Polymorphic Equality – No Tags Required

Martin Elsman

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.
E-mail: mael@diku.dk

Abstract. Polymorphic equality is a controversial language construct. While being convenient for the programmer, it has been argued that polymorphic equality (1) invites to violation of software engineering principles, (2) lays a serious burden on the language implementor, and (3) enforces a runtime overhead due to the necessity of tagging values at runtime. We show that neither (2) nor (3) are inherent to polymorphic equality by showing that one can compile programs with polymorphic equality into programs without polymorphic equality in such a way that there is no need for tagging or for runtime type analysis. Also, the translation is the identity on programs that do not use polymorphic equality. Experimental results indicate that even for programs that use polymorphic equality, the translation gives good results.

1 Introduction

Often, statically typed languages, like ML, provide the programmer with a generic function for checking structural equality of two values of the same type. To avoid the possibility of testing functional values for equality, the type system of Standard ML [11] distinguishes between ordinary type variables, which may be instantiated to any type, and equality type variables, which may be instantiated only to types that admit equality (i.e., types not containing ordinary type variables or function types). In this paper, we show how polymorphic equality may be eliminated entirely in the front-end of a compiler by a type based translation called equality elimination. The translation is possible for expressions that are typable according to the Standard ML type discipline[11]. We make three main contributions:

1. Identification and application of equality elimination in a call-by-value language without garbage collection, including treatment of parametric datatypes and side effects. Equality elimination eliminates the last obligation for tagging values in Standard ML and opens for efficient data representations and easier foreign language interfacing.
2. Measurements of the effect of equality elimination in the ML Kit with Regions [23,3,5] (from hereon just the Kit) and a discussion of the possibilities for data representations made possible by the translation.
3. Demonstration of semantic correctness of the translation. It has been considered non-trivial to demonstrate semantic correctness for type classes in Haskell [13, Sect. 4].

As an example of equality elimination, consider the following ML program, which declares the function `member` using polymorphic equality to test if a given value is among the elements of a list:

```
let fun member y [] = false
      | member y (x::xs) = (y = x) orelse member y xs
in (member 5 [3,5], member true [false])
end
```

The function `member` gets type scheme $\forall\varepsilon.\varepsilon \to \varepsilon\ \mathit{list} \to \mathit{bool}$, where ε is an *equality type variable* (i.e., a type variable that ranges over equality types). In the example, the function `member` is used with instances *int* and *bool*, which both admit equality.

On the other hand, the function `map`, presented below, gets type scheme $\forall\alpha\beta.(\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$, where α and β are ordinary type variables and hence may be instantiated to any particular type.

```
fun map f [] = []
  | map f (x::xs) = f x :: map f xs
```

To eliminate polymorphic equality, it is possible to pass extra arguments to equality polymorphic functions as `member` above – one for each abstracted equality type variable in the type scheme for the function. Using type information, the example is translated into the program

```
let fun member eq y [] = false
      | member eq y (x::xs) = eq (y,x) orelse member eq y xs
in (member eq_int 5 [3,5], member eq_bool true [false])
end
```

For each use of an equality polymorphic function, appropriate instances of the equality primitive are passed as arguments. In the translated program above, `eq_int` and `eq_bool` denote primitive equality functions for testing integers and booleans for equality. These primitive functions are functions on base types and can be implemented efficiently by the backend of a compiler without the requirement that values be tagged. An important property of the translation is that it is the identity on expressions that do not use polymorphic equality. Thus, one pays for polymorphic equality only when it is used. In particular, the translation is the identity on the `map` function.

In the next section, we give an overview of related work. The language that we consider is described in the sections to follow. We then proceed to present a translation for eliminating polymorphic equality. In Sect. 7 and Sect. 8, we demonstrate type correctness and semantic soundness of the translation. In Sect. 9 and Sect. 10, we show how the approach is extended to full ML and how it is implemented in the Kit. We then proceed to present experimental results. Finally, we conclude.

2 Related Work

A type based dictionary transformation similar to equality elimination allows type classes in Haskell to be eliminated at compile time [25,13,16]. However, the motivation for equality elimination is different from the motivation behind the dictionary transformation, which is to separate dictionary operations from values at runtime. In lazy languages such as Haskell, tagging cannot be eliminated even if tag-free garbage collection is used. A more aggressive elimination of dictionaries is possible by generating specialised versions of overloaded functions [8]. This technique does not work well with separate compilation and may lead to unnecessary code duplication. No work on dictionary transformations demonstrates semantic soundness.

Harper and Stone present an alternative semantics to Standard ML in terms of a translation into an intermediate typed language [6]. Similar to the translation we present here, polymorphic equality is eliminated during the translation. However, because the semantics of their source language is given by the translation, they cannot show correctness of the translation.

Ohori demonstrates how Standard ML may be extended with polymorphic record operations in such a way that these operations can be translated into efficient indexing operations [14]. His translation is much similar to equality elimination in that record indices are passed to instantiations of functions that use record operations polymorphically; Ohori demonstrates both type correctness and semantic soundness for the approach.

The TIL compiler, developed at Carnegie Mellon University, uses intensional polymorphism and nearly tag-free garbage collection to allow tag-free representations of values at runtime [20]. An intermediate language of TIL allows a function to take types as arguments, which can then be inspected by the function. This means that polymorphic equality can be encoded in the intermediate language, thus, eliminating the primitive notion of polymorphic equality. However, for nearly tag-free garbage collection, records and other objects stored in the heap are still tagged in order for the garbage collector to trace pointers. It has been reported, however, that the nearly tag-free scheme can be extended to an entirely tag-free scheme [21].

3 Language and Types

We consider a typed lambda calculus extended with pairs, conditionals, a polymorphic equality primitive, and a let-construct to allow polymorphic bindings. First, we introduce some terminology. A *finite* map is a map with finite domain and if f and g are such maps we denote by $\mathrm{Dom}(f)$ the domain of f and by $\mathrm{Ran}(f)$ the range of f. Further, we write $f + g$ to mean the *modification* of f by g with domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and values $(f + g)(x) =$ if $x \in \mathrm{Dom}(g)$ then $g(x)$ else $f(x)$.

We assume a denumerably infinite set of *equality type variables*, ranged over by ε, and a denumerably infinite set of *ordinary type variables*, ranged over by

α. Types, ranged over by τ, and type schemes, ranged over by σ, are defined as follows:

$$\tau ::= \varepsilon \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \mathit{bool}$$
$$\sigma ::= \forall \vec{\varepsilon}\vec{\alpha}.\tau$$

A type τ *admits equality* if either $\tau = \mathit{bool}$ or $\tau = \varepsilon$ or $\tau = \tau_1 \times \tau_2$ and τ_1 and τ_2 admit equality.

3.1 Substitutions

A *substitution* S is a pair $(S^\varepsilon, S^\alpha)$, where S^ε is a finite map from equality type variables to types such that, for all $\tau \in \mathrm{Ran}(S^\varepsilon)$, τ admits equality and S^α is a finite map from ordinary type variables to types. When A is any object and $S = (S^\varepsilon, S^\alpha)$ is a substitution, we write $S(A)$ to mean simultaneous capture free substitution of S^ε and S^α on A.

For any type scheme $\sigma = \forall\varepsilon_1 \cdots \varepsilon_n\alpha_1 \cdots \alpha_m.\tau$ and type τ', we say that τ' is an *instance of σ (via S)*, written $\sigma \geq \tau'$, if there exists a substitution $S = (\{\varepsilon_1 \mapsto \tau_1, \ldots, \varepsilon_n \mapsto \tau_n\}, \{\alpha_1 \mapsto \tau'_1, \ldots, \alpha_m \mapsto \tau'_m\})$ such that $S(\tau) = \tau'$. The *instance list of S*, written $\mathit{il}(S)$, is the pair $([\tau_1, \ldots, \tau_n], [\tau'_1, \ldots, \tau'_m])$. Pairs of the above form are referred to as instance lists and we use il to range over them.

When A is any object, we denote by $\mathrm{ftv}(A)$ a pair of a set of equality type variables free in A and a set of ordinary type variables free in A. Further, we denote by $\mathrm{fetv}(A)$ the set of equality type variables that occur free in A.

3.2 Typed Expressions

In the following, we use x and y to range over a denumerably infinite set of *lambda variables*. The grammar for typed expressions is as follows:

$$\begin{aligned} e ::= {} & \lambda x : \tau.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i\ e \mid x_{il} \\ & \mid \texttt{let}\ x : \sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{true} \mid \texttt{false} \\ & \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{eq}_\tau \end{aligned}$$

We sometimes abbreviate $x_{([],[])}$ with x.

4 Static Semantics

The static semantics for the language is described by a set of inference rules. Each of the rules allows inferences among sentences of the form $\Delta, TE \vdash e : \tau$, where, Δ is a set of equality type variables, TE is a *type environment*, mapping lambda variables to type schemes, e is a typed lambda expression, and τ is a type. Sentences of this form are read "under assumption (Δ, TE), the expression e has type τ."

A type τ is *well-formed* with respect to a set of equality type variables Δ, written $\Delta \vdash \tau$, if $\Delta \supseteq \mathrm{fetv}(\tau)$. Moreover, an instance list $\mathit{il} = ([\tau_1, \ldots, \tau_n], [\tau'_1, \ldots, \tau'_m])$ is *well-formed* with respect to a set of equality type variables Δ, written $\Delta \vdash \mathit{il}$, if $\Delta \vdash \tau_i$, $i = 1..n$ and $\Delta \vdash \tau'_i$, $i = 1..m$.

Expressions $\boxed{\Delta, TE \vdash e : \tau}$

$$\frac{\Delta, TE + \{x \mapsto \tau\} \vdash e : \tau'}{\Delta, TE \vdash \lambda x : \tau . e : \tau \to \tau'} \ (1) \qquad \frac{\Delta, TE \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta, TE \vdash e_2 : \tau_1}{\Delta, TE \vdash e_1 \ e_2 : \tau_2} \ (2)$$

$$\frac{\Delta, TE \vdash e_1 : \tau_1 \quad \Delta, TE \vdash e_2 : \tau_2}{\Delta, TE \vdash (e_1, e_2) : \tau_1 \times \tau_2} \ (3) \qquad \frac{i \in \{1, 2\} \quad \Delta, TE \vdash e : \tau_1 \times \tau_2}{\Delta, TE \vdash \pi_i \ e : \tau_i} \ (4)$$

$$\frac{TE(x) \geq \tau \text{ via } S \quad \Delta \vdash il(S)}{\Delta, TE \vdash x_{il(S)} : \tau} \ (5) \qquad \frac{\sigma = \forall \vec{\varepsilon}\vec{\alpha}.\tau \quad \text{ftv}(\vec{\varepsilon}\vec{\alpha}) \cap \text{ftv}(\Delta, TE) = \emptyset \quad \Delta \cup \text{fetv}(\vec{\varepsilon}), TE \vdash e_1 : \tau \quad \Delta, TE + \{x \mapsto \sigma\} \vdash e_2 : \tau'}{\Delta, TE \vdash \texttt{let } x : \sigma \texttt{ = } e_1 \texttt{ in } e_2 : \tau'} \ (6)$$

$$\frac{}{\Delta, TE \vdash \texttt{true} : bool} \ (7) \qquad \frac{\Delta, TE \vdash e : bool \quad \Delta, TE \vdash e_1 : \tau \quad \Delta, TE \vdash e_2 : \tau}{\Delta, TE \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau} \ (8)$$

$$\frac{}{\Delta, TE \vdash \texttt{false} : bool} \ (9) \qquad \frac{\Delta \vdash \tau \quad \tau \text{ admits equality}}{\Delta, TE \vdash \texttt{eq}_\tau : \tau \times \tau \to bool} \ (10)$$

There are only a few comments to note about the rules. In the rule for applying the equality primitive to values of a particular type, we require the type to be well-formed with respect to quantified equality type variables. Similarly, in the variable rule, we require the instance list be well-formed with respect to quantified equality type variables.

For simplifying the type system in languages with imperative updates and polymorphism, there is a tendency to restrict polymorphism to bindings of non-side-effecting terminating expressions. This tendency is known as the *value restriction*, which is enforced by both the Objective Caml system [10] and the Standard ML language [11]. To simplify the presentation, we do not enforce the value restriction in rule 6. We return to this issue later, in Sect. 8.

5 Dynamic Semantics

The dynamic semantics for the language is, as the static semantics, described by a set of inference rules.

An *untyped expression* may be obtained from a typed expression by eliminating all type information. In the rest of this section, we use e to range over untyped expressions.

A *dynamic environment*, $\mathcal{E}$, maps lambda variables to values, which again are defined by the grammar:

$$v ::= \texttt{clos}(\lambda x.e, \mathcal{E}) \mid \texttt{true} \mid \texttt{false} \mid (v_1, v_2) \mid \texttt{eq}$$

The rules of the dynamic semantics allow inferences among sentences of the forms $\mathcal{E} \vdash e \Downarrow v$ and $\vdash_{\texttt{eq}} (v_1, v_2) \Downarrow v$, where $\mathcal{E}$ is a dynamic environment, e is an untyped expression, and v, v_1, and v_2 are values. Sentences of the former form are read "under assumptions $\mathcal{E}$, the expression e evaluates to v." Sentences of the latter form are read "equality of values v_1 and v_2 is v."

Expressions $\boxed{\mathcal{E} \vdash e \Downarrow v}$

$$\frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \Downarrow v} \ (11) \qquad \frac{}{\mathcal{E} \vdash \lambda x.e \Downarrow \texttt{clos}(\lambda x.e, \mathcal{E})} \ (12)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow \texttt{clos}(\lambda x.e, \mathcal{E}_0) \\ \mathcal{E} \vdash e_2 \Downarrow v \\ \mathcal{E}_0 + \{x \mapsto v\} \vdash e \Downarrow v'\end{array}}{\mathcal{E} \vdash e_1 \ e_2 \Downarrow v'} \ (13) \qquad \frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow \texttt{eq} \\ \mathcal{E} \vdash e_2 \Downarrow v \qquad \vdash_{\texttt{eq}} v \Downarrow v'\end{array}}{\mathcal{E} \vdash e_1 \ e_2 \Downarrow v'} \ (14)$$

$$\frac{}{\mathcal{E} \vdash \texttt{true} \Downarrow \texttt{true}} \ (15) \qquad \frac{}{\mathcal{E} \vdash \texttt{false} \Downarrow \texttt{false}} \ (16)$$

$$\frac{}{\mathcal{E} \vdash \texttt{eq} \Downarrow \texttt{eq}} \ (17) \qquad \frac{\mathcal{E} \vdash e_1 \Downarrow v_1 \quad \mathcal{E} \vdash e_2 \Downarrow v_2}{\mathcal{E} \vdash (e_1, e_2) \Downarrow (v_1, v_2)} \ (18)$$

$$\frac{i \in \{1, 2\} \quad \mathcal{E} \vdash e \Downarrow (v_1, v_2)}{\mathcal{E} \vdash \pi_i \ e \Downarrow v_i} \ (19) \qquad \frac{\mathcal{E} \vdash e \Downarrow \texttt{true} \quad \mathcal{E} \vdash e_1 \Downarrow v}{\mathcal{E} \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v} \ (20)$$

$$\frac{\mathcal{E} \vdash e \Downarrow \texttt{false} \quad \mathcal{E} \vdash e_2 \Downarrow v}{\mathcal{E} \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow v} \ (21) \qquad \frac{\begin{array}{c}\mathcal{E} \vdash e_1 \Downarrow v_1 \\ \mathcal{E} + \{x \mapsto v_1\} \vdash e_2 \Downarrow v_2\end{array}}{\mathcal{E} \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v_2} \ (22)$$

Equality of Values $\boxed{\vdash_{\text{eq}} (v_1, v_2) \Downarrow v}$

$$\frac{\begin{array}{c} v_1 = v_2 \\ v_1, v_2 \in \{\texttt{true}, \texttt{false}\} \end{array}}{\vdash_{\text{eq}} (v_1, v_2) \Downarrow \texttt{true}} \ (23) \qquad \frac{\vdash_{\text{eq}} (v_{11}, v_{21}) \Downarrow \texttt{false}}{\vdash_{\text{eq}} ((v_{11}, v_{12}), (v_{21}, v_{22})) \Downarrow \texttt{false}} \ (24)$$

$$\frac{\begin{array}{c} v_1 \neq v_2 \\ v_1, v_2 \in \{\texttt{true}, \texttt{false}\} \end{array}}{\vdash_{\text{eq}} (v_1, v_2) \Downarrow \texttt{false}} \ (25) \qquad \frac{\begin{array}{c} \vdash_{\text{eq}} (v_{11}, v_{21}) \Downarrow \texttt{true} \\ \vdash_{\text{eq}} (v_{12}, v_{22}) \Downarrow v \end{array}}{\vdash_{\text{eq}} ((v_{11}, v_{12}), (v_{21}, v_{22})) \Downarrow v} \ (26)$$

The dynamic semantics includes rules for the polymorphic equality primitive (rules 23 through 26). If the equality primitive is only ever applied to values of type *bool* (if rules 24 and 26 are not used), the primitive need not distinguish booleans from values of pair type and no runtime tagging is required.

6 Equality Elimination

In this section, we present inference rules for translating typable expressions into typable expressions for which the equality primitive is used only with instance *bool*. The translation is the identity for typable expressions that do not use polymorphic equality.

A *translation environment*, E, is a finite map from equality type variables to lambda variables. We occasionally need to construct a function for checking structural equality on a pair of values of the same type. We define a relation that allows inferences among sentences of the form $E \vdash_{\text{eq}} \tau \Rightarrow e$, where E is a translation environment, τ is a type, and e is an expression. Sentences of this form are read "under the assumptions E, e is an equality function for values of type τ."

Equality Function Construction $\boxed{E \vdash_{\text{eq}} \tau \Rightarrow e}$

$$\frac{E(\varepsilon) = x}{E \vdash_{\text{eq}} \varepsilon \Rightarrow x} \ (27) \qquad \frac{}{E \vdash_{\text{eq}} \mathit{bool} \Rightarrow \texttt{eq}_{\mathit{bool}}} \ (28)$$

$$\frac{\begin{array}{c} E \vdash_{\text{eq}} \tau_1 \Rightarrow e_1 \quad E \vdash_{\text{eq}} \tau_2 \Rightarrow e_2 \quad x \text{ fresh} \\ e = e_2\ (\pi_2\ (\pi_1\ x), \pi_2\ (\pi_2\ x)) \\ e' = \texttt{if}\ e_1\ (\pi_1\ (\pi_1\ x), \pi_1\ (\pi_2\ x))\ \texttt{then}\ e\ \texttt{else}\ \texttt{false} \end{array}}{E \vdash_{\text{eq}} \tau_1 \times \tau_2 \Rightarrow \lambda x : (\tau_1 \times \tau_2) \times (\tau_1 \times \tau_2).e'} \ (29)$$

Each rule for the translation of expressions allows inferences among sentences of the form $E \vdash e \Rightarrow e'$, where e and e' are expressions and E is a translation environment. Sentences of this form are read "under the assumptions E, e translates to e'."

Expressions $\boxed{E \vdash e \Rightarrow e'}$

$$\frac{E \vdash e \Rightarrow e'}{E \vdash \lambda x : \tau.e \Rightarrow \lambda x : \tau.e'} \ (30) \qquad \frac{E \vdash e_1 \Rightarrow e_1' \quad E \vdash e_2 \Rightarrow e_2'}{E \vdash e_1\ e_2 \Rightarrow e_1'\ e_2'} (31)$$

$$\frac{E \vdash e_1 \Rightarrow e_1' \quad E \vdash e_2 \Rightarrow e_2'}{E \vdash (e_1, e_2) \Rightarrow (e_1', e_2')} (32) \qquad \frac{E \vdash e \Rightarrow e'}{E \vdash \pi_i\ e \Rightarrow \pi_i\ e'} \ (33)$$

$$\frac{\begin{array}{c} il = ([\tau_1, \ldots, \tau_n], [\ldots]) \quad n \geq 0 \\ E \vdash_{\mathrm{eq}} \tau_i \Rightarrow e_i \quad i = 1..n \end{array}}{E \vdash x_{il} \Rightarrow (\cdots (x_{il}\ e_1) \cdots e_n)} (34) \qquad \frac{E \vdash_{\mathrm{eq}} \tau \Rightarrow e}{E \vdash \mathtt{eq}_\tau \Rightarrow e} \ (35)$$

$$\frac{\begin{array}{c} \sigma = \forall \varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau \quad y_1 \cdots y_n \text{ fresh} \quad n \geq 0 \\ E + \{\varepsilon_1 \mapsto y_1, \ldots, \varepsilon_n \mapsto y_n\} \vdash e_1 \Rightarrow e_1' \\ \tau_i = \varepsilon_i \times \varepsilon_i \to \mathit{bool} \quad i = 1..n \\ e_1'' = \lambda y_1 : \tau_1. \cdots . \lambda y_n : \tau_n . e_1' \quad E \vdash e_2 \Rightarrow e_2' \\ \sigma' = \forall \varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau_1 \to \cdots \to \tau_n \to \tau \end{array}}{E \vdash \mathtt{let}\ x : \sigma\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \Rightarrow \mathtt{let}\ x : \sigma'\ \mathtt{=}\ e_1''\ \mathtt{in}\ e_2'} (36)$$

$$\frac{}{E \vdash \mathtt{true} \Rightarrow \mathtt{true}} \ (37) \qquad \frac{}{E \vdash \mathtt{false} \Rightarrow \mathtt{false}} \ (38)$$

$$\frac{E \vdash e \Rightarrow e' \quad E \vdash e_1 \Rightarrow e_1' \quad E \vdash e_2 \Rightarrow e_2'}{E \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Rightarrow \mathtt{if}\ e'\ \mathtt{then}\ e_1'\ \mathtt{else}\ e_2'} (39)$$

In the translation rule for the let-construct, we generate abstractions for equality functions for each bound equality type variable in the type scheme for the let-bound variable. Accordingly, in the rule for variable occurrences, appropriate equality functions are applied according to type instances for abstracted equality type variables. In rule 35, we generate a function for checking equality of values of type τ.

7 Type Correctness

In this section, we demonstrate that the translation preserves types and that all typable expressions may be translated.

7.1 Type Preservation

We first give a few definitions for relating type environments and translation environments.

Definition 1. (Extension) A type scheme $\sigma = \forall\varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau$ *extends* another type scheme $\sigma' = \forall\varepsilon'_1 \cdots \varepsilon'_m \vec{\alpha}'.\tau'$, written $\sigma \succ \sigma'$, if $n = m$ and $\varepsilon_i = \varepsilon'_i$, $i = 1..n$ and $\vec{\alpha} = \vec{\alpha}'$ and $\tau = (\varepsilon_1 \times \varepsilon_1 \to \mathit{bool}) \to \cdots \to (\varepsilon_n \times \varepsilon_n \to \mathit{bool}) \to \tau'$.

A type environment TE' *extends* another type environment TE, written $TE' \succ TE$, if $\mathrm{Dom}(TE') \supseteq \mathrm{Dom}(TE)$ and $TE'(x) \succ TE(x)$ for all $x \in \mathrm{Dom}(TE)$.

Definition 2. (Environment Matching) A translation environment E *matches* a type environment TE, written $E \sqsubseteq TE$, if $TE(E(\varepsilon)) = \varepsilon \times \varepsilon \to \mathit{bool}$ for all $\varepsilon \in \mathrm{Dom}(E)$.

The following proposition states that the equality function generated for a specific type that admits equality has the expected type.

Proposition 3. *If* $E \vdash_{\mathrm{eq}} \tau \Rightarrow e$ *and* $E \sqsubseteq TE$ *and* $\Delta \vdash \tau$ *then* $\Delta, TE \vdash e : \tau \times \tau \to \mathit{bool}$.

Proof. By induction over the structure of τ. □

We can now state a proposition saying that the translation preserves types.

Proposition 4. (Type Preservation) *If* $\Delta, TE \vdash e : \tau$ *and* $E \vdash e \Rightarrow e'$ *and* $E \sqsubseteq TE' \succ TE$ *then* $\Delta, TE' \vdash e' : \tau$.

Proof. By induction over the structure of e. We show the three interesting cases.

CASE $e = x_{il}$ From (5), we have $TE(x) = \sigma$ and $\sigma = \forall\varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau'$ and $\sigma \geq \tau$ via S and $\Delta \vdash il$ and $\Delta, TE \vdash x_{il} : \tau$, where $il = il(S)$. From (34), we have that $il = ([\tau_1, \ldots, \tau_n], [\ldots])$ and $E \vdash_{\mathrm{eq}} \tau_i \Rightarrow e_i$, $i = 1..n$ and $E \vdash x_{il} \Rightarrow (\cdots (x_{il}\ e_1) \cdots e_n)$.

Because $\Delta \vdash il$, we have $\Delta \vdash \tau_i$, $i = 1..n$ and because $E \sqsubseteq TE'$ follows from assumptions, we have by Proposition 3 that $\Delta, TE' \vdash e_i : \tau_i \times \tau_i \to \mathit{bool}$, $i = 1..n$.

Because $TE' \succ TE$ follows from assumptions, we have $TE'(x) = \sigma'$, where $\sigma' = \forall\varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.(\varepsilon_1 \times \varepsilon_1 \to \mathit{bool}) \to \cdots \to (\varepsilon_n \times \varepsilon_n \to \mathit{bool}).\tau'$, and because $\sigma \geq \tau$ via S and $S(\varepsilon_i) = \tau_i$, $i = 1..n$ follows from $il(S) = ([\tau_1, \ldots, \tau_n], [\ldots])$, we have $\sigma' \geq \tau''$ via S, where $\tau'' = (\tau_1 \times \tau_1 \to \mathit{bool}) \to \cdots \to (\tau_n \times \tau_n \to \mathit{bool}) \to \tau$. Because $\Delta \vdash il$, we can now apply (5) to get $\Delta, TE' \vdash x_{il} : \tau''$.

Now, because $\Delta, TE' \vdash e_i : \tau_i \times \tau_i \to \mathit{bool}$, $i = 1..n$, we can apply (2) n times to get $\Delta, TE' \vdash (\cdots (x_{il}\ e_1) \cdots e_n) : \tau$, as required.

CASE $e = \texttt{let}\ x : \sigma\ \texttt{=}\ e_1\ \texttt{in}\ e_2$ From (6), we have $\sigma = \forall \varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau$ and $\mathrm{ftv}(\varepsilon_1 \cdots \varepsilon_n \vec{\alpha}) \cap \mathrm{ftv}(\Delta, TE) = \emptyset$ and $\Delta \cup \{\varepsilon_1, \ldots, \varepsilon_n\}, TE \vdash e_1 : \tau$ and $\Delta, TE + \{x \mapsto \sigma\} \vdash e_2 : \tau'$ and $\Delta, TE \vdash e : \tau'$.

Further, from (36), we have $y_1 \cdots y_n$ fresh and $E + \{\varepsilon_1 \mapsto y_1, \ldots, \varepsilon_n \mapsto y_n\} \vdash e_1 \Rightarrow e_1'$ and $\tau_i = \varepsilon_i \times \varepsilon_i \to \mathit{bool}$, $i = 1..n$ and $e_1'' = \lambda y_1 : \tau_1. \cdots . \lambda y_n : \tau_n . e_1'$ and $E \vdash e_2 \Rightarrow e_2'$ and $\sigma' = \forall \varepsilon_1 \cdots \varepsilon_n \vec{\alpha}.\tau_1 \to \cdots \to \tau_n \to \tau$ and $E \vdash e \Rightarrow \texttt{let}\ x : \sigma'\ \texttt{=}\ e_1''\ \texttt{in}\ e_2'$.

It now follows from assumptions and from the definitions of extension and matching that $E + \{\varepsilon_1 \mapsto y_1, \ldots, \varepsilon_n \mapsto y_n\} \sqsubseteq TE'' \succ TE$, where $TE'' = TE' + \{y_1 \mapsto \tau_1, \ldots, y_n \mapsto \tau_n\}$, because $\mathrm{Dom}(TE') \cap \{y_1, \ldots y_n\} = \emptyset$ and $\mathrm{Dom}(E) \cap \{\varepsilon_1, \ldots, \varepsilon_n\} = \emptyset$ can be assumed by appropriate renaming of bound type variables of σ. We can now apply induction to get $\Delta \cup \{\varepsilon_1, \ldots, \varepsilon_n\}, TE'' \vdash e_1' : \tau$. By applying (1) n times, we get $\Delta \cup \{\varepsilon_1, \ldots, \varepsilon_n\}, TE' \vdash e_1'' : \tau_1 \to \cdots \to \tau_n \to \tau$.

To apply induction the second time, we observe that $E \sqsubseteq TE' + \{x \mapsto \sigma'\} \succ TE + \{x \mapsto \sigma\}$ by assumptions and definitions of matching and extension and because $\mathrm{Dom}(TE') \cap \{x\} = \emptyset$ can be assumed by appropriate renaming of x in e. By induction, we have $\Delta, TE' + \{x \mapsto \sigma'\} \vdash e_2' : \tau'$.

Because we can assume $\mathrm{ftv}(\varepsilon_1 \cdots \varepsilon_n \vec{\alpha}) \cap \mathrm{ftv}(TE') = \emptyset$ by appropriate renaming of bound equality type variables and type variables in σ, we can apply (6) to get $\Delta, TE' \vdash \texttt{let}\ x : \sigma'\ \texttt{=}\ e_1''\ \texttt{in}\ e_2' : \tau'$, as required.

CASE $e = \texttt{eq}_\tau$ From (10), we have $\Delta \vdash \tau$ and τ admits equality and $\Delta, TE \vdash \texttt{eq}_\tau : \tau \times \tau \to \mathit{bool}$. From (35), we have $E \vdash_{\mathrm{eq}} \tau \Rightarrow e'$ and $E \vdash \texttt{eq}_\tau \Rightarrow e'$.

By assumptions, we have $E \sqsubseteq TE'$ and because $\Delta \vdash \tau$, we can apply Proposition 3 to get $\Delta, TE' \vdash e' : \tau \times \tau \to \mathit{bool}$, as required. □

7.2 Typable Expressions are Translatable

We now demonstrate that all typable expressions may indeed be translated by the translation rules. The following proposition states that for a type that admits equality it is possible to construct a function that checks for equality on pairs of values of this type.

Proposition 5. *If τ admits equality and $\mathrm{fetv}(\tau) \subseteq \mathrm{Dom}(E)$ then there exists an expression e such that $E \vdash_{\mathrm{eq}} \tau \Rightarrow e$.*

Proof. By induction over the structure of τ. □

The following proposition states that all typable expressions may be translated.

Proposition 6. (Typable Expressions are Translatable) *If $\Delta, TE \vdash e : \tau$ and $\Delta = \mathrm{Dom}(E)$ and $\mathrm{Dom}(TE) \cap \mathrm{Ran}(E) = \emptyset$ then there exists e' such that $E \vdash e \Rightarrow e'$.*

Proof. By induction over the structure of e. We show the three interesting cases.

CASE $e = x_{il}$ From (5), we have $TE(x) = \sigma$ and $\sigma \geq \tau$ via S and $\Delta \vdash il(S)$ and $\Delta, TE \vdash x_{il(S)} : \tau$.

Let $il(S)$ be written as $([\tau_1, \ldots, \tau_n], [\ldots])$. Because $\Delta \vdash il(S)$, we have $\Delta \vdash \tau_i$, $i = 1..n$, hence, $\mathrm{fetv}(\tau_i) \subseteq \mathrm{Dom}(E)$, $i = 1..n$. Further, from the definition of substitution, we have τ_i admits equality, $i = 1..n$. We can now apply Proposition 5 to get, there exists an expression e_i such that $E \vdash_{\mathrm{eq}} \tau_i \Rightarrow e_i$, $i = 1..n$. By applying (34), we have $E \vdash x_{il(S)} \Rightarrow (\cdots (x_{il(S)}\ e_1) \cdots e_n)$, as required.

CASE $e = \mathtt{let}\ x : \sigma\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2$ From (6), we have $\sigma = \forall \vec{\varepsilon}\vec{\alpha}.\tau$ and $\mathrm{ftv}(\vec{\varepsilon}\vec{\alpha}) \cap \mathrm{ftv}(\Delta, TE) = \emptyset$ and $\Delta \cup \mathrm{fetv}(\vec{\varepsilon}), TE \vdash e_1 : \tau$ and $\Delta, TE + \{x \mapsto \sigma\} \vdash e_2 : \tau'$ and $\Delta, TE \vdash e : \tau'$.

Write $\vec{\varepsilon}$ as $\varepsilon_1 \cdots \varepsilon_n$ and let $y_1 \cdots y_n$ be fresh. Further, let $E' = E + \{\varepsilon_1 \mapsto y_1, \ldots, \varepsilon_n \mapsto y_n\}$. By assumptions, we have $\Delta \cup \mathrm{fetv}(\vec{\varepsilon}) = \mathrm{Dom}(E')$ and $\mathrm{Dom}(TE) \cap \mathrm{Ran}(E') = \emptyset$. We can now apply induction to get, there exists an expression e_1' such that $E' \vdash e_1 \Rightarrow e_1'$. Also, let $e_1'' = \lambda y_1 : \tau_1. \cdots . \lambda y_n : \tau_n . e_1'$, where $\tau_i = \varepsilon_i \times \varepsilon_i \to \mathit{bool}$, $i = 1..n$.

By assumptions and by appropriate renaming of x in e, we have $\mathrm{Dom}(TE + \{x \mapsto \sigma\}) \cap \mathrm{Ran}(E) = \emptyset$, hence, we can apply induction to get, there exists e_2' such that $E \vdash e_2 \Rightarrow e_2'$. Letting $\sigma' = \forall \vec{\varepsilon}\vec{\alpha}.\tau_1 \to \cdots \to \tau_n \to \tau$, we can apply (36) to get $E \vdash e \Rightarrow \mathtt{let}\ x : \sigma' \mathtt{=} e_1''\ \mathtt{in}\ e_2'$, as required.

CASE $e = \mathtt{eq}_\tau$ From (10), we have $\Delta \vdash \tau$ and τ admits equality and $\Delta, TE \vdash \mathtt{eq}_\tau : \tau \times \tau \to \mathit{bool}$. Because $\Delta = \mathrm{Dom}(E)$ follows from assumptions and $\Delta \vdash \tau$, we have $\mathrm{fetv}(\tau) \subseteq \mathrm{Dom}(E)$, hence, from Proposition 5, we have, there exists an expression e' such that $E \vdash_{\mathrm{eq}} \tau \Rightarrow e'$. From (35), we now have $E \vdash e \Rightarrow e'$, as required. □

8 Semantic Soundness

In this section, we demonstrate semantic soundness of the translation inspired by other proofs of semantic soundness of type systems [9,22].

Because equality functions are represented differently in the original program and the translated program, the operational semantics may assign different values to them. For this reason, we define a notion of *semantic equivalence* between values corresponding to the original program and values corresponding to the translated program. We write it $\Gamma \models v : \tau \approx v'$. The type is needed to correctly interpret the values and to ensure well-foundedness of the definition. The environment Γ is formally a pair $(\Gamma^\varepsilon, \Gamma^\alpha)$ providing interpretations of equality type variables and ordinary type variables in τ. Interpretations are non-empty sets $\mathcal{V}$ of pairs (v_1, v_2) of values. We often abbreviate projections from Γ and injections in Γ. For instance, when $\Gamma = (\Gamma^\varepsilon, \Gamma^\alpha)$, we write $\Gamma(\varepsilon)$ to mean $\Gamma^\varepsilon(\varepsilon)$ and $\Gamma + \{\alpha \mapsto \mathcal{V}\}$ to mean $(\Gamma^\varepsilon, \Gamma^\alpha + \{\alpha \mapsto \mathcal{V}\})$, for any ε, α, and $\mathcal{V}$.

- $\Gamma \models \mathtt{true} : \mathit{bool} \approx \mathtt{true}$

- $\Gamma \models \texttt{false} : \mathit{bool} \approx \texttt{false}$
- $\Gamma \models (v_1, v_2) : \tau_1 \times \tau_2 \approx (v_1', v_2')$ iff $\Gamma \models v_1 : \tau_1 \approx v_1'$ and $\Gamma \models v_2 : \tau_2 \approx v_2'$
- $\Gamma \models \texttt{eq} : \mathit{bool} \times \mathit{bool} \to \mathit{bool} \approx \texttt{eq}$
- $\Gamma \models \texttt{eq} : \tau \times \tau \to \mathit{bool} \approx \texttt{clos}(\lambda x.e, \mathcal{E})$ iff for all values v_1, v_2, v_1' such that $\Gamma \models v_1 : \tau \times \tau \approx v_1'$ and $\vdash_{\texttt{eq}} v_1 \Downarrow v_2$, we have $\mathcal{E} + \{x \mapsto v_1'\} \vdash e \Downarrow v_2$
- $\Gamma \models \texttt{clos}(\lambda x.e, \mathcal{E}) : \tau_1 \to \tau_2 \approx \texttt{clos}(\lambda x.e', \mathcal{E}')$ iff for all values v_1, v_2, v_1' such that $\Gamma \models v_1 : \tau_1 \approx v_1'$ and $\mathcal{E} + \{x \mapsto v_1\} \vdash e \Downarrow v_2$, there exists a value v_2' such that $\mathcal{E}' + \{x \mapsto v_1'\} \vdash e' \Downarrow v_2'$ and $\Gamma \models v_2 : \tau_2 \approx v_2'$
- $\Gamma \models v : \alpha \approx v'$ iff $(v, v') \in \Gamma(\alpha)$
- $\Gamma \models v : \varepsilon \approx v'$ iff $(v, v') \in \Gamma(\varepsilon)$

The semantic equivalence relation extends to type schemes and environments:

- $\Gamma \models v : \forall \alpha_1 \cdots \alpha_n.\tau \approx v'$ iff for all interpretations $\mathcal{V}_1^\alpha \cdots \mathcal{V}_m^\alpha$, we have $\Gamma + \{\alpha_1 \mapsto \mathcal{V}_1^\alpha, \cdots, \alpha_n \mapsto \mathcal{V}_n^\alpha\} \models v : \tau \approx v'$
- $\Gamma \models v : \forall \varepsilon_1 \cdots \varepsilon_n \alpha_1 \cdots \alpha_m.\tau \approx \texttt{clos}(\lambda y_1.\cdots.\lambda y_n.e, \mathcal{E})$ iff for all interpretations $\mathcal{V}_1^\varepsilon \cdots \mathcal{V}_n^\varepsilon \mathcal{V}_1^\alpha \cdots \mathcal{V}_m^\alpha$, values $v_1 \cdots v_n$ and semantic environments Γ', such that $\Gamma' \models \texttt{eq} : \varepsilon_i \times \varepsilon_i \to \mathit{bool} \approx v_i$, $i = 1..n$ and $\Gamma' = \Gamma + \{\varepsilon_1 \mapsto \mathcal{V}_1^\varepsilon, \cdots, \varepsilon_n \mapsto \mathcal{V}_n^\varepsilon, \alpha_1 \mapsto \mathcal{V}_1^\alpha, \cdots, \alpha_m \mapsto \mathcal{V}_m^\alpha\}$, we have there exists a value v' such that $\Gamma' \models v : \tau \approx v'$ and $\mathcal{E} + \{y_1 \mapsto v_1, \cdots, y_n \mapsto v_n\} \vdash e \Downarrow v'$
- $\Gamma \models \mathcal{E} : TE \approx_E \mathcal{E}'$ iff $\mathrm{Dom}(\mathcal{E}) = \mathrm{Dom}(TE)$ and $\mathrm{Dom}(\mathcal{E}) \subseteq \mathrm{Dom}(\mathcal{E}')$ and for all $x \in \mathrm{Dom}(\mathcal{E})$ we have $\Gamma \models \mathcal{E}(x) : TE(x) \approx \mathcal{E}'(x)$. Further, for all $\varepsilon \in \mathrm{Dom}(E)$ we have $\Gamma \models \texttt{eq} : \varepsilon \times \varepsilon \to \mathit{bool} \approx \mathcal{E}'(E(\varepsilon))$

The following proposition states that a generated equality function for a given type has the expected semantics. We leave elimination of type information from typed expressions implicit.

Proposition 7. *If $E \vdash_{\texttt{eq}} \tau \Rightarrow e$ and for all $\varepsilon \in \mathrm{Dom}(E)$ we have $\Gamma \models \texttt{eq} : \varepsilon \times \varepsilon \to \mathit{bool} \approx \mathcal{E}(E(\varepsilon))$ then there exists a value v such that $\mathcal{E} \vdash e \Downarrow v$ and $\Gamma \models \texttt{eq} : \tau \times \tau \to \mathit{bool} \approx v$.*

Proof. By induction over the structure of τ. □

The semantic equivalence relation is closed with respect to substitution.

Proposition 8. *Let S be a substitution $(\{\varepsilon_1 \mapsto \tau_1, \cdots, \varepsilon_n \mapsto \tau_n\}, \{\alpha_1 \mapsto \tau_1', \cdots, \alpha_m \mapsto \tau_m'\})$. Define $\mathcal{V}_i^\varepsilon = \{(v, v') \mid \Gamma \models v : \tau_i \approx v'\}$, $i = 1..n$ and $\mathcal{V}_i^\alpha = \{(v, v') \mid \Gamma \models v : \tau_i' \approx v'\}$, $i = 1..m$.*

Then $\Gamma + \{\varepsilon_1 \mapsto \mathcal{V}_1^\varepsilon, \cdots \varepsilon_n \mapsto \mathcal{V}_n^\varepsilon, \alpha_1 \mapsto \mathcal{V}_1^\alpha, \cdots, \alpha_m \mapsto \mathcal{V}_m^\alpha\} \models v : \tau \approx v'$ iff $\Gamma \models v : S(\tau) \approx v'$.

Proof. By induction over the structure of τ. □

We can now state a semantic soundness proposition for the translation.

Proposition 9. (Semantic Soundness) *If $\Delta, TE \vdash e : \tau$ and $E \vdash e \Rightarrow e'$ and $\Gamma \models \mathcal{E} : TE \approx_E \mathcal{E}'$ and $\mathcal{E} \vdash e \Downarrow v$ then there exists a value v' such that $\mathcal{E}' \vdash e' \Downarrow v'$ and $\Gamma \models v : \tau \approx v'$.*

Proof. By induction over the structure of e. We show the three interesting cases.

CASE $e = x_{il}$, $il = ([\tau_1, \cdots, \tau_n], [\tau'_1, \cdots, \tau'_m])$, $n \geq 1$ From assumptions, (11), (5), the definition of semantic equivalence, and the definition of instantiation, we have $\Gamma \models v : \sigma \approx v''$ and $v'' = \mathcal{E}'(x)$ and $\sigma = \forall \varepsilon_1 \cdots \varepsilon_n \alpha_1 \cdots \alpha_m . \tau'$ and $TE(x) = \sigma$ and $S = (\{\varepsilon_1 \mapsto \tau_1, \cdots, \varepsilon_n \mapsto \tau_n\}, \{\alpha_1 \mapsto \tau'_1, \cdots, \alpha_m \mapsto \tau'_m\})$. Because $n \geq 1$, we have $v'' = \texttt{clos}(\lambda y_1. \cdots . \lambda y_n . e', \mathcal{E}'')$, for some lambda variables $y_1 \cdots y_n$, expression e', and dynamic environment $\mathcal{E}''$.

From assumptions and (34), we have $\Gamma \models \mathcal{E} : TE \approx_E \mathcal{E}'$ and $E \vdash_{\text{eq}} \tau_i \Rightarrow e_i$, $i = 1..n$, hence, we can apply Proposition 7 n times to get, there exist values v_i, $i = 1..n$ such that $\Gamma \models \texttt{eq} : \tau_i \times \tau_i \to \mathit{bool} \approx v_i$ and $\mathcal{E}' \vdash e_i \Downarrow v_i$, $i = 1..n$.

Letting $\mathcal{V}^{\varepsilon}_i = \{(v, v') | \Gamma \models v : \tau_i \approx v'\}$, $i = 1..n$ and $\mathcal{V}^{\alpha}_i = \{(v, v') | \Gamma \models v : \tau'_i \approx v'\}$, $i = 1..m$ and $\Gamma' = \Gamma + \{\varepsilon_1 \mapsto \mathcal{V}^{\varepsilon}_1, \cdots, \varepsilon_n \mapsto \mathcal{V}^{\varepsilon}_n, \alpha_1 \mapsto \mathcal{V}^{\alpha}_1, \cdots, \alpha_m \mapsto \mathcal{V}^{\alpha}_m\}$, we can apply Proposition 8 to get $\Gamma' \models \texttt{eq} : \varepsilon_i \times \varepsilon_i \to \mathit{bool} \approx v_i$, $i = 1..n$.

From the definition of semantic equivalence, we now have, there exists a value v' such that $\Gamma' \models v : \tau' \approx v'$ and $\mathcal{E}'' + \{y_1 \mapsto v_1, \cdots, y_n \mapsto v_n\} \vdash e' \Downarrow v'$. Now, because $v'' = \mathcal{E}'(x)$ and $\mathcal{E}' \vdash e_i \Downarrow v_i$, $i = 1..n$, we can derive $\mathcal{E}' \vdash (\cdots (x\ e_1) \cdots e_n) \Downarrow v'$ from (13), (11), and (12). By applying Proposition 8 again, we get $\Gamma \models v : \tau \approx v'$, as required.

CASE $e = \texttt{eq}_{\tau'}$ From assumptions, (17), (35), and the definition of semantic equivalence, we have from Proposition 7 that there exists a value v' such that $\mathcal{E}' \vdash e' \Downarrow v'$ and $\Gamma \models \texttt{eq} : \tau' \times \tau' \to \mathit{bool} \approx v'$, as required.

CASE $e = \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2$, $\sigma = \forall \varepsilon_1 \cdots \varepsilon_n \vec{\alpha} . \tau$, $n \geq 1$ Write $\vec{\alpha}$ in the form $\alpha_1 \cdots \alpha_m$. Let $\mathcal{V}^{\varepsilon}_1 \cdots \mathcal{V}^{\varepsilon}_n \mathcal{V}^{\alpha}_1 \cdots \mathcal{V}^{\alpha}_m$ be interpretations, let $v_1^{eq} \cdots v_n^{eq}$ be values, and let Γ' be a semantic environment such that $\Gamma' = \Gamma + \{\varepsilon_1 \mapsto \mathcal{V}^{\varepsilon}_1, \cdots, \varepsilon_n \mapsto \mathcal{V}^{\varepsilon}_n, \alpha_1 \mapsto \mathcal{V}^{\alpha}_1, \cdots, \alpha_m \mapsto \mathcal{V}^{\alpha}_m\}$ and $\Gamma' \models \texttt{eq} : \varepsilon_i \times \varepsilon_i \to \mathit{bool} \approx v_i^{eq}$, $i = 1..n$.

From assumptions and from (36), we have $y_1 \cdots y_n$ are chosen fresh and $E' \vdash e_1 \Rightarrow e'_1$ and $e''_1 = \lambda y_1 : \tau_1. \cdots . \lambda y_n : \tau_n . e'_1$, where $\tau_i = \varepsilon_i \times \varepsilon_i \to \mathit{bool}$, $i = 1..n$ and $E' = E + \{\varepsilon_1 \mapsto y_1, \cdots, \varepsilon_n \mapsto y_n\}$. From the definition of semantic equivalence, we can now establish $\Gamma' \models \texttt{eq} : \varepsilon \times \varepsilon \to \mathit{bool} \approx \mathcal{E}''(E'(\varepsilon))$, for all $\varepsilon \in \text{Dom}(E')$, where $\mathcal{E}'' = \mathcal{E}' + \{y_1 \mapsto v_1^{eq}, \cdots, y_n \mapsto v_n^{eq}\}$, and hence $\Gamma' \models \mathcal{E} : TE \approx_{E'} \mathcal{E}''$. From assumptions, (6), and (22), we have $\Delta \cup \text{fetv}(\varepsilon_1 \cdots \varepsilon_n)$, $TE \vdash e_1 : \tau$ and $\mathcal{E} \vdash e_1 \Downarrow v_1$ and because we have $E' \vdash e_1 \Rightarrow e'_1$, we can apply induction to get, there exists a value v'_1 such that $\mathcal{E}'' \vdash e'_1 \Downarrow v'_1$ and $\Gamma' \models v_1 : \tau \approx v'_1$.

Letting $v''_1 = \texttt{clos}(\lambda y_1. \cdots . \lambda y_n . e'_1, \mathcal{E})$, we have from the definition of semantic equivalence that $\Gamma \models v_1 : \sigma \approx v''_1$ and $\Gamma \models \mathcal{E} + \{x \mapsto v_1\} : TE + \{x \mapsto \sigma\} \approx_E \mathcal{E}' + \{x \mapsto v''_1\}$. From assumptions and from (22), (6), and (36), we have $\mathcal{E} + \{x \mapsto v_1\} \vdash e_2 \Downarrow v_2$ and $\Delta, TE + \{x \mapsto \sigma\} \vdash e_2 : \tau'$ and $\mathcal{E} \vdash e_2 \Rightarrow e'_2$, hence, we can apply induction a second time to get, there exists a value v'_2 such that $\mathcal{E}' + \{x \mapsto v''_1\} \vdash e'_2 \Downarrow v'_2$ and $\Gamma \models v_2 : \tau' \approx v'_2$.

We can now apply (12) to get $\mathcal{E}' \vdash e''_1 \Downarrow v''_1$, hence, we can apply (22) to get $\mathcal{E}' \vdash e' \Downarrow v'_2$, as required. □

We now return to the value restriction issue. The translation rule for the let-construct does not preserve semantics unless (1) e_1 is known to terminate

and not to have side effects or (2) no equality type variables are generalised. In the language we consider, (1) is always satisfied. For Standard ML, the value restriction always enforces either (1) or (2). However, the restriction is enforced by limiting generalisation to so called non-expansive expressions, which include function applications. Adding such a requirement to the typing rule for the let-construct makes too few programs typable; to demonstrate type correctness for the translation, applications of functions to generated equality functions must also be considered non-expansive.

9 Extension to Full ML

It is straightforward to extend equality elimination to allow imperative features and to allow a letrec-construct for declaration of recursive functions. We now demonstrate how the approach is extended to deal with parametric datatypes and modules.

9.1 Datatype Declarations

In Standard ML, lists may be implemented by a datatype declaration

$$\texttt{datatype}\ \alpha\ \mathit{list} = :: \ \texttt{of}\ \alpha \times \alpha\ \mathit{list}\ \mid\ \mathit{nil}$$

Because lists are declared to be parametric in the type of the elements, it is possible to write polymorphic functions to manipulate the elements of any list. In general datatype declarations may be parametric in any number of type variables and they may even be declared mutually recursive with other datatype declarations. The datatype declaration for lists elaborates to the type environment

$$\{\mathit{list} \mapsto (t,\ \{:: \mapsto \forall\alpha.\alpha \times \alpha\ t \to \alpha\ t,\ \mathit{nil} \mapsto \forall\alpha.\alpha\ t\})\}$$

where t is a fresh type name [11]. Every type name t possess a boolean attribute that denotes whether t admits equality. In the example, t will indeed be inferred to admit equality. This property of the type name t allows values of type $\tau\ t$ to be checked for equality if τ admits equality.

When a datatype declaration elaborates to a type environment, an equality function is generated for every fresh type name t in the type environment such that t admits equality. For a parametric datatype declaration, such as the *list* datatype declaration, the generated equality function is parametric in equality functions for parameters of the datatype.

The Kit does not allow all valid ML programs to be compiled using equality elimination. Consider the datatype declaration

$$\texttt{datatype}\ \alpha\ t = A\ \texttt{of}\ (\alpha \times \alpha)\ t\ \mid\ B\ \texttt{of}\ \alpha$$

Datatypes of the above form are called *non-uniform* datatypes [15, page 86]. It is possible to declare non-uniform datatypes in ML, but they are of limited

use, because ML does not support polymorphic recursive functions. In particular, it is not possible to declare a function in ML that checks values of non-uniform datatypes for structural equality. However, the problem is not inherent to equality elimination. Adding support for polymorphic recursion in the intermediate language would solve the problem. Other compilation techniques also have troubles dealing with non-uniform datatypes. The TIL compiler developed at Carnegie Mellon University does not support non-uniform datatypes due to problems with compiling constructors of such datatypes in the framework of intensional polymorphism [12, page 166].

9.2 Modules

The translation extends to Standard ML Modules [11]. However, to compile functors separately, structures must contain equality functions for each type name that admits equality and that occurs free in the structure. Moreover, when constraining a structure to a signature, it is necessary to enforce the implementation of a function to follow its type by generating appropriate stub code. The body of a functor may then uniformly extract equality functions from the formal argument structure.

10 Implementation

The Kit compiles the Standard ML Core language by first elaborating and translating programs into an intermediate typed lambda language. At this point, polymorphic equality is eliminated. Then, a simple optimiser performs various optimisations inspired by [1] and small recursive functions are specialised as suggested in [17].

The remaining phases of the compiler are based on region inference [24]. Each value generated by the program resides in a region and region inference is the task of determining when to allocate and deallocate regions. Various analyses determine how to represent different regions at runtime [3]. Some regions can be determined to only ever contain word-sized unboxed values, such as integers and booleans. Such regions need never be allocated. Other regions can be determined to only ever hold one value at runtime. Such regions may be implemented on the stack. Other regions are implemented using a stack of linked pages.

The backend of the Kit implements a simple graph coloring technique for register allocation and emits code for the HP PA-RISC architecture [5].

10.1 Datatype Representation

The Kit supports different schemes for representing datatypes at runtime. The simplest scheme implements all constructed values (except integers and booleans) as boxed objects at runtime. Using this scheme, the list $[1, 2]$, for instance, is represented as shown in Fig. 1.

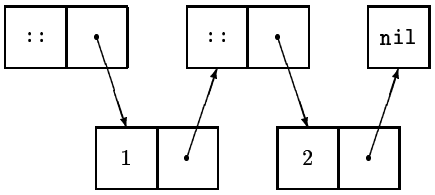

Fig. 1. Boxed representation of the list [1, 2] with untagged integers.

The Standard ML of New Jersey compiler version 110 (SML/NJ) implements lists as shown in Fig. 2, using the observation that pointers are four-aligned on most modern architectures [1]. In this way, the two least significant bits of pointers to constructed values may be used to represent the constructor. However, because SML/NJ implements polymorphic equality and garbage collection by following pointers, only one bit remains to distinguish constructed values.

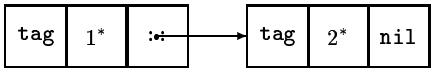

Fig. 2. Unboxed representation of the list [1, 2] with tagged tuples and tagged integers.

Utilising the two least significant bits of pointers to constructed values, we say that a type name associated with a datatype declaration is *unboxed* if the datatype binding declares at-most three unary constructors (and any number of nullary constructors) and for all argument types τ for a unary constructor, τ is not a type variable and τ is not unboxed (for recursion, we initially assume that the declared type names of the declaration are unboxed.) A type τ is *unboxed* if it is on the form $(\tau_1, \ldots, \tau_n)\ t$ and t is unboxed.

The Kit treats all values of unboxed types as word-sized unboxed objects. Using this scheme, lists are represented uniformly at runtime as shown in Fig. 3. Efficient unboxed representations of many tree structures are also obtained using this scheme.

In the context of separate compilation of functors, as implemented in Standard ML of New Jersey, version 0.93, problems arise when a unique representation of datatypes is not used [2]. If instead functors are specialised for each application, no restrictions are enforced on datatype representations and no representation overhead is introduced by programming with Modules. Current research addresses this idea.

11 Experimental Results

In this section, we present some experimental results obtained with the Kit and the Standard ML of New Jersey compiler version 110 (SML/NJ). The purpose of the experiments are (1) to assess the feasibility of eliminating polymorphic

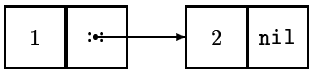

Fig. 3. Unboxed representation of the list $[1, 2]$ with untagged tuples and untagged integers.

equality, (2) to assess the importance of efficient datatype representations, and (3) to compare the code generated by the Kit with that generated by SML/NJ.

All tests are run on a HP PA-RISC 9000s700 computer. For SML/NJ, executables are generated using the `exportFn` built-in function. We use `KitT` to mean the Kit with a tagging approach to implement polymorphic equality. Further, `KitE` is the Kit with equality elimination enabled. In `KitE`, tagging of values is disabled as no operations need tags at runtime. Finally, `KitEE` is `KitE` with efficient representation of datatypes enabled. All versions of the Kit generate efficient equality checks for values that are known to be of base type (e.g., int or real).

Measurements are shown for eight benchmark programs. Four of these are non-trivial programs based on the SML/NJ distribution benchmarks (`life`, `mandelbrot`, `knuth-bendix` and `simple`). The program `fib35` is the simple Fibonacci program and `mergesort` is a program for sorting 200,000 pseudo-random integers. The programs `life` and `knuth-bendix` use polymorphic equality extensively. The program `lifem` is a monomorphic version of `life` for which polymorphic functions are made monomorphic by insertion of type constraints. The program `sieve` computes all prime numbers in the range from 1 to 2000, using the Sieve of Eratosthenes.

Running times for all benchmarks are shown in Fig. 4. Equality elimination, and thus, elimination of tags, appears to have a positive effect on the running time for most programs. In particular, the `life` benchmark runs 48 percent faster under `KitE` than under `KitT`. However, programs do exist for which equality elimination has a negative effect on the running time of the program. There are potentially two reasons for a slowdown. First, extra function parameters to equality polymorphic functions may lead to less efficient programs. Second, functions generated by `KitE` and `KitEE` for checking two structural values for equality do not check if the values are located on the same address. This check is performed by the polymorphic equality primitive of `KitT`. In principle, such a check could also be performed by equality functions generated by `KitE` and `KitEE`. The `knuth-bendix` benchmark runs slightly slower under `KitE` than under `KitT`.

Not surprisingly, efficient representation of datatypes improves the running time of most programs – with up to 40 percent for the `sieve` benchmark.

The Kit does not implement the minimum typing derivation technique for decreasing the degree of polymorphism [4]. Decreasing the degree of polymorphism has been reported to have a great effect on performance; it makes it possible to transform slow polymorphic equality tests into fast monomorphic ones [19,18]. Due to the decrease in polymorphism, the `lifem` benchmark is 12 percent faster than the `life` benchmark (under `KitEE`).

Running time	KitT	KitE	KitEE	SML/NJ
fib35	10.9	10.2	10.1	18.5
sieve	9.01	6.18	3.71	9.24
life	35.4	18.5	18.1	5.28
lifem	35.2	16.2	16.0	5.25
mergesort	12.9	11.9	9.25	15.9
mandelbrot	35.4	32.3	31.9	7.17
knuth-bendix	26.4	26.7	23.3	17.7
simple	47.1	40.7	40.6	15.5

Fig. 4. Running times in seconds for code generated by three versions of the Kit and SML/NJ, measured using the UNIX `time` program.

Space usage for the different benchmarks is shown in Fig. 5. No benchmark program uses more space due to elimination of equality. For programs allocating a large amount of memory, equality elimination, and thus, elimination of tags, reduces memory significantly – with up to 31 percent for the `simple` program. Efficient datatype representation reduces space usage further up to 33 percent for the `mergesort` program.

Space usage	KitT	KitE	KitEE	SML/NJ
fib35	108	108	108	1,380
sieve	1,248	1,052	736	6,180
life	428	376	272	1,408
lifem	428	376	272	1,420
mergesort	16,000	13,000	8,728	18,000
mandelbrot	304	296	296	712
knuth-bendix	4,280	3,620	2,568	2,724
simple	1,388	960	748	2,396

Fig. 5. Space used for code generated by the three versions of the Kit and SML/NJ. All numbers are in kilobytes and indicate maximum resident memory used, measured using the UNIX `top` program.

Sizes of executables for all benchmarks are shown in Fig. 6. Equality elimination does not seem to have a dramatic effect on the sizes of the executables. Efficient datatype representation reduces sizes of executables with up to 22 percent for the `life` benchmark.

The Kit and SML/NJ are two very different compilers. There can be dramatic differences between using region inference and reference tracing garbage collection, thus, the numbers presented here should be read with caution. The Kit currently only allows an argument to a function to be passed in one register. Moreover, the Kit does not allocate floating point numbers in registers. Instead, floating point numbers are always boxed. The benchmark programs `mandelbrot` and `simple` use floating point operations extensively. No doubt, efficient calling

Program size	KitT	KitE	KitEE	SML/NJ
fib35	0	0	0	0
sieve	16	20	16	29
life	92	92	72	17
lifem	92	92	72	17
mergesort	20	24	20	40
mandelbrot	8	12	12	37
knuth-bendix	160	168	140	71
simple	356	352	328	199

Fig. 6. Sizes of executables (with the size of the empty program subtracted) for code generated by three versions of the Kit and SML/NJ. All numbers are in kilobytes.

conventions and register allocation of floating point numbers will improve the quality of the code generated by the Kit.

12 Conclusion

The translation suggested in this paper makes it possible to eliminate polymorphic equality completely in the front-end of a compiler. Experimental results show that equality elimination can lead to important space and time savings even for programs that use polymorphic equality.

Although tags may be needed at runtime to implement reference tracing garbage collection, it is attractive to eliminate polymorphic equality at an early stage during compilation. Various optimisations, such as boxing analysis [9,7], must otherwise treat polymorphic equality distinct from other primitive operations. Checking two arbitrary values for equality may cause both values to be traversed to any depth. This is in contrast to how other polymorphic functions behave. Further, no special demands are placed on the implementor of the runtime system and the backend of the compiler. For instance, there is no need to flush all values represented in registers into the heap prior to testing two values for equality.

Acknowledgements. I would like to thank Lars Birkedal, Niels Hallenberg, Fritz Henglein, Tommy Højfeld Olesen, Peter Sestoft, and Mads Tofte for valuable comments and suggestions.

References

1. Andrew Appel. *Compiling With Continuations.* Cambridge University Press, 1992.
2. Andrew Appel. A critique of Standard ML. In *Journal of Functional Programming*, pages 3(4):391–429, October 1993.
3. Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23st ACM Symposium on Principles of Programming Languages*, January 1996.

4. Nikolaj Bjørner. Minimal typing derivations. In *ACM Workshop on Standard ML and its Applications*, June 1994.
5. Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project, July 1995.
6. Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical report, Carnegie Mellon University, June 1997. CMU-CS-97-147.
7. Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *21st ACM Symposium on Principles of Programming Languages*, pages 213–226, January 1994.
8. Mark Jones. Dictionary-free overloading by partial evaluation. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida*, June 1994.
9. Xavier Leroy. Unboxed objects and polymorphic typing. In *19th ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.
10. Xavier Leroy. The Objective Caml system. Software and documentation available on the Web, 1996.
11. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
12. Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1995.
13. Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *7'th International Conference on Functional Programming and Computer Architecture*, June 1995.
14. Atsushi Ohori. A Polymorphic Record Calculus and its Compilation. *ACM Transactions on Programming Languages and Systems*, 17(6), November 1995.
15. Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, September 1996.
16. John Peterson and Mark Jones. Implementing type classes. In *ACM Symposium on Programming Language Design and Implementation*, June 1993.
17. Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Second International Symposium on Static Analysis*, pages 366–381, September 1995.
18. Zhong Shao. Typed common intermediate format. In *1997 USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct 1997.
19. Zhong Shao and Andrew Appel. A type-based compiler for Standard ML. Technical report, Yale University and Princeton University, November 1994.
20. David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Symposium on Programming Language Design and Implementation*, 1996.
21. David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, 1996.
22. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.
23. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical report, Department of Computer Science, University of Copenhagen, April 1997.
24. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
25. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, January 1989.