An Abstract Interpretation for ML Equality Kinds[*]

Carl A. Gunter
University of Pennsylvania

Elsa L. Gunter
AT&T Bell Laboratories

David B. MacQueen
AT&T Bell Laboratories

March 1991

Abstract

The definition of Standard ML provides a form of generic equality which is inferred for certain types, called *equality types,* on which it is possible to define an equality relation in ML. However, the standard definition is incomplete in the sense that there are interesting and useful types which are *not* inferred to be equality types but for which an equality relation can be defined in ML in a uniform manner. In this paper, a refinement of the Standard ML system of equality types is introduced and is proven sound and *complete* with respect to the existence of a definable equality. The technique used here is based on an abstract interpretation of ML operators as monotone functions over a three point lattice. It is shown how the equality relation can be defined (as an ML program) from the definition of a type with our equality property. Finally, a sound, efficient algorithm for inferring the equality property which corrects the limitations of the standard definition in all cases of practical interest is demonstrated.

1 Equality Types in Standard ML

The ML language provides an extensible algebra of type constructions. The Standard ML dialect divides types into two classes, those which *admit equality* (also called *equality types*) and those which do not. This distinction is based on the structure of types. Primitive types like `int` and `string` have a predefined equality operation, while equality can be defined over compound types built up from primitive types using "concrete" constructions like product and disjoint union in the usual componentwise manner. Function types on the other hand do not posses a definable equality relation (the existence of such a relation would solve the halting problem), nor do user-defined abstract types (the compiler cannot determine when two concrete representations correspond to the same abstract value). As a first approximation, therefore, the types admitting equality can be identified with the "hereditarily concrete" types built from primitive types using concrete type

[*]To appear in: **Information and Computation.**

constructions. Some recursively defined datatype constructors such as `list` also qualify as concrete, producing equality types when applied to equality types.

Having identified a class of types possessing canonical equality operations, the next step is to introduce a restricted form of polymorphism by abstracting over polymorphic type variables which are constrained to range only over such types. Using this restricted form of polymorphism one can define functions like the generic list membership function:

```
fun member x nil = false
  | member x (y::r) = if x = y then true else member x r
```

which searches a list for an appearance of a value. The type inferred for `member` is

```
member : ''a * ''a list -> bool
```

where `''a` is a polymorphic type variable ranging over equality types. (In Standard ML, an ordinary polymorphic type variable ranging over arbitrary types begins with a single quote, e.g. `'a`.)

This paper addresses some problems that arise from oversimplifications in the treatment of equality types in the Definition of Standard ML [MTH90, MT91]. We propose a refined treatment of equality types using equality *kinds* defined in terms of an abstract interpretation of type expressions that we prove to be sound and complete with respect to the denotational semantics of Standard ML types.

In the Definition of Standard ML, a unary type constructor `'a F` is said to *admit equality* if `t F` is an equality type whenever the parameter type `t` is. A constructed type `t F` admits equality only if both `t` and `F` admit equality. This extends to n-ary type constructors in the obvious way. Unfortunately, this definition is incomplete for the inference of equality properties because of the presence of certain special type constructors that have stronger equality properties. For example, the type `t ref` admits equality regardless of whether `t` does. Therefore a type constructor defined as

```
datatype ('a,'b) F = mkF of 'a * 'b ref
```

has a more complex equality preservation behavior than the standard definition is capable of expressing.[1] For example, `(int, unit->int) F` should admit equality even though one of its arguments, `(unit->int)`, does not. However this will not be inferred based on the definition.

To correct this problem requires a more precise notion of equality properties of type constructors. A simple binary property distinguishing between type constructors which admit equality and those which do not must be replaced by an *equality kind* that specifies how the equality property of the result depends on the equality properties of the arguments of the constructor.

[1]Indeed, an example very similar to this one was sent as a 'bug report' to the implementors of the Standard ML of New Jersey compiler.

We start in Section 2 by developing a standard denotational interpretation of types and an abstract interpretation mapping types into a three point lattice $\mathcal{E} = \{\mathsf{void}, \mathsf{eq}, \mathsf{type}\}$, where the top element type represents arbitrary types, the middle element eq represents types admitting equality and the bottom element void represents empty types (*i.e.* types containing no defined elements).

This abstract interpretation of types is also extended to type constructors, whose interpretations will be mappings from appropriate products of $\mathcal{E}$ to $\mathcal{E}$. To interpret recursively defined constructors we simply calculate a least fixed point of the abstract interpretations. For example, if we define

```
datatype ('a,'b) F = A of 'a | B of (unit -> 'b, 'b) G
     and ('c,'d) G = C of ('c ref, 'd) F | D of 'd
```

then neither F nor G are considered to admit equality according to the Standard ML definition. However, both F and G admit equality under the abstract interpretation, and their interpretations f, g satisfy $f(\mathsf{eq}, \mathsf{eq}) = \mathsf{eq}$ and $g(\mathsf{type}, \mathsf{eq}) = \mathsf{eq}$ respectively.

In Section 3 we relate the denotational and abstract interpretations by showing that the denotation of a type is a flat domain if and only if the abstract interpretation of that type is eq.

In Section 4 we show that if a type has eq as its abstract interpretation we can define an equality relation for that type in ML. This involves defining equality functionals corresponding to the type constructors used to build the type. It is also shown that only the flat domains may have a definable equality relation, so that the equality types are exactly the types having a definable equality. Hereafter when we speak of a "definable" relation, we mean definability of the relation as an ML program.

The structure of the recursive definition of a type constructor is used as a format for creating a recursive definition of the corresponding equality function. In the case of F and G, this recursive function is parameterized by equality tests for 'a, 'b, 'd and a "dummy parameter" for 'c. In fact, the equality test for 'c will be never be invoked in an equality test for a type built with F or G because it is not used to compute equality on 'c ref.

In Section 5 we show that by avoiding void the abstract interpretation can be simplified, and *equality kinds* are introduced as succinct characterizations of the interpretation of type constructors. In practice, normal type constructor definitions are indeed "void-avoiding".

We conclude by discussing future research directions, particularly the interaction of equality types and ML modules, and the impact of equality types on implementations of ML.

2 Interpretations of Types

For the purposes of this paper, we shall assume that the expressions of the type algebra in ML are given by the following grammar:

$$t ::= \mathbf{void} \mid \mathbf{unit} \mid t * t \mid t + t \mid t \to t \mid \mathbf{ref}\ t \mid F_i(t_1, \ldots, t_{n_i}) \qquad (i = 1, \ldots m)$$

where the type constructors $F_1, \ldots F_m$ are all the user-defined datatypes. Associated with the user-defined type constructors there is a system of equations

$$\begin{aligned} F_1(\alpha_1, \ldots, \alpha_{n_1}) &= \mathcal{F}_1(F_1, \ldots, F_m)(\alpha_1, \ldots, \alpha_{n_1}) \\ &\vdots \\ F_m(\alpha_1, \ldots, \alpha_{n_m}) &= \mathcal{F}_m(F_1, \ldots, F_m)(\alpha_1, \ldots, \alpha_{n_m}). \end{aligned}$$

which any interpretation should treat as one large mutual recursion. For example, the recursive definition from the previous section would use the following operators:

$$\begin{aligned} \mathcal{F} &= \lambda(F,G)\lambda(\alpha,\beta).\alpha + G(\mathbf{unit} \to \beta, \beta) \\ \mathcal{G} &= \lambda(F,G)\lambda(\gamma,\delta).F(\mathbf{ref}\ \gamma, \delta) + \delta \end{aligned}$$

We shall refer to the non-recursive type constructors, namely **void**, **unit**, $*$, $+$, $\to$, and **ref**, as *basic* constructors. In the above recursive equations, the functions $\mathcal{F}_i$ are second-order λ-expressions over our type algebra, containing no free occurrences of the symbols $F_1, \ldots, F_m$. That is, the bodies of the functions $\mathcal{F}_i$ are composed only of first-order and second-order bound variables and basic constructors.

In this section we wish to define two interpretations of these type expressions, one domain-theoretic and the other an abstract interpretation. In order to do so, it will be beneficial in both cases to have associated with each of our recursive constructors, $F_1, \ldots, F_m$, a sequence of functions which are first-order λ-expressions over our basic type algebra whose bodies are composed only of first-order variables and basic constructors. Under each of the interpretations these functions will provide finite approximates to the recursive constructors. These functions are given by the following recursive definition:

$$\begin{aligned} F_i^0(\alpha_1, \ldots, \alpha_{n_i}) &= \mathbf{void} \\ F_i^{j+1}(\alpha_1, \ldots, \alpha_{n_i}) &= \mathcal{F}_i(F_1^j, \ldots, F_m^j)(\alpha_1, \ldots, \alpha_{n_1}). \end{aligned}$$

With these we are now in a position to describe our two interpretations.

2.1 A Domain-theoretic Interpretation

We now sketch the standard fixed-point semantics of ML's types. To do this we must briefly introduce some domain-theoretic terminology. A somewhat fuller discussion of domain theory can be found in several sources (see [GS90] and the references there). A subset $M \subseteq D$ of a poset D is *directed* if, for every finite set $u \subseteq M$, there is an upper bound $x \in M$ for u. D is a *complete* partial order (cpo) if every directed subset $M \subseteq D$ has a least upper bound $\bigvee M$ and there is a least element $\perp_D$ in D. To interpret ML's types, we need a collection of operators on cpo's.

Given cpo's D and E, we define the *coalesced sum* $D \oplus E$ to be the set

$$\Big((D - \{\bot_D\}) \times \{0\}\Big) \cup \Big((E - \{\bot_E\}) \times \{1\}\Big) \cup \{\bot_{D\oplus E}\}$$

where $D-\{\bot_D\}$ and $E-\{\bot_E\}$ are the sets D and E with their respective bottom elements removed and $\bot_{D\oplus E}$ is a new element which is not a pair. It is ordered by taking $\bot_{D\oplus E} \leq z$ for all $z \in D \oplus E$ and taking $(x,m) \leq (y,n)$ if and only if $m = n$ and $x \leq y$.

Given a cpo D, we define the *lift* of D to be the poset obtained by adding a new bottom to D. More precisely, the set $D_\bot = (D \times \{0\}) \cup \{\bot\}$, where $\bot$ is a new element which is not a pair, together with a partial ordering $\leq$ which is given by stipulating that $(x,0) \leq (y,0)$ whenever $x \leq y$ and $\bot \leq z$ for every $z \in D_\bot$.

For cpo's D and E, the *smash product* $D \otimes E$ is the set

$$\{(x,y) \in D \times E \mid x \neq \bot \text{ and } y \neq \bot\} \cup \{\bot_{D\otimes E}\}$$

where $\bot_{D\otimes E}$ is some new element which is not a pair. The ordering on pairs is coordinatewise and we stipulate that $\bot_{D\otimes E} \leq z$ for every $z \in D \otimes E$.

The two point lattice $\mathbf{1}$ is a unit for the smash product: $D \otimes \mathbf{1} \cong \mathbf{1} \otimes D \cong D$. The one point lattice $\mathbf{0}$ is a unit for the coalesced sum, $D \oplus \mathbf{0} \cong \mathbf{0} \oplus D \cong D$, and an eliminator for the smash product, $D \otimes \mathbf{0} \cong \mathbf{0} \otimes D \cong \mathbf{0}$. A cpo is said to be *void* if it is isomorphic to $\mathbf{0}$. A cpo D is said to be *flat* if it is not void and any two distinct elements of D are comparable only when one of them is $\bot$. Up to isomorphism, there is a unique countably infinite flat cpo which we denote $N_\bot$. The domain $\mathbf{B}$ of *booleans* is the flat domain with three distinct elements $\mathsf{true}, \mathsf{false}, \bot$. The *equality function* $=_D$ on a flat domain D is a mapping from $D \otimes D$ into $\mathbf{B}$ such that

- $=_D (x,y)$ is true when $x = y \neq \bot$
- $=_D (x,y)$ is false when $x, y \neq \bot$ and $x \neq y$
- $=_D (x,y)$ is $\bot$ when x or y is $\bot$.

A monotone function between two cpo's is *continuous* if it preserves least upper bounds of directed collections. A function between cpo's is *strict* if it takes $\bot$ to $\bot$. Given two cpo's D and E, the space of all strict continuous functions between D and E, denoted by $D \circ\!\!\to E$, is again a cpo under the point-wise ordering.

For an ML type expression t, let $\check{t}$ be the standard domain-theoretic interpretation of t. This definition can be given inductively as follows. First of all, we define $\check{\mathbf{void}} = \mathbf{0}$ and $\check{\mathbf{unit}} = \mathbf{1}$. The interpretations of the basic constructors are defined on domains D and E as follows:

- $D \mathbin{\check{*}} E = D \otimes E$
- $D \mathbin{\check{+}} E = D \oplus E$

- $D \check{\rightarrow} E = (D \circ\!\!\!\rightarrow E)_\perp$
- $\check{\mathbf{ref}}\ \mathbf{0} = \mathbf{0}$
- $\check{\mathbf{ref}}\ D = N_\perp \qquad \text{if } D \neq \mathbf{0}$

The interpretation of recursive types can be given as described in [SP82] using colimits. These methods also apply to provide a semantics for the recursively defined type constructors provided by ML. For example, an ML definition of lists such as

```
datatype 'a list = Cons of 'a * 'a list | Nil
```

is a recursive definition of a constructor `list`. At the domain-theoretic level, this is a recursive definition of a *functor.* The solution is obtained as a colimit of a sequence of functors, where the colimit is obtained in a category of functors and natural transformations. To prove that the equality functions we define later as ML programs are indeed the ones we expect, it is essential for us to know something about the exact mathematical operator which we obtain as the solution of this equation.

Fortunately, it is not necessary to work in a functor category in order to do this. Instead, we can employ a technique of Scott which uses a *universal domain.* The first use of the idea appears in [Sco76] using what one might call a "closure-universal" domain, but we will employ a related technique introduced in [Sco82b, Sco82a] using a "projection-universal" domain. Both techniques are described and illustrated in [GS90]. For the purposes of the remainder of this paper, a *domain* is a bounded complete algebraic cpo (these are sometimes called "Scott domains"). It will not be necessary for us to define these structures here since we will simply rely on properties of their universal domain. The universal domain technique can be summarized as follows. Given a domain D, let us say that a subset E is a subdomain of D and write $E \lhd D$ if E forms a domain under the ordering inherited from D and there is a projection from D onto E, *i.e.* there is is a mapping $p : D \to E$ such that $p \circ p = p$ and $p(x) \leq x$ for each $x \in D$. Roughly speaking, a *universal domain* is a domain U which has a copy of every other domain D as a subdomain, *i.e.* $D \lhd U$ up to isomorphism. Moreover, the set of all subdomains of U again forms a domain, and hence there is a special subdomain $T \lhd U$, called the *type of types*, which is isomorphic to the domain of subdomains of U. More specifically, there is a bijection τ between T and the domain of subdomains of U such that $D \lhd E$ iff $\tau(D) \leq \tau(E)$ for any pair of subdomains $D, E \lhd U$. In the remainder of the paper we will make no distinction between a domain D (which is to be viewed as a subdomain of U) and its image in T under τ.

The existence of a universal domain allows us to interpret operators on *types* as continuous functions on the *domain* T. For example, the function space operator $\to$ can be viewed as a continuous function from $T \times T$ into T. Hence, a fixed point specification such as the one given for `list` above can be solved as a fixed point equation over a cpo without the need to introduce functor

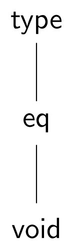

Figure 1: The Equality Properties Lattice $\mathcal{E}$.

categories, *etc.* explicitly (see Theorem 7.10 of [GS90]). Therefore, if D_1, …, D_{n_i} are domains, then we can define

$$\check{F}_i(D_1, \ldots, D_{n_i}) = \bigvee(\check{F_i^j}(D_1, \ldots, D_{n_i}))$$

where the least upper bound is being taken in T and the isomorphism between domains and elements of T is being taken for granted.

Another useful perspective that we are able to obtain by working in a universal domains is a simple way to compare functions between domains. If we are given a continuous function $f : D \to E$ between subdomains D and E, then we may view this as a continuous function $f' : U \to U$ where $f'(x) = f(p(x))$ where p is the projection onto D. In particular, if $D \lhd D'$ are flat subdomains, then their equality functions are related $=_D \leq =_{D'}$ (where we are suppressing the distinction between the equality functions on the domains and their extensions to all of U). One further note which will be important to our discussion later is that when we have a chain of flat domains $D_0 \lhd D_1 \lhd \cdots$, then their limit in T corresponds to their *union* $\bigcup_i D_i$ as subdomains of U. In particular, the limit of their equality functions is the equality function on their limit (the union of the D_i's).

2.2 An Abstract Interpretation

Next we wish to describe an abstract interpretation function mapping closed ML type expressions into the three point lattice $\mathcal{E}$ pictured in Figure 1. To do so, we will define the interpretation on the constructors and extend by structural induction to closed type expressions. For any $v_1, v_2 \in \mathcal{E}$, we have:

- $\hat{\textbf{void}} = \mathsf{void}$
- $\hat{\textbf{unit}} = \mathsf{eq}$
- $v_1 \mathbin{\hat{+}} v_2 = \max\{v_1, v_2\}$
- If either $v_1 = \mathsf{void}$ or $v_2 = \mathsf{void}$ then $v_1 \mathbin{\hat{*}} v_2 = \mathsf{void}$, if $v_1 = v_2 = \mathsf{eq}$ then $v_1 \mathbin{\hat{*}} v_2 = \mathsf{eq}$ and otherwise $v_1 \mathbin{\hat{*}} v_2 = \mathsf{type}$

- If $v_1 = \mathsf{void}$ or $v_2 = \mathsf{void}$, then $v_1 \hat{\to} v_2 = \mathsf{eq}$, and otherwise $v_1 \hat{\to} v_2 = \mathsf{type}$
- $\hat{\mathbf{ref}}\ \mathsf{void} = \mathsf{void}$ and if $v \neq \mathsf{void}$ then $\hat{\mathbf{ref}}\ v = \mathsf{eq}$.

Notice that each of the the basic constructors is interpreted as a monotone function over the n-ary product ($n = 0, 1, 2$) of $\mathcal{E}$ with itself.

Having defined our interpretation for the basic constructors, by structural induction we have the interpretations $\hat{F}_i^j$ for the functions F_i^j, since they are composed only of basic constructors. Using these, we define the interpretations of the recursive constructors by

$$\hat{F}_i(v_1, \ldots, v_{n_i}) = \max_j \{\hat{F}_i^j(v_1, \ldots, v_{n_i})\}.$$

By structural induction, we can in fact extend our interpretation function to the second-order functions $\mathcal{F}_i$. Since these are also composed only of basic constructors (and first- and second-order bound variables), and since all the basic constructors are interpreted as monotone functions, the interpretation of $\mathcal{F}_i$ will itself be a function which is monotone in both its first-order and second-order arguments.

Lemma 1 *For all i, j, we have $\hat{F}_i^j \leq \hat{F}_i^{j+1}$. Moreover, there exists a k such that for all i, $\hat{F}_i^k = \hat{F}_i^{k+1}$, and hence for all i, $\hat{F}_i = \hat{F}_i^k$.*

Proof. For the first part, the proof is by induction on j. Suppose $v_1, \ldots, v_{n_i} \in \mathcal{E}$. For the base step, $\hat{F}_i^0(v_1, \ldots, v_{n_i}) = \mathsf{void} \leq \hat{F}_i^1(v_1, \ldots, v_{n_i})$. For the inductive step suppose that for all i

$$\hat{F}_i^{j-1}(v_1, \ldots, v_{n_i}) \leq \hat{F}_i^j(v_1, \ldots, v_{n_i})$$

Then, by applying $\hat{\mathcal{F}}_i$, for each i, since $\hat{\mathcal{F}}_i$ is monotonic, we have

$$\hat{\mathcal{F}}_i(\hat{F}_1^j, \ldots, \hat{F}_m^j)(v_1, \ldots, v_{n_i}) \leq \hat{\mathcal{F}}_i(\hat{F}_1^{j+1}, \ldots, \hat{F}_m^{j+1})(v_1, \ldots, v_{n_i})$$

and hence

$$\hat{F}_i^j(v_1, \ldots, v_{n_i}) \leq \hat{F}_i^{j+1}(v_1, \ldots, v_{n_i}).$$

For the second part, since the set of functions mapping $\mathcal{E}^{n_i}$ into $\mathcal{E}$ is finite and the $\hat{F}_i^j$'s form an increasing sequence, it is immediate that there exists a k such that for all i we have $\hat{F}_i^k = \hat{F}_i^{k+1}$. By the definition of the F_i^j's, for all $j > k$ we therefore have $\hat{F}_i^j = \hat{F}_i^k$. Again since the $\hat{F}_i^j$'s form an increasing sequence, we have that $\hat{F}_i = max_j\{\hat{F}_i^j\} = \hat{F}_i^k$. ▮

Notice that the previous lemma tells us that the computation of the $\hat{F}_i$'s is a finite process.

3 Relating Interpretations

The purpose of this section is to demonstrate that, for any type expression t, the standard interpretation $\check{t}$ is flat if and only if $\hat{t} = \mathsf{eq}$. This describes the soundness and completeness property of our interpretation. Because of the presence of recursive definitions and the constant type **void** itself, it is necessary to deal with the possibility that there are type expressions t such that every program of type t is divergent. Evidently, it is possible to define a type directly in terms of itself:

```
datatype money = Invest of money
```

No programs of this type converge. But this could happen more subtly in a mutually recursive definition:

```
datatype chicken = Hatch of egg
       | egg = Lay of chicken
```

(After all, which comes first?) The following definition and lemma show how such types are abstractly interpreted as void.

Definition 2 A closed type expression t *has property* $\mathcal{V}$ provided that $\hat{t} = \mathsf{void}$ iff $\check{t} = \mathbf{0}$. ▮

Lemma 3 *1. The types* **void** *and* **unit** *both have property* $\mathcal{V}$.

2. If the types t_1 and t_2 both have property $\mathcal{V}$, then so do $t_1 + t_2$ and $t_1 * t_2$.

3. For all types t_1 and t_2, the type $t_1 \to t_2$ has property $\mathcal{V}$.

4. If a type t has property $\mathcal{V}$ then so does **ref** *t.*

5. If types $t_1, \ldots, t_{n_i}$ have property $\mathcal{V}$, then so does $F_i(t_1, \ldots, t_{n_i})$.

Proof. 1) The type **void** has property $\mathcal{V}$ since $\widehat{\mathbf{void}} = \mathsf{void}$ and $\check{\mathbf{void}} = \mathbf{0}$. The type **unit** has property $\mathcal{V}$ since $\widehat{\mathbf{unit}} = \mathsf{eq} \neq \mathsf{void}$ and $\check{\mathbf{unit}} = \mathbf{1} \neq \mathbf{0}$.

2) Suppose t_1 and t_2 both have property $\mathcal{V}$. Then

$$\begin{aligned} \hat{t_1} \mathbin{\hat{+}} \hat{t_2} = \mathsf{void} &\Leftrightarrow \text{both } \hat{t_1} = \mathsf{void} \text{ and } \hat{t_2} = \mathsf{void} \\ &\Leftrightarrow \text{both } \check{t_1} = \mathbf{0} \text{ and } \check{t_2} = \mathbf{0} \\ &\Leftrightarrow \check{t_1} \mathbin{\check{+}} \check{t_2} = \mathbf{0}. \end{aligned}$$

Therefore, $t_1 + t_2$ has property $\mathcal{V}$. Also,

$$\begin{aligned} \hat{t_1} \mathbin{\hat{*}} \hat{t_2} = \mathsf{void} &\Leftrightarrow \text{either } \hat{t_1} = \mathsf{void} \text{ or } \hat{t_2} = \mathsf{void} \\ &\Leftrightarrow \text{either } \check{t_1} = \mathbf{0} \text{ or } \check{t_2} = \mathbf{0} \\ &\Leftrightarrow \check{t_1} \mathbin{\check{*}} \check{t_2} = \mathbf{0}. \end{aligned}$$

Therefore, $t_1 * t_2$ has property $\mathcal{V}$.

3) For all types t_1 and t_2, the domain $\check{t_1} \check{\rightarrow} \check{t_2} = (\check{t_1} \circ\!\!\rightarrow \check{t_2})_\perp$ always has at least two elements, namely $\perp$ and $\lambda x.\perp$, and hence is not $\mathbf{0}$. Moreover, by the definition of $\hat{\rightarrow}$, $\hat{t_1} \hat{\rightarrow} \hat{t_2}$ is never equal to void. Therefore, $t_1 \rightarrow t_2$ always has property $\mathcal{V}$.

4) Suppose that the type t has property $\mathcal{V}$. Then

$$\begin{aligned} \hat{\mathbf{ref}}\ \hat{t} = \mathsf{void} &\Leftrightarrow \hat{t} = \mathsf{void} \\ &\Leftrightarrow \check{t} = \mathbf{0} \\ &\Leftrightarrow \check{\mathbf{ref}}\ \check{t} = \mathbf{0} \end{aligned}$$

Therefore, $\mathbf{ref}\ t$ has property $\mathcal{V}$.

As a result of parts 1 through 4 of the lemma, we have by structural induction that any type operator that is composed solely of basic constructors preserves property $\mathcal{V}$.

5) Let $t_1, \ldots, t_{n_i}$ be a collection of types having property $\mathcal{V}$. By the previous remark, for each j, the type $F_i^j(t_1, \ldots, t_{n_i})$ has property $\mathcal{V}$. Therefore

$$\begin{aligned} \hat{F}_i(\hat{t_1}, \ldots, \hat{t_{n_i}}) = \mathsf{void} &\Leftrightarrow \hat{F_i^j}(\hat{t_1}, \ldots, \hat{t_{n_i}}) = \mathsf{void}, \text{ for all } j \\ &\Leftrightarrow \check{F_i^j}(\check{t_1}, \ldots, \check{t_{n_i}}) = \mathbf{0}, \text{ for all } j \\ &\Leftrightarrow \check{F}_i(\check{t_1}, \ldots, \check{t_{n_i}}) = \bigvee_j \check{F_i^j}(\check{t_1}, \ldots, \check{t_{n_i}}) = \mathbf{0}. \end{aligned}$$

where the least upper bound is taken in T, the type of types. Therefore, $F_i(t_1, \ldots, t_{n_i})$ has property $\mathcal{V}$. ▮

Corollary 4 *For all closed ML type expressions t, we have that $\hat{t} = \mathsf{void}$ iff $\check{t} = \mathbf{0}$.*

Proof. By structural induction and the previous lemma, all closed type expressions in ML have property $\mathcal{V}$. ▮

Our primary interest is not in types which are void, but in those which are equality types. We may now characterize the types having eq as their abstract interpretation as exactly those with a flat standard interpretation.

Definition 5 A closed type expression t *has property $\mathcal{SC}$* (for "sound and complete") provided that $\hat{t} = \mathsf{eq}$ iff $\check{t}$ is flat. ▮

Lemma 6 *1. The types* **void** *and* **unit** *both have property $\mathcal{SC}$.*

2. If the types t_1 and t_2 both have property $\mathcal{SC}$, then so do $t_1 + t_2$ and $t_1 * t_2$.

3. *For all types t_1 and t_2, the type $t_1 \to t_2$ has property $\mathcal{SC}$.*

4. *For all types t, the type* **ref** *t has property $\mathcal{SC}$.*

5. *If types t_1, …, t_{n_i} have property $\mathcal{SC}$, then so does $F_i(t_1, \ldots, t_{n_i})$.*

Proof. 1) Since $\hat{\mathbf{void}} \neq \mathsf{eq}$ and $\check{\mathbf{void}}$ is not flat, **void** has property $\mathcal{SC}$. Since $\hat{\mathbf{unit}} = \mathsf{eq}$ and $\check{\mathbf{unit}}$ is flat, **unit** also has property $\mathcal{SC}$.

2) Suppose that both types t_1 and t_2 have property $\mathcal{SC}$. Then

$\hat{t_1} \mathbin{\hat{+}} \hat{t_2} = \mathsf{eq}$

$\Leftrightarrow$ each of $\hat{t_1}$ and $\hat{t_2}$ is either void or eq, and at least one of them is eq

$\Leftrightarrow$ each of $\check{t_1}$ and $\check{t_2}$ is either void or flat (by Corollary 3 and property SC), and at least one is not void

$\Leftrightarrow$ $\check{t_1} \mathbin{\check{+}} \check{t_2}$ is flat.

Therefore, $t_1 + t_2$ has property $\mathcal{SC}$. Also,

$\hat{t_1} \mathbin{\hat{*}} \hat{t_2} = \mathsf{eq}$

$\Leftrightarrow$ both $\hat{t_1}$ and $\hat{t_2}$ are eq

$\Leftrightarrow$ both $\check{t_1}$ and $\check{t_2}$ are flat

$\Leftrightarrow$ $\check{t_1} \mathbin{\check{*}} \check{t_2}$ is flat,

and hence, $t_1 * t_2$ has property $\mathcal{SC}$.

3) For any types t_1 and t_2, we have that

$$\begin{aligned}
\hat{t_1} \mathbin{\hat{\to}} \hat{t_2} = \mathsf{eq} &\Leftrightarrow \hat{t_1} = \mathsf{void} \text{ or } \hat{t_1} = \mathsf{void} \\
&\Leftrightarrow \check{t_1} = \mathbf{0} \text{ or } \check{t_1} = \mathbf{0} \text{ (by Corollary 3)} \\
&\Leftrightarrow \check{t_1} \mathbin{\check{\to}} \check{t_2} = \mathbf{1}\text{, which is flat.}
\end{aligned}$$

Therefore, $t_1 \to t_2$ has property $\mathcal{SC}$.

4) Given any type t, we have

$$\begin{aligned}
\hat{\mathbf{ref}}\ \hat{t} \neq \mathsf{eq} &\Leftrightarrow \hat{t} = \mathsf{void} \\
&\Leftrightarrow \check{t} = \mathbf{0} \text{ (by Corollary 3)} \\
&\Leftrightarrow \check{\mathbf{ref}}\ \check{t} = \mathbf{0} \\
&\Leftrightarrow \check{\mathbf{ref}}\ \check{t} \text{ is not flat.}
\end{aligned}$$

Therefore, **ref** t has property $\mathcal{SC}$.

As before, by 1 through 4 of this lemma, we know by structural induction that any type operator that is composed solely of basic constructors preserves property $\mathcal{SC}$.

5)Let t_1, …, t_{n_i} be a collection of types having property $\mathcal{SC}$. By the previous remark we have that $F_i^j(t_1, \ldots, t_{n_i})$ has property $\mathcal{SC}$, for each j. Thus

$\hat{F}_i(\hat{t_1}, \ldots, \hat{t_{n_i}}) = \mathsf{eq}$
$\Leftrightarrow$ there exists a k such that $\hat{F_i^j}(\hat{t_1}, \ldots, \hat{t_{n_i}}) = \mathsf{void}$ for all $j < k$, and $\hat{F_i^j}(\hat{t_1}, \ldots, \hat{t_{n_i}}) = \mathsf{eq}$ for all $j \geq k$ (by Lemma 1).
$\Leftrightarrow$ there exists a k such that $\check{F_i^j}(\check{t_1}, \ldots, \check{t_{n_i}}) = \mathbf{0}$ for all $j < k$, and $\check{F_i^j}(\check{t_1}, \ldots, \check{t_{n_i}})$ is (non-void) flat for all $j \geq k$
$\Leftrightarrow$ $\check{F}_i(\check{t_1}, \ldots, \check{t_{n_i}})$ is flat, being the least upper bound of a chain of flat domains.

Therefore, $F_i(t_1, \ldots, t_{n_i})$ has property $\mathcal{SC}$. ▮

Corollary 7 *(Soundness and Completeness) For all closed ML type expressions* t*, we have that* $\check{t}$ *is flat iff* $\hat{t} = \mathsf{eq}$.

Proof. By structural induction and the previous lemma, all closed type expressions in ML have property $\mathcal{SC}$. ▮

4 Equality Functions

Having derived an abstract interpretation for equality types, we now have a theory that tells us when we should expect to find an equality function on a type. However, there is no *a priori* reason to believe that this function is definable in ML or that we can provide a way to uniformly produce a program for computing the function from the structure of the type. However, it is not at all difficult to see that we can do this for the basic operators. For example, to get the equality function on a *product* `s * t`, given equality functions `f` and `g` on `s` and `t` respectively, one just uses the given equality functions to compute the equality on the respective coordinates of the product:

```
fun eqtimes (f,g) ((x,y), (x',y'))
   = f(x,x') andalso g(y,y')
```

The *sum* is similar; the given equality functions should be used in their respective components:

```
fun eqsum (f,g) (inl x, inl y) = f(x,y)
  | eqsum (f,g) (inr x, inr y) = g(x,y)
  | eqsum (f,g) _ = false
```

where the sum type is represented by the following concrete type operator:

```
datatype ('a,'b) sum = inl of 'a | inr of 'b;
```

What should be done for the *arrow* types? These are never equality types except when the domain or codomain of the type is void. In this case, the interpretation of the type has two elements; one of these represents the undefined program at the type and the other represents "delayed divergence". Hence, if two arguments to an equality test for such a type both converge, then they are equal. Noting the call-by-value evaluation of ML programs, we may therefore take the following definition:

```
fun eqarrow (f,g) = fn (x,y) => true
```

Note that the equality function parameters `f` and `g` are not used. That this is the "correct" equality function on arrow types presupposes that it will only be used in the case where the arrow type is flat.

Equality on *reference* types must be computed by a primitive function which determines identity of memory locations.

How is the equality function on *recursive* types computed? Recursively, of course! For example, consider the definition of the operator `list`:

```
 datatype 'a list = Cons of 'a * 'a list | Nil
```

given earlier. To calculate equality on `''a list`, given an equality function `aeq` for `''a`, the constructors which build the list must be recursively unwound:

```
fun eqlist aeq (Cons (x,l), Cons (y, m))
      = (aeq (x,y)) andalso (eqlist aeq (l, m))
  | eqlist aeq (Nil, Nil) = true
  | eqlist _ _ = false
```

Now we give the formal definitions of the equality interpretation of types. Given a type t, we define the equality function $\overline{\overline{t}}$ by induction on the structure of t. First, the equality function on products is given by

$$(f \mathbin{\overline{\overline{*}}} g)(x,y) = \begin{cases} \mathsf{true} & \text{if } x = (x_1,x_2) \text{ and } y = (y_1,y_2) \\ & \quad \text{and } f(x_1,y_1) = g(x_2,y_2) = \mathsf{true} \\ \mathsf{false} & \text{if } x = (x_1,x_2) \text{ and } y = (y_1,y_2) \\ & \quad \text{and } f(x_1,y_1) = \mathsf{false} \text{ or } g = (x_2,y_2)\mathsf{false} \\ \bot & \text{otherwise} \end{cases}$$

and on sums by

$$(f \mathbin{\overline{\overline{+}}} g)(x,y) = \begin{cases} f(x',y') & \text{if } x = (x',0) \text{ and } y = (y',0) \\ g(x',y') & \text{if } x = (x',1) \text{ and } y = (y',1) \\ \mathsf{false} & \text{if } x = (x',i) \text{ and } y = (y',j) \text{ and } i \neq j \\ \bot & \text{if } x = \bot \text{ or } y = \bot \end{cases}$$

A we saw with the definition of `eqarrow` the interpretation for the function spaces is essentially trivial

$$(f \overline{\overline{\to}}\, g)(x, y) = \begin{cases} \mathsf{true} & \text{if } x \neq \bot \text{ and } y \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

The interpretation for unit is similar:

$$(\overline{\overline{\mathbf{unit}}})(x, y) = \begin{cases} \mathsf{true} & \text{if } x \neq \bot \text{ and } y \neq \bot \\ \bot & \text{otherwise.} \end{cases}$$

$\overline{\overline{\mathbf{void}}}$ is the constant function to $\bot$. $\overline{\overline{\mathbf{ref}}}\ (f)$ is the equality function on $N_\bot$.

The equality function for the recursive type operators is the limit of the equality functions associated with their finite approximates:

$$\overline{\overline{F_i}} = \bigvee_i \overline{\overline{F_i^j}}.$$

Theorem 8 *For any type expression t, if $\hat{t} = \mathsf{eq}$, then $\overline{\overline{t}}$ is the equality function on $\check{t}$.*

Proof. The proof is by an induction on the structure of t. The cases involving the primitive operators are straightforward. For the recursive type constructors F_i, note first that $\check{F}_i^j(\check{t}_1, \ldots, \check{t}_{n_i})$ is flat if $\hat{F}_i^j(\hat{t}_1, \ldots, \hat{t}_{n_i}) = \mathsf{eq}$ by Corollary 7. If $\overline{\overline{F_i^j}}$ is the equality function on

$$D_j = \check{F}_i^j(\check{t}_1, \ldots, \check{t}_{n_i})$$

for each j then $\overline{\overline{F_i}}$ is a limit of equality functions on domains D_j. Since these domains are all flat their limit is simply their union $\bigcup_j D_j$ and the limit of the equality functions on the parts is the equality function on the whole. Hence $\overline{\overline{F_i}}$ is the equality function on $\check{F}_i(\check{t}_1, \ldots, \check{t}_{n_i})$. ▮

The reader may now be curious why we have restricted ourselves to types with flat interpretations for those having an equality property. Could there be other types on which equality could be defined? Our domain-theoretic semantics offers some guidance on this point. Let us generalize our earlier definition of an equality function $=_D$ by relaxing the requirement that D is flat. It is clear that there is a program denoting the equality function for the domain $\mathbf{0}$ that interprets `void.` But let us consider the simplest non-flat, non-trivial domain. This domain has three elements; indeed it is isomorphic to $\mathcal{E}$, but to avoid confusing matters, let us name its elements by $\bot < x < y$. This is the interpretation of the type **unit** $\to$ **unit**. Following the standard denotational interpretation of ML terms, the equality function on this type cannot be defined in ML because the equality function on this three point domain is not *monotone.* Indeed, no domain with a three element chain could have an ML-definable equality function for this reason. another standpoint, a computable equality on this would provide a solution to the halting problem! We may conclude that our abstract interpretation describes all and exactly the ML types on which a definable equality exists.

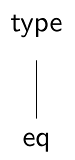

Figure 2: The Equality Kinds Lattice $\mathcal{O}$.

5 Calculating Equality Kinds

There is a problem with the abstract interpretation of types given in the previous sections. We cannot say of a type constructor that the type it yields will admit equality if and only if certain of its arguments admit equality. The difficulty is with the combination of the function space type constructor and void types. The type **unit** $\rightarrow$ **void** admits equality and the type **void** $\rightarrow$ **unit** admits equality, but **unit** $\rightarrow$ **unit** does not admit equality. There is a lack of independence between the two arguments to $\rightarrow$ when determining whether their resultant type admits equality. This example also shows why it was necessary for us to introduce void as a separate element of the equality properties lattice, $\mathcal{E}$. If we were to interpret $\hat{\textbf{void}}$ as eq, then what would be the correct value of $\mathsf{eq} \mathrel{\hat{\rightarrow}} \mathsf{eq}$? If we choose it to be eq, then we lose soundness, and if we choose it to be type, then we lose completeness. It is too naive to try to solve these problems by saying that "there are no elements of type void, so there is no reason to have it." Firstly, **void** may be a subexpression of a nonvoid type, such as **void** $\rightarrow$ **unit**. More importantly, we can only understand the recursive types by successive approximations, starting with the void type. Still, there is a useful, sensible theory that we can cull out based on the idea of banning void.

To begin with let us focus attention on the sublattice $\mathcal{O}$ of $\mathcal{E}$ consisting of the points $\{\mathsf{eq}, \mathsf{type}\}$ as picture in Figure 2. In this section, we will develop another abstract interpretation of ML types, using $\mathcal{O}$ instead of $\mathcal{E}$. This new interpretation has a succinct representation, which is readily computed from the types and type constructors. Moreover, if our recursive type constructors satisfy a reasonable void-avoiding property, when we restrict to the subalgebra of types not involving **void**, the two abstract interpretations turn out to be the same. Therefore, on this subalgebra, this new abstract interpretation will also turn out to be sound and complete.

As before, the definition of the abstract interpretation over $\mathcal{O}$ is given by first defining it on the constructors, and then extending it to closed type expressions by structural induction. The definition for the constructors is as follows:

- $\tilde{\textbf{void}} = \mathsf{eq}$;
- $\tilde{\textbf{unit}} = \mathsf{eq}$;
- $v_1 \mathbin{\tilde{+}} v_2 = \max\{v_1, v_2\}$;

- $v_1 \mathbin{\tilde{*}} v_2 = \max\{v_1, v_2\}$;
- for all $v_1, v_2 \in \mathcal{O}$ we have $v_1 \mathbin{\tilde{\to}} v_2 = \mathsf{type}$;
- for all $v \in \mathcal{O}$ we have $\tilde{\mathbf{ref}}\ v = \mathsf{eq}$; and
- $\tilde{F}_i(v_1, \ldots, v_{n_i}) = \max\limits_j \{\tilde{F}_i^j(v_1, \ldots, v_n)\}$.

Lemma 9 *For all i, j, we have $\tilde{F}_i^j \leq \tilde{F}_i^{j+1}$. Moreover, there exists a k such that for all i, $\tilde{F}_i^k = \tilde{F}_i^{k+1}$, and hence, $\tilde{F}_i = \tilde{F}_i^k$.*

Proof. The proof is the same as for Lemma 1. ▮

Lemma 10 *Given any n-ary type operator G over our type algebra, either for all $(v_1, \ldots, v_n) \in \mathcal{O}^n$ we have $\tilde{G}(v_1, \ldots, v_n) = \mathsf{type}$, or there exists a point $(z_1, \ldots, z_n)$ such that $\tilde{G}(v_1, \ldots, v_n) = \mathsf{eq}$ iff $v_i \leq z_i$, $i = 1, \ldots, n$.*

Proof. The proof is by structural induction on the body of G. The result follows immediately for the basic constructors. Therefore, by structural induction, we have the result for type operators composed solely of the basic operators. In particular, we have the result for the operators F_i^j. But then, the result for the recursive constructors follows immediately from the previous lemma, since for some k, $\tilde{F}_i = \tilde{F}_i^k$. ▮

Definition 11 Given any n-ary type operator G over our type algebra, if for all $(v_1, \ldots, v_n) \in \mathcal{O}^n$ we have $\tilde{G}(v_1, \ldots, v_n) = \mathsf{type}$, then the *equality kind* of G is $\not\bigcirc$, and we say that G does *not* admit equality. Otherwise, the *equality kind* of G is the point $(z_1, \ldots, z_n)$ such that $\tilde{G}(v_1, \ldots, v_n) = \mathsf{eq}$ iff $v_i \leq z_i$ for all $i = 1, \ldots, n$. ▮

In particular, if t is a closed (nullary) type expression, then either it does not admit equality, and therefore has equality kind $\not\bigcirc$, or it does admit equality and has equality kind $()$.

The equality kinds for the basic constructors is as follows:

- The equality kind of **void** is $()$.
- The equality kind of **unit** is $()$.
- The equality kind of both $+$ and $*$ is $(\mathsf{eq}, \mathsf{eq})$.
- The equality kind of $\to$ is $\not\bigcirc$.
- The equality kind of **ref** is type.

Notice that for n-ary type operators F and G that admit equality, we have that $\tilde{F} \leq \tilde{G}$ iff $(w_1, \ldots, w_n) \geq (z_1, \ldots, z_n)$ where $(w_1, \ldots, w_n)$ is the equality kind of F and $(z_1, \ldots, z_n)$ is the equality kind of G.

With these definitions it is possible to describe how to calculate the equality kind of a recursive type constructor. One simply carries out the iterations of the fixed point. By Lemma 9, this will terminate. The number of iterations required is bounded by the number of parameters in the type recursion, so the algorithm is quite efficient. This calculation will miss some types for which equality is definable, but only in cases that are uninteresting in practice. To state a crisp theorem, we must formulate a notion of "void avoidance". To do this, we now restrict our attention to that subalgebra of type expressions over basic constructors **unit**, $+$, $*$, $\rightarrow$, and **ref**, and the recursive operators, provided that the associated recursive equations are over just these basic constructors.

Definition 12 A set of recursive type constructors F_1, …, F_m is *void avoiding* provided that the second-order recursive operators $\mathcal{F}_i$ giving the recursive equations associated with them involve only the basic constructors **unit**, $+$, $*$, $\rightarrow$, and **ref**, and whenever the constructor F_i is applied to argument types t_1 …, t_{n_i}, each of which has a non-void domain-theoretic interpretation, the resulting type has a domain-theoretic interpretation which is non-void, *i.e.* $\check{F_i}(\check{t_1}, \ldots, \check{t_{n_i}}) \neq \mathbf{0}$. ∎

Lemma 13 *Suppose that the set of recursive type operators F_1, …, F_m are void-avoiding. Then, for every closed type expression t not containing* **void** *as a subexpression, we have that $\hat{t} = \tilde{t}$.*

Proof. First notice that any closed type expression in this subalgebra will have an interpretation under $\frown$ of either eq or type. Therefore, we may view the abstract interpretation under $\frown$ of any n-ary type operator as a function from $\mathcal{O}^n$ into $\mathcal{O}$. To prove the lemma, it suffices to show that given any n-ary type operator G, the function $\hat{G}$, when restricted to $\mathcal{O}^n$ is the same as the function $\tilde{G}$. By structural induction, in fact it suffices to show this for the basic constructors (excluding **void**) and for the recursive constructors. The result follows immediately from the definitions of $\frown$ and $\sim$ for the basic constructors **unit**, $+$, $*$, $\rightarrow$, and **ref**.

Since the second-order operators $\mathcal{F}_i$ are composed solely of basic constructors, by structural induction we have that $\hat{\mathcal{F}_i} = \tilde{\mathcal{F}_i}$. Also, since $\hat{G} \leq \tilde{G}$ for every basic constructor G, we have that for each of the finite approximates $\hat{F}_i^j \leq \tilde{F}_i^j$. By Lemma 1, there exists a k such that

$$\hat{F}_i^k = \hat{\mathcal{F}}_i(\hat{F}_1^k, \ldots, \hat{F}_m^k) = \tilde{\mathcal{F}}_i(\hat{F}_1^k, \ldots, \hat{F}_m^k)$$

and $\hat{F}_i = \hat{F}_i^k$. Therefore, the operators $\hat{F}_i^k$ form a fixed point of the system $\tilde{\mathcal{F}}_1, \ldots, \tilde{\mathcal{F}}_m$. However, by their construction the functions $\tilde{F}_i$, $i = 1, \ldots, m$ form the least fix point of the operators $\tilde{\mathcal{F}}_i$. Since $\hat{F}_i = \hat{F}_i^k \leq \tilde{F}_i^k \leq \tilde{F}_i$, we must have that $\hat{F}_i = \tilde{F}_i$. ∎

Corollary 14 *(Soundness and Completeness) Suppose that the set of recursive type operators F_1, …, F_m are void-avoiding. Let G be a type operator defined in terms of the basic constructors* **unit**, *$+$, $*$, $\rightarrow$ and* **ref** *and the recursive constructors. Then G admits equality iff the equality kind of G is not $\bigcirc\!\!\!\!\!\neq$. Moreover, if G admits equality with equality kind $(z_1, \ldots, z_n)$, then for any types $t_1, \ldots t_n$ which do not contain* **void** *as a subexpression, $\check{G}(\check{t}_1, \ldots, \check{t}_n)$ is flat iff $\tilde{t}_i \leq z_i$ for all $i = 1, \ldots, n$.*

6 Conclusions and Future Work

We have provided a sound and complete semantic analysis of the equality property for ML types and demonstrated an efficient algorithm for carrying out the inference of equality properties for void-avoiding systems of user-defined types. Our results are based on theorems that relate the standard denotational semantics of type constructors to an abstract interpretation that describes the equality kind of the operator.

Our algorithm expands the number of types that will be judged to admit equality. In existing compilers, the new types admitted under our scheme can be handled in exactly the same way as the ones currently accepted, so no new approach to the implementation is implied. The specification of equality in Section 4 can be viewed as a specification of equality functions on ML types rather than a prescription for how the equality functions must be implemented. (Although our specification is, arguably, the most natural approach to the implementation.)

The motivation for this work was to provide a more accurate version of the notion of equality types in Standard ML. Introducing the refined notion of equality kinds into Standard ML itself raises the question of how they would be integrated with the module system.

The easiest problem is specifying the equality kinds of type constructors in signatures. The current language definition provides a simple type specification

```
type ('a,'b) F
```

that does not constrain the equality kind of `F` at all, and the equality type specification

```
eqtype ('a,'b) F
```

that specifies that `F` has equality kind with succinct representation $(\mathtt{eq}, \mathtt{eq})$. To specify that `F` has the equality kind $(\mathtt{eq}, \mathtt{type})$ we might use the following notation:

```
type F: (eq,ty) => eq
```

A more complex interaction with modules involves the effect of sharing constraints in signatures. If two type constructor specifications are identified as a consequence of sharing constraints, it seems clear that they should have the same equality kind. This brings up the issue of compatibility of equality kind specifications and the problem of determining the resultant kind when two specifications share.

A third problem is how equality kinds are affected by functor applications. The definition of a type constructor in the body of a functor may depend on type constructors in the functor parameter with unspecified equality kinds, making it impossible to completely infer the equality kind of the defined constructor. When the functor is applied, the actual parameter supplies additional information that should be taken into account to recalculate the equality kind of the defined type constructor. This suggests partial and incremental calculation of equality kind information may be required.

These problems of integrating equality kinds with the module system are the subject of continuing research, with the experience with the current Standard ML treatment of equality kinds providing a starting point.

It is our belief that there are broader issues relating to equality types that involve other properties and operators which are defined uniformly from the structure of types. One might refer to this as *structural polymorphism.* It has come up in other contexts such as the study of subtyping and coercions between recursive types. For instance, [BGS89] describes how coercions between such types are generated from the type definitions in a manner very similar to the one used in Section 4 of this paper for the equality relations. Whether there is any general theory that connects these apparently similar phenomenon remains to be seen.

References

[BGS89] V. Breazu-Tannen, C. Gunter, and A. Scedrov. *Denotational Semantics for Subtyping between Recursive Types.* Research Report MS-CIS-89-63/Logic & Computation 12, Department of Computer and Information Science, University of Pennsylvania, 1989.

[GS90] C. A. Gunter and D. S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 633–674, North Holland, 1990.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Sco76] D. S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

[Sco82a] D. S. Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *International Colloquium on Automata, Languages and Programs*, pages 577–613, *Lecture Notes in Computer Science vol. 140,* Springer, 1982.

[Sco82b] D. S. Scott. Lectures on a mathematical theory of computation. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 145–292, *NATO Advanced Study Institutes Series,* D. Reidel, 1982.

[SP82] M. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.