THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

PROGRAMMING WITH PRIVATE STATE

KAMAL ABOUL-HOSN

FALL 2001

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Approved: _____ Date: _____
Dr. John Hannan
Thesis Supervisor/Honors Adviser

_____ _____
Dr. Dale Miller
Second Reader

Abstract

In a purely-functional language, reasoning about program equivalence is rather straightforward. However, the inclusion of states and references can complicate such reasoning. The problem is that expressions have the ability to alter the state, and therefore depend on the state for their values. Consequently, we must resort to a notion of contextual equivalence, where equivalence depends on two expressions having the same value no matter what the complete program in which they appear. We consider privatizing state to functions by adding a new function declaration to ML. If the use of references in the state is limited, proving equivalence may not need to go as far as contextual equivalence. We present the operational semantics and type system for the language, as well as a type soundness proof. Finally, we present some examples in the language, as well as its encoding in the Elf programming language.

Contents

List of Figures

1 Introduction

Functional programming offers us many advantages when attempting to reason about program properties. However, purely-functional programming languages are somewhat limited in their capabilities. One feature that can expand the capability of a functional programming language is state. Such an addition, however, complicates some properties that makes functional programming so appealing, namely the ability to reason about programs easily.

For instance, consider the desire to determine if two pieces of programming code are equivalent. When references and state are allowed in the language, equivalence is difficult to reason about. Pitts used a notion of "contextual equivalence," whereby two expressions are contextually equivalent if the first expression can be replaced by the second in a complete program without affecting the observable results [8]. Even this definition makes determining equivalence in an obvious way difficult.

Take for example the following pieces of Caml code from Pitts' paper [8].

```
p  ≡  let a = ref 0 in
      fun(x : int) -> (a := !a + x ; !a)

m  ≡  let b = ref 0 in
      fun(y : int) -> (b := !b - y ; 0 - !b)
```

As Pitts argues, one can determine that these two pieces of code are contextually equivalent, since an invariant is maintained between the values in the locations a and b: a = -b. At the same time, the following pieces of code are not equivalent, although it is difficult to see from just looking at the expressions.

```
f  ≡  let a = ref 0 in
      let b = ref 0 in
      fun(x : intref) -> (if x == a then b else a)
```

```
g  ≡  let c = ref 0 in
      let d = ref 0 in
      fun(y : intref) -> (if y == d then d else c)
```

Upon a cursory examination, one may claim these two expressions are equivalent. Consider the following expression, used as a context for f and g.

```
t  ≡  fun (h : int ref -> int ref) ->
      let z = ref 0 in h (h z) == h z
```

Notice that t f evaluates to false while t g evaluates to true. Hence, these two functions are not contextually equivalent.

The major reason for the difficulty in reasoning about expression equivalence with references is the ability of expressions to change the state, which maps locations to values. With global references, the equivalence cannot be considered without also looking at the state. By placing restrictions on the use of references, it may be possible to reason about expression equivalence without using contextual equivalence and the entire reference state.

In many practical applications of state, references are inherently used on a restricted basis. For example, consider a simple function intended to maintain and increment a counter:

```
count  =  let c = ref 0 in
          fun() -> c := !c + 1; !c
```

If the cell c is only to be accessed via the function count(), then the reference is local to the function. Nonetheless, we have no guarantee in the operational semantics via a simple ref declaration that a reference is local to a certain piece of code. If we could declare the location c so that its locality to the function were explicitly given, perhaps reasoning about a program that included this function would be easier.

It is with this notion in mind that we consider *private state* with *static references*. In essence, the idea is to limit the use of references to particular blocks of code, i.e., function bodies. Functions may have associated with them a private state, containing references to which only a single function or group of functions has access. By giving functions private state, we hope to maintain the ease with which programs can be reasoned about while at the same time adding references to functional language.

2 ML with Static References and Private State

We want a reference that can be associated with an expression's private state, yet not deallocated immediately upon use. This notion fits exactly into the realm of static variables. In our case, we can create a static reference, which belongs to the private state of an expression, yet may be maintained beyond a single execution of the body of a function. Our notion of a static reference is similar to the use of a static reference in C, since the private states containing these references are contained within a global environment.

However, the references themselves are not globally accessible in the state. The references are only accessible when the function to which the references belong has been called. Once the function call has completed, the references in its private state must no longer be accessible. With this addition, functions need to keep track of all the references in their private state. We can accomplish this via function closures stored in an environment that include the private reference state.

2.1 Expressions

The language serving as the basis for our analysis is a λ-based calculus extended to include the state operations reference, dereference, and update. We also expand the language to include expressions to declare functions with static references.

$$\begin{aligned} e \;::=\; & c \mid x \mid \text{if } e \text{ then } e \text{ else } e \mid \lambda x.e \mid e \mathbin{@} e \\ & \mid f \mid \mathsf{letfun}\ \overline{f = e}\ \mathsf{with}\ \overline{x = \mathsf{sref}\ e}\ \mathsf{in}\ e \\ & \mid () \mid \ell \mid\ !e \mid e := e \end{aligned}$$

We use the shorthand $\mathsf{let}\ x\ \mathsf{be}\ e_1\ \mathsf{in}\ e_2$, to represent $(\lambda x.e_2) \mathbin{@} e_1$. The state operators include a reference access (ℓ), dereference ($!e$), and update ($x := e$) are expressions as well. The ℓ is the actual reference cell as opposed to being a vari-

able that points to a location, which is mapped to a value.

The most interesting of the expressions is the one that declares functions with static references, $\mathsf{letfun}\ \overline{f = e}\ \mathsf{with}\ \overline{x = \mathsf{sref}\ e}\ \mathsf{in}\ e$. This expression declares multiple functions $\overline{f = e}$, which can use static references declared as $\mathsf{with}\ \overline{x = \mathsf{sref}\ e}$. These functions can in turn be used in the body. However, the body itself does not have access to the statically-declared references; these are exclusively to be used by the declared functions.

2.2 Operational Semantics

The judgment for the language's operational semantics is of the form

$$\rho; \sigma; e \hookrightarrow \rho'; \sigma'; v$$

The statement can be read as "in the environment ρ with the state σ, the expression e evaluates to the value v with the new environment ρ' and the new state σ'." The language's operational semantics uses both substitutions and environments, with only functions with private state using the latter. The motivation for this is that only one copy of a static reference should exist, either in a function closure in the environment or in the state σ. If functions use substitution semantics, then a copy of the private state would be made for each expression into which the function is substituted. On the other hand, regular variables and functions need not have this constraint; not all instances of them need to be the identical copy.

The environment contains the function closures for the functions with private state. Multiple functions can appear in a single closure, all associated with the same state. Closures are defined as follows:

$$cl \quad ::= \quad \langle \overline{(f_i, e_i)}, \sigma \rangle$$

It is interesting to note that closures do not contain the environment. Instead, the environment is global, and all expressions use the global environment. The reason for a global environment is similar to the reason for using environments at all. If each function closure contains an environment, it is possible that the state with static references will be copied and two expressions will call the same function, each of which accesses a different static reference. If this happens, then the reference is private to each instance of the function, not to the function itself. Only one copy of a static reference should exist at any given time.

We maintain the singular existence of static references through two functions for environments–one to lookup items and one to update items–shown in Figure 1. The lookup function takes an environment and a function name and returns its body, its private state, and a new environment. When the function retrieves the private state, it replaces the private state in the closure with the empty state. This ensures that only one copy of the static references in the state exists. The update function takes an environment, a function, and a new state (denoted as (f, σ'_f) and returns environment with the private state put back in the closure, denoted as $\rho\langle(f, e), \sigma'_f\rangle$.

$$\frac{((f,e),\sigma_f)) \in \rho \quad \rho' = \rho((f,e),\{\})}{\rho(f) = (\sigma_f, e, \rho')} \text{ Lookup}$$

$$\frac{((f,e),\sigma_f) \in \rho}{\rho(f,\sigma'_f) = \rho\langle(f,e),\sigma'_f\rangle} \text{ Update}$$

Figure 1: Environment Functions

Since we have references, it is necessary to maintain a state, which maps locations to base values. In our case, a state is present only in association with a function that uses static references. The collection of these states in the operational

semantics is σ.

The question of how to represent σ and manage the private states is important. We have two options available, each with their own advantages. The first option considers σ to be a stack of states, or more simply, a stack of reference bindings. Whenever a function containing a private state is called, the references from the private state are retrieved from the environment and placed on the top of the stack, if they are not already there. If they are already in the state, then an earlier function call must have put them there, and our lookup function to retrieve the state returns the empty state. When the function's execution is complete, the state–which has possibly been changed–is popped off of the stack and written back to the environment.

The second representation attempts to manage the collection states and recognize the scope of static references. Consider a function f, which has in its body a call to another function g. If the static references declared for use with f are not accessed by g, then the private state belonging to f need not be in the collection of states during the call to g. Hence, its private state would be removed from the collection of states for the duration of executing g. Once g has completed execution, f's private state would be added back to the collection of states.

The first representation has the advantage of simplicity. Every call to a function with private state is the same: look up the function and its private state in the environment, put the static references in the state, execute the function, remove the references from the state, and write the private state back to the environment. Even if a function's static references are already in the state, the procedure is the same, since it will simply add and remove no references to the state. Reasoning about the correctness of such an implementation requires fewer cases than the second representation.

The second representation is obviously more complex and therefore harder to reason about. However, it more exactly implements our notion of private state. If a function's private state is not to be accessed by a piece of code, it *cannot* be accessed, because it is removed from the collection of states. This representation might be more desirable in a security setting, where guarantees about the privacy of state may be necessary.

For this thesis, we use a stack structure to represent σ. Static references that need added to the state will be pushed onto the top of the state while a function with the private state is executed. When the function call is complete, the function's references are popped from the state.

In the same way we needed a lookup function and update function for the environment, we need these functions for the state to work with references. The lookup function is trivial, since it need not change the contents of the state. However, the update function must find the private state which contains the reference and then update it. The update function never adds a location mapping to the state, it only changes the mappings that are already in the state. The update function takes as arguments the reference and the new value (denoted as (ℓ, v')) and returns a new state with the value of ℓ changed, denoted as $\sigma\langle \ell \mapsto v' \rangle$. The reference lookup and update functions are in Figure 2.

$$\frac{(\ell \mapsto v) \in \sigma}{\sigma(\ell) = v} \text{ Lookup}$$

$$\frac{(\ell \mapsto v) \in \sigma_f}{\sigma(\ell, v') = \sigma\langle \ell \mapsto v' \rangle} \text{ Update}$$

Figure 2: State Functions

Expressions have the ability to alter the environment and the state. In an expression, if a subexpression calls a function that has static references, then its private state may be updated in the environment so that other subexpressions will have access to the most recent value of the references. The significance of this fact will be demonstrated in examples in Section 4.1.

The operational semantics for basic expressions is given in Figure 3. Note that in the rule (f), a function symbol representing a function with static references evaluates to itself. In order to ensure that a call to a function with static references uses the most recent version of the private state, a function name must evaluate to itself instead of its closure. Only after all other subexpressions have been evaluated should the private state be retrieved and used. Otherwise, another subexpression could have called the function and altered the private state, leaving the original call with an outdated copy of the function's private state. Moreover, evaluating a function f to itself does not change the state or the environment.

$$\frac{}{\rho;\sigma;c \hookrightarrow \rho;\sigma;c}\ \text{(const)}$$

$$\frac{\rho;\sigma;e_1 \hookrightarrow \rho'';\sigma'';\mathsf{true} \quad \rho'';\sigma'';e_2 \hookrightarrow \rho';\sigma';v}{\rho;\sigma;\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \rho;\sigma';v}\ \text{(if\_true)}$$

$$\frac{\rho;\sigma;e_1 \hookrightarrow \rho'';\sigma'';\mathsf{false} \quad \rho'';\sigma'';e_3 \hookrightarrow \rho';\sigma';v}{\rho;\sigma;\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \rho;\sigma';v}\ \text{(if\_false)}$$

$$\frac{}{\rho;\sigma;\lambda x.e \hookrightarrow \rho;\sigma;\lambda x.e}\ \text{(lam)}$$

$$\frac{}{\rho;\sigma;f \hookrightarrow \rho;\sigma;f}\ (f)$$

Figure 3: Basic Expression Rules

The next group of operational semantics are those that deal with references, given in Figure 4. Reference cells evaluate to themselves in the *ref* rule, since we

use substitution for everything except functions with static references. Dereferencing and updating a reference are handled by they (deref) rule and (upref) rule, respectively.

The *lets* rule is the one that declares functions with private state. Multiple functions can be declared to have access to the same state. In the environment, these functions are all grouped together with their bodies and associated with the same state, rather than making a copy of the state for each function. We have an entry in the environment for every *group* of functions declared.

First, all of the static references are evaluated. The body is then evaluated, with an environment extended to include the functions declared and their private state. We include the functions in the environment in order to allow mutual recursion. One need not necessarily declare any static references, either. When we allow no references to be declared, we can create a regular recursive function without any private state with the letfun expression by declaring only one function. Consequently, our letfun expression inherently handles recursion, making a separate rule unnecessary.

The g_is are new names for the functions. The reason we need to replace the f_is with these names is to make sure that multiple declarations of a private state are unique. If they do not and two sets of functions have the same name, we may not know which private state to retrieve from the environment when one of the functions is called. These new names are substituted into the body of the letfun expression. While executing, a function is accessed via the unique identifier for the group of functions declared and i, indicating the index of the function in the declaration list.

We also substitute for the reference variables in the functions, putting the unique reference names, ℓ_is, in for the x_is. If we do not use these new names, then

a function that accesses static references from more than one private state (i.e., its own private state and also the state of a parent function) could have references with the same name, so we would not know which one to access. The references are accessed the same way as functions: with unique identifiers for the static references declared. However, we do not substitute these unique identifiers into the body of the letfun expression, since none of these references should be there in the first place.

$$\frac{}{\rho; \sigma; \ell \hookrightarrow \rho; \sigma; \ell} \text{ (ref)}$$

$$\frac{\rho; \sigma; e \hookrightarrow \rho'; \sigma'; \ell \quad \sigma'(\ell) = v}{\rho; \sigma; !e \hookrightarrow \rho'; \sigma'; v} \text{ (deref)}$$

$$\frac{\rho; \sigma; e_1 \hookrightarrow \ell; \rho''; \sigma'' \quad \rho''; \sigma''; e_2 \hookrightarrow v; \rho'; \sigma''' \quad \sigma'''(\ell, v) = \sigma'}{\rho; \sigma; e_1 := e_2 \hookrightarrow \rho'; \sigma'; ()} \text{ (upref)}$$

$$\frac{\overline{\rho; \sigma; a_j \hookrightarrow; \rho''; \sigma''; v_j} \quad \rho''\overline{\langle (g_i, e_i'[g_i/f_i][\ell_j/x_j]), \overline{(\ell_j, v_j)}\rangle}; \sigma''; e\overline{[g_i/f_i]} \hookrightarrow \rho'\overline{\langle (g_i, e_i'[g_i/f_i][\ell_j/x_j]), \overline{(\ell_j, v_j')}\rangle}; \sigma'; v}{\rho, \sigma; \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e \hookrightarrow \rho'; \sigma'; v} \text{ (lets)}$$

Figure 4: Reference Rules

$$\frac{\rho; \sigma; e_1 \hookrightarrow \rho'', \sigma''; \lambda x.e_1' \quad \rho''; \sigma''; e_2 \hookrightarrow \rho''', \sigma'''; v' \quad \rho'''; \sigma'''; e_1'[v'/x] \hookrightarrow \rho', \sigma'; v}{\rho; \sigma; e_1 \mathbin{@} e_2 \hookrightarrow \rho'; \sigma'; v} \text{ (ap1)}$$

$$\frac{\rho; \sigma; e_1 \hookrightarrow \rho''; \sigma''; f \quad \rho''; \sigma''; e_2 \hookrightarrow \rho'''; \sigma'''; v' \quad \rho'''(f) = (\sigma_f, e_i', \rho'''') \quad \rho''''; \sigma'''\sigma_f; e_i'[v'/x] \hookrightarrow \rho'; \sigma'\sigma_f'; v}{\rho; \sigma; e_1 \mathbin{@} e_2 \hookrightarrow \rho'(f, \sigma_f'); \sigma'; v} \text{ (ap2)}$$

Figure 5: Application Rules

The final group of operational semantics is the applications rules. Since some functions carry along with them a private state, we must have two application rules, given in Figure 5. The major difference between the application rules is the retrieval of the private state. The first rule, *ap1*, handles the application of a basic λ-abstraction to an argument. In this case, it is not necessary to retrieve any sort of state.

The rule *ap2* handles the application of a function with a private state to an expression. In this case, we must retrieve the private state and the body of the function from the environment. The private state, which is identified by the function's unique identifier created in the *lets* rule, is retrieved from the environment and its references are added to the state.

Our lookup function ensures that the most recent versions of the references are in the state. If the private state's references are not already in the state, the lookup function copies them over to the state and then replaces the private state in the environment with the empty state. However, if the private state was already removed from the environment, then the lookup function retrieves the empty state–which has no bearing on the state–and then replaces the empty private state with the empty private state.

Once the application has been evaluated, the private state is written back to the proper function closure in the environment. If the lookup function retrieved an empty state, it will write back an empty state. We know that the function's state will inevitably be written back to the environment, since the only way the lookup could have returned an empty state is if we are within a block of code that already retrieved that particular private state. Hence, once we leave that block, the private state will be written back to the environment.

It is important to mention that we do not need to have a separate application rule for when a function is applied to a function with static references. In this case, the application returns the function name that is the argument, for which we need not look up the state, only return the name of the function.

2.3 Type System

In typing expressions, we must be sure that references declared statically are truly static. References are not static if the reference itself is the return value of an expression or the reference is used outside the scope of the bodies of the functions declared in the same letfun expression. In order to determine which references are static, we must use a system that keeps track of the difference between expressions that contain and can return static references and those that cannot. As a basis, we start with an annotated type system for references [2]. However, an important difference exists between Hannan's system and ours; expressions that can return static references should not type at all.

At first, it may seem that keeping track of the static references themselves would be enough in typing the expressions. However, the possibility of aliasing exists–an expression could alias a reference location to another variable, thus making it globally accessible. Consider the following example:

```
letfun f    =    λx.let y=a in y
with a      =    sref 3
in          f @ 3
```

In this expression, the cell belonging to the static reference a escapes. Consequently, we must keep track not of the static variables themselves, but the cells they create and access.

The type system we define gives us information about the lifetime of the reference cells. The types are defined as

$$\phi \quad ::= \quad \iota \mid \phi \ \ \mathsf{ref} \ \ (\Delta) \mid \phi \xrightarrow{\Delta} \phi$$

in which ι ranges over the set of base types, **Constants**. In our system, Δ ranges over sets of static cells used to contain data for static references, as well as the names of functions with static references used in the type. Both reference types and function types contain these sets. In a reference, Δ refers to the static cells the reference itself can access. In a function, the Δ over the arrow ranges over the reference cells and functions that can be accessed during the execution of the function body.

In order to use these types, we must be able to determine which static reference cells and functions are possibly accessed during the evaluation of an expression. We can do so through a function, $LS(\phi)$, which returns the set of live static reference cells and live functions with static references within the type ϕ.

$$\begin{aligned} &LS(\iota) = \{\} \\ &LS(\phi \ \ \mathsf{ref} \ \ (\Delta)) = \Delta \bigcup LS(\phi) \\ &LS(\phi_1 \xrightarrow{\Delta} \phi_2) = \Delta \bigcup LS(\phi_1) \bigcup LS(\phi_2) \end{aligned}$$

Figure 6: $LS(\phi)$ function

The first rule is rather clear. The live static references for a reference is the set of cells, Δ, that the cell can possibly reference and the set of cells the type it is referencing can reference. For a function, the set of live static references is the set of live static references for its argument type and return type unioned with the set Δ over the arrow. The reason for including Δ is to keep track of the references

and functions used within one of these functions to ensure they do not escape.

With this function, we can test to make sure that no static references escape as one does in the example above. The important place to use this function is in the typing of a letfun declaration. We must ensure that the static references declared do not appear anywhere in the body of the expression. Additionally, we make sure that functions that use these static references do not escape.

Since we have a set associated with types, we may not have identical types in function application, i.e., the argument to a function may not have the same set of static reference cells and functions as the input type of the function. We must be sure that the set of the argument's static reference cells and functions is a subset of the set of cells for the function input. We ensure this property through the $\leq$ relation. We need only define this relation on reference types and function types, since they are the only two that contain a set Δ. The rules for the $\leq$ relation are shown in Figure 7.

$$\frac{}{\iota \leq \iota}$$

$$\frac{\phi_1 \leq \phi_2 \quad \Delta_1 \subseteq \Delta_2}{\phi_1 \ \mathsf{ref} \ (\Delta_1) \leq \phi_2 \ \mathsf{ref} \ (\Delta_2)}$$

$$\frac{\phi_1' \leq \phi_1 \quad \phi_2 \leq \phi_2' \quad \Delta \subseteq \Delta'}{\phi_1 \xrightarrow{\Delta} \phi_2 \leq \phi_1' \xrightarrow{\Delta'} \phi_2'}$$

Figure 7: $\leq$ Relation for Types

The fact that $\phi_1' \leq \phi_1$ in the function rule may seem counterintuitive. The explanation of why we must put it this way is left until after the discussion of typing a function application, where the $\leq$ relation will actually be used.

When we type expressions, we also want to define the $\leq$ relation for the type environment Γ. This definition is based on the $\leq$ relation for types:

Definition 2.1 $\Gamma' \leq \Gamma$ *iff* $dom(\Gamma) = (\Gamma')$*and*$\forall x \in dom(\Gamma)$, $\Gamma'(x) \leq \Gamma(x)$.

The type inference rules for basic expressions are listed in Figure 8. Note that when an f is typed, it is added to the set Δ, since the function is being accessed.

Unlike the operational semantics, we need not have separate application rules for regular λ-abstractions and functions with private state. While the evaluation of the two kinds of functions is different, their typing is the same; both are of some type $\phi_1 \xrightarrow{\Delta} \phi_2$.

With the application rule defined, we can understand why $\phi_1' \leq \phi_1$ in the λ-abstraction case of the $\leq$ relation. Consider the following example:

$$(\lambda \mathsf{f}.\mathsf{f} \ @ \ \mathsf{a}) \ @ \ \mathsf{g})$$

Assume f, a, and g to have the types $\phi_1' \xrightarrow{\Delta'} \phi_2'$, ϕ_1'', and $\phi_1 \xrightarrow{\Delta} \phi_2$, respectively. From the application rule, we know:

$$\phi_1'' \leq \phi_1' \quad (1)$$

$$\phi_1 \xrightarrow{\Delta} \phi_2 \leq \phi_1' \xrightarrow{\Delta'} \phi_2' \quad (2)$$

We also need to ensure that $\phi_1'' \leq \phi_1$. The only way we can make this assurance is if we know $\phi_1' \leq \phi_1$. Hence, our rule for a λ-abstraction needs this contravariance in the source of the function type in order for the $\leq$ relation to hold.

All of the typing rules related to static references are in Figure 9. When typing a location ℓ, the reference itself is returned as a singleton set for Δ.

In the letfun expression, we must first type the static references. For each reference, we generate a new static cell ℓ_j. Then we can type the function, with the assumption that the static references have type ϕ_j and the functions have the type ϕ_i, since we allow recursion. Finally, we can type the expression e. We must know that the static references do not have the ability to escape. Consequently,

$$\frac{}{\Gamma \rhd c : (\iota, \{\})}$$

$$\frac{\Gamma(x) = \phi}{\Gamma \rhd x : (\phi, \{\})}$$

$$\frac{\Gamma(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$

$$\frac{\Gamma\{x : \phi_1\} \rhd e : (\phi_2, \Delta)}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$

$$\frac{\Gamma \rhd e_1 : (\text{bool}, \Delta) \quad \Gamma \rhd e_2 : (\phi_2, \Delta_1) \quad \Gamma \rhd e_3 : (\phi_3, \Delta_2) \quad \phi_2 \leq \phi \quad \phi_3 \leq \phi}{\Gamma \rhd \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

$$\frac{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \quad \Gamma \rhd e_2 : (\phi_1', \Delta_2)}{\Gamma \rhd e_1 \mathbin{@} e_2 : (\phi_2, LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)} \; \phi_1' \leq \phi_1$$

Figure 8: Type Inference Rules for Basic Expressions

we check to make sure that the set of live static references in e's type ϕ contains none of the static cells associated with the static reference.

$$\frac{\Gamma(\ell) = \phi \text{ ref } (\Delta)}{\Gamma \rhd \ell : (\phi \text{ ref } (\Delta), \{\ell\})}$$

$$\frac{\Gamma \rhd e : (\phi \text{ ref } \Delta, \Delta')}{\Gamma \rhd !e : (\phi, \Delta \bigcup \Delta')}$$

$$\frac{\Gamma \rhd e_1 : (\phi \ \ \mathsf{ref} \ \ (\Delta), \Delta_1) \quad \Gamma \rhd e_2 : (\phi, \Delta_2)}{\Gamma \rhd e_1 := e_2 : (\text{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

$$\frac{\begin{array}{l}\Gamma \rhd \overline{a_j : (\phi_j, \Delta_j)} \\ \Gamma\{\overline{\ell_j : \phi_j \ \mathsf{ref}\ (\{\ell_j\} \cup \Delta)}, \overline{g_i : \phi_i}\} \rhd \overline{e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_k, \{\})} \\ \Gamma\{\overline{g_i : \phi_i}\} \rhd e[\overline{g_i/f_i}] : (\phi, \Delta)\end{array}}{\Gamma \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta \bigcup \overline{\Delta_j} \bigcup \overline{LS(\phi_j)} \bigcup \overline{LS(\phi_k)} - \{\overline{\ell_j}\} \bigcup \{\overline{g_i}\})}$$

$$\text{where} \quad \begin{array}{l} LS(\phi_k) \bigcap (\{\overline{\ell_j}\} \uplus \{\overline{g_i}\}) = \{\} \\ LS(\phi) \bigcap \{\overline{g_i}\} = \{\} \end{array}$$

Figure 9: Type Inference Rules for Reference Operations

3 Type Soundness

In this section, we provide the type soundness proof for ML with private state. However, we must first define typing a value, and also prove several auxiliary lemmas we need.

3.1 Types for Values Using Maximal Fixed Points

In order to prove type soundness, we must understand what it means for a value to have a type. Values can include base values, λ-abstractions, references, function symbols, and a function closure. The relation for typing values is of the form $\rho, \sigma \models v : \phi$. We can read this relation as "with respect to the environment ρ and the state σ, the value v has the type ϕ." This relation should have the following property

Property 3.1

$$
\begin{array}{lcl}
\rho, \sigma & \models & c : \phi, \ \textit{where } (c : \phi) \in \textit{Constants} \\
\rho, \sigma & \models & \lambda x.e : \phi \ \textit{iff there exist } \Gamma \textit{ and } \phi' \textit{ such that} \\
& & \quad dom(\Gamma) = dom(\rho) \biguplus dom(\sigma), \\
& & \quad \forall \ell \in dom(\sigma), \rho, \sigma \models \sigma(\ell) : \Gamma(\ell), \\
& & \quad \forall f \in dom(\rho), \rho, \sigma \models \rho(\ell) : \Gamma(\rho), \\
& & \quad \Gamma \rhd \lambda x.e : (\phi', \{\}) \textit{ is derivable, and } \phi' \leq \phi \\
\rho, \sigma & \models & \ell : \phi \ \mathsf{ref}\ (\Delta) \textit{ iff } \rho, \sigma \models \sigma(\ell) : \phi \textit{ and } \ell \in \Delta \\
\rho, \sigma & \models & f : \phi_1 \xrightarrow{\Delta} \phi_2 \textit{ iff } \rho(f) = (\lambda x.e, \sigma_e, \rho'), \textit{ and} \\
& & \quad \textit{there exist } \Gamma, \phi_1', \phi_2', \textit{ and } \Delta' \textit{ such that} \\
& & \quad dom(\Gamma) = dom(\rho) \biguplus dom(\sigma), \\
& & \quad \forall \ell \in dom(\sigma), \rho, \sigma \models \sigma(\ell) : \Gamma(\ell), \\
& & \quad \forall f \in dom(\rho), \rho, \sigma \models \rho(\ell) : \Gamma(\rho), \\
& & \quad \Gamma \rhd \lambda x.e : (\phi_1' \xrightarrow{\Delta'} \phi_2', \{\}) \textit{ is derivable,} \\
& & \quad \phi_1' \xrightarrow{\Delta'} \phi_2' \leq \phi_1 \xrightarrow{\Delta} \phi_2 \textit{ and} \\
& & \quad LS(\phi_2') \bigcap dom(\sigma_e) = \{\}
\end{array}
$$

This property does not define a unique relation $\models$, since it is not well-founded. The reason that it is not founded is that circularity can exist in the

locations when they are updated. Consider the following example

$$\begin{array}{rrl} \mathsf{letfun} & \mathsf{circ} = & \lambda \mathsf{x.a} := \mathsf{circ}\ ;\ 0 \\ \mathsf{with} & \mathsf{a} = & \mathsf{sref}\ \lambda \mathsf{x.0} \\ \mathsf{in} & \ldots & \end{array}$$

Notice that the reference a is updated with a value which includes the reference a. Such a circular definition would fail to meet our property. Consequently, we must use maximum fixed points to define our relation.

Instead of giving an inductive definition, we will constrain the relation using maximal fixed points. We have a function, $F : P(U) \to P(U)$, where $P(U)$ is the powerset of U and

$$U = Env \times Ste \times Val \times Type$$

where Env is the set of environments, Ste is the set of states, Val is the set of values $c, \lambda x.e, \ell, andf$, and $Type$ is the set of types.

Now we define the relation F:

Definition 3.1

$$\begin{aligned} F(Q) \;=\; & \{(\rho,\sigma,c,\iota) \mid (c:\phi) \in Constants\} \bigcup \\ & \{(\rho,\sigma,\lambda x.e,\phi) \mid \textit{there exists a } \Gamma \textit{ such that } dom(\Gamma) = dom(\rho) \biguplus dom(\sigma), \\ & \quad \forall \ell \in dom(\sigma), (\rho,\sigma,\sigma(\ell),\Gamma(\ell)) \in Q, \\ & \quad \forall f \in dom(\rho), (\rho,\sigma,\rho(\ell),\Gamma(\rho)) \in Q, \\ & \quad \Gamma \rhd \lambda x.e : \phi' \textit{ is derivable, and } \phi' \leq \phi\} \bigcup \\ & \{(\rho,\sigma,\ell,\phi\ \mathsf{ref}\ (\Delta)) \mid (\rho,\sigma,\sigma(\ell),\phi) \in Q \textit{ and } \ell \in \Delta\} \bigcup \\ & \{(\rho,\sigma,f,\phi_1 \xrightarrow{\Delta} \phi_2) \mid \rho(f) = (\lambda x.e,\sigma_e,\rho') \textit{ and} \\ & \quad \textit{there exists a } \Gamma \textit{ such that } dom(\Gamma) = dom(\rho) \biguplus dom(\sigma), \\ & \quad \forall \ell \in dom(\sigma), (\rho,\sigma,\sigma(\ell),\Gamma(\ell)) \in Q, \\ & \quad \forall f \in dom(\rho), (\rho,\sigma,\rho(\ell),\Gamma(\rho)) \in Q, \\ & \quad \Gamma \rhd \lambda x.e : (\phi_1' \xrightarrow{\Delta'} \phi_2', \{\}) \textit{ is derivable,} \\ & \quad \phi_1' \xrightarrow{\Delta'} \phi_2' \leq \phi_1 \xrightarrow{\Delta} \phi_2 \textit{ and} \\ & \quad \textit{and } LS(\phi_2') \bigcap dom(\sigma_e) = \{\}\} \end{aligned}$$

It is important that F be monotonic (i.e., that $Q \subseteq Q'$ implies $F(Q) \subseteq F(Q')$). The F we have chosen is in fact monotonic. Moreover, we know that $(\mathcal{P}(U), \subseteq)$ is a complete lattice. From these two facts, we can conclude that F has a maximum fixed point by the Knaster-Tarski Theorem.

$$Q^{max} = \bigcup\{Q \subseteq U | Q \subseteq F(Q)\}$$

We want to use the maximal fixed point because we want examples such as the one given above to be included in the language and typable.

Now we shall take our definition of $\models$ to be

$$\rho, \sigma \models v : \phi \text{ iff } (\rho, \sigma, v, \phi) \in Q^{max}$$

We also need a definition for a relation between the environment ρ, the state σ, and the type environment Γ.

Definition 3.2 *$\rho, \sigma : \Gamma$ iff:*

1. *$dom(\Gamma) = dom(\rho) \uplus dom(\sigma)$,*
2. *$\forall g \in dom(\rho)$, iff $\models \rho(g) : \Gamma(g)$, and*
3. *$\forall \ell \in dom(\sigma)$, iff $\models \sigma(\ell) : \Gamma(\ell)$*

3.2 Type Soundness Lemmas

Before we can prove type soundness, we must prove the following Propositions and Lemmas:

First of all, we want to know that looking up a function in the environment does not change the domain of the environment. Otherwise, the lookup function for environments is removing or adding static functions, which we do not want.

Proposition 3.1 *If $\rho(f) = (\sigma_f, e, \rho')$ then $dom(\rho) = dom(\rho')$.*

Proof: This Proposition follows directly from inspecting the Lookup function for environments in Figure 1.

Similarly to Proposition 3.1, we want to be sure that updating the state σ with a new value for a reference does not change the domain of the states.

Proposition 3.2 *If* $\sigma(\ell, v') = \sigma'$ *then* $dom(\sigma) = dom(\sigma')$.

Proof: This Proposition follows directly from inspecting the Update function for states in Figure 2.

Another important property we want to be sure holds is that anything that types Γ types in a larger environment $\Gamma\Gamma'$.

Proposition 3.3 *If* $\Gamma \rhd e : (\phi, \Delta)$ *then* $\forall \Gamma' where dom(\Gamma) \bigcap dom(\Gamma') = \{\}, \Gamma\Gamma' \rhd e : (\phi, \Delta)$.

Proof: We can prove this Proposition via induction on typing derivations Ξ.

1. Assume Ξ is $\overline{\Gamma \rhd c : (\phi, \{\})}$. This case is straightforward, since a constant can be typed in any Γ.

2. Assume Ξ is
$$\frac{\Gamma(x) = \phi}{\Gamma \rhd x : (\phi, \{\})}$$
This case is straightforward, since the domains of Γ and Γ' are disjoint.

3. Assume Ξ is
$$\frac{\Gamma(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$
We know that Γ' cannot redefine f, since all names for functions are unique. Hence, we know
$$\frac{\Gamma\Gamma'(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma\Gamma' \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$
which is what we wanted to show.

4. Assume Ξ is
$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma\{x : \phi_1\} \rhd e : (\phi_2, \Delta)\end{array}}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$
We know by induction on Ξ_1 that
$$\Xi_1' \quad :: \quad \Gamma\{x : \phi_1\}\Gamma' \rhd e : (\phi_2, \Delta) \tag{3}$$
From (3), we can conclude that
$$\Xi_2' \quad :: \quad \Gamma\Gamma' \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})$$
which is what we wanted to show.

5. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\text{bool}, \Delta)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi, \Delta_1)} \quad \overset{\Xi_3}{\Gamma \rhd e_3 : (\phi, \Delta_2)}}{\Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

We know via induction on Ξ_1, Ξ_2, and Ξ_3 that

$$\Xi_1' :: \Gamma\Gamma' \rhd e_1 : (\text{bool}, \Delta) \tag{4}$$
$$\Xi_2' :: \Gamma\Gamma' \rhd e_2 : (\phi_2, \Delta_1) \tag{5}$$
$$\Xi_3' :: \Gamma\Gamma' \rhd e_3 : (\phi_3, \Delta_2) \tag{6}$$

From (4), (5), and (6), we can construct a derivation of

$$\Xi_4' :: \Gamma\Gamma' \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)$$

which is what we wanted.

6. Assume Ξ is

$$\frac{\Gamma(\ell) = \phi\ \mathsf{ref}\ (\Delta)}{\Gamma \rhd \ell : (\phi\ \mathsf{ref}\ (\Delta), (\ell)}$$

We know that $\Gamma\Gamma' = \phi\ \mathsf{ref}\ (\Delta)$ because the names for locations are unique. Hence we know we have

$$\frac{\Gamma\Gamma'(\ell) = \phi\ \mathsf{ref}\ (\Delta)}{\Gamma\Gamma' \rhd \ell : (\phi\ \mathsf{ref}\ (\Delta), (\ell)}$$

which is what we needed for this case.

7. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta')}}{\Gamma \rhd !e : (\phi, \Delta \bigcup \Delta')}$$

We know via induction on Ξ_1 that

$$\Xi_1' :: \Gamma\Gamma' \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta') \tag{7}$$

We can use (7) to construct a derivation of

$$\Xi_2' :: \Gamma\Gamma' \rhd !e : (\phi, \Delta \bigcup \Delta')$$

which is what we wanted to show.

8. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi\ \ \mathsf{ref}\ \ (\Delta), \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi, \Delta_2)}}{\Gamma \rhd e_1 := e_2 : (\mathsf{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

We know via induction on Ξ_1 and Ξ_2 that

$$\Xi_3' :: \Gamma\Gamma' \rhd e_1 : (\phi \ \mathsf{ref} \ (\Delta), \Delta_1) \quad (8)$$
$$\Xi_4' :: \Gamma\Gamma' \rhd e_2 : (\phi, \Delta_2 \quad (9)$$

We can use (8) and (9) to construct a derivation of

$$\Xi_5' :: \Gamma\Gamma' \rhd e_1 := e_2 : (\mathsf{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)$$

which is what we wanted.

9. Assume Ξ is

$$\frac{\Xi_j \quad \Xi_k \quad \Gamma\{\overline{g_i : \phi_i}\} \rhd \overset{\Xi'}{e}[\overline{g_i/f_i}] : (\phi, \Delta)}{\Gamma \rhd \mathsf{letfun} \ \overline{f_i = e_i} \ \mathsf{with} \ \overline{x_j = \mathsf{sref} \ a_j} \ \mathsf{in} \ e : (\phi, \Delta \bigcup \overline{\Delta_j} \bigcup \overline{LS(\phi_j)} \bigcup \overline{LS(\phi_k)} - \{\overline{\ell_j} \bigcup \overline{g_i}\})}$$

where

$$\begin{array}{rcl} \Xi_j & :: & \Gamma \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \\ \Xi_k & :: & \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref} \ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_i, \{\}) \\ & & \text{for } 1 \leq i \leq n \\ & & LS(\phi_k) \bigcap (\{\overline{\ell_j}\} \biguplus \{\overline{g_i}\} = \{\}) \\ & & LS(\phi) \bigcap \{\overline{g_i}\} = \{\} \end{array}$$

We know via induction on Ξ_j, Ξ_k, and Ξ' that

$$\Xi_1' :: \Gamma\Gamma' \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \quad (10)$$
$$\Xi_2' :: \Gamma\Gamma'\{\overline{\ell_j : \phi_j \ \ \mathsf{ref} \ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_i, \{\}) \quad (11)$$
$$1 \leq i \leq n$$
$$\Xi_3' :: \Gamma\Gamma'\{\overline{g_i : \phi_i}\} \rhd e[\overline{g_i/f_i}] : (\phi, \Delta)$$

From (10), (11), and (12) we have a new derivation

$$\begin{array}{rcl} \Xi_4' & :: & \Gamma \rhd \mathsf{letfun} \ \overline{f_i = e_i} \ \mathsf{with} \ \overline{x_j = \mathsf{sref} \ a_j} \ \mathsf{in} \ e : (\phi, \Delta_{lf} \\ \Delta_{lf} & = & \Delta \bigcup \overline{\Delta_j} \bigcup \overline{LS(\phi_j)} \bigcup \overline{LS(\phi_k)} - \{\overline{\ell_j} \bigcup \overline{g_i}\}) \end{array}$$

which is what we wanted.

10. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \overset{\Delta}{\rightarrow} \phi_2, \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1', \Delta_2)}}{\Gamma \rhd e_1 \ @ \ e_2 : (\phi_2, LS(\phi_1 \overset{\Delta}{\rightarrow} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2)} \ \phi_1' \leq \phi_1$$

We know via induction on Ξ_1 and Ξ_2 that

$$\Xi_1' :: \Gamma\Gamma' \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \tag{12}$$
$$\Xi_2' :: \Gamma\Gamma' \rhd e_2 : (\phi_1', \Delta_2) \tag{13}$$

From (12) and (13), we can construct

$$\Xi_3' :: \Gamma\Gamma' \rhd e_1 \ @ \ e_2 : (\phi_2, LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2)$$

which is what we needed. □

When we have a value that satisfies our $\models$ relation for a certain type ϕ', it stands to reason that it would satisfy the relation at a greater type as well. Such a property is important, since we depend on an ordering on our types.

Lemma 3.1 *If $\rho, \sigma \models v : \phi'$ and $\phi' \leq \phi$, then $\rho, \sigma \models v : \phi$.*

Proof: We can prove this Lemma via induction on the type ϕ'.

1. Assume $v = c$. This case is trivial, since a base type will always have the same type.

2. Assume $v = \lambda x.e$ and $\rho, \sigma \models \lambda x.e : \phi'$ iff there exist Γ and ϕ such that $\rho, \sigma : \Gamma, \Gamma \rhd \lambda x.e : (\phi, \{\})$ is derivable, and $\phi \leq \phi'$. We know from the $\models$ relation that there exists a Γ such that

$$\Xi_1' :: \Gamma \rhd \lambda x.e : (\phi, \{\}) \tag{14}$$
$$\phi \leq \phi' \tag{15}$$
$$\rho, \sigma : \Gamma \tag{16}$$

From (14) and (15) we can conclude that we have a relationship such that

$$\rho, \sigma \models \lambda x.e : \phi'' \tag{17}$$

for any $\phi' \leq \phi''$ Hence, we know either $\phi \leq \phi''$, which is what we need.

3. Assume $v = \ell$. From the definition of the $\models$ relation for a location, we know that

$$\rho, \sigma \models \sigma(\ell) : \phi$$

By induction on the type, we know that

$$\rho, \sigma \models \sigma(\ell) : \phi'$$
$$\rho' \leq \rho \tag{18}$$

This is enough to show that

$$\rho, \sigma \models \sigma(\ell) : \phi' \ \mathsf{ref}\ (\Delta)$$

where $\phi' \ \mathsf{ref}\ (\Delta) \leq \phi \ \mathsf{ref}\ (\Delta)$.

4. Assume $v = f$ and the $\models$ relation holds for f. We know that this case immediately reduces to a λ-abstraction case. Hence, we know that there exist $\phi_1, \phi_2, and \Delta$ such that

$$\begin{aligned} \rho', \sigma\sigma_e \models \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2 \\ \phi_1' \xrightarrow{\Delta'} \phi_2' \leq \phi_1 \xrightarrow{\Delta} \phi_2 \end{aligned} \tag{19}$$

From (19), we can conclude that

$$\rho, \sigma \models f : \phi_1 \xrightarrow{\Delta} \phi_2$$

which is what we needed to show. □

When we type an expression, we us the environment Γ, which may contain more information than we need to type an expression. Since Γ contains the static references and functions, we want to be able to limit the environment to the items needed to type an expression, i.e., $LS(\phi)$.

Lemma 3.2 *If* $\Gamma \rhd e : (\phi, \Delta)$ *then* $\Gamma|_{LS(\phi) \bigcup \Delta} \rhd e : (\phi, \Delta)$.

Proof: We can prove the Lemma via induction on derivations Ξ of the form $\Gamma \rhd e : (\phi, \Delta)$.

1. Assume Ξ is $\overline{\Gamma \rhd c : (\phi, \{\})}$. The case is straightforward, since $LS(\phi) = \{\}$ and we can naturally type constants in an empty Γ.

2. Assume Ξ is

$$\frac{\Gamma(x) = \phi)}{\Gamma \rhd x : (\phi, \{\}}$$

It follows that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi)} \rhd x : (\phi, \{\}) \Gamma(x) = \phi \tag{20}$$

3. Assume Ξ is

$$\frac{\Gamma(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$

It follows that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \{f\}} \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})$$

is derivable.

4. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma\{x:\phi_1\} \rhd e : (\phi_2, \Delta)}}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$

We know via induction on Ξ_1 that

$$\Xi_1' \quad :: \quad \Gamma\{x:\phi_1\}|_{LS(\phi_2)\bigcup\Delta} \rhd e : (\phi_2, \Delta)$$

We also notice that

$$LS(\phi_1 \xrightarrow{\Delta} \phi_2) \quad = \quad \Delta \bigcup LS(\phi_1) \bigcup LS(\phi_2) \tag{21}$$

Hence, it follows from Proposition 3.3 that

$$\Xi_2' \quad :: \quad \Gamma|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2)} \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})$$

which is what we wanted to show.

5. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\text{bool}, \Delta)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_2, \Delta_1)} \quad \overset{\Xi_3}{\Gamma \rhd e_3 : (\phi_3, \Delta_2)}}{\Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

By induction on Ξ_1 we know that

$$\Xi_1' \quad :: \quad \Gamma|_\Delta \rhd e_1 : (\text{bool}, \Delta) \tag{22}$$

By induction on Ξ_2 we know that

$$\Xi_2' \quad :: \quad \Gamma|_{LS(\phi)\bigcup\Delta_1} \rhd e_2 : (\phi_2, \Delta_1) \tag{23}$$

By induction on Ξ_2 we know that

$$\Xi_3' \quad :: \quad \Gamma|_{LS(\phi)\bigcup\Delta_2} \rhd e_3 : (\phi_3, \Delta_2) \tag{24}$$

From (22), (23), (24), and Proposition 3.3, we know that the entire expression can be typed in $\Gamma|_{LS(\phi)\bigcup\Delta\bigcup\Delta_1\bigcup\Delta_2}$. Hence, we can conclude that

$$\Xi_4' \quad :: \quad \Gamma|_{LS(\phi)\bigcup\Delta\bigcup\Delta_1\bigcup\Delta_2} \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$$

from Proposition 3.3, which is what we wanted to show.

6. Assume Ξ is

$$\frac{\Gamma(\ell) = \phi\ \mathsf{ref}\ (\Delta)}{\Gamma \rhd \ell : (\phi\ \mathsf{ref}\ (\Delta), \{\ell\})}$$

We must be sure that $\ell \in dom(\Gamma)|_{LS(\phi\ \mathsf{ref}\ (\Delta))} \bigcup \{\ell\}$, which we can see is explicit. We must also be sure that all static references and functions of the type of the location are included, which they are. Hence, we can conclude that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi) \bigcup \Delta} \rhd \ell : (\phi\ \mathsf{ref}\ (\Delta), \Delta)$$

This is what we wanted to show.

7. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta')}}{\Gamma \rhd !e : (\phi, \Delta \bigcup \Delta')}$$

We know via induction on Ξ_1 that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi\ \mathsf{ref}\ (\Delta)) \bigcup \Delta'} \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta') \tag{25}$$

We also know that

$$LS(\phi\ \mathrm{ref}\ \Delta) \ = \ LS(\phi) \bigcup \Delta \tag{26}$$

From (25) and (26), we can conclude that

$$\Xi_2' \ :: \ \Gamma|_{LS(\phi) \bigcup \Delta \bigcup \Delta'} \rhd !e : (\phi, \Delta \bigcup \Delta')$$

which is what we wanted to show.

8. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi\ \mathsf{ref}\ (\Delta), \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi, \Delta_2)}}{\Gamma \rhd e_1 := e_2 : (\mathsf{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

Via induction on Ξ_1, we know that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi\ \mathsf{ref}\ (\Delta)) \bigcup \Delta_1} \rhd e_1 : (\phi\ \mathsf{ref}\ (\Delta), \Delta_1) \tag{27}$$

We know by induction on Ξ_2 that

$$\Xi_2' \ :: \ \Gamma|_{LS(\phi) \bigcup \Delta_2} \rhd e_1 : (\phi, \Delta_2) \tag{28}$$

We also know that

$$LS(\phi\ \mathrm{ref}\ \Delta) \ = \ LS(\phi) \bigcup \Delta \tag{29}$$

from the definition of $LS(\phi\ \mathsf{ref}\ (\Delta))$. Hence, we can conclude from (27), (28), (29), and Proposition 3.3 that

$$\Xi_3' \ :: \ \Gamma|_{LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2} \rhd e_1 := e_2 : (\mathsf{unit}, \Delta \bigcup \Delta_1 \bigcup \Delta_2)$$

since $LS(\mathsf{unit}) = \{\}$.

9. Assume Ξ is

$$\frac{\Xi_j \quad \Xi_k \quad \Gamma\{\overline{g_i : \phi_i}\} \rhd \overset{\Xi'}{e}[\overline{g_i/f_i}] : (\phi, \Delta)}{\Gamma \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta \bigcup \overline{\Delta_j} \bigcup \overline{LS(\phi_j)} \bigcup \overline{LS(\phi_k)} - \{\overline{\ell_j} \bigcup \overline{g_i}\})}$$

where

$$\begin{array}{rcl} \Xi_j & :: & \Gamma \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \\ \Xi_k & :: & \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_i, \{\}) \\ & & 1 \leq i \leq n \\ & & LS(\phi_k) \bigcap \{\overline{\ell_j}\} \biguplus \{\overline{g_i}\} = \{\} \\ & & LS(\phi) \bigcap \{\overline{g_i}\} = \{\} \end{array}$$

We know via induction on Ξ_j that

$$\Xi'_1 \quad :: \quad \Gamma|_{LS(\phi_j) \bigcup \Delta_j} \rhd a_j : (\phi_j, \Delta_j)$$

We know via induction on Ξ_k that

$$\Xi'_2 \quad :: \quad \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\}|_{LS(\phi_k)} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_k, \{\})$$

Finally, we know via induction on Ξ' that

$$\Xi'_3 \quad :: \quad \Gamma\{\overline{g_i : \phi_i}\}|_{LS(\phi) \bigcup \Delta} \rhd e[\overline{g_i/f_i}] : (\phi, \Delta)$$

Hence we know by Proposition 3.3 that

$$\Xi'_4 \quad :: \quad \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\}|_{lsf} \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta_{lf})$$

where

$$\begin{array}{rcl} \Delta_{lf} & = & LS(\phi_j) \bigcup \Delta_j \bigcup LS(\phi_k) \bigcup \Delta \\ lsf & = & LS(\phi) \bigcup \Delta_{lf} \end{array}$$

Furthermore, we can subtract the ℓ_js and the g_is out of the set for the expression because they cannot occur free in the expression. Hence, we have

$$\begin{array}{rcl} \Xi'_5 & :: & \Gamma|_{lsf} \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta'_{lf}) \\ \Delta'_{lf} & = & LS(\phi_j) \bigcup \Delta_j \bigcup LS(\phi_k) \bigcup \Delta - \{\overline{\ell_j}\} \bigcup \{\overline{g_i}\}) \end{array}$$

which is what we wanted to prove.

10. Assume Ξ is

$$\frac{\begin{array}{cc}\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1)} & \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1', \Delta_2)}\end{array}}{\Gamma \rhd e_1 \ @ \ e_2 : (\phi_2, LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2)} \ \phi_1' \leq \phi_1$$

We know via induction on Ξ_1 that

$$\Xi_1' \ :: \ \Gamma|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1} \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1) \tag{30}$$

Via induction on Ξ_2, we know that

$$\Xi_2' \ :: \ \Gamma|_{LS(\phi_1') \bigcup \Delta_2} \rhd e_2 : (\phi_1', \Delta_2)$$

However, since $\phi_1' \leq \phi_1$, we know that $LS(\phi_1') \subseteq LS(\phi)$. Therefore,

$$\Xi_3' \ :: \ \Gamma|_{LS(\phi_1) \bigcup \Delta_2} \rhd e_2 : (\phi_1', \Delta_2) \tag{31}$$

From (30) and (31), we can conclude that

$$\Xi_4' \ :: \ \Gamma|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2} \rhd e_1 \ @ \ e_2 : (\phi_2, LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2) \tag{32}$$

using Proposition 3.3, which is what we wanted to show. □

We can define a notion of extending the environment ρ and the state σ. We use the notation $\rho\rho'$ and $\sigma\sigma'$ to indicate the environment and state have been extended to include the items in ρ' and σ'. With these extensions, it would make sense that any values for which the $\models$ relation held would still hold, since unique names are used for functions and references.

Lemma 3.3 *If $\rho, \sigma \models v : \phi$, $dom(\rho) \bigcap dom(\rho') = \{\}$, and $dom(\sigma) \bigcap dom(\sigma') = \{\}$, then $\rho\rho', \sigma\sigma' \models : \phi$.*

Proof: We can prove this by induction types.

1. Assume $v = c$. This case holds trivially, since a constant can be typed in any environment and state.
2. Assume $v = \lambda x.e$ and $\rho, \sigma \models \lambda x.e : \phi$ iff there exist Γ and ϕ' such that $\rho, \sigma : \Gamma, \Gamma \rhd \lambda x.e : (\phi', \{\})$ is derivable, and $\phi' \leq \phi$. From Lemma 3.3, we know that

$$\Xi_1 \ :: \ \Gamma\Gamma' \rhd \lambda x.e : (\phi', \{\}) \tag{33}$$

We can reason that there is a $\rho\rho'$ and $\sigma\sigma'$ such that $\rho\rho', \sigma\sigma' : \Gamma\Gamma'$. Thus, we have

$$\rho\rho', \sigma\sigma' \models \lambda x.e : \phi' \tag{34}$$

From Lemma 3.1, we know that

$$\rho\rho', \sigma\sigma' \models \lambda x.e : \phi$$

from (34) and our assumption that $\phi' \leq \phi$, which is what we wanted to show.

3. Assume $v = \ell$. We know that all names for references are unique, hence nothing in σ' will replace the value in σ. By induction on the type, we know

$$\rho\rho', \sigma\sigma' \models \sigma(\ell) : \phi \tag{35}$$

which we can use to as the condition in the $\models$ relation for a location to show that

$$\rho\rho', \sigma\sigma' \models \ell : \phi$$

which is what we needed to show.

4. Assume $v = f$. This case immediately reduces to the case for a λ-abstraction. Thus, we know that

$$\rho\rho', \sigma\sigma' : \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2 \tag{36}$$

Since ρ and ρ' are disjoint sets by assumption, we know that $\rho\rho'(f) = \rho(f)$. Thus, we can conclude that

$$\rho\rho', \sigma\sigma' \models f : \phi_1 \xrightarrow{\Delta} \phi_2$$

which is what we needed to show. □

In the same way we proved Lemma 3.2 to allow us to restrict a typing environment to the live static references and functions, we want to be able to make the same restriction on ρ and σ and still satisfy the $\models$ relation.

Lemma 3.4 *If $\rho, \sigma \models v : \phi$, then $\rho', \sigma' \models v : \phi$ where $\rho' = \rho|_{LS(\phi)}$ and $\sigma' = \sigma|_{LS(\phi)}$.*

Proof: We can prove this by looking at the types for values. We will also refer to the definition of $LS(\phi)$, found in Figure 6.

1. Assume $v = c$. This case holds trivially, since a constant can type in any ρ and σ.

2. Assume $v = \lambda x.e$ and $\rho, \sigma \models \lambda x.e : \phi$ iff there exist Γ and ϕ' such that $\rho, \sigma : \Gamma, \Gamma \rhd \lambda x.e : (\phi', \{\})$ is derivable, and $\phi' \leq \phi$. We know from Lemma 3.2 that

$$\Xi_1' \quad :: \quad \Gamma|_{LS(\phi')} \rhd \lambda x.e : (\phi', \{\})$$

If we have $\rho' = \rho|_{LS(\phi')}$ and $\sigma' = \sigma|_{LS(\phi')}$, then we know

$$\rho', \sigma' : \Gamma|_{LS(\phi')}$$

This is enough to show that

$$\rho'\sigma' \models \lambda x.e : \phi'$$

By Lemma 3.3, we know that

$$\rho'\sigma' \models \lambda x.e : \phi$$

which is what we needed to prove.

3. Assume $v = \ell$ and $\rho, \sigma \models \ell : \phi \ \mathsf{ref}\ (\Delta)$ iff $\rho, \sigma \models \sigma(\ell) : \phi$ and $\ell \in \Delta$. We know by induction on the type that

$$\rho', \sigma' \models \sigma(\ell) : \phi$$

However, this is sufficient to show that

$$\rho', \sigma' \models \ell : \phi \ \mathsf{ref}\ (\Delta)$$

which is what we needed to show.

4. Assume $f = e$ and $\rho, \sigma \models f : \phi_1 \xrightarrow{\Delta} \phi_2$ iff $\rho(f) = (\lambda x.e, \sigma_e, \rho')$, $\rho', \sigma\sigma_e \models \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2$ and $LS(\phi_2) \bigcap dom(\sigma_e) = \{\}$. We know that this case reduces to a λ-abstraction case. Thus, we have the following satisfiability relation

$$\rho'|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2)} \sigma\sigma_e|_{\phi_1 \xrightarrow{\Delta} \phi_2} \models \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2$$

From Lemma 3.3, we know that

$$\rho'|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2)} \sigma_{\phi_1 \xrightarrow{\Delta} \phi_2} \sigma_e \models \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2$$

Since $\sigma\sigma_e|_{\phi_1 \xrightarrow{\Delta} \phi_2} \subseteq \sigma|_{\phi_1 \xrightarrow{\Delta} \phi_2} \sigma_e$. We can use this to conclude that

$$\rho|_{LS(\phi_1 \xrightarrow{\Delta} \phi_2)} \models f : \phi_1 \xrightarrow{\Delta} \phi_2$$

which is what we wanted to prove. $\square$

Since we can relate more than one ρ and σ to a single Γ, we want to be sure that the $\models$ relation for a value holds for any environment and state related to Γ.

Lemma 3.5 *If $\rho, \sigma \models v : \phi$, and there exists a Γ such that $\rho, \sigma : \Gamma$ and $\rho', \sigma' : \Gamma$, then $\rho', \sigma' \models v : \phi$.*

Proof: We can prove this Lemma by analyzing our value types and the $\models$ relation.

1. Assume $v = c$. This case holds trivially, since $c : \phi$ holds in any ρ and σ.

2. Assume $v = \lambda x.e$ and $\rho, \sigma \models \lambda x.e : \phi$ iff there exist Γ and ϕ' such that $\rho, \sigma : \Gamma, \Gamma \rhd \lambda x.e : (\phi', \{\})$ is derivable, and $\phi' \leq \phi$. Let us use as our Γ the one in the above $\models$ definition. We know we have a Γ such that

$$\rho', \sigma' : \Gamma \tag{37}$$
$$\Xi'_1 \quad :: \quad \Gamma \rhd \lambda x.e : (\phi', \{\}) \tag{38}$$

   such that $\phi' \leq \phi$. We can conclude from (37) and (38) that

$$\rho', \sigma' \models \lambda x.e : \phi$$

   which is what we needed to show.

3. Assume $v = \ell$ and $\rho, \sigma \models \ell : \phi \ \mathsf{ref}\ (\Delta)$ iff $\rho, \sigma \models \sigma(\ell) : \phi$ and $\ell \in \Delta$. From our assumption that $\rho, \sigma : \Gamma$, we know

$$\Gamma(\ell) \quad = \quad \phi \ \mathsf{ref}\ (\Delta) \tag{39}$$

   From our assumption that $\rho', \sigma' : \Gamma$ and (39), we know

$$\rho', \sigma' \models \sigma'(\ell) : \phi \tag{40}$$

   We can use (40) to prove

$$\rho', \sigma' \models \ell : \phi \ \mathsf{ref}\ (\Delta)$$

   where we know $\ell \in \Delta$. Hence, we have proven the Lemma for this case.

4. Assume $f = e$ and $\rho, \sigma \models f : \phi_1 \xrightarrow{\Delta} \phi_2$ iff $\rho(f) = (\lambda x.e, \sigma_e, \rho')$, $\rho', \sigma\sigma_e \models \lambda x.e : \phi_1 \xrightarrow{\Delta} \phi_2$ and $LS(\phi_2) \bigcap dom(\sigma_e) = \{\}$. From our assumption that $\rho, \sigma : \Gamma$, we know that

$$\Gamma(f) \quad = \quad \phi_1 \xrightarrow{\Delta} \phi_2 \tag{41}$$

   We know from (41) and our assumption that $\rho', \sigma' : \Gamma$ that

$$\rho', \sigma' \models f : \phi_1 \xrightarrow{\Delta} \phi_2$$

   This is what we needed to show. □

If an expression satisfies the $\models$ relation, it should be typable. However, we have to be careful, since the type may not be the same one, but instead a lesser type.

Lemma 3.6 *If $\rho, \sigma \models v : \phi$ then there exists a Γ and Δ such that $\rho, \sigma : \Gamma$, $\Gamma \rhd v : (\phi', \Delta)$ and $\phi' \leq \phi$.*

Proof: We can prove this Lemma via induction on the structure of the types of values.

1. Assume $v = c$ and $\rho, \sigma \models c : \phi$, where $c : \phi \in Constants$. This case holds trivially, since a constant can be typed in any environment to have the type ϕ, where $\phi \leq \phi$.

2. Assume $v = \lambda x.e$ and $\rho, \sigma \models \lambda x.e : \phi$ iff there exist Γ and ϕ' such that $\rho, \sigma : \Gamma, \Gamma \rhd \lambda x.e : (\phi', \{\})$ is derivable, and $\phi' \leq \phi$. The Γ we can use is the one we assume exists. By definition, $\phi' \leq \phi$.

3. Assume $v = \ell$ and $\rho, \sigma \models \ell : \phi \ \mathsf{ref}\ (\Delta)$ iff $\rho, \sigma \models \sigma(\ell) : \phi$ and $\ell \in \Delta$. By induction on the type $\phi \ \mathsf{ref}\ (\Delta)$, we know that there exists a Γ and ϕ' such that:

$$\begin{array}{rcl} & & \rho, \sigma : \Gamma \\ & & \phi' \leq \phi \\ \Xi_1' & :: & \Gamma \rhd \sigma(\ell) : (\phi', \{\}) \end{array} \tag{42}$$

   We can use this to construct a reference to the type ϕ', $\phi' \ \mathsf{ref}\ (\{\ell\})$. Since $\sigma(\ell) \in dom(\Gamma)$ and $\rho, \sigma : \Gamma$, we have

$$\Gamma(\ell) = \phi \ \mathsf{ref}\ (\Delta) \tag{43}$$

   which we can use to construct a derivation

$$\Xi_2' \ :: \ \Gamma \rhd \ell : (\phi' \ \mathsf{ref}\ \{\ell\})$$

   We know that $\phi' \ \mathsf{ref}\ \{\ell\} \leq \phi \ \mathsf{ref}\ \Delta$, as a result of (42) and the fact that $\{\ell\} \subseteq \Delta$. This is what we wanted to show.

4. Assume $v = f$. As we can see, this case immediately reduces to a λ-abstraction. Hence, we know we have

$$\begin{array}{rcl} & & \rho', \sigma\sigma_e : \Gamma' \\ \Xi_1' & :: & \Gamma' \rhd \lambda x.e : (\phi_1' \xrightarrow{\Delta'} \phi_2', \{\}) \\ & & (\phi_1' \xrightarrow{\Delta'} \phi_2') \leq (\phi_1 \xrightarrow{\Delta} \phi_2) \end{array} \tag{44}$$

We know from the definition for the Lookup function for an environment that $dom(\rho) = dom(\rho')$, since the only difference between ρ and ρ' is that the latter contains the empty private state instead of σ_e. Hence, we know

$$\rho, \sigma\sigma_e : \Gamma \tag{45}$$

From (45) and the definition of $\rho, \sigma : \Gamma$, we know that $\rho, \sigma \models \rho(f) : \Gamma(f)$ holds. We can use this to construct a derivation of the type of f

$$\Xi'_2 \quad :: \quad \Gamma \rhd f : (\phi'_1 \xrightarrow{\Delta'} \phi'_2, \{f\}) \tag{46}$$

We notice that the typing of the f did not require the references in σ_e. In fact, it *cannot* use them. Hence, we know

$$\rho, \sigma : \Gamma \tag{47}$$

for some Γ such that $dom(\Gamma) \bigcap dom(\sigma_e) = \{\}$. We have shown what we needed with (46), (47), and (44). □

Lemma 3.7 (Substitution Lemma) *If* $\Gamma, x : \phi_1 \rhd e : (\phi_2, \Delta)$, $\Gamma \rhd e_1 : (\phi'_1, \Delta')$, $\phi'_1 \leq \phi_1$, *then* $\Gamma \rhd e[e_1/x] : (\phi'_2, \Delta)$ *for some* $\phi'_2 \leq \phi_2$.

Proof: We prove the Substitution Lemma via induction on deductions Ξ of the form $\Gamma \rhd e : (\phi_2, \Delta)$.

1. Assume $e = c$ It follows immediately that $\Gamma \rhd c[e_1/x] : (\phi,)$ such that $\phi \leq \phi$.
2. Assume $e = y$ where $x \neq y$. It follows immediately that $\Gamma \rhd y[e_1/x] : (\phi_2, \Delta)$ such that $\phi_2 \leq \phi_2$.
3. Assume $e = x$. In this case, $\phi_1 = \phi_2$ and $\Gamma \rhd e_1 : (\phi'_1, \Delta')$, where $\phi'_1 \leq \phi_1$.
4. Assume Ξ is
$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma\{x : \phi_1, y : \phi_2\} \rhd e : (\phi_2, \Delta)\end{array}}{\Gamma \rhd \lambda y.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$
By induction on Ξ_1 with our inductive assumptions, we know that
$$\begin{array}{rcl}\Xi'_1 & :: & \Gamma\{y : \phi_1\} \rhd e[e_1/x] : (\phi'_2, \Delta) \\ & & \phi'_2 \leq \phi_2\end{array} \tag{48}$$
From (48), we know that
$$\Xi'_2 \quad :: \quad \Gamma \rhd \lambda y.e(\phi_1 \xrightarrow{\Delta} \phi'_2, \{\}) \tag{49}$$
Since we know that $\phi_1 \leq \phi_1$ and $\phi'_2 \leq \phi_2$, we can conclude that
$$(\phi_1 \xrightarrow{\Delta} \phi'_2) \leq (\phi_1 \xrightarrow{\Delta} \phi_2) \tag{50}$$
Thus, we have what we need, (49) and (50).

5. Assume Ξ is

$$\frac{\Gamma(f) = \phi \xrightarrow{\Delta} \phi_2}{\Gamma\{x : \phi_1\} \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$

This case follows directly that $\Gamma \rhd f[e_1/x] : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})$.

6. Assume Ξ is

$$\frac{\Gamma(\ell) = \phi}{\Gamma \rhd \ell : (\phi, \{\})}$$

This case follows immediately that $\Gamma \rhd \ell[e_1/x] : (\phi, \{\})$

7. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma\{x : \phi_1\} \rhd e : (\text{bool}, \Delta)} \quad \overset{\Xi_2}{\Gamma\{x : \phi_1\} \rhd e_2 : (\phi, \Delta_1)} \quad \overset{\Xi_3}{\Gamma\{x : \phi_1\} \rhd e_3 : (\phi, \Delta_2)} \quad \begin{array}{l}\phi_2 \leq \phi \\ \phi_3 \leq \phi\end{array}}{\Gamma\{x : \phi_1\} \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

By induction on Ξ_1 with our inductive assumption, we know that

$$\Xi_1' \quad :: \quad \Gamma \rhd e[e_1/x] : (\text{bool}, \Delta) \tag{51}$$

By induction on Ξ_2 with our induction assumption, we know that

$$\begin{aligned} \Xi_2' \quad :: \quad & \Gamma \rhd e_2[e_1/x] : (\phi', \Delta_1) && (52) \\ & \phi_2' \leq \phi_2 \end{aligned}$$

By induction on Ξ_3 with our induction assumption, we know that

$$\begin{aligned} \Xi_3' \quad :: \quad & \Gamma \rhd e_3[e_1/x] : (\phi', \Delta_2) && (53) \\ & \phi_3' \leq \phi_3 && (54) \end{aligned}$$

Hence, we can use (51), (52), (53) to form a new typing

$$\begin{aligned} \Xi_4' \quad :: \quad & \Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi', \Delta \bigcup \Delta_1 \bigcup \Delta_2) && (55) \\ & \phi' \leq \phi && (56) \end{aligned}$$

We have our results for Lemma for the if statement with (54) and (55).

8. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma\{x : \phi_1\} \rhd e_0 : (\phi \xrightarrow{\Delta} \phi_2, \Delta_1)} \quad \overset{\Xi_2}{\Gamma\{x : \phi_1\} \rhd e_2 : (\phi_3, \Delta_2)}}{\Gamma\{x : \phi_1\} \rhd e_0 \mathbin{@} e_2 : (\phi_2, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}\ \phi_3 \leq \phi$$

By induction on Ξ_1 with our inductive assumption, we know that

$$\begin{aligned} \Xi_1' \quad :: \quad & \Gamma \rhd e_0[e_1/x] : (\phi' \xrightarrow{\Delta} \phi_2', \Delta_1) \\ & (\phi' \xrightarrow{\Delta} \phi_2') \leq (\phi \xrightarrow{\Delta} \phi_2) && (57) \end{aligned}$$

From (57), it follows that

$$\begin{array}{l} \phi \leq \phi' \\ \phi'_2 \leq \phi_2 \end{array} \tag{58}$$

By induction on Ξ_2 with our induction assumption, we know that

$$\begin{array}{rcll} \Xi'_2 & :: & \Gamma \rhd e_2[e_1/x] : (\phi'_3, \Delta_2) & (59) \\ & & \phi'_3 \leq \phi_3 & (60) \end{array}$$

From (60), the fact that $\phi_3 \leq \phi$, and (58) we can conclude that

$$\phi'_3 \leq \phi'$$

From (57), (59), and (61), we have

$$\Xi'_3 \quad :: \quad \Gamma \rhd e_0 \ @ \ e_2[e_1/x] : (\phi'_2, \Delta \bigcup \Delta_1 \bigcup \Delta_2) \tag{61}$$

Therefore, we have proven the Lemma for this case with (61) and (59).

9. Assume Ξ is

$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma \rhd e : (\phi \ \mathsf{ref} \ \Delta, \Delta')\end{array}}{\Gamma \rhd !e : (\phi, \Delta')}$$

10. Assume e is a reference update $e_0 := e_2$. The case follows immediately, since unit is a base type.

11. Assume Ξ is

$$\frac{\Xi_j \quad \Xi_i \quad \begin{array}{c}\Xi' \\ \Gamma\{\overline{g_i : \phi_i}, x : \phi_1\} \rhd e[\overline{g_i/f_i}] : (\phi, \Delta)\end{array}}{\Gamma\{x : \phi_1\} \rhd \mathsf{letfun} \ \overline{f_i = e_i} \ \mathsf{with} \ \overline{x_j = \mathsf{sref} \ a_j} \ \mathsf{in} \ e : (\phi, \Delta_f)} \quad \begin{array}{l} LS(\phi_k)(\bigcap\{\overline{\ell_j}\} \uplus \{\overline{g_i}\}) = \{\} \\ LS(\phi) \bigcap \{\overline{g_i}\} = \{\} \end{array}$$

where

$$\begin{array}{rcl} \Delta_l & = & \Delta \bigcup \overline{\Delta_i} - \{\overline{\ell_j} \bigcup \overline{g_i}\}) \\ \Xi_j & :: & \Gamma\{x : \phi_1\} \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \\ \Xi_k & :: & \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref} \ \ \{\ell_j\}}, \overline{g_i : \phi_i}, x : \phi_1\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_i, \{\}), 1 \leq i \leq n \end{array}$$

We know via induction on Ξ_j with our inductive assumption that

$$\begin{array}{rcll} \Xi'_1 & :: & \Gamma \rhd a_j[e_1/x] : (\phi'_j, \Delta_j), 1 \leq j \leq m & (62) \\ & & \phi'_j \leq \phi_j, 1 \leq j \leq m & \end{array}$$

We know via induction on Ξ_k with our inductive assumption and the results in (62) that

$$\begin{array}{rcll} \Xi_2' & :: & \Gamma\{\overline{\ell_j : \phi_j' \ \mathsf{ref}\ (\Delta_j), g_i : \phi_i'}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}][e_1/x] : (\phi_i', \{\}) & (63) \\ & & \phi_i' \leq \phi_i & \\ & & 1 \leq i \leq n & \end{array}$$

By induction on Ξ' with our inductive assumption and our results in (63) we know that

$$\begin{array}{rcll} \Xi_3' & :: & \Gamma\{\overline{g_i : \phi_i'}\} \rhd e[\overline{g_i/f_i}][e_1/x] : (\phi', \Delta) & (64) \\ & & \phi' \leq \phi & (65) \end{array}$$

With (62), (63), and (64), we have a new derivation

$$\Xi_4' \ :: \ \Gamma \rhd (\mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e[e_1/x]) : (\phi', \Delta_e) \quad (66)$$

where

$$\Delta_e \ = \ \Delta \bigcup \overline{\Delta_i} - \{\overline{\ell_j} \bigcup \overline{g_i}\})$$

From (66) and (65), we have proven the Lemma for this case. □

When two typing environments are related to the same ρ and σ, we want there to be a single Γ_3–less than both typing environments–that is related to the environment and state. Such a property will help us in the type soundness proof when we need to make sure the typing environments are the same, particularly in applying the Substitution Lemma.

Lemma 3.8 *Conjecture If $\rho, \sigma : \Gamma_1$ and $\rho, \sigma : \Gamma_2$, then there exists a Γ_3 such that $\Gamma_3 \leq \Gamma_2$, $\Gamma_3 \leq \Gamma_1$, and $\rho, \sigma : \Gamma_3$. The proof requires a notion of least types.*

We should be be able to type an expression to have a smaller type in a smaller type environment. As we have already defined, a smaller type environment has smaller types for all types in the environment.

Lemma 3.9 *If $\Gamma \rhd e : (\phi, \Delta)$ and $\Gamma' \leq \Gamma$, then $\Gamma' \rhd e : (\phi', \Delta')$ for some $\phi' \leq \phi$ and $\Delta' \subseteq \Delta$.*

Proof: We can prove this via induction on derivations Ξ of the form $\Gamma \rhd e : (\phi, \Delta)$.

1. Assume Ξ is $\overline{\Gamma \rhd c : (\phi, \{\})}$. This case is straightforward, since $\phi \leq \phi$ and Δ and Δ' are the empty set.

2. Assume Ξ is
$$\frac{\Gamma(x) = \phi}{\Gamma \rhd x : (\phi, \{\})}$$
If $\Gamma' \leq \Gamma$, then by definition $x \in dom(\Gamma')$ and $\phi' \leq \phi$ for some ϕ' such that
$$\Gamma'(x) = \phi' \tag{67}$$
which we can use to create a deduction
$$\Xi_1' \quad :: \quad \Gamma' \rhd x : (\phi', \{\}) \tag{68}$$

3. Assume Ξ is
$$\frac{\Gamma(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$
By our assumption, we know that
$$\begin{aligned} \Gamma'(f) &= (\phi_1' \xrightarrow{\Delta'} \phi_2') \\ (\phi_1' \xrightarrow{\Delta'} \phi_2') &\leq (\phi_1 \xrightarrow{\Delta} \phi_2) \end{aligned} \tag{69}$$
Hence, we know
$$\Xi_1' \quad :: \quad \Gamma' \rhd f : (\phi_1' \xrightarrow{\Delta'} \phi_2', \{f\}) \tag{70}$$
where (69) holds and $\{f\} \subseteq \{f\}$. This is what we wanted to prove.

4. Assume Ξ is
$$\frac{\begin{array}{c}\Xi_1 \\ \Gamma\{x : \phi_1\} \rhd e : (\phi_2, \Delta)\end{array}}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})}$$
We know via induction on Ξ_1 that
$$\begin{aligned} \Xi_1' \quad :: \quad & \Gamma'\{x : \phi_1\} \rhd e : (\phi_2', \Delta') && (71) \\ & \phi_2' \leq \phi_2 && (72) \\ & \Delta' \subseteq \Delta && (73) \\ & \Gamma' x : \phi_1 \leq \Gamma x : \phi_1 && (74) \end{aligned}$$
We can use (71) to construct a derivation of
$$\Xi_2' \quad :: \quad \Gamma' \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta'} \phi_2, \{\})$$
where $\phi_1 \xrightarrow{\Delta'} \phi_2' \leq \phi_1 \xrightarrow{\Delta} \phi_2$ by (72) and (73). This is what we needed to show.

5. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\text{bool}, \Delta)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_2, \Delta_1)} \quad \overset{\Xi_3}{\Gamma \rhd e_3 : (\phi_2, \Delta_2)} \quad \phi_2 \leq \phi}{\Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2) \quad \phi_3 \leq \phi}$$

We know via induction on Ξ_1 that

$$\begin{aligned} \Xi_1' \ :: \ & \Gamma' \rhd e_1 : (\text{bool}, \Delta') && (75)\\ & \Delta' \subseteq \Delta && (76) \end{aligned}$$

By induction on Ξ_2, we know

$$\begin{aligned} \Xi_2' \ :: \ & \Gamma' \rhd e_2 : (\phi_2', \Delta_1') && (77)\\ & \phi_2' \leq \phi_2 && (78)\\ & \Delta_1' \subseteq \Delta_1 && (79) \end{aligned}$$

By induction on Ξ_3, we know

$$\begin{aligned} \Xi_3' \ :: \ & \Gamma' \rhd e_3 : (\phi_3', \Delta_2') && (80)\\ & \phi_3' \leq \phi_3 && (81)\\ & \Delta_2' \subseteq \Delta_2 && (82) \end{aligned}$$

From (75), (77), and (80), we can construct a new derivation

$$\Xi \ :: \ \Gamma' \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi', \Delta' \bigcup \Delta_1' \bigcup \Delta_2') \tag{83}$$

where

$$\frac{\phi' \leq \phi}{\Delta' \bigcup \Delta_1' \bigcup \Delta_2' \subseteq \Delta \bigcup \Delta_1 \bigcup \Delta_2}$$

by (76), (79), and (82). This is what we needed to prove in this case.

6. Assume Ξ is

$$\frac{\Gamma(\ell) = \phi\ \mathsf{ref}\ (\Delta)}{\Gamma \rhd \ell : (\phi\ \mathsf{ref}\ (\Delta), \{\ell\})}$$

By our assumption, we know

$$\begin{aligned} \Xi_1' \ :: \ & \Gamma'(\ell) = \phi'\ \mathsf{ref}\ (\Delta') && (84)\\ & \phi' \leq \phi && (85)\\ & \Delta' \subseteq \Delta && (86) \end{aligned}$$

This is sufficient to show that

$$\Xi_2' \ :: \ \Gamma' \rhd \ell : (\phi'\ \mathsf{ref}\ (\Delta'), \{\ell\}) \tag{87}$$

where

$$\phi'\ \mathsf{ref}\ (\Delta') \leq \phi'\ \mathsf{ref}\ (\Delta') \tag{88}$$

by (85), (86), and $\{\ell\} \subseteq \{\ell\}$ trivially.

7. Assume Ξ is

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta_1)\end{array}}{\Gamma \rhd !e : (\phi, \Delta \bigcup \Delta_1)}$$

We know by induction on Ξ_1 that

$$\begin{aligned} \Xi_1' \quad :: \quad & \Gamma' \rhd e : (\phi'\ \mathsf{ref}\ (\Delta'), \Delta_1') && (89)\\ & \phi' \leq \phi && (90)\\ & \Delta' \subseteq \Delta && (91)\\ & \Delta_1' \subseteq \Delta_1 && (92) \end{aligned}$$

From (89), we have a new derivation

$$\Xi_2' \quad :: \quad \Gamma' \rhd !e : (\phi', \Delta' \bigcup \Delta_1')$$

We know the type and sets Δ in (93) are smaller than those in Ξ by (90), (91), and (92).

8. Assume Ξ is

$$\frac{\begin{array}{cc}\Xi_1 & \Xi_2\\ \Gamma \rhd e_1 : (\phi\ \ \mathsf{ref}\ \ (\Delta), \Delta_1) & \Gamma \rhd e_2 : (\phi, \Delta_2)\end{array}}{\Gamma \rhd e_1 := e_2 : (\mathsf{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

We know via induction on Ξ_1 that

$$\begin{aligned} \Xi_1' \quad :: \quad & \Gamma' \rhd e_1 : (\phi'\ \mathsf{ref}\ \Delta', \Delta_1') && (93)\\ & \phi' \leq \phi && (94)\\ & \Delta' \subseteq \Delta && (95)\\ & \Delta_1' \subseteq \Delta_1 && (96) \end{aligned}$$

By induction on Ξ_2, we have

$$\begin{aligned} \Xi_2' \quad :: \quad & \Gamma' \rhd e_2 : (\phi', \Delta_2') && (97)\\ & \phi' \leq \phi && \\ & \Delta_2' \subseteq \Delta_2 && (98) \end{aligned}$$

We can use (93) and (97) to construct a new derivation

$$\Xi_3' \quad :: \quad \Gamma \rhd e_1 := e_2 : (\mathrm{unit}, LS(\phi) \bigcup \Delta' \bigcup \Delta_1' \bigcup \Delta_2') \qquad (99)$$

We know that $\mathsf{unit} \leq \mathsf{unit}$ trivially, and we know that the Δ set for (99) is smaller than the set in Ξ, by (94), (95), (96), and (98), which is what we needed to show.

9. Assume Ξ is

$$\frac{\Xi_j \quad \Xi_k \quad \Gamma\{\overline{g_i : \phi_i}\} \rhd \overset{\Xi'}{e}[\overline{g_i/f_i}] : (\phi, \Delta)}{\Gamma \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta_{lf})}$$

where

$$\begin{array}{rcl}
\Xi_j & :: & \Gamma \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \\
\Xi_k & :: & \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_k, \{g_i\}), 1 \leq i \leq n \\
 & & \Delta \bigcup \overline{\Delta_j} \bigcup \overline{LS(\phi_j)} \bigcup \overline{LS(\phi_k)} - \{\overline{\ell_j} \bigcup \overline{g_i}\} \\
 & & LS(\phi_k) \bigcap \{\overline{\ell_j}\} \biguplus \{\overline{g_i}\} = \{\} \\
 & & LS(\phi) \bigcap \{\overline{g_i}\} = \{\}
\end{array}$$

We know via induction on Ξ_j that

$$\begin{array}{rcll}
\Xi'_1 & :: & \Gamma' \rhd a_j : (\phi'_j, \Delta'_j), 1 \leq j \leq m & (100) \\
 & & \phi'_j \leq \phi_j & (101) \\
 & & \Delta'_j \subseteq \Delta_j & (102)
\end{array}$$

Because $\Gamma'\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \leq \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\}$, by induction on Ξ_k, we can conclude that

$$\begin{array}{rcll}
\Xi'_2 & :: & \Gamma'\{\overline{\ell_j : \phi \ \ \mathsf{ref}\ \ (\Delta)}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi'_k, \{g_i\}) & (103) \\
 & & 1 \leq i \leq n & \\
\phi'_k \leq \phi_k & & &
\end{array}$$

From induction on Ξ' and the fact that $\Gamma'\{\overline{g_i : \phi_i}\} \leq \Gamma\{\overline{g_i : \phi_i}\}$, we know

$$\begin{array}{rcll}
\Xi'_3 & :: & \Gamma'\{\overline{g_i : \phi_i}\} \rhd e[\overline{g_i/f_i}] : (\phi', \Delta') & (104) \\
 & & \phi' \leq \phi & (105) \\
 & & \Delta' \subseteq \Delta & (106)
\end{array}$$

From (100), (103), and (104), we have a new deduction

$$\begin{array}{rcl}
\Xi'_4 & :: & \Gamma' \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi', \Delta_{lf'} \\
\Delta_{lf'} & = & \Delta' \bigcup \overline{\Delta'_j} \bigcup \overline{LS(\phi'_j)} \bigcup \overline{LS(\phi'_k)} - \{\overline{\ell_j} \bigcup \overline{g_i}\})
\end{array}$$

We know that the Δ set in (107) is less than the Δ set in Ξ from (101), (102), (104), (105), and (106). This is what we needed to show.

10. Assume Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi'_1, \Delta_2)}}{\Gamma \rhd e_1 \ @\ e_2 : (\phi_2, LS(\phi_1 \xrightarrow{\Delta} \phi_2) \bigcup \Delta_1 \bigcup \Delta_2)}\ \phi'_1 \leq \phi_1$$

We know via induction on Ξ_1 that

$$\Xi_1' :: \Gamma' \rhd e_1 : (\phi_1'' \xrightarrow{\Delta'} \phi_2', \Delta_1') \quad (107)$$
$$\phi_1'' \xrightarrow{\Delta'} \phi_2' \leq \phi_1 \xrightarrow{\Delta} \phi_2 \quad (108)$$
$$\Delta_1' \subseteq \Delta_1 \quad (109)$$

By induction on Ξ_2, we know

$$\Xi_2' :: \Gamma' \rhd e_2 : (\phi_1''', \Delta_2') \quad (110)$$
$$\phi_1''' \leq \phi_1'$$
$$\Delta_2' \subseteq \Delta_2 \quad (111)$$

We can construct a new derivation from (107) and (110)

$$\Xi_3' :: \Gamma \rhd e_1 \ @ \ e_2 : (\phi_2', LS(\phi_1'' \xrightarrow{\Delta'} \phi_2') \bigcup \Delta_1' \bigcup \Delta_2')$$
$$\phi_2' \leq \phi_2 \quad (112)$$

where we know our relation on they types and Δ sets holds by (108), (109), and (111). $\square$

3.3 Type Soundness Proof

Type soundness allows us to be sure that our operational semantics have the desired behavior. If we evaluate an expression, we want to be sure that the value returned is the same type as the expression itself. Our type soundness theorem can be stated as follows:

Theorem 3.1 *Conjecture If* $\Gamma \rhd e : (\phi, \Delta)$, $\rho; \sigma; e \hookrightarrow \rho'; \sigma'; v$, *and* $\rho; \sigma : \Gamma$ *then* $\rho', \sigma' \models v : \phi$ *and* $\rho', \sigma' : \Gamma$.

Until we prove Lemma 3.8, the type soundness theorem remains a conjecture. We will proceed on the assumption that the Lemma is provable.

Proof: We prove the type soundness theorem via induction on deductions Π, of the form $\rho; \sigma; e \hookrightarrow \rho'; \sigma'; v$.

1. Assume Π is $\overline{\rho; \sigma; c \hookrightarrow \rho; \sigma; c}$ and Ξ is $\overline{\Gamma \rhd c : (\phi, \{\})}$. It follows immediately that $\rho; \sigma \models c : \phi$ such that $(c : \phi) \in Constants$.

2. Assume Π is
$$\frac{}{\rho; \sigma; \lambda x.e \hookrightarrow \rho; \sigma; \lambda x.e} \ (\text{lam})$$
and Ξ is
$$\frac{\Gamma\{x : \phi_1\} \rhd e : (\phi_2, \Delta)}{\Gamma \rhd \lambda x.e : (\phi_1 \xrightarrow{\Delta} \phi_2, \{\})\Xi_1}$$
The proof follows directly that $\rho; \sigma \models \lambda x.e_1 : \phi_1 \xrightarrow{\Delta} \phi_2$ and $\rho, \sigma : \Gamma$.

3. Assume Π is $\overline{\rho; \sigma; f \hookrightarrow \rho; \sigma; f}$ and Ξ is

$$\frac{\Gamma(f) = \phi_1 \xrightarrow{\Delta} \phi_2}{\Gamma \rhd f : (\phi_1 \xrightarrow{\Delta} \phi_2, \{f\})}$$

We know that $\rho, \sigma \models \rho(f) : \Gamma(f)$ from our assumption that $\rho, \sigma : \Gamma$, which implies that $\rho, \sigma \models f : \Gamma(f)$. WE also have that $\rho, \sigma : \Gamma$. These two items are what we needed.

4. Assume Π is

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ \rho; \sigma; e_1 \hookrightarrow \rho''; \sigma''; \mathsf{true} & \rho''; \sigma''; e_2 \hookrightarrow \rho'; \sigma'; v\end{array}}{\rho; \sigma; \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \rho'; \sigma'; v}$$

and Ξ is

$$\frac{\begin{array}{ccc}\Xi_1 & \Xi_2 & \Xi_3\\ \Gamma \rhd e_1 : (\mathrm{bool}, \Delta) & \Gamma \rhd e_2 : (\phi_2, \Delta_1) & \Gamma \rhd e_3 : (\phi_3, \Delta_2)\end{array} \quad \begin{array}{c}\phi_2 \leq \phi\\ \phi_3 \leq \phi\end{array}}{\Gamma \rhd \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\phi, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

By induction on Π_1 and Ξ_1 with our inductive assumption, we know that

$$\rho'', \sigma'' : \Gamma \tag{113}$$

We know via induction on Π_2 with Ξ_2 and (113), that

$$\begin{array}{l}\rho', \sigma' \models v : \phi_2\\ \rho', \sigma' : \Gamma\end{array} \tag{114}$$

We can conclude that

$$\rho', \sigma' \models v : \phi$$

which is what we wanted to show, along with (114).

5. Assume Π is

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ \rho; \sigma; e_1 \hookrightarrow \rho''; \sigma''; \mathsf{false} & \rho''; \sigma''; e_3 \hookrightarrow \rho'; \sigma'; v\end{array}}{\rho; \sigma; \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \rho; \sigma'; v}$$

The proof is symmetric to the previous one.

6. Assume Π is $\overline{\rho; \sigma; \ell \hookrightarrow \rho; \sigma; \ell}$ and Ξ is

$$\frac{\Gamma(\ell) = \phi}{\Gamma \rhd \ell : (\phi, \{\})}$$

From $\rho, \sigma : \Gamma$, we know that $\rho, \sigma \models \sigma(\ell) : \Gamma(\ell)$. This implies that $\rho, \sigma \models \ell : \Gamma(\ell)$, which is what we need.

7. Assume Π is

$$\frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ \rho;\sigma;e \hookrightarrow \rho';\sigma';\ell & \sigma'(\ell) = v\end{array}}{\rho;\sigma;!e \hookrightarrow \rho';\sigma';v}$$

and Ξ is

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \rhd e : (\phi\ \mathsf{ref}\ (\Delta), \Delta')\end{array}}{\Gamma \rhd !e : (\phi, \Delta')}$$

We know via induction on Π_1 with Ξ_1 and the assumption that $\rho, \sigma : \Gamma$ that

$$\frac{\rho', \sigma' \models \ell : \phi\ \mathsf{ref}}{\rho', \sigma' : \Gamma} \tag{115}$$

By our value typing rule for a location, we know that

$$\rho', \sigma' \models \sigma'(\ell) : \phi$$

We can therefore conclude that

$$\rho', \sigma' \models v : \phi$$

8. Assume Π is

$$\frac{\begin{array}{ccc}\Pi_1 & & \Pi_2\\ \rho;\sigma;e_1 \hookrightarrow \ell;\rho'';\sigma'' & \rho'';\sigma'';e_2 \hookrightarrow v;\rho';\sigma''' & \sigma'''(\ell, v) = \sigma'\end{array}}{\rho;\sigma;e_1 := e_2 \hookrightarrow \rho';\sigma';()}$$

and Xi is

$$\frac{\begin{array}{c}\Xi_1\\ \Gamma \rhd e_1 : (\phi\ \ \mathsf{ref}\ \ (\Delta), \Delta_1)\end{array}}{\Gamma \rhd e_1 := e_2 : (\mathrm{unit}, LS(\phi) \bigcup \Delta \bigcup \Delta_1 \bigcup \Delta_2) \Gamma \rhd e_2 : (\phi, \Delta_2) \Xi_2}$$

The proof follows directly, since the unit type is one of our base types. We also know that $\rho', \sigma' : \Gamma$ by straightforward induction on Π_1 and Π_2.

9. Assume Π is

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\Pi_j}\\ \multicolumn{2}{c}{\rho;\sigma;a_j \hookrightarrow;\rho'';\sigma'';v_j} & \Pi'\end{array}}{\rho, \sigma; \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e \hookrightarrow \rho';\sigma';v}$$

where

$$\Pi' :: \rho''\langle\overline{(g_i, e_i'\overline{[g_i/f_i][\ell_j/x_j]})}, \overline{(\ell_j, v_j)}\rangle;\sigma'';e\overline{[g_i/f_i]} \hookrightarrow \rho'\langle\overline{(g_i, e_i'\overline{[g_i/f_i][\ell_j/x_j]})}, \overline{(\ell_j, v_j)}\rangle;\sigma';v$$

and Ξ is

$$\frac{\Xi_j \quad \Xi_k \quad \Gamma\{\overline{g_i : \phi_i}\} \rhd \overset{\Xi'}{e}[\overline{g_i/f_i}] : (\phi, \Delta)}{\Gamma \rhd \mathsf{letfun}\ \overline{f_i = e_i}\ \mathsf{with}\ \overline{x_j = \mathsf{sref}\ a_j}\ \mathsf{in}\ e : (\phi, \Delta \bigcup \overline{\Delta_i} - \{\overline{\ell_j} \bigcup \overline{g_i}\})}$$

where

$$\begin{array}{rcl} \Xi_j & :: & \Gamma \rhd a_j : (\phi_j, \Delta_j), 1 \leq j \leq m \\ \Xi_k & :: & \Gamma\{\overline{\ell_j : \phi_j \ \ \mathsf{ref}\ \ \{\ell_j\}}, \overline{g_i : \phi_i}\} \rhd e_k[\overline{\ell_j/x_j}][\overline{g_i/f_i}] : (\phi_i, \{\}), 1 \leq i \leq n \\ LS(\phi_k) \bigcap (\{\overline{\ell_j}\} \biguplus \{\overline{g_i}\}) & = & \{\} \\ LS(\phi) \bigcap \{\overline{g_i}\} & = & \{\} \end{array}$$

We know via induction on the Π_js with the Ξ_js and the assumption that $\rho, \sigma : \Gamma$ that

$$\begin{array}{l} \rho'', \sigma'' \models v_j : \phi_j \\ \rho'', \sigma'' : \Gamma \end{array} \tag{116}$$

for all $1 \leq j \leq n$. We must show that

$$\rho_1, \sigma'' : \Gamma_1 \tag{117}$$

where

$$\begin{array}{rcl} \rho_1 & = & \\ \rho''\langle\overline{(g_i, \overline{e_i'[g_i/f_i][\ell_j/x_j]})}, \overline{(\ell_j, v_j)}\rangle & & \\ \Gamma_1 & = & \\ \Gamma\{\overline{g_i : \phi_i}\} & & \end{array}$$

First we must know that

$$dom(\rho_1) \biguplus dom(\sigma'') = dom(\Gamma_1) \tag{118}$$

We already know that

$$dom(\rho) \biguplus dom(\sigma'') = dom(\Gamma)$$

Moreover, for every g_i we add to ρ to get ρ', we add a corresponding typing in Γ to get Γ'. Hence, (118) holds. We must show that

$$\forall \ell \in dom(\sigma''), \rho_1, \sigma'' \models \sigma''(\ell) : \Gamma_1(\ell) \tag{119}$$

$$\forall g \in dom(\rho_1), \rho_1, \sigma'' \models \rho_1(g) : \Gamma_1(g) \tag{120}$$

For (119), we notice that

$$LS(\Gamma_1(\ell)) \quad \subseteq \quad dom(\sigma'') \biguplus dom(\rho') \tag{121}$$

Hence, we know that (119) follows from Lemma 3.4 and (121). We have two cases for (120). The first case is when g is not one of the g_is. We know (121) holds for these gs, from Lemma 3.4 and (121). The other case is when the g_is are the ones being defined in the letfun expression. In this case, we must use co-induction, as described by Milner and Tofte [7]. We must show

$$\forall g \in dom(\rho_1), (\rho_1, \sigma'', \rho_1(g), \Gamma_1(g)) \in Q^{max}$$

Let

$$Q_0 \quad = \quad Q^{max} \bigcup \{(\rho_1, \sigma'', \rho_1(g_i), \Gamma_1(g_i))\} \text{ for } 1 \leq i \leq n$$

We need to show that Q_0 is consistent, i.e., $Q_0 \subseteq F(Q_0)$. For all $q \in Q^{max}$, $q \in F(Q_0)$ follows from Q^{max} being a fixed point.

$$Q^{max} \quad \subseteq \quad Q_0$$

by definition. Since F is monotonic, we know that

$$F(Q^{max}) \quad \subseteq \quad F(Q_0)$$

Moreover, since Q^{max} is a fixed point,

$$Q^{max} \subseteq F(Q_0)$$

Therefore, all $q \in Q^{max}$ are also in $F(Q_0)$. Now we are left with showing that

$$(\rho_1, \sigma'', \rho_1(g_i), \Gamma_1(g_i)) \in F(Q_0) \text{ for } 1 \leq i \leq n$$

Let

$$\begin{array}{rcl} \rho(g_i) & = & (e_i, \sigma_e, \rho_2) \\ \Gamma(g_i) & = & \phi_i \end{array} \tag{122}$$

We need to show that

$$(\rho_2, \sigma''\sigma_e, e_i, \phi_i) \in F(Q_0) \text{ for } 1 \leq i \leq n$$

To prove (122) we must show that there exists a Γ' and ϕ_i' such that

$$\begin{array}{rcl} dom(\Gamma') & = & dom(\rho_2) \biguplus dom(\sigma''\sigma_e) \\ & & \Gamma' \rhd e_i \rhd (\phi_1', \{\}) \end{array}$$

We know this is true from the typing derivations we have, Ξ_k. Now we must show that

$$\forall \ell \in \sigma'' \sigma_e, (\rho_2, \sigma'' \sigma_e, \sigma'' \sigma_e(\ell), \Gamma(\ell)) \in Q_0 \tag{123}$$
$$\forall g \in \rho_2, (\rho_2, \sigma'' \sigma_e, \rho_2(g), \Gamma(g)) \in Q_0 \tag{124}$$

For (123), if $\ell \in \sigma''$, then the statement follows from (116) and Lemma 3.4. If $\ell \in \sigma_e$, then we know we have a typing derivation from Ξ_j. For (124), if $g \in Q^{max}$, then it follows directly in the same way it did for locations. If the g is one of the new ones we are defining, then we know it is in Q_0, since we explicitly defined Q_0 to include them. Now, we have proven (117). Finally, we can perform induction on Π' with Ξ' and (117) to conclude that

$$\rho_1, \sigma' \models v : \phi \tag{125}$$

Using Lemma 3.4 using (125) and $LS(\phi) \bigcap \{\overline{\ell_j} \uplus \overline{g_i}\} = \{\}$, we know that

$$\rho', \sigma' \models v : \phi$$

which is what we wanted to show.

10. Assume Π is

$$\frac{\overset{\Pi_1}{\rho; \sigma; e_1 \hookrightarrow \rho'', \sigma''; \lambda x.e_1'} \quad \overset{\Pi_2}{\rho''; \sigma''; e_2 \hookrightarrow \rho''', \sigma'''; v'} \quad \overset{\Pi_3}{\rho'''; \sigma'''; e_1'[v'/x] \hookrightarrow \rho', \sigma'; v}}{\rho; \sigma; e_1 \mathbin{@} e_2 \hookrightarrow \rho'; \sigma'; v}$$

and Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1', \Delta_2)}}{\Gamma \rhd e_1 \mathbin{@} e_2 : (\phi_2, \Delta \bigcup \Delta_1 \bigcup \Delta_2)} \; \phi_1' \leq \phi_1$$

We know by induction on Π_1 with Ξ_1 and our assumption that $\rho, \sigma : \Gamma$ that

$$\rho'', \sigma'' \models \lambda x.e_1' : \phi_1 \xrightarrow{\Delta} \phi_2$$
$$\rho'', \sigma'' : \Gamma \tag{126}$$

We also know from the way the Ξ_1 derivation was formed that we have a deduction

$$\Xi_1' \quad :: \quad \Gamma\{x : \phi_1\} \rhd e_1' : (\phi_2, \Delta) \tag{127}$$

By induction on Π_2 with Ξ_2 and (126), we know that

$$\rho''', \sigma''' \models v' : \phi_1'$$
$$\phi_1' \leq \phi_1 \tag{128}$$
$$\rho''', \sigma''' : \Gamma \tag{129}$$

From Lemma 3.6, we know there exists a Γ' and such that

$$\rho''', \sigma''' : \Gamma' \tag{130}$$
$$\Xi_2' \;::\; \Gamma' \rhd v' : (\phi_4, \Delta''')$$
$$\phi_4 \leq \phi' \tag{131}$$

Moreover, by the value typing definition for a λ-abstraction, there exists a Γ'', ϕ_1'', and ϕ_2'' such that

$$\rho''', \sigma''' : \Gamma'' \tag{132}$$
$$\Xi_3' \;::\; \Gamma'' \rhd \lambda x.e_1' : (\phi_1'' \xrightarrow{\Delta'} \phi_2'', \{\})$$
$$(\phi_1'' \xrightarrow{\Delta'} \phi_2'') \leq (\phi_1 \xrightarrow{\Delta} \phi_2) \tag{133}$$

From (133), we know

$$\phi_1 \leq \phi_1'' \tag{134}$$
$$\phi_2'' \leq \phi_2 \tag{135}$$

From (128) and (134), we can conclude that

$$\phi_4 \leq \phi_1''$$

By Lemma 3.8, we know there exists a Γ''' such that $\Gamma''' \leq \Gamma$, $\Gamma''' \leq \Gamma'$, and $\Gamma''' \leq \Gamma''$, as a result of (129), (131), and (132). Hence, we know from Lemma 3.9

$$\Xi \;::\; \Gamma''' \rhd \lambda x.e_1' : (\phi_1''' \xrightarrow{\Delta''} \phi_2''', \{\}) \tag{136}$$
$$\Xi \;::\; \Gamma''' \rhd v' : (\phi_1'''', \{\}) \tag{137}$$
$$\phi_1''' \xrightarrow{\Delta''} \phi_2''' \leq \phi_1'' \xrightarrow{\Delta'} \phi_2'' \tag{138}$$
$$\phi_1'''' \leq \phi_4 \tag{139}$$

By the Substitution Lemma (Lemma 3.7) using (136), (137), and (139), we know that

$$\Xi_3 \;::\; \Gamma''' \rhd e_1'[v'/x] : \phi_2''''$$
$$\phi_2'''' \leq \phi_2''' \tag{140}$$

However, it follows from (140), (138), and (135) that

$$\phi_2'''' \leq \phi_2 \tag{141}$$

Via induction on Π_3 with Ξ_3 and (129), we can conclude

$$\rho', \sigma' \models v : \phi_2'''' \tag{142}$$

From Lemma 3.1 using (142) and (141), we know that

$$\rho', \sigma' \models v : \phi_2$$

This is the what we needed to prove.

11. Assume Π is

$$\frac{\overset{\Pi_1}{\rho;\sigma;e_1 \hookrightarrow \rho'';\sigma'';f} \quad \overset{\Pi_2}{\rho'';\sigma'';e_2 \hookrightarrow \rho''';\sigma''';v'} \quad \rho'''(f) = (\sigma_f, \lambda x.e_i', \rho'''') \quad \Pi_3}{\rho;\sigma;e_1 \mathbin{@} e_2 \hookrightarrow \rho'(f,\sigma_f');\sigma';v}$$

where

$$\Pi_3 :: \rho'''';\sigma'''\sigma_f;e_i'[v'/x] \hookrightarrow \rho';\sigma'\sigma_f';v$$

and that Ξ is

$$\frac{\overset{\Xi_1}{\Gamma \rhd e_1 : (\phi_1 \xrightarrow{\Delta} \phi_2, \Delta_1)} \quad \overset{\Xi_2}{\Gamma \rhd e_2 : (\phi_1', \Delta_2)}}{\Gamma \rhd e_1 \mathbin{@} e_2 : (\phi_2, \Delta \bigcup \Delta_1 \bigcup \Delta_2)} \; \phi_1' \leq \phi_1$$

We know from induction on Π_1 with Ξ_1 and our assumption that $\rho, \sigma : \Gamma$ that

$$\begin{aligned} &\rho'', \sigma'' \models f : \phi_1 \xrightarrow{\Delta} \phi_2 \\ &\rho'', \sigma'' : \Gamma \end{aligned} \tag{143}$$

By induction on Π_2 with Ξ_2 and (143)

$$\begin{aligned} &\rho''', \sigma''' \models v' : \phi_1' \\ &\rho''', \sigma''' : \Gamma \end{aligned} \tag{144}$$

From (144), we know using Lemma 3.6 that there exists a Γ' such that

$$\begin{aligned} &\rho''', \sigma''' : \Gamma' \\ \Xi_1' \quad :: \quad &\Gamma' \rhd v' : (\phi_4, \Delta''') \\ \phi_4 \leq \phi_1 & \end{aligned} \tag{145}$$

By the $\models$ relation for a function closure, we know

$$LS(\phi_2'') \bigcap dom(\sigma_f) \quad = \quad \{\} \tag{146}$$

From the $\models$ relation for a λ-abstraction, there exists a Γ'', ϕ_1'', and ϕ_2'' such that

$$\rho'''', \sigma'''\sigma_f : \Gamma'' \tag{147}$$

$$\Xi_2' \quad :: \quad \Gamma'' \rhd \lambda x.e_i' : (\phi_1'' \xrightarrow{\Delta'} \phi_2'', \{\}) \tag{148}$$

$$(\phi_1'' \xrightarrow{\Delta'} \phi_2'') \leq (\phi_1 \xrightarrow{\Delta} \phi_2) \tag{149}$$

From (149), we know

$$\phi_1 \leq \phi_1'' \tag{150}$$

$$\phi_2'' \leq \phi_2 \tag{151}$$

From the fact that $\phi_1' \leq \phi_1$ and (150), we can conclude that

$$\phi_1''' \leq \phi_1'' \tag{152}$$

By Lemma 3.3, we know

$$\begin{aligned} \Xi_3' \quad :: \quad & \Gamma'' \rhd v' : (\phi_1'''', \{\}) && (153)\\ & \phi_1''' \xrightarrow{\Delta''} \phi_2''' \leq \phi_1'' \xrightarrow{\Delta'} \phi_2'' && (154)\\ & \phi_1'''' \leq \phi_1''' && (155) \end{aligned}$$

We also know from the way the Ξ_2' derivation was formed that we have a deduction

$$\Xi_4' \quad :: \quad \Gamma''\{x : \phi_1''\} \rhd e_i' : (\phi_2'', \Delta') \tag{156}$$

By the Substitution Lemma (Lemma 3.7) using (156), (153), and (155), we know that

$$\begin{aligned} \Xi_3 \quad :: \quad & \Gamma'' \rhd e_i'[v'/x] : \phi_2''' && (157)\\ & \phi_2''' \leq \phi_2'' && (158) \end{aligned}$$

However, it follows from (158), (154), and (151) that

$$\phi_2'''' \leq \phi_2 \tag{159}$$

Hence, via induction on Π_3 with Ξ_3 and (147), we can conclude

$$\rho', \sigma'\sigma_f' \models v : \phi_2'''' \tag{160}$$

We know from (146) and (159) that

$$LS(\phi_2'''') \bigcap dom(\sigma_f) \quad = \quad \{\} \tag{161}$$

From Lemma 3.4 using (160) and (161), we know that

$$\rho', \sigma' \models v : \phi_2'''' \tag{162}$$

From (162) and (159) using Lemma 3.1), we know that

$$\rho', \sigma' \models v : \phi_2$$

This is exactly what we needed to show for this case. □

4 Examples

We can use the language we have created in this thesis to represent many examples we have seen in other articles related to the topic of reasoning about references. However, we first try to justify some of the operational semantics with an example.

4.1 Reasons for a Single Copy of a State

If we did not make sure that only one copy of a static reference exists, our language may exhibit undesirable characteristics. Recall that our original application rule for functions with a private state is

$$\frac{\begin{array}{l}\rho;\sigma;e_1 \hookrightarrow \rho'';\sigma'';f \\ \rho'';\sigma'';e_2 \hookrightarrow \rho''';\sigma''';v' \\ \rho'''(f) = (\sigma_f, e_i', \rho'''') \\ \rho'''';\sigma'''\sigma_f;e_i'[v'/x] \hookrightarrow \rho';\sigma'\sigma_f';v\end{array}}{\rho;\sigma;e_1 \mathbin{@} e_2 \hookrightarrow \rho'(f,\sigma_f');\sigma';v} \text{ (ap2)}$$

Consider if our *ap2* rule was of the following form:

$$\frac{\begin{array}{l}\rho;\overline{\sigma};e_1 \hookrightarrow \rho'';\overline{\sigma}'';((f,e_i'),\sigma_f) \\ \rho'';\overline{\sigma}'';e_2 \hookrightarrow \rho''';\overline{\sigma}''';v' \\ \rho''';\overline{\sigma}'''\sigma_f;e_i'[v'/x] \hookrightarrow \rho';\overline{\sigma}'\sigma_f';v\end{array}}{\rho;\overline{\sigma};e_1 \mathbin{@} e_2 \hookrightarrow \rho'(f,\sigma_f');\overline{\sigma}';v} \text{ (ap2\_b)}$$

where $((f, e_i'), \sigma_f$ is the function closure for the function f. Consider the evaluation of this code:

```
letfun  total =  λx.(a := !a + x; !a)
with      a =    sref 0
in               total @ ((total @ 3))
```

Our real operational semantics returns the value 6. For the body of the expression, the *ap2* rule evaluates total to itself. Next the argument is resolved using the *ap2* rule again. The function total evaluates to itself, as does the argument

3. Now we look up the state and function body in the environment. The state returned contains the single static reference a with a value of 0, which is added to the state. After adding 3 to the value of a, the reference is written back to the state. Now that the execution of (total @ 3) is complete, the state is removed from the state and written back to the function closure in the environment.

The value returned for this part of the expression is 3, which is then used as the argument to the outer execution of the total function. To execute this function, we retrieve the function's body and private state from the environment similar to the inner execution. However, the value of the reference a is now 3. We execute the body of the function, again similar to the previous execution. Finally, the expression returns the value 6.

On the other hand, the incorrect operational semantics would return the value 3. When the outer total is seen, the private state is immediately retrieved from the environment. Although the execution of (total @ 3) will update the reference a, the outer total will not see the update, since the state was already retrieved for it. Consequently, the value of a is still 0, so the value returned is 3.

4.2 Pitts Examples

Now we can encode the examples in Section 1 into our language and type check them. The first example of expressions p and m would be encoded like this:

```
letfun  p =  λx.(a := !a + x; !a)
with    a =  sref 0
in       ...

letfun  q =  λy.(b := !b - y; 0 - !b)
with    b =  sref 0
in       ...
```

These expressions would type properly, assuming the body of the expressions did not contain any direct accesses to the static references and the body typed properly. However, we can see that in the other example with the expressions f and g, a representation of these functions in our language would not type.

```
letfun  f =  λx.(if x == a then b else a)
with    a =  sref 0
        b =  sref 0

letfun  g =  λy.(if y == d then d else c)
with    c =  sref 0
        d =  sref 0
in       ...
```

In both cases, the static references are able to escape via the function f or g. Our type system will realize this when it checks the intersection of the type of f and g with the contents of the set of static cells for b or d, respectively. The set will be non-empty, since these static reference cells can be returned by the functions. Consequently, these functions cannot be converted to functions with static references.

5 Encoding In Elf

To reason about our new static reference and private state, we have encoded a basic version of the language into Elf [6]. Since we do not allow functions with private state to escape, we can use Elf's eigenvariables to generate unique names for functions and locations. For the initial coding, we will only allow one function and one static reference in a letfun declaration.

5.1 Expressions and Values

Expressions translate into Elf rather intuitively. The only expression that requires some extra care is the letfun expression, which is of the form
letfun $\overline{f = e}$ with $\overline{x = \mathsf{sref}\ e}$ in e. We can encode the letfun expression as:

```
letfun : (ident -> ident -> exp) -> exp
  -> (ident -> exp) -> exp.
```

The `ident` is meant to be a unique identifier associated with each declaration of functions with private state. The question is, how do we get these unique identifiers? We can use Elf's pi variables to create the unique identifiers. However, these new constants are going to go out of scope whenever outside the pi declaration. Hence, we abstract out the identifiers for the function names and the reference cell names so that unique identifiers can be created every time one of the functions is applied. However, the names need not be abstracted out of the reference cells, since they can refer to neither the functions nor the static references.

These variables that are abstracted out will also be used when applying one of these functions. Such a function is encoded as:

```
fun : ident -> exp.
```

Static references have a representation for the same reason as functions with private state. All of the code for expressions and values can be found in Appendix A.1.

5.2 Closures, States, and Environments

We can use Elf's pi variables to generate the unique identifiers we need for functions and their states. We abstract the unique identifier for the private state out of function closures in the environment so that they can be generated by Elf's eigenvariables whenever a function with private state is called.

A function closure is encoded as follows:

```
funclo_ls : (ident -> exp) -> state -> funclo.
```

The `exp` is the function body. The `state` is the private state for the functions, which, when containing items, is of the form:

```
list_state : exp -> state.
```

The empty state is also a state. The environment is a list of these function closures, so it looks like:

```
list_env  : env -> ident -> funclo -> env.
```

where ident is the unique identifier associated with funclo.

Our state mirrors our notion of a stack of states. Note that the abstraction from the function closure does not exist in the state σ. The reason is that a private state is only added to the state when a function that uses that private state is called. Consequently, the unique identifier will have already been generated and accessible. The state takes the form:

```
list_states : states -> ident -> state -> states.
```

where ident is the unique identifier associated with state.

See Appendix A.2 for all the Elf code pertaining to function closures, states, and environments.

5.3 Evaluation

Our operational semantics can be encoded in Elf with only a few minor changes, namely the introduction of pi constants to generate unique identifiers in rules associated with functions with private state. In general, our evaluation function mirrors the form of the judgment for our operational semantics:

```
eval : env -> states -> exp
-> env -> states -> exp -> type.
```

Evaluation takes an initial environment `env`, collection of states `states`, and an expression `exp` to a value `exp`, a new environment `env`, and a new collection of states `states`. The *letfun* and *ap2* rules are worth noting in particular. Refer to Figure 4 for the operational semantics of the *letfun* function. Its encoding in Elf is as follows:

```
eval_lets : eval Env Ste (letfun Fun Er E) Env' Ste' V
             <- eval Env Ste Er Env'' Ste'' Vr
             <- {f:ident} eval (list_env Env'' f
              (funclo_ls (Fun f) (list_state Vr)))
              Ste'' (E f) (list_env Env' f Funclo) Ste' V.
```

Each line of the code corresponds exactly to the parts of the operational semantics: In order to evaluate the letfun expression, first the static reference must be evaluated, then the expression `e` must be evaluated, with the function being given a unique identifier. We generate this unique identifier via the eigenvariable, i.e., {`f:ident`}. Once we have the new constant f, we evaluate the expression `e`

applied to the new identifier. After the final evaluation, the unique variable will go out of scope and hence cannot be returned as a part of `V`. Additionally, the environment can contain no references to `f`, so we must remove it before returning the environment of the entire letfun expression. This is exactly the kind of behavior we wanted.

The *ap2* rule will take advantage of the unique quantifier we have created via the `eval_lets` rule:

```
eval_@2  : eval Env Ste (E1 @ E2) Fenv Ste' V
            <- eval Env Ste E1 Env'' Ste'' (fun F)
            <- eval Env'' Ste'' E2 Env''' Ste''' V'
            <- lookup_fun F Env''' Sig_n E1' Env''''
            <- {x:ident} eval Env'''' (list_states Ste''' x Sig_n)
               ((E1' x) @ V') Env' (list_states Ste' x Sig_n') V
            <- update_env Env' F Sig_n' Fenv.
```

Since functions with private state cannot be returned as values, we know that any application of such a function will be in the body of a letfun expression. Consequently, we know the unique identifier for the function will have already been created. Therefore, we retrieve the function from the environment and its private state using the `lookup_fun` function. We need to create the additional eigenvariable x to refer to the static references. At this point, we can evaluate the application and then update the environment with the new private state.

5.4 Type Checking

The encoding of the type system and a type checker in Elf follows directly from our rules. We have declarations for the type environment Γ and the set of static reference cells Δ:

```
list_gamma  : typepair -> gamma -> gamma.
delta_list   : cell -> delta -> delta.
```

The `typepair` item is the mapping of an expression to a type, and is encoded as:

```
typepair_ls : exp -> tp -> typepair.
```

One of the functions we have in our typing system is the $\leq$ relation, which is defined as follows for the function type, for example:

```
lessthan_fun   : lessthan (--> Phi1 Delta Phi2) (--> Phi1' Delta' Phi2')
                  <- lessthan Phi1' Phi1
                  <- lessthan Phi2 Phi2'
                  <- subset Delta Delta'.
```

Our type inference rules translate into Elf as the evaluation rules did. The most interesting of the rules is the letfun rule, which takes advantage of Elf's eigenvariables:

```
typeof_letfun : typeof Gamma (letfun E1 E2 E3) Phi (Delta u Delta_r)
                 <- typeof Gamma E2 Phi_r Delta_r
                 <- ({f:ident} {x:ident} {c:cell} typeof (list_gamma
                  (typepair_ls (fun f) (--> Phi_f1 Delta_f Phi_f2))
                  (list_gamma (typepair_ls (ref x) (Phi_r ref_tp
                  (delta_list c empty_delta))) empty_gamma)) (E1 f x)
                  (--> Phi_f1 Delta_f Phi_f2) Delta_f)
                 <- ({f:ident} {x:ident} typeof (list_gamma
                  (typepair_ls (fun f) (--> Phi_f1 Delta_f Phi_f2))
                  Gamma) (E3 f) Phi Delta).
```

All three parts of the typing rule are there: typing the reference, typing the function body, and typing the body of the letfun expression. However, we need not explicitly check that $\mathsf{LS}(\phi) \cap \{\overline{c}\} = \{\}$. We get this check for free, since we generate a new name for the cell, which will go out of scope once we have left the typing rule for the function. In fact, we do not need to define the LS function at all, since this is the only place we used it in the typing rules.

However, we must take special care in typing $e_1 := e_2$ in our Elf encoding. If we are not careful, no function we try to type will work, because of the way we generate new cell names. Recall that the type inference rule for update is:

$$\frac{\Gamma \rhd e_1 : (\phi \ \ \mathsf{ref} \ \ (\Delta), \Delta_1) \quad \Gamma \rhd e_2 : (\phi, \Delta_2)}{\Gamma \rhd e_1 := e_2 : (\text{unit}, \Delta \bigcup \Delta_1 \bigcup \Delta_2)}$$

The set of cells returned in the entire typing includes the set Δ. However, in our Elf code, this would not be possible, since the unique identifier for the cell will go out of scope. Therefore, we extract out the identifier for the cell and replace it with an arbitrary one, called `dummy`. Since update does not itself have ability to return a reference, we know that it is okay to make this change. Our encoding of update looks like:

```
typeof_up : typeof Gamma (up E1 E2) unit_tp ((delta_list dummy Delta')
      u Delta1 u Delta2)
             <- typeof Gamma E1 (Phi ref_tp (delta_list C Delta')) Delta1
             <- typeof Gamma E2 Phi Delta2.
```

The entire Elf encoding of the type system can be found in Appendix A.4.

5.5 Examples

We want to ensure that our encoding in Elf represents our examples correctly. Let us start with some simple expressions:

letfun	f =	λy.x	`letfun ([f:ident] [x:ident]`
with	x =	sref c	`(lam [y:exp] ((ref x)))`
in	f @ c		`c`
			`([f:ident] (fun f @ c)))`

letfun	f =	λy.!x	`letfun ([f:ident] [x:ident]`
with	x =	sref c	`(lam [y:exp] (! (ref x)))`
in	f @ c		`c`
			`([f:ident] (fun f @ c)))`

The first expression will not type, which our Elf program confirms. The reason is that the function f returns the static reference. When the system attempts to type this expression, the cell associated with it goes out of scope, and thus cannot be returned in the set Δ.

On the other hand, the second expression does type as `iota` with the set `(empty_delta u empty_delta u empty_delta) u empty_delta`, which is obviously the empty set. This is exactly the type the expression should have. Furthermore, when we evaluate the expression, we get the value c, the value we expect the expression to return.

The expression should also fail to type if a static reference is used in the body of the letfun expression. Consider the following expression and its encoding:

letfun	f =	λy.x	`letfun ([f:ident] [x:ident]`
with	x =	sref c	`(lam [y:exp] ((ref x)))`
in	!x		`c`
			`([f:ident] (! (ref x))))`

Elf will not even attempt to type check this, returning the error "Undeclared constant x," since x is free in the body of the letfun expression. The same is true when we type check the expression by hand: the x should not be in scope.

In order to look at more complex examples, let us consider the encoding of the first Pitts example, the functions p and m. The two functions would appear in Elf as:

```
letfun ([f:ident] [x:ident]
       (lam [y:exp] (up (ref x) (plus (! (ref x)) y)) ; (! (ref x))))
       (int 0)
       ([f:ident] (fun f) @ (int 3)))
```

```
letfun ([f:ident] [x:ident]
       (lam [y:exp] (up (ref x) (minus (! (ref x)) y)) ;
       (minus (int 0) ! (ref x))))
       (int 0)
       ([f:ident] (fun f) @ (int 3))
```

We have arbitrarily applied each function to the number 3. When type checked and evaluated, both functions return the same type and value. As one would expect, the type of the expressions is `int_tp` with the set `(((delta_list dummy empty_delta...)))`, where the rest of the set is the union of more `empty_delta`s. Note that the set Δ includes the `dummy` item. The inclusion of this item is a result of the update on the reference, although we do not regard `dummy` as referring to the cell for `x`. Consequently, the typing of the expressions do conform to our constraint that the reference cells declared for the static references are subtracted out of the set Δ. The value of the expressions is `int 3`.

6 Related Work

Others have approached the problem of languages with references. A notion of local state even existed in Scheme, as described by Abelson and Sussman [1]. They use as an example withdrawing money from a bank account. In this example, the amount of money in the account is maintained as a local variable, since we do not want others to be able to access it. They define a function `new-withdraw` to encapsulate the balance of the account in function:

```
(define new-withdraw
  (let ((balance 100))
     (lambda (amount)
        (if (>= balance amount)
            (sequence (set! balance (- balance amount))
                      balance)
            ‘‘Insufficient funds’’))))
```

We can even encode the example in our language.

```
letfun   new-withdraw =   λ x.if (!balance >= x)
                          then (balance := !balance - x) ; !balance
                          else -1
with     balance =        sref 100

in       ...
```

Mason and Talcott consider a λ-calculus extended with the state operations [4, 5]. They define a formal system with one conclusion of the form $\Sigma \vdash \Phi$, where Σ is the set of constraints and Φ is an assertion. Mason and Talcott give this system an operational semantics, which they use to prove the equivalence of certain expressions. They also consider the relation $\sqsubseteq^{ciu}$, meaning that all closed instantiations of all uses are trivially approximate. as a method of proving contextual equivalence of expressions [5].

In another work with Honsell and Smith, Mason and Talcott mention the difficultly in making assertions about contexts when the have the ability to alter references [3]. As a solution, they propose localizing the statements they make about contextual equivalence in their logic. This is exactly the approach we have taken in this thesis, limiting the use of references.

Throughout this thesis, we have referred to Pitts' work [8, 10, 9]. In one paper, Pitts and Stark consider a language that uses local state, called ReFS. Although the references are used at a local level–which allows them to use a local invariant to reason about expression equivalence–the references are still accessible globally in certain contexts. Moreover, Pitts and Stark consider only references to integers, with the hope of expanding their definition to references of any type.

In one paper, Pitts explains the difficulty in determining contextual equivalence [9]. As he explains, the definition of contextual equivalence is unclear for two reasons:

1. What is a "complete program context"?

2. What do we consider to be "observable behavior"?

For his paper, Pitts defines a "complete program context" to be any well-typed expression. Furthermore, he defines the "observable results" by a recursive function *obs(v,s)*, where v is a value s is a state. Although Pitts says its unlikely redefining the notion of "observable results" would impact contextual equivalence, he still mentions it as a possibility. Our hope is that we can avoid the ambiguity in the definition of contextual equivalence and move towards a notion of functional equivalence.

7 Future Work

This thesis has set up the basis for more easily reasoning about programming languages with state. The language could have many applications. One application might be the design of an Object-Oriented ML. In this section, we talk about the work planned with regard to the language.

7.1 Reasoning about Equivalence with Private State

Our goal from the beginning has been to improve the ability to reason about equality in a language that contains state. Although the language we have defined is more specific in its use of state than traditional semantic definitions, the language alone does not make reasoning about state easier. Consider two notions of equivalence typically used:

1. **Functional Equivalence**: Two functions are equivalent if they yield identical outputs for identical inputs.

2. **Contextual Equivalence**: Two functions are equivalent if they have the same observable behavior in any complete program context.

Our notion of private state does not move us completely into the realm of functional equivalence. However, if two functions access only their private state, then the only states we need to consider in showing equivalence are these private states. We know what the contents of these private states will be, since we explicitly define them. Consequently, we need not resort to contextual equivalence to prove that two functions are equivalent if they only use private state.

A relation still must exist between the private states. The relation is much simpler than the relation between states used by Pitts. Consider the example from

Pitts we have been using. The function f_1 shall be defined as fun(x : int) -> (a := !a + x ; !a) and its state σ_1 be the private state for the reference a. The function f_2 is defined as fun(y : int) -> (b := !b - y ; 0 - !b), with its state σ_2, containing the reference b. We must define an invariant relation between the states: $R(\sigma_1, \sigma_2)$ iff $\sigma_1(a) = -\sigma_2(b)$. Moreover, there must exist a similar relation between the environments: $R(\rho_1, \rho_2)$. With these relations defined, we can state the definition of equivalence between f_1 and f_2:

Definition 7.1 *$f_1 \equiv_R f_2$ iff $\forall x, \rho_1, \rho_2, \sigma_1, \sigma_2$, where $R(\rho_1, \rho_2)$ and $R(\sigma_1, \sigma_2)$,*

1. *$\forall \rho_1', \sigma_1', v$, if $\rho_1, \sigma_1, (f_1 x) \hookrightarrow \rho_1', \sigma_1', v$ then there exist ρ_2', σ_2' such that $\rho_2, \sigma_1, (f_2 x) \hookrightarrow \rho_2', \sigma_2', v$ and $R(\rho_1', \rho_2')$ and $R(\sigma_1', \sigma_2')$;*
2. *$\forall \rho_2', \sigma_2', v$, if $\rho_2, \sigma_2, (f_2 x) \hookrightarrow \rho_2', \sigma_2', v$ then there exist ρ_1', σ_1' such that $\rho_1, \sigma_1, (f_1 x) \hookrightarrow \rho_1', \sigma_1', v$ and $R(\rho_1', \rho_2')$ and $R(\sigma_1', \sigma_2')$*

The definition seems to follow well the informal argument we use for the equivalence of these two functions. Proving that this definition is correct shall be addressed in future research.

7.2 Translation from a Standard Operational Semantics

Another goal of our work is to design a translation relation to convert from an ML language with only global state to one with private state. With the translation, one need not worry about explicitly programming static references, yet could still benefit from the properties of static references. The translation function, $\xrightarrow{tr}$, converts all ref declarations into sref declarations if the references are static. The translation also partitions the global state into private states. Once the translation occurs, we should be able to reason about the expression more easily, since it would have a private state.

7.3 Correctness with Regard to a Standard Operational Semantics

Another goal of our work is to demonstrate the correctness of our operational semantics with regard to a traditional operational semantics. What we need is the ability to translate from a traditional operational semantics to our semantics, and then make sure the typing and evaluation yield the same type and value, respectively. However, we must consider certain issues when comparing two expressions. For instance, how do we handle expressions that contain references that are not static? Such expressions do not have any meaning in our language right now.

7.4 Expanding the Features of the Language

The language we have created is limited, even if it does handle all of the examples we have shown. The most limiting feature is that these functions with private state are limited in scope to the body of a letfun. Our hope is to relax this restriction, allowing these functions to be the return value for a letfun and appear in blocks outside of where they are declared. In this system, our representation of σ would be better represented as a managed collection of states, as described in Section 2.2. The other goal we have with regard to expanding the language is allowing polymorphic types.

A Elf Code

A.1 Expressions

```
%use equality/integers.
%use inequality/integers.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Natural numbers for states
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nat : type.
z   : nat.
s   : nat -> nat.


%%%%%%%%%%%%%%%%%%%%%%%%
% Expressions and Values
%%%%%%%%%%%%%%%%%%%%%%%%
ident : type. %name ident I.
exp : type. %name exp E.
val : type.

% Conditional
if    : exp -> exp -> exp -> exp.
true  : exp.
false : exp.


% Function application
@   : exp -> exp -> exp.  %infix right 11 @.

% Let
let : exp -> (exp -> exp) -> exp.

% Let functions with static references
letfun : (ident -> ident -> exp) -> exp
   -> (ident -> exp) -> exp.

% Dereference
!   : exp -> exp.  %prefix 11 !.

% Update
up  : exp -> exp -> exp.
```

```
% Constants
int : integer -> exp.
c   : exp.

% Lambda abstractions
lam : (exp -> exp) -> exp.

% Static reference function
fun   : ident -> exp.

% Reference
ref   : ident -> exp.

% Unit
unit : exp.

% Built-in integer functions
plus  : exp -> exp -> exp.
minus : exp -> exp -> exp.
times : exp -> exp -> exp.
iless : exp -> exp -> exp.
imore : exp -> exp -> exp.
ieq   : exp -> exp -> exp.

;  : exp -> exp -> exp.  %infix left 1000 ;.
```

A.2 Environment, State, and Closures

```
%%%%%%%%%%%%%%%%%%
% Reference states
%%%%%%%%%%%%%%%%%%
state      : type.  %name state Sig.

% Empty reference state
empty_state : state.

% State with items
list_state  : exp -> state.

% Stackection of states
states : type.

% Empty stackection of states
empty_states : states.

% Non-empty stackection of states
list_states : states -> ident -> state -> states.


%%%%%%%%%%%%%%%%%%
% Function Closures
%%%%%%%%%%%%%%%%%%
funclo  : type.
funclo_ls : (ident -> exp) -> state -> funclo.


%%%%%%%%%%%%%%
% Environments
%%%%%%%%%%%%%%
env  : type. %name env Rho.

% Empty environment
empty_env : env.

% Non-empty environment
list_env  : env -> ident -> funclo -> env.
```

A.3 Operational Semantics

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Evaluation of reference set
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
eval : env -> states -> exp
-> env -> states -> exp -> type. %name eval Eval.
% %mode eval +Env +State +Exp -Env2 -State2 -V.
eval_exp  :  exp -> exp -> type.

eval_exp_1 : eval_exp E V
      <- eval empty_env empty_states E Env Ste V.

%%%%%%%%%%%%%%%%
% Function lookup
%%%%%%%%%%%%%%%%
empty_clo : funclo -> funclo -> type.
empty_clo_1 : empty_clo (funclo_ls Funs State)
       (funclo_ls Funs empty_state).

lookup_fun      : ident -> env -> state
   -> (ident -> exp) -> env -> type.

lookup_fun_tail : lookup_fun F (list_env Env F (funclo_ls Fun Sig))
   Sig Fun (list_env Env F Fun')
   <- empty_clo (funclo_ls Fun Sig) Fun'.

% lookup_fun_head : lookup_fun F (list_env Env G Fun) Sig
%    V (list_env Env' G Fun)
%    <- lookup_fun F Env Sig V Env'.


%%%%%%%%%%%%%%%%
% Reference Lookup
%%%%%%%%%%%%%%%%
lookup_ref :  states -> ident -> exp -> type.

lookup_ref_tail : lookup_ref (list_states Ste X (list_state V)) X V.

% lookup_ref_head : lookup_ref (list_states Ste Y State) X V
%    <- lookup_ref Ste X V.


%%%%%%%%%%%%%%%%%%%%%%
```

```
% Update Functions
%%%%%%%%%%%%%%%%%%%%%%
update_env      : env -> ident -> state -> env -> type.
% put_in_state      : state -> exp -> state -> type.
update_env_tail : update_env (list_env Env F (funclo_ls Fun
   empty_state)) F Sig'
   (list_env Env F (funclo_ls Fun Sig')).
% update_env_head : update_env (list_env Env G F%unclo)
%    F Sig' (list_env Env' G Funclo)
%    <- update_env Env F Sig' Env'.

update_state      : states -> exp -> exp
     -> states -> type.
update_state_tail : update_state (list_states States X
     (list_state V)) (ref X) V'
     (list_states States X (list_state V')).

% update_state_head : update_state (list_states States Y State)
%      (ref X) V'  (list_states States' Y State)
%      <- update_state States (ref X) V' States'.


%%%%%%%%%%%%%%%%%%%%%%
% Operational Semantics
%%%%%%%%%%%%%%%%%%%%%%
% Constants
eval_c : eval Env Ste c Env Ste c.

% Integers and functions
eval_int   : eval Env Ste (int I) Env Ste (int I).
eval_plus  : eval Env Ste (plus E1 E2) Env' Ste' (int (I1 + I2))
      <- eval Env Ste E1 Env'' Ste'' (int I1)
      <- eval Env'' Ste'' E2 Env' Ste' (int I2).
eval_minus : eval Env Ste (minus E1 E2) Env' Ste' (int (I1 - I2))
      <- eval Env Ste E1 Env'' Ste'' (int I1)
      <- eval Env'' Ste'' E2 Env' Ste' (int I2).
eval_times : eval Env Ste (times E1 E2) Env' Ste' (int (I1 * I2))
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2).
eval_iless : eval Env Ste (iless E1 E2) Env' Ste' false
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I1 >= I2.
eval_iless : eval Env Ste (iless E1 E2) Env' Ste' true
```

```
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I2 >= (I1 + 1).
eval_imore : eval Env Ste (imore E1 E2) Env' Ste' false
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I2 >= I1.
eval_imore : eval Env Ste (imore E1 E2) Env' Ste' true
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I1 >= (I2 + 1).
eval_ieq   : eval Env Ste (ieq E1 E2) Env' Ste' true
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I2 >= I1
     <- I1 >= I2.
eval_ieq : eval Env Ste (ieq E1 E2) Env' Ste' false
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I1 >= I2
     <- I1 >= (I2 + 1).
eval_ieq : eval Env Ste (ieq E1 E2) Env' Ste' false
     <- eval Env Ste E1 Env'' Ste'' (int I1)
     <- eval Env'' Ste'' E2 Env' Ste' (int I2)
     <- I2 >= I1
     <- I2 >= (I1 + 1).

% Lambda abstractions
eval_lam : eval Env Ste (lam E) Env Ste (lam E).

% Functions with static references
eval_fun : eval Env Ste (fun F) Env Ste (fun F).

% Conditional
eval_if_true  : eval Env Ste (if E1 E2 E3) Env' Ste' V
 <- eval Env Ste E1 Env'' Ste'' true
 <- eval Env'' Ste'' E2 Env' Ste' V.

eval_if_false : eval Env Ste (if E1 E2 E3) Env' Ste' V
 <- eval Env Ste E1 Env'' Ste'' false
 <- eval Env'' Ste'' E3 Env' Ste' V.

% Function application, lambda
eval_@1  : eval Env Ste (E1 @ E2) Env' Ste' V
```

```
    <- eval Env Ste E1 Env'' Ste'' (lam E1')
    <- eval Env'' Ste'' E2 Env''' Ste''' V'
    <- eval Env''' Ste''' (E1' V') Env' Ste' V.

% Function application, f
eval_@2  : eval Env Ste (E1 @ E2) Fenv Ste' V
    <- eval Env Ste E1 Env'' Ste'' (fun F)
    <- eval Env'' Ste'' E2 Env''' Ste''' V'
    <- lookup_fun F Env''' Sig_n E1' Env''''
    <- {x:ident} eval Env'''' (list_states Ste''' x Sig_n)
       ((E1' x) @ V') Env' (list_states Ste' x Sig_n') V
    <- update_env Env' F Sig_n' Fenv.

% Let
eval_let : eval Env Ste (let E1 E2) Env' Ste' V
    <- eval Env Ste E1 Env'' Ste'' V'
    <- eval Env'' Ste'' (E2 V') Env' Ste' V.

% Let w/static references
eval_lets : eval Env Ste (letfun Fun Er E) Env' Ste' V
     <- eval Env Ste Er Env'' Ste'' Vr
     <- {f:ident} eval (list_env Env'' f
 (funclo_ls (Fun f) (list_state Vr)))
         Ste'' (E f) (list_env Env' f Funclo) Ste' V.

% Reference
eval_ref  : eval Env Ste (ref X) Env Ste (ref X).

% Dereference
eval_!   : eval Env Ste (! E) Env' Ste' V
    <- eval Env Ste E Env'' Ste'' (ref X)
    <- lookup_ref Ste'' X V.

% Update
eval_up  : eval Env Ste (up E1 E2) Env' Ste' unit
    <- eval Env Ste E1 Env'' Ste'' (ref X)
    <- eval Env'' Ste'' E2 Env' Ste''' V
    <- update_state Ste''' (ref X) V Ste'.

eval_;   : eval Env Ste (E1 ; E2) Env' Ste' V
    <- eval Env Ste E1 Env'' Ste'' V'
    <- eval Env'' Ste'' E2 Env' Ste' V.
```

A.4 Typing Rules

```
%%%%%%%%%%%%
% Type System
%%%%%%%%%%%%
tp     : type.
cell   : type.


%%%%%%%%%%%%%
% Set of cells
%%%%%%%%%%%%%
delta  : type.

empty_delta  : delta.
delta_list   : cell -> delta -> delta.

% Delta set manipulation/comparison
% Union
u          : delta -> delta -> delta.  %infix right 6 u.
member     : delta -> cell -> type.
subset    : delta -> delta -> type.
member_head : member (delta_list X Delta) X.
member_tail : member (delta_list Y Delta) X
       <- member Delta X.


subset_yes : subset Delta1 Delta2
      <- ({x:cell} member Delta1 x -> member Delta2 x).

%%%%%%%
% Types
%%%%%%%
% Unit
unit_tp : tp.

% Base
iota    : tp.
bool : tp.
int_tp : tp.
% Reference
ref_tp  : tp -> delta -> tp.  %infix right 6 ref_tp.

% Function
```

```
-->   : tp -> delta -> tp -> tp.


%%%%%%%%%%%%%%%%%%%%%%%%
% Type Ordering Relation
%%%%%%%%%%%%%%%%%%%%%%%%
lessthan : tp -> tp -> type.

lessthan_iota  : lessthan iota iota.
lessthan_int   : lessthan int_tp int_tp.
lessthan_bool  : lessthan bool bool.
lessthan_fun   : lessthan (--> Phi1 Delta Phi2) (--> Phi1' Delta' Phi2')
  <- lessthan Phi1' Phi1
  <- lessthan Phi2 Phi2'
  <- subset Delta Delta'.
lessthan_ref   : lessthan (Phi ref_tp Delta) (Phi' ref_tp Delta')
  <- lessthan Phi Phi'
  <- subset Delta Delta'.


%%%%%%%%%%%%%%%%%
% Type Environment
%%%%%%%%%%%%%%%%%
typepair  :  type.

typepair_ls : exp -> tp -> typepair.

gamma       : type.
empty_gamma : gamma.
list_gamma  : typepair -> gamma -> gamma.

member_gamma : gamma -> exp -> tp -> type.

member_gamma_head : member_gamma (list_gamma (typepair_ls X T) Gamma) X T.

member_gamma_tail : member_gamma (list_gamma Tpair Gamma) X T
     <- member_gamma Gamma X T.


%%%%%%%%%%%%%%%%%%%%%%
% Type Inferrence Rules
%%%%%%%%%%%%%%%%%%%%%%
typeof   : gamma -> exp -> tp -> delta -> type.
type_exp : exp -> tp -> delta -> type.
```

```
type_exp_1 : type_exp E T D
      <- typeof empty_gamma E T D.

% Constants
typeof_c   : typeof Gamma c iota empty_delta.

% Constants
typeof_c   : typeof Gamma c iota empty_delta.
typeof_bt  : typeof Gamma true bool empty_delta.
typeof_bf  : typeof Gamma false bool empty_delta.

% Integers and functions
typeof_int  : typeof Gamma (int I) int_tp empty_delta.
typeof_plus  : typeof Gamma (plus E1 E2) int_tp (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.
typeof_minus : typeof Gamma (minus E1 E2) int_tp (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.
typeof_times : typeof Gamma (times E1 E2) int_tp (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.
typeof_iless : typeof Gamma (iless E1 E2) bool (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.
typeof_imore : typeof Gamma (imore E1 E2) bool (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.
typeof_ieq   : typeof Gamma (ieq E1 E2) bool (Delta1 u Delta2)
        <- typeof Gamma E1 int_tp Delta1
        <- typeof Gamma E2 int_tp Delta2.

% Conditional
typeof_if   : typeof Gamma (if E1 E2 E3) Phi (Delta1 u Delta2 u Delta3)
       <- typeof Gamma E1 bool Delta1
       <- typeof Gamma E2 Phi Delta2
       <- typeof Gamma E3 Phi Delta3.

% Functions with private state
typeof_f   : typeof Gamma (fun F) Phi_f empty_delta
      <- member_gamma Gamma (fun F) Phi_f.

% Lambda abstractions
typeof_lam : typeof Gamma (lam E) (--> Phi1 Delta' Phi2)
```

```
      empty_delta
      <- {x:exp} typeof Gamma x Phi1 empty_delta
       -> typeof Gamma (E x) Phi2 Delta.


% Function application
typeof_app : typeof Gamma (E1 @ E2) Phi2 (Delta u Delta1 u Delta2)
      <- typeof Gamma E1 (--> Phi1 Delta Phi2) Delta1
      <- typeof Gamma E2 Phi1' Delta2
      <- lessthan Phi1' Phi1.

% Let
typeof_let : typeof Gamma (let E1 E2) Phi2 (Delta1 u Delta2)
      <- typeof Gamma E1 Phi1 Delta1
      <- {x:exp} ({Phi:tp} typeof Gamma x Phi Delta
      <- typeof Gamma E1 Phi Delta)
      -> typeof Gamma (E2 x) Phi2 Delta2.

% Letfun
typeof_letfun : typeof Gamma (letfun E1 E2 E3) Phi (Delta u Delta_r)
 <- typeof Gamma E2 Phi_r Delta_r
 <- ({f:ident} {x:ident} {c:cell} typeof (list_gamma
                  (typepair_ls (fun f) (--> Phi_f1 Delta_f Phi_f2))
                  (list_gamma (typepair_ls (ref x) (Phi_r ref_tp
                  (delta_list c empty_delta))) empty_gamma)) (E1 f x)
                  (--> Phi_f1 Delta_f Phi_f2) Delta_f)
 <- ({f:ident} {x:ident} typeof (list_gamma
                  (typepair_ls (fun f) (--> Phi_f1 Delta_f Phi_f2))
                  Gamma) (E3 f) Phi Delta).

% Reference
typeof_ref : typeof Gamma (ref X) Phi empty_delta
      <- member_gamma Gamma (ref X) Phi.

% Dereference
typeof_! : typeof Gamma (! E) Phi Delta'
    <- typeof Gamma E (Phi ref_tp Delta) Delta'.

% Update
typeof_up : typeof Gamma (up X E) unit_tp (Delta u Delta1 u Delta2)
     <- typeof Gamma X (Phi ref_tp Delta) Delta1
     <- typeof Gamma E Phi Delta2.

typeof_;  : typeof Gamma (E1 ; E2) Phi (Delta1 u Delta2)
```

```
<- typeof Gamma E1 Phi' Delta1
<- typeof Gamma E2 Phi Delta2.
```

A.5 Examples

```
%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (up (ref x) (plus (! (ref x)) y)) ; (! (ref x))))
      (int 0)
      ([f:ident] (fun f) @ (int 3))) T D).

%query * 1 (eval_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (up (ref x) (plus (! (ref x)) y)) ; (! (ref x))))
      (int 0)
      ([f:ident] (fun f) @ (int 3))) V).

%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (up (ref x) (minus (! (ref x)) y)) ;
      (minus (int 0) ! (ref x))))
      (int 0)
      ([f:ident] (fun f) @ (int 3))) T D).
%query * 1 (eval_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (up (ref x) (minus (! (ref x)) y)) ;
      (minus (int 0) ! (ref x))))
      (int 0)
      ([f:ident] (fun f) @ (int 3))) V).

%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (! (ref x))))
      c
      ([f:ident] (fun f))) T D).

%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (! (ref x))))
      c
      ([f:ident] (fun f @ c))) T D).

%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] ((ref x))))
      c
      ([f:ident] (fun f @ c))) T D).

%query * 1 (type_exp (letfun ([f:ident] [x:ident]
      (lam [y:exp] (! (ref x))))
      c
      ([f:ident] (ref x))) T D).
```

References

[1] H. Abelson and G. Sussman. Structure and interpretation of computer programs, 1985.

[2] John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998.

[3] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computatio*, 119(1):55–90, 1995.

[4] I. A. Mason and C. L. Talcott. Axiomatizing operational equivalence in the presence of effects. In *Fourth Annual Symposium on Logic in Computer Science*. IEEE, 1989.

[5] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[6] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In Lars Hallnas, editor, *Extensions of Logic Programming*, pages 299–344. Springer-Verlag LNCS 596, 1992. A preliminary version is available as Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science, Saarbrucken, Germany, August 1991.

[7] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[8] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[9] A. M. Pitts. Operational semantics and program equivalence. Technical report, INRIA Sophia Antipolis, 2000. Lectures at the International Summer School On Applied Semantics, APPSEM 2000, Caminha, Minho, Portugal, 9-15 September 2000.

[10] A. M. Pitts and I. D. B. Stark. Operational reasoning in functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

A Academic Vita

Academic Vita
Kamal Aboul-Hosn

Address	175 Irish Hollow Rd.
	Bellefonte, PA 16823
Phone	(814) 355-8738
Email	aboulhos@cse.psu.edu
Web page	http://www.cse.psu.edu/~aboulhos/

Education

Dec. 2001 Honors Bachelor of Science in Computer Science
Minor in Mathematics
The Pennsylvania State University

Academic Work

2001	Honors Thesis	"Programming with Private State"
		Adviser : Dr. John Hannan
	Research Interests	Logic Programming
		Programming Languages Semantics
	Class Project	Online Patient Scheduling Tool for Physicians

Work Experience

2000 - **Technology Learning Assistant**
Pennsylvania State University
- · Consulted professors to discuss and implement integrating technology into their classes to fit their personal needs
- · Designed and managed web pages and online discussion sections
- · Created step-by-step instruction guides for professors and students

2000 **Medical Records Clerk/Computer Consultant**
Bellefonte Medical Clinic
- · Coordinated installation and maintenance of computer network
- · Created custom billing and scheduling documents with *Medisoft*
- · Trained nurses and staff in general computing and *Medisoft*

Honors/Awards

Penn State Schreyer Honors College Scholar
Recipient of 2000 and 2001 Frances S. and E. Keith Anderson Scholarship
Member of Golden Key National Honor Society
Member of Phi Kappa Phi

Computing Skills

Languages	C/C++, Java, Javascript, HTML, PHP, ASP, Tcl/Tk Prolog, λProlog, SML, Elf
Servers	Apache Web Server, MySQL Database, Qmail email, BIND DNS