S\,\tau_1) \to Imp_T(S\,\tau_2)$ is a well-formed type expression, we must have that τ'' is a well-formed type expression. Now, suppose that $\alpha \in \mathrm{ITV}(\Gamma'', \mathcal{R}') \cap \mathrm{FTV}(\tau'')$. If $\alpha \in \mathrm{ITV}(\Gamma', \mathcal{R}')$ then since $\Gamma', \mathcal{R}' \triangleright e'\colon \tau'$ is a well-formed typing judgment we have that $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_2))$ and hence $\alpha \in \mathrm{ITV}(\tau'')$. And, if $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_1))$ then since τ' is a well-formed type expression we get that $\alpha \in \mathrm{ITV}(Imp_T(S\,\tau_2))$ and hence $\alpha \in \mathrm{ITV}(\tau'')$. We next observe that $\tau'' = Imp_T(S(W(\tau_2)))$, since for any substitution S, $S(W(\tau)) = W(S(\tau))$ if $W(\tau)$ is defined. Hence using substitution S, $T_x = T$, we can obtain $\Gamma'', \mathcal{R}' \triangleright e'\colon \tau''$ as an instance of $\Gamma, x\colon \tau_1, \mathcal{R} \triangleright e'\colon W(\tau_1)$ which is therefore derivable by the induction hypothesis.

■

As corollaries of Lemma 6.2, we obtain statements describing the behaviour of derivable type-judgements under substitutions, and adding variables and locations to the context. In our type system adding assumptions to the context may have the effect of forcing some additional variables in the type of the expression to be imperative.

Corollary 6.3 (Type Substitution) *Suppose* $\vdash \Gamma, \mathcal{R} \triangleright e\colon \tau$ *. Let* S *be any substitution and* T *any set of type variables such that* $S\Gamma, S\mathcal{R} \triangleright e\colon \mathit{Imp}_T(S\,\tau)$ *is a well-formed typing judgement. Then,* $\vdash S\Gamma, S\mathcal{R} \triangleright e\colon \mathit{Imp}_T(S\,\tau)$ *.*

Corollary 6.4 (Adding Variables and Locations) *Suppose* $\vdash \Gamma, \mathcal{R} \triangleright e\colon \tau$ *. Let* $\Gamma' \supseteq \Gamma, \mathcal{R}' \supseteq \mathcal{R}$ *. Then* $\vdash \Gamma', \mathcal{R}' \triangleright e\colon \mathit{Imp}_{ITV(\Gamma',\mathcal{R}')}\,\tau$ *.*

Proof For any well-formed monotype τ and any set of type variables T, $\mathit{Imp}_T(\tau)$ is still a well-formed monotype. ■

We now establish a term-substitution property which is crucial to proving the subject reduction property for β-reduction and the reduction of `let`-expressions. The property we are interested in is Corollary 6.6 which is not provable directly by induction. We therefore prove a stronger statement in the following lemma. The statement of the lemma also provides formal justification for allowing type variables to be made imperative in the (var) rule.

Lemma 6.5 *Assume that* $\vdash \Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e\colon \tau'$ *and* $\{\alpha_1, \ldots, \alpha_n\} \cap FTV(\Gamma, \mathcal{R}) = \emptyset$ *. If* $\vdash \Gamma, \mathcal{R} \triangleright v\colon \mathit{Imp}_{ITV(\Gamma,\mathcal{R})}\,\tau$ *then* $\vdash \Gamma, \mathcal{R} \triangleright [v/x]e\colon \tau'$ *.*

Proof By induction on the proof of $\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e\colon \tau'$.

(var) $e = x'$. If $x' \neq x$ then the lemma is trivially true. Otherwise, there is a substitution S with domain $\{\alpha_1, \ldots, \alpha_n\}$ such that $\tau' = \mathit{Imp}_T(S\tau)$. By the definition of well-formed typing judgements, we can assume that $T \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$. Since $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma, \mathcal{R}) = \emptyset$ we have that $S\Gamma, S\mathcal{R} = \Gamma, \mathcal{R}$. Thus, applying Lemma 6.2 to $\Gamma, \mathcal{R} \triangleright v\colon \mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau$, we can derive the judgment

$$\Gamma, \mathcal{R} \triangleright v\colon \mathit{Imp}_T(S(\mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau))$$

And, since $T \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$, $\mathit{Imp}_T(S(\mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau)) = \tau'$. Hence $\vdash \Gamma, \mathcal{R} \triangleright [v/x]x\colon \tau'$.

(abs) $e = \lambda x'.\, e_1$. Then,

$$\frac{\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, x'\colon \tau_1, \mathcal{R} \triangleright e_1\colon W(\tau_2)}{\Gamma, x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright \lambda x'.\, e_1\colon \tau_1 \rightarrow \tau_2}$$

Let $\{\beta_1, \ldots, \beta_n\}$ be distinct variables such that $\beta_i \notin \mathrm{FTV}(\Gamma, \mathcal{R}) \cup \{\alpha_1 \ldots, \alpha_n\}$. Let S be the substitution defined by $S(\alpha_i) = \beta_i^0$, $\Gamma' = \Gamma, x'\colon S\tau_1$. By the definition of well-formed typing judgements and the nature of S, all imperative variables of $\Gamma', x\colon \forall\alpha_1 \ldots, \alpha_n.\tau, \mathcal{R}$ can only appear imperatively in $S(W(\tau_2))$. Hence, using Lemma 6.2, and substitution S on the premise of the *(abs)* rule, we can derive

$$\Gamma', x\colon \forall\alpha_1 \ldots \alpha_n.\tau, \mathcal{R} \triangleright e_1\colon S(W(\tau_2))$$

since $S\Gamma, S\mathcal{R} = \Gamma, \mathcal{R}$. By the choice of S, $\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma', \mathcal{R}) = \emptyset$. We have that $\vdash \Gamma, \mathcal{R} \triangleright v\colon \tau''$, where $\tau'' = \mathit{Imp}_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau$. Using Corollary 6.4, $\vdash \Gamma', \mathcal{R} \triangleright v\colon \mathit{Imp}_{\mathrm{ITV}(\Gamma',\mathcal{R})}\,\tau''$. Since $\mathrm{ITV}(\Gamma', \mathcal{R}) \supseteq \mathrm{ITV}(\Gamma, \mathcal{R})$, we thus have that $\vdash \Gamma', \mathcal{R} \triangleright v\colon \mathit{Imp}_{\mathrm{ITV}(\Gamma',\mathcal{R})}\,\tau$. Hence, by induction hypothesis, we can derive

$$\Gamma', \mathcal{R} \triangleright [v/x]e_1\colon S(W(\tau_2))$$

Let $S'(\beta_i) = \alpha_i^0$. Using S' with Corollary 6.4, we can derive

$$\Gamma', x'\colon \tau_1, \mathcal{R} \triangleright [v/x]e_1\colon W(\tau_2)$$

Now, using the (abs) rule, we can get the desired judgement.

The (loc), $(const)$, (app) cases are trivially true, and the (let) case is similar to the (abs) case. ■

Corollary 6.6 (Term Substitution) *If* $\vdash \Gamma, x{:}\,\forall\alpha_1 \ldots \alpha_n.\tau \rhd e{:}\,\tau'$ *and* $\vdash \Gamma \rhd v{:}\,\tau$ *with* $\{\alpha_1, \ldots, \alpha_n\} \cap FTV(\Gamma) = \phi$ *then* $\vdash \Gamma \rhd [v/x]e{:}\,\tau'$.

Proof We have that $\Gamma, \mathcal{R} \rhd v{:}\,\tau$ is derivable. By the restriction on well-formed typing judgements, $Imp_{\mathrm{ITV}(\Gamma,\mathcal{R})}\,\tau = \tau$. Hence, by Lemma 6.5, we have the desired result. ■

Using the above lemmas, we are now ready to connect the type system with the operational semantics. To establish the correspondence with the semantics, we need to assume some properties of the typings for the term constructors, since our proof system depends on them. The following definition states what we require of the typings for the term constructors.

Definition 6.7 *The typing for term-constructors given by Typeof is* **admissible** *if the resulting proof system for deriving typing judgments has the following properties:*

(i) *For all values* $v \in V$ *(i.e., other than* λ*-abstractions,* `ref`*, and location constants), if* $\vdash \phi, \mathcal{R} \rhd v{:}\,\tau$ *for any store typing* $\mathcal{R}$*, type* τ*, then* τ *is of the form* $b(\tau_1, \ldots, \tau_n)$ *for some type constructor* b*.*

(ii) *Let* $\langle e_1, e_2\rangle \in F$*. For any* $\mathcal{R}, \tau$*, if* $\vdash \phi, \mathcal{R} \rhd e_1{:}\,\tau$ *then* $\vdash \phi, \mathcal{R} \rhd e_2{:}\,\tau$*.*

It is a consequence of (i) that the only functional values are λ-abstractions and `ref`, and the only values that are reference cells are location constants. (Indeed, we could have eliminated `ref` from being a value as well, by having the redex $\texttt{ref} \leadsto \lambda x.\texttt{ref}\ x$). Condition (ii) requires the typings for term constructors to be consistent with their operational properties as given by their functional reductions. We assume in the following lemmas that the given typings for term-constructors is admissible.

Our operational semantics was described by defining the notions of redexes, evaluation contexts, and evaluation steps, in that order. The presentation of the lemmas for establishing the soundness of our type system follows the same order. First, we prove subject reduction for the purely functional reductions.

Lemma 6.8 (Subject Reduction for Functional Reductions) *If* $\vdash \phi, \mathcal{R} \rhd e_1{:}\,\tau$ *and* $e_1 \leadsto e_2$ *then* $\vdash \phi, \mathcal{R} \rhd e_2{:}\,\tau$*.*

Proof By case analysis on $e_1 \leadsto e_2$.

Case $(\lambda x.\, e)\ v \leadsto [v/x]e$. Assume $\vdash \phi, \mathcal{R} \rhd (\lambda x.\, e)\ v{:}\,\tau$. Then $\vdash \phi, \mathcal{R} \rhd \lambda x.\, e{:}\,\tau_1 \to \tau_2, \vdash \phi, \mathcal{R} \rhd v{:}\,\tau_1$ and $\tau = W(\tau_2)$. From $\vdash \phi, \mathcal{R} \rhd \lambda x.\, e{:}\,\tau_1 \to \tau_2$, we have $\vdash x{:}\,\tau_1, \mathcal{R} \rhd e{:}\,W(\tau_2)$. Using Corollary 6.6 we get $\vdash \phi, \mathcal{R} \rhd [v/x]e{:}\,\tau$.

Case $(\texttt{let}\ x = v\ \texttt{in}\ e) \leadsto [v/x]e$. Assume $\vdash \phi, \mathcal{R} \rhd \texttt{let}\ x = v\ \texttt{in}\ e{:}\,\tau$. Then

$$\vdash \phi, \mathcal{R} \rhd v{:}\,\tau_1, \vdash x{:}\,Clos_\phi \tau_1, \mathcal{R} \rhd e{:}\,\tau$$

Since $\phi, \mathcal{R} \rhd v{:}\,\tau_1$ is a well-formed typing judgement, if $\alpha \in \mathrm{ATV}(\tau_1)$ then $\alpha \notin \mathrm{FTV}(\mathcal{R})$. Hence, using Corollary 6.6 we get $\vdash \phi, \mathcal{R} \rhd [v/x]e{:}\,\tau$.

Case $v_1\ v_2 \leadsto \texttt{wrong}$. Since v_1 is not a λ-abstraction or `ref`, it must be a location constant or a value in V. The former can only have a type of the form $\tau'\ ref$ and the latter can only have a type of the form $b(\tau_1, \ldots, \tau_n)$ by condition (i) of admissibility. Hence, $\not\vdash \phi, \mathcal{R} v_1 \tau_1 \to \tau_2$ for any $\mathcal{R}, \tau_1, \tau_2$. Hence $\not\vdash \phi, \mathcal{R} \rhd v_1\ v_2{:}\,\tau$ for any $\mathcal{R}, \tau$ and the statement is true vacuously.

Otherwise $e_1 \leadsto e_2$ is a redex involving term-constructors and the statement is true by condition (ii) of admissibility. ■

We might expect a proof of soundness to proceed by extending Lemma 6.8 to establish subject reduction for evaluation. In more detail, suppose that $(e, s) \longmapsto (e', s')$. We would like to establish that if e has type τ, then so does e'. Note that because of the possiblity of creation of reference cells, e' may mention location constants that do not appear in e'. For this reason, the typing of e' must involve assumptions about additional location constants. By our restriction on typing judgements, e' can then have type τ only if the type variables in the types of the new location constants appearing in e' all appear imperatively in τ. Unfortunately this condition on type variables may not be true, since there is nothing in our type system to force "new" type variables to be used whenever possible. As a result, an applicative type variable in τ could "accidentally" be the same as one in the type of a new reference cell created. However, as (iii) of the following lemma shows, such applicative variables in τ can be suitably renamed so that e' can be given the renamed type. The form of the renaming is the crucial formal justification for allowing all applicative variables to be treated polymorphically in the (let) rule. It is also in the proof of (iii) that we see the formal origin of our restrictions on the syntactic form of types and typing judgements. Parts (i) and (ii) express obvious properties, but have been stated explicitly merely for ease of reference in the proof of Lemma 6.10.

Lemma 6.9 *Let $E[\,]$ be any evaluation context and suppose that $\vdash \phi, \mathcal{R} \triangleright E[e]\colon \tau$. Then,*

(i) $\vdash \phi, \mathcal{R} \triangleright e\colon \tau_1$ *for some type* τ_1.

(ii) *If* $\vdash \phi, \mathcal{R} \triangleright e'\colon \tau_1$, *then* $\vdash \phi, \mathcal{R} \triangleright E[e']\colon \tau$.

(iii) *Suppose that for some $\mathcal{R}' \supseteq \mathcal{R}$ and some expression e', we have that $\vdash \phi, \mathcal{R}' \triangleright e'\colon \tau_1$. Let $\{\alpha_1, \ldots, \alpha_n\} = ATV(\tau)$. Then for some distinct variables $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin FTV(\mathcal{R}')$ we have that $\vdash \phi, \mathcal{R}' \triangleright E[e']\colon S_E\tau$ where $S_E = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$.*

Proof By induction on the structure of $E[\,]$. Statements (i) and (ii) can be proved trivially, so we only show details for (iii). Note that for any distinct $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}')$ we have that $\phi, \mathcal{R} \triangleright E[e']\colon [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]\tau$ is a well-formed judgement since $\mathrm{ATV}([\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]\tau) = \{\beta_1, \ldots, \beta_n\}$.

- $E[\,] \equiv [\,]$. Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau_1)$. Choose any $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}')$. Since $\phi, \mathcal{R}' \triangleright e'\colon \tau_1$ is a well-formed judgement and $\mathrm{FTV}(\mathcal{R}') = \mathrm{ITV}(\mathcal{R}')$ we have that $\alpha_i \notin \mathrm{FTV}(\mathcal{R}')$. Hence for $S_{[\,]} = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$, $S_{[\,]}\mathcal{R}' = \mathcal{R}'$. Hence using Lemma 6.2, $\vdash \phi, \mathcal{R}' \triangleright e'\colon S_{[\,]}\tau_1$.

- $E[\,] \equiv E_1\, e_2$. Then we can derive

$$\phi, \mathcal{R} \triangleright E_1[e]\colon \tau' \to \tau_2 \tag{1}$$
$$\phi, \mathcal{R} \triangleright e_2\colon \tau' \tag{2}$$

  and $\tau = W(\tau_2)$. Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau)$. Clearly $\mathrm{ITV}(\tau_2) \subseteq \mathrm{ITV}(\tau)$. Hence $\mathrm{ATV}(\tau) \subseteq \mathrm{ATV}(\tau_2) \subseteq \mathrm{ATV}(\tau' \to \tau_2)$. Thus, $\mathrm{ATV}(\tau' \to \tau_2) = \{\alpha_1, \ldots, \alpha_n, \alpha_{n+1}, \ldots, \alpha_p\}$ for some variables $\alpha_{n+1}, \ldots, \alpha_p$. By inductive hypothesis about $E_1[\,]$,

$$\vdash \phi, \mathcal{R}' \triangleright E_1[e']\colon S_{E_1}(\tau') \to S_{E_1}(\tau_2)$$

where $S_{E_1}(\alpha_i) = \beta_i^0, 1 \leq i \leq p$ for some variables $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}')$. Since (1) is a well-formed judgement, $\mathrm{Dom}(S_{E_1}) \cap \mathrm{FTV}(\mathcal{R}') = \phi$. Hence $S_{E_1}\mathcal{R} = \mathcal{R}$. Note that $\phi, \mathcal{R}' \triangleright e_2 : S_{E_1}\tau'$ is a well-formed judgement since $\mathrm{ATV}(S_{E_1}\tau') \subseteq \{\beta_1, \ldots, \beta_p\}$ and $\{\beta_1, \ldots, \beta_p\} \cap \mathrm{FTV}(\mathcal{R}') = \phi$. Hence using Lemma 6.2 on (2) with substitution S_{E_1}, we get that $\vdash \phi, \mathcal{R}' \triangleright e_2 : S_{E_1}\tau'$. Thus using (app), $\vdash \phi, \mathcal{R}' \triangleright E[e'] : S_{E_1}(W(\tau_2))$, *i.e.*,

$$\vdash \phi, \mathcal{R}' \triangleright E[e'] : S_{E_1}(W(\tau_2)) \tag{3}$$

since $W(S_{E_1}\tau_2) = S_{E_1}(W(\tau_2))$, as $W\tau_2$ is defined. Let $S' = [\alpha_{n+1}^0/\beta_{n+1}, \ldots, \alpha_p^0/\beta_p]$. Take $S_E = S' \circ S_{E_1}$. Since $\mathrm{Dom}(S') \cap \mathcal{R}' = \phi$, using substitution S' and Corollary 6.3 on (3) we get $\vdash \phi, \mathcal{R}' \triangleright E[e'] : S_E(\tau)$.

- $E[\,] \equiv v\ E_1$. Then we can derive

$$\begin{gathered}\phi, \mathcal{R} \triangleright v : \tau' \to \tau_2 \\ \phi, \mathcal{R} \triangleright E_1[e] : \tau'\end{gathered}$$

  and $\tau = W(\tau_2)$. Let $\mathrm{ATV}(\tau') = \{\alpha_1, \ldots, \alpha_n\}$. By inductive hypothesis about E_1, we have variables $\beta_1, \ldots, \beta_n \notin \mathrm{FTV}(\mathcal{R}')$ such that

$$\phi, \mathcal{R}' \triangleright E_1[e'] : S_{E_1}(\tau')$$

  with $S_{E_1} = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$. Since $\mathrm{ATV}(\tau') \subseteq \mathrm{ATV}(\tau' \to \tau_2)$, we have $\mathrm{ATV}(\tau' \to \tau_2) = \{\alpha_1, \ldots, \alpha_n, \alpha_{n+1}, \ldots, \alpha_p\}$ for some variables $\alpha_{n+1}, \ldots, \alpha_p$. Choose any distinct variables $\beta_{n+1}, \ldots, \beta_p$ such that $\beta_j \notin \mathrm{FTV}(\mathcal{R}'), n+1 \leq j \leq p$. Consider $S' = [\beta_1^0/\alpha_1, \ldots, \beta_p^0/\alpha_p]$.

  We claim that $S'\tau' = S_{E_1}\tau'$. For consider any $\alpha \in \mathrm{FTV}(\tau')$. If $\alpha \in \mathrm{ATV}(\tau')$ then clearly $S'(\alpha) = S_{E_1}(\alpha)$. Otherwise $\alpha \in \mathrm{ITV}(\tau')$ whence $\alpha \in \mathrm{ATV}(\tau_2)$ by out restriction on the well-formed type $\tau' \to \tau_2$. *This is the formal origin of our restriction on well-formed types.* Hence $\alpha \in \mathrm{Dom}(S')$ and $S'(\alpha) = S_{E_1}(\alpha)$. Then we use similar reasoning as in the previous case.

- $E[\,] \equiv \texttt{let}\ x = E_1\ \texttt{in}\ e_2$. Then we can derive

$$\phi, \mathcal{R} \triangleright E_1[e] : \tau' \tag{4}$$

$$x : \mathit{Clos}_\phi \tau', \mathcal{R} \triangleright e_2 : \tau \tag{5}$$

  By induction hypothesis,

$$\vdash \phi, \mathcal{R}' \triangleright E_1[e'] : S_{E_1}\tau' \tag{6}$$

  for some suitable S_{E_1}. By the form of substitution S_{E_1}, $\mathit{Clos}_\phi \tau_1 = \mathit{Clos}_\phi S_{E_1}(\tau_1)$, *which was precisely the purpose of the inductive statement in (iii).*

  Let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ATV}(\tau)$. Choose any distinct $\beta_1, \ldots, \beta_n$ such that $\beta_i \notin \mathrm{FTV}(\mathcal{R}') \cup \mathrm{ITV}(\tau')$. Let $S_E = [\beta_1^0/\alpha_1, \ldots, \beta_n^0/\alpha_n]$. Then clearly

$$x : \mathit{Clos}_\phi \tau_1, \mathcal{R}' \triangleright e_2 : S_E\tau$$

  is a well-formed judgement. Also since $\mathrm{FTV}(\mathit{Clos}_\phi \tau_1) = \mathrm{ITV}(\mathit{Clos}_\phi \tau_1)$ and since (5) is a well-formed judgement, $\alpha_i \notin \mathrm{FTV}(\mathit{Clos}_\phi \tau_1)$. Thus using Lemma 6.2 on (5) with substitution S_E we have

$$\vdash x : \mathit{Clos}_\phi(S_{E_1}\tau'), \mathcal{R}' \triangleright e_2 : S_E\tau$$

  Combining with (6),

$$\vdash \phi, \mathcal{R}' \triangleright E[e'] : S_E\tau$$

Other cases of evaluation contexts are similar. ∎

Under suitable assumptions about the store in which an expression is evaluated, we can now establish that evaluation of typable expressions leads to typable expressions.

Lemma 6.10 (Preservation of Typability) *Suppose* $(e, s) \longmapsto (e', s')$ *and* $\mathcal{R}$ *is a store typing such that* $s\colon \mathcal{R}$ *and* $\vdash \phi, \mathcal{R} \triangleright e\colon \tau$ *. Then there is a store typing* $\mathcal{R}'$ *such that* $s'\colon \mathcal{R}'$ *and for some type* τ' *,* $\vdash \phi, \mathcal{R}' \triangleright e'\colon \tau'$ *.*

Proof By case analysis on $\longmapsto$.

- $(E[e], s) \longmapsto (E[e'], s)$ with $e \leadsto e'$. Assume that $\vdash \phi, \mathcal{R} \triangleright E[e]\colon \tau$. From Lemma 6.9(i), we have that $\vdash \phi, \mathcal{R} \triangleright e\colon \tau_1$ for some type τ_1. By Lemma 6.8, $\vdash \phi, \mathcal{R} \triangleright e'\colon \tau_1$. Now, using Lemma 6.9(ii), $\vdash \phi, \mathcal{R} \triangleright E[e']\colon \tau$. And, by assumption $s\colon \mathcal{R}$.

- $(E[\mathtt{ref}\ v], s) \longmapsto (E[l], s[l \mapsto v])$.
  By Lemma 6.9(i), $\vdash \phi, \mathcal{R} \triangleright \mathtt{ref}\ v\colon \tau_1$ for some type τ_1. Hence $\vdash \phi, \mathcal{R} \triangleright v\colon \tau_1'$ and $\tau_1 = W(Set_1(\tau_1'))\ ref$.

  Consider $\mathcal{R}' = \mathcal{R}, l\colon Set_0(\tau_1')$. For $l' \neq l$, since $\vdash \phi, \mathcal{R} \triangleright s(l')\colon \mathcal{R}(l')$ and $\mathcal{R}(l')$ is imperative, we have by Corollary 6.4, $\vdash \phi, \mathcal{R}' \triangleright s(l')\colon \mathcal{R}(l')$, *i.e.*, $\vdash \phi, \mathcal{R}' \triangleright s'(l')\colon \mathcal{R}'(l')$ where $s' = s[l \mapsto v]$. And, since $\vdash \phi, \mathcal{R} \triangleright v\colon \tau_1'$, by Corollary 6.4, $\vdash \phi, \mathcal{R}' \triangleright v\colon Imp_{\mathrm{ITV}(\mathcal{R}')}\tau_1'$. Clearly, $Imp_{\mathrm{ITV}(\mathcal{R}')}\tau_1' = Set_0(\tau_1')$. Hence, $\vdash \phi, \mathcal{R}' \triangleright s'(l)\colon \mathcal{R}'(l)$ and thus $s'\colon \mathcal{R}'$. Trivially, $\vdash \phi, \mathcal{R}' \triangleright l\colon \tau_1$, since $W(Set_1(\tau_1)') = Set_0(\tau_1')$. Thus, by Lemma 6.9(iii), $\vdash \phi, \mathcal{R}' \triangleright E[l]\colon \tau'$ for $\tau' = S_E(\tau)$.

- $(E[!l], s) \longmapsto (E[s(l)], s)$. By Lemma 6.9(i), $\vdash \phi, \mathcal{R} \triangleright !l\colon \tau_1$. Hence, $\vdash \phi, \mathcal{R} \triangleright l\colon \tau_1\ ref$, *i.e.*, $\mathcal{R}(l) = \tau_1$. Since $s\colon \mathcal{R}$, we have that $\vdash \phi, \mathcal{R} \triangleright s(l)\colon \mathcal{R}(l)$, *i.e.*, $\vdash \phi, \mathcal{R} \triangleright s(l)\colon \tau_1$. Now, using Lemma 6.9(ii), we get that $\vdash \phi, \mathcal{R} \triangleright E[s(l)]\colon \tau$.

- $(E[l := v], s) \longmapsto (E[v], s[l \mapsto v])$. Then $\vdash \phi, \mathcal{R} \triangleright l\colon \tau_1\ ref, \vdash \phi, \mathcal{R} \triangleright v\colon \tau_1$. Hence $s[l \mapsto v]\colon \mathcal{R}$. And since $\vdash \phi, \mathcal{R} \triangleright v\colon \tau_1$, by Lemma 6.9(ii), $\vdash \phi, \mathcal{R} \triangleright E[v]\colon \tau_1$.

- $(E[e], s) \longmapsto (\mathtt{wrong}, s)$, where $e \leadsto \mathtt{wrong}$. By Lemma 6.8, since $\mathtt{wrong}$ is not typeable, there is no $\mathcal{R}, \tau_1$ such that $\vdash \phi, \mathcal{R} \triangleright e\colon \tau_1$. By Lemma 6.9(i), this implies that $\not\exists \mathcal{R}, \tau$ such that $\vdash \phi, \mathcal{R} \triangleright E[e]\colon \tau$ and the Lemma is thus true vacuously.

- $(E[!v], s) \mapsto (\mathtt{wrong}, s)$, where $v \notin Loc$. By admissibility condition (i) and the *(abs)* rule, $\not\vdash \phi, \mathcal{R} \triangleright v\colon \tau_1\ ref$ for any $\mathcal{R}, \tau_1$. Hence $\not\vdash \triangleright \triangleright \phi\colon, \mathcal{R}\colon !v\tau_1$ for any $\mathcal{R}, \tau_1$. Then similar to previous case.

- $(E[v_1 := v_2], s) \longmapsto (\mathtt{wrong}, s)$, where $v_1 \notin Loc$. This is similar to previous case.

∎

As a corollary of Lemma 6.10, we can now establish that starting in the initial store, evaluation of any typable user program leads to typable expressions.

Corollary 6.11 *Let* e *be a program, i.e., a closed expression with no location constants. If* e *is typeable, i.e.,* $\vdash \phi, \phi \triangleright e\colon \tau$ *for some type* τ *then for any* e', s' *such that* $(e, \mathbf{init}) \longmapsto\!\!\!\!\rightarrow (e', s')$ *we have that* e' *is typeable.*

Proof The store **init** has the empty store typing, *i.e.*, $\mathbf{init}\colon \phi$. Now the statement follows from Lemma 6.10. ∎

Finally, we can establish our main theorem, that well-typed programs do not lead to run-time errors.

Theorem 6.12 *Let e be a typeable program. Then $Eval(e) \neq \texttt{wrong}$, i.e., e does not lead to a run-time error.*

Proof There is no $\mathcal{R}', \tau'$ such that $\vdash \phi, \mathcal{R}' \triangleright \texttt{wrong}: \tau'$. Hence, by Corollary 6.11 we cannot have $(e, \mathbf{init}) \longmapsto\!\!\!\rightarrow (\texttt{wrong}, s')$, for any store s'. ∎

7 Conclusion

We present a type system, in the form of typing rules , using weak type variables. Intuitively, a weak type variable is a variable with a numeric index, indicating the number of function applications needed to produce a reference cell containing a value of this type. Although the Standard ML of New Jersey source code is sufficiently complex to make direct proof difficult, extensive experiments with the compiler and examination of the source code suggest that our type system strictly subsumes the implemented type system. Moreover, as noted in Section 5, our type system is sufficiently flexible to give all benchmark examples listed in [LW91] the correct type, without the more complicated type expressions used in [LW91, TJ92, Wri92]. In light of the comments on the limitations of SML-NJ in [LW91] and elsewhere, our type system shows that simple indexed type variables allow more flexible typing of imperative programs than previously believed. To the extent that our type system subsumes the implemented algorithm, Theorem 6.12 shows not only the semantic soundness of our typing rules, but also the soundness of the SML-NJ algorithm that has been used for several years. Since our type system is more flexible than the one used in the SML-NJ, but requires very little or no change in displayed type expressions, we believe it is a good candidate for later implementations of ML and other polymorphic programming languages that have imperative or control constructs.

References

[AM87] A.W. Appel and D.B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer LNCS 274, 1987.

[AM91] A.W. Appel and D.B. MacQueen. Standard ML of New Jersey. In *Int'l Symp. Prog. Lang. Implementation and Logic*, pages 1–13. Springer-Verlag, 1991.

[CF58] H.B Curry and R. Feys. *Combinatory Logic I.* North-Holland, Amsterdam, 1958.

[Dam85] L. Damas. *Type assignment in programming languages.* PhD thesis, Edinburgh University CST-33-85, 1985.

[Hin69] J.R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. AMS*, 146:29–60, 1969.

[HL91] R. Harper and M. Lillibridge. A bug in sml polymorphic typing for continuations. Email to `types@theory.lcs.mit.edu`, 1991.

[JG91] P. Jouvelot and D. Gifford. Algebraic reconstruction of types and effects. In *18th ACM Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[KMM91] P.C. Kanellakis, H.G. Mairson, and J.C. Mitchell. Unification and ML type reconstruction. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[Koe92] A. Koenig. An anecdote about ml type inference. Unpublished Manuscript, 1992.

[KTU90] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is Dexptime-complete. In *Proc. 15th Colloq. on Trees in Algebra and Programming*, pages 206–220. Springer LNCS 431, 1990. To appear in *J. Assoc. Comput. Machinery* under title, "An Analysis of ML Typability".

[LW91] X. Leroy and P. Weis. Polymorphis type inference and assignment. In *18th ACM Symposium on Principles of Programming Languages*, pages 291–302, 1991.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17:348–375, 1978.

[TJ92] J.-P. Talpin and P. Jouvelot. The type and effects discipline. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 162–173, 1992.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988. Available as Edinburgh University Laboratory for Foundations of Computer Science Technical Report ECS–LFCS–88–54.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[WF92] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, 1992.

[Wri92] A.K. Wright. Typing references by effect inference. In *Proceedings of the European Symposium on Programming*, 1992.