A Type Soundness Proof for Variables in LCF ML †

Dennis Volpano
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943, USA
volpano@cs.nps.navy.mil

Geoffrey Smith
School of Computer Science
Florida International University
Miami, FL 33199, USA
smithg@fiu.edu

Abstract

We prove the soundness of a polymorphic type system for a language with variables, assignments, and first-class functions. As a corollary, this proves the soundness of the Edinburgh LCF ML rules for typing variables and assignments, thereby settling a long-standing open problem.

Keywords: Type theory, formal semantics, variables and assignment.

1 Introduction

A type system is presented for a language with a letvar construct to allocate variables, which are implicitly dereferenced and whose addresses are not first-class values, as in traditional imperative languages. Edinburgh LCF ML [GMW78] had such a construct, which it called letref. We show that the restriction that a variable must have weak type only if it is assigned to inside a λ-abstraction within its scope is sound. As a corollary then, LCF ML restriction 2(i)(b) (pg. 49 [GMW78]), which requires a variable to have a monotype (a type with no type variables) if the variable is assigned to inside a λ-abstraction within its scope, is also sound since every monotype is weak. This restriction was never proved sound, according to Tofte [Tof90].

2 The Type System

The syntax of the language we consider is core ML with a letvar construct and assignment. Following Tofte [Tof90], we distinguish a subset of the expressions called Values:

†Appeared in Information Processing Letters, 56(3), November 1995, pp.141-146. This material is based upon activities supported by the National Science Foundation under Agreements No. CCR-9400592 and CCR-9414421. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.
**A Type Soundness Proof for Variables in LCF ML**

Department of Computer Science Naval Postgraduate School Monterey, CA 93943

Approved for public release, distribution unlimited

<table>
<thead>
<tr>
<th>1. REPORT DATE</th>
<th>2. REPORT TYPE</th>
<th>3. DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 NOV 1995</td>
<td>N/A</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Type Soundness Proof for Variables in LCF ML</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5a. CONTRACT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5b. GRANT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5c. PROGRAM ELEMENT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5d. PROJECT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5e. TASK NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5f. WORK UNIT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department of Computer Science Naval Postgraduate School Monterey, CA 93943</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10. SPONSOR/MONITOR'S ACRONYM(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. SPONSOR/MONITOR’S REPORT NUMBER(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. SUBJECT TERMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. SECURITY CLASSIFICATION OF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. REPORT: unclassified</td>
</tr>
<tr>
<td>b. ABSTRACT: unclassified</td>
</tr>
<tr>
<td>c. THIS PAGE: unclassified</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17. LIMITATION OF ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>UU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>18. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Expressions

\[ \begin{align*}
(\text{Expressions}) & \quad e ::= v \mid l \mid e_1 e_2 \mid e_1 := e_2 \\
& \quad \text{let } x = e_1 \text{ in } e_2 \\
& \quad \text{letvar } x := e_1 \text{ in } e_2
\end{align*} \]

Values

\[ \begin{align*}
(\text{Values}) & \quad v ::= x \mid \text{unit} \mid \lambda x. e
\end{align*} \]

Meta-variable \( x \) ranges over identifiers. The \texttt{letvar} construct binds \( x \) to a new cell initialized to the value of \( e_1 \). The scope of the binding is \( e_2 \) and the lifetime of the cell is unbounded. Dereferencing of variables created with \texttt{letvar} is implicit. Locations are denoted by meta-variable \( l \) and are not values.

The types of the language are stratified as follows.

\[ \begin{align*}
\tau ::= \alpha \mid \text{unit} \mid \tau \rightarrow \tau' \quad (\text{data types}) \\
\sigma ::= \forall \alpha . \sigma \mid \tau \quad (\text{type schemes}) \\
\rho ::= \sigma \mid \tau \text{ var} \quad (\text{phrase types})
\end{align*} \]

The meta-variable \( \alpha \) ranges over \textit{type variables}. Type variables are partitioned into \textit{weak} and \textit{strong} type variables, written \( \omega \alpha \) and \( \alpha \) respectively. These variables correspond to the imperative and applicative type variables respectively of Tofte’s system. We say that a type scheme \( \sigma \) is \textit{weak} iff \( \sigma \) is unquantified and every type variable in \( \sigma \) is weak. Type \( \tau \text{ var} \) is the type of locations storing values of type \( \tau \).

The rules of the type system are formulated as they are in Harper’s system [Har94] and are given in Figure 1. It is a deductive proof system used to assign types to expressions. Typing judgements have the form

\[ \lambda ; \gamma \vdash e : \rho \]

meaning that expression \( e \) has type \( \rho \) assuming that \( \lambda \) prescribes type schemes for locations in \( e \) and \( \gamma \) prescribes phrase types for the free identifiers of \( e \). Meta-variable \( \gamma \) ranges over identifier typings. An \textit{identifier typing} \( \gamma \) is a finite function mapping identifiers to phrase types; \( \gamma(x) \) is the phrase type assigned to \( x \) by \( \gamma \) and \( \gamma[x : \rho] \) assigns phrase type \( \rho \) to \( x \) and to variable \( x' \neq x \), phrase type \( \gamma(x') \).

Meta-variable \( \lambda \) ranges over location typings. Unlike other approaches [Tof90, Har94, SmVo93], a \textit{location typing} here is a finite function mapping locations to \textit{type schemes}. This is the most novel aspect of the type system. The notational conventions for location typings are similar to those for identifier typings.

The \textit{generalization} of a type scheme \( \sigma \) relative to \( \lambda \) and \( \gamma \), written \( \text{Close}_\lambda,\gamma(\sigma) \), is the type scheme \( \forall \alpha . \sigma \), where \( \tilde{\alpha} \) is the set of all type variables occurring free in \( \sigma \) but not in \( \lambda \) or in \( \gamma \). We write \( \lambda \vdash e : \tau \) and \( \text{Close}_\lambda(\sigma) \) when \( \gamma = \emptyset \). A restricted form of generalization, written \( \text{AppClose}_\lambda,\gamma(\sigma) \), is defined to be the same as \( \text{Close}_\lambda,\gamma(\sigma) \) except that only strong type variables are generalized; any weak ones remain free. As in Tofte [Tof90], the \textit{generic instance} relation \( (\geq) \) of Damas and Milner [DaM82] is restricted by requiring universally quantified weak type variables to be instantiated only with weak types.

Finally, we write \( \lambda ; \gamma \vdash e : \sigma \iff \lambda ; \gamma \vdash e : \tau \) whenever \( \sigma \geq \tau \).

3 Semantics and Soundness

In this section, we establish type soundness using the framework of Harper [Har94]. First we give a structured operational semantics for the language. An expression is evaluated
\begin{align*}
\text{(var)} & \quad \lambda; \gamma \vdash x : \tau \quad \gamma(x) = \tau \\
\text{(ident)} & \quad \lambda; \gamma \vdash x : \tau \quad \gamma(x) \geq \tau \\
\text{(varloc)} & \quad \lambda; \gamma \vdash I : \tau \quad \lambda(I) \geq \tau \\
\text{(unit)} & \quad \lambda; \gamma \vdash \text{unit} : \text{unit} \\
\text{(-intro)} & \quad \lambda; \gamma \vdash [x : \pi_1] e : \pi_0 \quad \frac{}{\lambda; \gamma \vdash \lambda x. e : \pi_1 \to \pi_2} \\
\text{(-elim)} & \quad \lambda; \gamma \vdash e_1 : \pi_2, \lambda; \gamma \vdash e_2 : \pi_1 \\
\text{\quad \lambda; \gamma \vdash e_1 e_2 : \pi_2} \\
\text{(let-val)} & \quad \lambda; \gamma \vdash v : \pi_1, \lambda; \gamma \vdash [x : \text{Close}_{\lambda; \gamma}(\pi_1)] e : \pi_2 \\
\text{\quad \frac{}{\lambda; \gamma \vdash \text{let } x = v \text{ in } e : \pi_2}} \\
\text{(let-ord)} & \quad \lambda; \gamma \vdash e_1 : \tau_1, \lambda; \gamma \vdash [x : \text{AppClose}_{\lambda; \gamma}(\pi_1)] e_2 : \tau_2 \\
\text{\quad \frac{}{\lambda; \gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2}} \\
\text{(letvar)} & \quad \lambda; \gamma \vdash e_1 : \tau_1, \lambda; \gamma \vdash [x : \tau_1 \varnothing] e_2 : \tau_2 \\
\text{\quad If } x \text{ is assigned to in a } \lambda \text{-abstraction in } e_2 \text{ then } \tau_1 \text{ is weak.} \\
\text{\quad \frac{}{\lambda; \gamma \vdash \text{letvar } x := e_1 \text{ in } e_2 : \tau_2}} \\
\text{(r-val)} & \quad \lambda; \gamma \vdash e : \tau \quad \frac{}{\lambda; \gamma \vdash e : \tau} \\
\text{(assign)} & \quad \lambda; \gamma \vdash e_1 : \tau \quad \lambda; \gamma \vdash e_2 : \tau \\
\text{\quad \frac{}{\lambda; \gamma \vdash e_1 := e_2 : \text{unit}}} \\
\end{align*}

Figure 1: Rules of the Type System
(VAL) \[ \mu \vdash v \Rightarrow v, \mu \]

(DEREF) \[ \mu \vdash l \Rightarrow \mu(l), \mu \]

(APPLY) \[ \begin{align*} 
&\mu \vdash e_1 \Rightarrow \lambda x. e_1', \mu_1 \\
&\mu_1 \vdash e_2 \Rightarrow v_2, \mu_2 \\
&\mu_2 \vdash [v_2/x]e_1' \Rightarrow v, \mu' \\
&\mu \vdash e_1 \; e_2 \Rightarrow v, \mu' 
\end{align*} \]

(UPDATE) \[ \begin{align*} 
&\mu \vdash e \Rightarrow v, \mu' \\
&\mu \vdash l := e \Rightarrow \text{unit}, \mu'[l := v] 
\end{align*} \]

(BIND) \[ \begin{align*} 
&\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\
&\mu_1 \vdash [v_1/x]e_2 \Rightarrow v_2, \mu_2 \\
&\mu \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2, \mu_2 
\end{align*} \]

(BINDVAR) \[ \begin{align*} 
&\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\
&l \notin \text{dom}(\mu_1) \\
&\mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2 \\
&\mu \vdash \text{letvar } x := e_1 \text{ in } e_2 \Rightarrow v_2, \mu_2 
\end{align*} \]

Figure 2: The Evaluation Rules

relative to a memory \( \mu \), which is a finite function from locations to values. The contents of a location \( l \in \text{dom}(\mu) \) is the value \( \mu(l) \), and we write \( \mu[l := v] \) for the memory that assigns value \( v \) to location \( l \), and value \( \mu(l') \) to a location \( l' \neq l \). Note that \( \mu[l := v] \) is an update of \( \mu \) if \( l \in \text{dom}(\mu) \) and an extension of \( \mu \) if \( l \notin \text{dom}(\mu) \). The range of \( \mu \) is the set of all values \( \mu(l) \), for \( l \in \text{dom}(\mu) \).

Our evaluation rules are given in Figure 2. They allow us to derive judgements of the form

\[ \mu \vdash e \Rightarrow v, \mu' \]

which is intended to assert that evaluating closed expression \( e \) in memory \( \mu \) results in value \( v \) and new memory \( \mu' \). We write \([e'/x]e\) to denote the capture-avoiding substitution of \( e' \) for all free occurrences of \( x \) in \( e \). The use of substitution in the rules allows us to avoid environments and closures in the semantics, so that the result of evaluating an expression is just another expression.

The basic idea behind showing soundness is to show that if \( \vdash e : \tau \) and \( \vdash e \Rightarrow v, \mu' \), then \( \vdash v : \tau \), a property called subject reduction. But since \( e \) can allocate locations and since these locations can occur in \( v \), the conclusion must actually be that there exists a location typing \( \lambda' \) such that \( \lambda' \vdash v : \tau \) and such that \( \mu' : \lambda' \). The latter condition asserts that \( \lambda' \) is consistent with \( \mu' \). More precisely we say that \( \mu : \lambda \) iff \( \text{dom}(\mu) = \text{dom}(\lambda) \) and for every \( l \in \text{dom}(\mu) \), \( \lambda \vdash \mu(l) : \lambda(l) \).

It is the location typing \( \lambda' \) that makes soundness delicate. As observed by Tofte [Tof90], we may generalize a type variable \( \alpha \) in typing \( \vdash e : \tau \), only to find that \( \alpha \) occurs free in \( \lambda' \), and therefore cannot be generalized in typing \( \lambda' \vdash e : \tau \). For example, we can
define list reversal as follows:

\[
\text{let } r := \lambda x. x \text{ in } \smallskip
\begin{align*}
  r &:= \lambda x. \text{if } x = [] \text{ then } [] \text{ else } (r (\text{tl } x)) @ [h d x]; \\
  r &\text{ end}
\end{align*}
\]

This expression has type \(\forall a. \text{a list} \rightarrow \text{a list}\) in our type system. But when the expression is evaluated, a location \(l\) of type \(\text{a list} \rightarrow \text{a list}\) is allocated for \(r\) and \(l\) appears in the resulting value as well as in the domain of the resulting location typing \(\lambda'.\)

The solution proposed here is to use the quantified type \(\forall a. \text{a list} \rightarrow \text{a list}\) for \(l\) in \(\lambda',\) thereby eliminating the free occurrence of \(a.\) Of course, it is not always reasonable to give a location a quantified type. For example, if \(\lambda(l) = \forall a. a \rightarrow a,\) then the program \(l := \text{not}\; l;\) can be given type \(\text{int} \rightarrow \text{int},\) yet it evaluates to \(\text{not}\) of type \(\text{bool} \rightarrow \text{bool}.\) Our subject reduction theorem allows only read-only locations to be given quantified types.

We now turn to the soundness proof. First we introduce the relevant lemmas.

**Lemma 3.1 (Superfutiousness)** Suppose that \(\lambda;\gamma \vdash e : \tau.\) If \(l \not\in \text{dom}(\lambda),\) then \(\lambda[l : \sigma]\;\gamma \vdash e : \tau\) and if \(x \not\in \text{dom}(\gamma),\) then \(\lambda[l, x : \sigma]\;\gamma \vdash e : \tau.\)

**Lemma 3.2 (Substitution)** If \(\lambda;\gamma \vdash v : \sigma\) and \(\lambda;\gamma[x : \sigma] \vdash e : \tau,\) then \(\lambda;\gamma \vdash [v/x]e : \tau.\) Also, if \(\lambda;\gamma \vdash l : \tau\;\text{var}\) and \(\lambda;\gamma[x : \tau\;\text{var}] \vdash e : \tau',\) then \(\lambda;\gamma \vdash [l/x]e : \tau'.\)

The preceding two lemmas are straightforward variants of the lemmas given in [Har94]. We also need two new lemmas:

**Lemma 3.3 (Strengthening)** If \(\lambda[l : \sigma] \vdash e : \sigma\) and \(\sigma_2 \supseteq \sigma_1\) then \(\lambda[l : \sigma_2] \vdash e : \sigma.\)

**Lemma 3.4 (\(\forall\)-intro)** If \(\lambda \vdash e : \sigma\) and \(a\) does not occur free in \(\lambda,\) then \(\lambda \vdash e : \forall a : \sigma.\)

Finally, we note that in spite of \((\text{r-val})\) our typing rules are essentially "syntax directed":

**Lemma 3.5 ((r-val)-scope)** If the derivation of \(\lambda;\gamma \vdash e : \tau\) ends with \((\text{r-val}),\) then \(e\) is an identifier or a location.

**Proof.** If the derivation ends with \((\text{r-val}),\) then there must be a derivation of the hypothesis \(\lambda;\gamma \vdash e : \tau\;\text{var}.\) But to show that an expression has a type of the form \(\tau\;\text{var},\) there are only two possible rules that can be used: \((\text{var})\) and \((\text{varloc}).\) (The other rules all give data types to expressions.) So \(e\) must either be an identifier, in the case of \((\text{var}),\) or a location, in the case of \((\text{varloc}).\)  

We now give the soundness theorem:

**Theorem 3.6 (Subject Reduction)** Suppose

(a) \(\mu \vdash e \Rightarrow v, \mu',\)

(b) \(\lambda \vdash e : \tau,\)

(c) \(\mu : \lambda,\) and

(d) if a location \(l\) is assigned to in \(e,\) then \(\lambda(l)\) is unquantified; also, if \(l\) is assigned to

in the range of \(\mu\) or in a \(\lambda\)-abstraction in \(e,\) then \(\lambda(l)\) is weak.
Then there exists \( \lambda' \) such that
\[(e) \quad \lambda \subseteq \lambda', \]
\[(f) \quad \mu' : \lambda', \]
\[(g) \quad \lambda' \vdash v : \tau, \]
\[(h) \quad \text{any strong type variable free in } \lambda' \text{ is free in } \lambda, \text{ and} \]
\[(i) \quad \text{if a location } l' \text{ is assigned to in } v \text{ or in the range of } \mu', \text{ then } \lambda(l') \text{ is weak.} \]

Proof. The proof is by induction on the structure of the derivation of \( \mu \vdash e \Rightarrow v, \mu' \). Due to space limitations, we present only the most interesting cases, \((\text{update})\) and \((\text{bind var})\). We remark that property \((h)\) above makes the \((\text{bind})\) case routine.

\((\text{update})\). The evaluation must end with
\[
\mu \vdash e \Rightarrow v, \mu'
\]
and, by Lemma 3.5, the typing must end with
\[
\lambda \vdash l : \tau \text{ var}, \quad \lambda \vdash e : \tau
\]
Also, \( \mu : \lambda, \lambda(l) \) is unquantified, and if a location \( l' \) is assigned to in \( e \), then \( \lambda(l') \) is unquantified. And if \( l' \) is assigned to in the range of \( \mu \) or in a \( \lambda \)-abstraction in \( e \), then \( \lambda(l') \) is weak. By induction, there exists \( \lambda' \) such that
\[(e) \quad \lambda \subseteq \lambda', \]
\[(f) \quad \mu' : \lambda', \]
\[(g) \quad \lambda' \vdash v : \tau, \]
\[(h) \quad \text{any strong type variable free in } \lambda' \text{ is free in } \lambda, \text{ and} \]
\[(i) \quad \text{if a location } l' \text{ is assigned to in } v \text{ or in the range of } \mu', \text{ then } \lambda(l') \text{ is weak.} \]

Now we must show
\[(f) \quad \mu'[l := v] : \lambda', \]
\[(g) \quad \lambda' \vdash \text{unit} : \text{unit}, \]
\[(i) \quad \text{if a location } l' \text{ is assigned to in } \text{unit} \text{ or in the range of } \mu'[l := v], \]
then \( \lambda'(l') \) is weak.

\((g)\) follows immediately from typing rule \((\text{unit})\). \((i)\) follows by induction, since if a location \( l' \) is assigned to in the range of \( \mu'[l := v] \) then it is assigned to in \( v \) or in the range of \( \mu' \). Finally, we consider \((f)\), the most interesting case. For every \( l' \in \text{dom}(\mu') \) and \( l' \neq l \), we have
\[
X' \vdash \mu'[l := v](l') : X'(l')
\]
by induction. Since \( \lambda \vdash l : \tau \text{ var}, \lambda(l) \geq \tau \). But since \( \lambda(l) \) is unquantified, \( \lambda(l) = \tau \) and therefore \( \lambda'(l) = \tau \) since \( \lambda \subseteq \lambda' \). Since, by induction, \( X \vdash v : \tau \), we have
\[
X' \vdash \mu'[l := v](l) : X'(l)
\]
Thus we have \( \mu'[l := v] : \lambda' \). This completes \((\text{update})\).
Notice the role of condition (d) in proving $\lambda \vdash \mu'[l := v][i]: X(l)$ above. Since $i$ is assigned to in $l := e$, $\lambda(l)$ must be unquantified and consequently has only one generic instance, namely $\tau$. Therefore, $\lambda \vdash \mu'[l := v][i]: X(l)$ follows directly from $\lambda \vdash v : \tau$ of the induction. If $\lambda(l)$ were quantified, then it would not be possible to show $\lambda \vdash v : X(l)$. For example, if $\lambda(l) = \forall a . a \rightarrow a$, then on the program $l := \text{not}$ we would have to show that not has type $\forall a . a \rightarrow a$.

($\text{bindvar}$). The evaluation must end with

$$
\mu \vdash e_1 \Rightarrow v_1, \mu_1
$$

$$
l \notin \text{dom}(\mu_1)
$$

$$
\mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2
$$

and, by Lemma 3.5, the typing must end with

$$
\lambda \vdash e_1 : \tau_1
$$

$$
\lambda_1[x : \tau_1 \text{var}] \vdash e_2 : \tau_2
$$

If $x$ is assigned to in a $\lambda$-abstraction in $e_2$ then $\tau_1$ is weak.

$$
\lambda \vdash \text{letvar } x := e_1 \text{ in } e_2 \Rightarrow e_2, \mu_2
$$

Also, $\mu : \lambda$ and if a location $l'$ is assigned to in $e_1$ or in $e_2$, then $\lambda(l')$ is unquantified. And if $l'$ is assigned to in the range of $\mu$ or in a $\lambda$-abstraction in $e_1$ or in $e_2$, then $\lambda(l')$ is weak. By induction, there exists $\lambda_1$ such that

(c) $\lambda \subseteq \lambda_1$,

(f) $\mu_1 : \lambda_1$,

(g) $\lambda_1 \vdash v_1 : \tau_1$,

(h) any strong type variable free in $\lambda_1$ is free in $\lambda$, and

(i) if a location $l'$ is assigned to in $v_1$ or in the range of $\mu_1$, then $\lambda_1(l')$ is weak.

Since $l \notin \text{dom}(\lambda_1)$, $\lambda_1 \subseteq \lambda_1[l : \tau_1]$. Now, since $\lambda_1[l : \tau_1] \vdash l : \tau_1 \text{ var}$ and, by Lemma 3.1, $\lambda_1[l : \tau_1]; [x : \tau_1 \text{ var}] \vdash e_2 : \tau_2$, we can apply Lemma 3.2 to get

(b) $\lambda_1[l : \tau_1] \vdash [l/x]e_2 : \tau_2$

We also have, by Lemma 3.1,

(c) $\mu_1[l := v_1] : \lambda_1[l : \tau_1]$

Next, if a location $l'$ is assigned to in $[l/x]e_2$, then either $l'$ is assigned to in $e_2$ or $l' = l$. In the first case we have that $\lambda(l')$ is unquantified by hypothesis, and so $\lambda_1[l : \tau_1](l')$ is unquantified. In the second case we have $\lambda_1[l : \tau_1](l) = \tau_1$, which is unquantified. Also, if $l'$ is assigned to in the range of $\mu_1[l := v_1]$, then $l'$ is assigned to in $v_1$ or in the range of $\mu_1$, so by induction $\lambda_1(l')$ is weak, and hence $\lambda_1[l : \tau_1](l')$ is weak, since $\lambda_1 \subseteq \lambda_1[l : \tau_1]$. Finally, if $l'$ is assigned to in a $\lambda$-abstraction in $[l/x]e_2$, then either $l'$ is assigned to in a $\lambda$-abstraction in $e_2$ or $l' = l$ and $x$ is assigned to in a $\lambda$-abstraction in $e_2$. In the first case, $\lambda(l')$ is weak by hypothesis, and so $\lambda_1[l : \tau_1](l')$ is weak. In the second case, we have $\tau_1$ is weak by the restriction on the ($\text{letvar}$) rule, and so $\lambda_1[l : \tau_1](l')$ is weak. Therefore, we have

7
(d) if a location \( l' \) is assigned to in \([l/x]e_2\), then \( \lambda_1[l : \tau_1](l') \) is unquantified; also, if \( l' \) is assigned to in the range of \( \mu_1[l := v_1] \) or in a \( \lambda \)-abstraction in \([l/x]e_2\), then \( \lambda_1[l : \tau_1](l') \) is weak.

So by a second use of induction, there exists \( \lambda_2 \) such that

\[ \lambda_2[l : \tau_2] \subseteq \lambda_2, \]
\[ \mu_2 : \lambda_2, \]
\[ \lambda_2 \vdash v_2 : \tau_2, \]
\[ \text{any strong type variable free in } \lambda_2 \text{ is free in } \lambda_1[l : \tau_1], \]
\[ \text{if a location } l' \text{ is assigned to in } v_2 \text{ or in the range of } \mu_2, \text{ then } \lambda_2(l') \text{ is weak.} \]

At this point, \( \lambda_2 \) may contain free strong type variables that are not free in \( \lambda \), namely those of \( \tau_1 \). So we cannot take \( \lambda_2 \) as our final location typing. Instead, define \( \lambda' \) by

\[ \lambda'(l') = \text{AppClose}_\lambda(\lambda_2(l')), \]

for all \( l' \in \text{dom}(\lambda_2) \). Now we must establish

\[ \lambda \subseteq \lambda', \]
\[ \mu_2 : \lambda', \]
\[ \lambda' \vdash v_2 : \tau_2, \]
\[ \text{any strong type variable free in } \lambda' \text{ is free in } \lambda, \]
\[ \text{if a location } l' \text{ is assigned to in } v_2 \text{ or in the range of } \mu_2, \text{ then } \lambda'(l') \text{ is weak.} \]

To show (e), note that for any \( l' \in \text{dom}(\lambda) \), \( \lambda'(l') = \lambda_2(l') \), by the definition of \( \lambda' \). Since \( \lambda \subseteq \lambda_2 \), it follows that \( \lambda \subseteq \lambda' \).

Next we show (f). Since \( \mu_2 : \lambda_2 \), we have

\[ \lambda_2 \vdash \mu_2(l') : \lambda_2(l') \]

for any \( l' \in \text{dom}(\mu_2) \). Since \( \text{AppClose}_\lambda(\sigma) \geq \sigma \) for every \( \sigma \), by applying Lemma 3.3 repeatedly we get

\[ \lambda' \vdash \mu_2(l') : \lambda_2(l') \]

Finally, from Lemma 3.4 we get

\[ \lambda' \vdash \mu_2(l') : \text{AppClose}_\lambda(\lambda_2(l')), \]

since any type variables thereby quantified do not occur free in \( \lambda' \). Hence \( \mu_2 : \lambda' \).

To get (g), apply Lemma 3.3 to \( \lambda_2 \vdash v_2 : \tau_2 \). And (h) follows immediately from the definition of \( \lambda' \). Finally, for (i) suppose that \( l' \) is assigned to in \( v_2 \) or in the range of \( \mu_2 \). By the second use of induction, \( \lambda_2(l') \) is weak. Hence \( \lambda'(l') \) is weak, since \( \text{AppClose} \) quantifies only strong type variables. This completes (hndvar). \( \square \)

**Corollary 3.7** Restriction 2(i)(b) of LCF ML [GMW78], requiring a variable to have a monotype if the variable is assigned to in a \( \lambda \)-abstraction within its scope, is sound.

**Proof.** A monotype is a type with no type variables, so every such type is weak. So by Theorem 3.6, the LCF ML restriction is sound. \( \square \)
References


[SmVo95] Smith, G. and Volpano, D., Polymorphic Typing of Variables and References, to appear in *ACM Trans on Programming Languages and Systems*, 1996.