Existential Types for Imperative Languages*

Dan Grossman
danieljg@cs.cornell.edu

Cornell University

Abstract. We integrate existential types into a strongly typed C-like language. In particular, we show how a bad combination of existential types, mutation, and aliasing can cause a subtle violation of type safety. We explore two independent ways to strengthen the type system to restore safety. One restricts the mutation of existential packages. The other restricts the types of aliases of existential packages. We use our framework to explain why other languages with existential types are safe.

1 Introduction

Strongly typed programming languages prevent certain programming errors and provide a foundation on which the user can enforce strong abstractions. High-level languages usually have primitive support for data hiding (e.g., closures, objects), which mitigate the burden of strong typing. At lower levels of abstraction, such as in C, exposed data representation and a rich set of primitive operations make it difficult to provide strong typing without unduly restricting the set of legal programs. A powerful technique is to provide a rich type system that the programmer can use to express invariants that ensure a program's safety.

In particular, *existential types* (often written $\exists\alpha.\tau$ where τ is a type) are a well-known device for permitting consistent manipulation of data containing values of unknown types. Indeed, they have become the standard tool for modeling the data-hiding constructs of high-level languages. Mitchell and Plotkin's seminal work [11] explains how constructs for abstract types, such as the `rep` types in CLU clusters [8] and the `abstype` declarations in Standard ML [9], are really existential types. For example, an abstraction for the natural numbers could have the type $\exists\alpha\{\texttt{zero:}\alpha\texttt{; succ:}\alpha \to \alpha\}$. As desired, if we had two such abstractions, we could not apply one's `succ` function to the other's `zero` value.

Existential types also serve an essential role in many encodings of objects [1] and closures [10]. For example, we can represent a closure as a record of values for the original code's free variables (its environment) and a closed code pointer taking the environment as an extra parameter. By abstracting the environment's type with an existential type, source functions with the same type but different environments continue to have the same type after this encoding.

* This material is based on work supported in part by AFOSR under grant F49620-00-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publications are those of the author and do not reflect the view of this agency.

More recently, existential types have also proven useful in safe *low-level* languages [12, 3]. For example, many low-level interfaces let a client register a call-back with a server along with data to pass to the call-back when it is invoked. If the server specifies the data's type, then the interface is too restrictive. This idiom bears an uncanny resemblance to the closure encoding: A record holding the call-back and the data should have an existential type. Essentially, low-level languages do not provide data-hiding constructs directly, so it is sensible for the type system to be rich enough for programs to create them. Existential quantification has also been used to express connections beyond simple types. For example, Xanadu [20] lets programmers express that an integer holds the length of a particular array. An existential type lets us pack an array with its length.

In C, we must resort to `void*` and unchecked casts when existential types would be appropriate. For example, using the aforementioned encoding, a function closure that consumes an `int` and returns `void` could have the type `struct T {void (*f)(int,void*); void* env;};`. If `x` had this type, we could write `x.f(37,x.env)`, but nothing enforces that `x.env` has the type that `x.f` expects, nor can we prevent calling `x.f` with any other pointer. With existential types, we can enforce the intended usage by declaring `struct T` $\exists\alpha$`.{void (*f)(int,`α`); `α `env;};`. For this reason, Cyclone [2, 6], a safe C-like language developed by Trevor Jim, Greg Morrisett, the author, and others, allows `struct` declarations to have existential type variables.

However, it does not appear that the interaction of existential types with features like mutation and C's address-of (&) operator has been carefully studied. Orthogonality suggests that existential types in a C-like language should permit mutation and acquiring the address of fields, just as ordinary `struct` types do. Moreover, such abilities are genuinely useful. For example, a server accepting call-backs can use mutation to reuse the same memory for different call-backs that expect data of different types. Using & to introduce aliasing is also useful. As a small example, given a value `v` of type `struct T` $\exists\alpha$`.{`α `x; `α `y;};` and a polymorphic function $\forall\beta$. `void swap(`β`*, `β`*)` for swapping the contents of two locations, we would like to permit a call like `swap(&v.x, &v.y)`.

Unfortunately, a subtle interaction among all these features can violate type safety. Somewhat embarrassingly, Cyclone was unsafe for several months before the problem was discovered. In order to expose the problem's essential source and provide guidelines for using existential types in low-level languages, this paper explains the unsoundness, describes two independent solutions, proves the solutions are correct, and explores why this problem did not arise previously.

In the next section, we present a full example exploiting the unsoundness that assignment, aliasing, and existential types can produce. We use this example to explain how we restore soundness. Section 3 presents a small formal language suitable for arguing rigorously that we have, in fact, restored soundness. Section 4 describes the soundness proof for the formal language; the excruciating details appear in a companion technical report [4]. Section 5 discusses related work. In particular, it uses the insights of the preceding sections to explain why other languages with existential types are safe.

2 Violating Type Safety

In this section, we present a violation of type safety discovered in Cyclone [2, 6] (a safe C-like language) and how we fixed the problem. We describe only the Cyclone features necessary for our present purposes, and we take the liberty of using prettier syntax (e.g., Greek letters) than the actual language.

A `struct` declaration may declare *existentially-bound type variables* and use them in the types of fields. Repeating an earlier example, a value of the type

$$\texttt{struct T } \exists\alpha.\texttt{\{void (*f)(int, } \alpha\texttt{); } \alpha \texttt{ env;\};}$$

contains a function pointer in the `f` field that can be applied to an `int` and the `env` field *of the same value*.

Different values of type `struct T` can instantiate α with different types. Here is a program exploiting this feature. (The form `T(e1,e2)` is just a convenient way to create a `struct T` object with fields initialized to `e1` and `e2`.)

```
void ignore(int x, int y) {}
void assign(int x, int *y) { *y = x; }
void f(int* ptr) {
  struct T p1 = T(ignore, 0xabcd);
  struct T p2 = T(assign, ptr);
  /* use p1 and p2 ... */
}
```

The type-checker infers that in `T(ignore,0xabcd)`, for example, α is `int`. We call `int` the *witness type* for the *existential package* `T(ignore,0xabcd)`. The type-checker would reject `T(assign,0xabcd)` because there is no appropriate witness type. Witness types are not present at run time.

Because `p1` and `p2` have the same type, we could assign one to the other with `p2=p1`. As in C, this assignment *copies* the fields of `p1` into the fields of `p2`. Note that the assignment *changes* `p2`'s witness type.

We cannot access fields of existential packages with the "." or "→" operators.[1] Instead, Cyclone provides *pattern-matching* to bind variables to parts of aggregate objects. For existential packages, the pattern also *opens* (sometimes called "unpacking") the package by giving an abstract name to the witness type. For example, the function `f` could continue with

```
let T(g,arg)<β> = p2  in  g(37,arg);
```

The pattern binds `g` to (a copy of) `p2.f` and `arg` to (a copy of) `p2.env`. It also makes β a type. The scope of `g`, `arg`, and β is the statement after `in`. The types of `g` and `arg` are `void (*f)(int,`β`)` and β, respectively, so the function call is well-typed.

It is well-known that the typing rule for opening an existential package must forbid the introduced type variable (β) from occurring in the type assigned to

[1] `struct` declarations without existential type variables permit these operators.

the term in which β is in scope. In our case, this term is a statement, which has no type (or a unit type if you prefer), so this condition is trivially satisfied. Our unsoundness results from a different problem.

Another pattern form, which we call a *reference pattern*, is `*id`; it binds `id` to *the address of* part of a value. (So `*id` has the same type as that part of the value.) We need this feature for the `swap` example from the previous section. We can modify our previous example to use this feature gratuitously:

```
let T(g,*arg)<β> = p2  in  g(37,*arg);
```

Here `arg` is an alias for `&p2.env`, but `arg` has the *opened type*, in this case $\beta*$.

At this point, we have seen how to create existential packages, use assignment to modify memory that has an existential type, and use reference patterns to get aliases of existential-package fields. It appears that we have a smooth integration of several features that are natural for a language at the C level of abstraction. Unfortunately, these features conspire to violate type safety:

```
void f(int* ptr) {
  struct T p1 = T(ignore, 0xabcd);
  struct T p2 = T(assign, ptr);
  let T(g,*arg)<β> = p2  in  { p2 = p1; g(37,*arg); }
}
```

The call `g(37,*arg)` executes `assign` with `37` and `0xabcd`—we are passing an `int` where we expect an `int*`, allowing us to write to an arbitrary address.

What went wrong in the type system? We used β to express an equality between one of `g`'s parameter types and the type of value at which `arg` points. But after the assignment, which changes `p2`'s witness type, this equality is false.

We have developed two solutions. The first solution forbids using reference patterns to match against fields of existential packages. Other uses of reference patterns are sound because assignment to a package mutates only the fields of the package. We call this solution, "no aliases at the opened type." The second solution forbids assigning to an existential package (or an aggregate value that has an existential package as a field). We call this solution, "no witness changes."

These solutions are *independent*: Either suffices and we could use different solutions for different existential packages. That is, for each existential-type declaration we could let the programmer decide which restriction the compiler enforces. Our current implementation supports only "no aliases at the opened type" because we believe it is more useful, but both solutions are easy to enforce.

To emphasize the exact source of the problem, we mention some aspects that are not problematic. First, pointers to witness types are not a problem. For example, given `struct T2` $\exists\alpha$`.{void f(int,` α`); ` α`* env;};` and the pattern `T2(g,arg)<`β`>`, an intervening assignment changes a package's witness type but does *not* change the type of the value at which `arg` points. Second, assignment to a *pointer to* an existential package is not a problem because it changes which package a pointer refers to, but does *not* change any package's witness type.

Witness changes are more difficult with multithreading: A similar unsoundness results if the witness can change in-between the binding of `g` and `arg`. We

would need some mechanism for excluding a mutation while binding a package's fields. If we restricted (shared, witness-changeable) packages to a single field (a pointer), atomic reads and writes of a single word would suffice.

3 Sound Language

To investigate the essential source of the unsoundness described in Section 2, we present a small formal language with the same potential problem. Instead of type definitions, we use "anonymous" product types (pairs) and existential types. Instead of pattern-matching, we use `open` statements for destructing existential packages. We omit many features that have no relevance to our investigation, including loops and function calls. Such features can be added in typical fashion.

3.1 Syntax

Figure 1 presents the language's syntax. Types include a base type (int), products ($\tau_1 \times \tau_2$), pointers ($\tau*$), existentials ($\exists^\phi \alpha.\tau$), and type variables (α). Because aliasing is relevant, all uses of pointers are explicitly noted. In particular, a value of a product type is a record, not a pointer to a record.[2] To distinguish our two solutions to restoring soundness, we annotate existential types with δ (the witness type can change) or & (aliases at the opened type are allowed).

$$\ell \in \text{Lab} \qquad c \in \text{Int} \qquad x \in \text{Var} \qquad \alpha \in \text{Tyvar} \qquad H : \text{Lab} \rightharpoonup \text{Value}$$

$$\begin{array}{rcl}
\text{Type } \tau & ::= & \mathsf{int} \mid \alpha \mid \tau * \mid \tau_1 \times \tau_2 \mid \exists^\phi \alpha.\tau \\
\text{Exp } e & ::= & c \mid x \mid \ell p \mid (e_1, e_2) \mid e.i \mid \&e \mid *e \mid \mathbf{pack}\ \tau', e\ \mathbf{as}\ \exists^\phi \alpha.\tau \\
\text{Stmt } s & ::= & \mathbf{skip} \mid e_1 := e_2 \mid s_1; s_2 \mid \mathbf{let}\ x : \tau = e\ \mathbf{in}\ s \\
& \mid & \mathbf{open}\ e\ \mathbf{as}\ \alpha, x\ \mathbf{in}\ s \mid \mathbf{open}\ e\ \mathbf{as}\ \alpha, *x\ \mathbf{in}\ s \\
\text{Path } p & ::= & \cdot \mid ip \mid \mathtt{u}p \\
\text{Field } i & ::= & 0 \mid 1 \\
\text{Style } \phi & ::= & \delta \mid \&
\end{array}$$

$$\text{Value } v ::= c \mid \&\ell p \mid \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^\phi \alpha.\tau \mid (v_1, v_2)$$

Fig. 1. Syntax

Expressions include variables (x), constants (c), pairs ((e_1, e_2)), field accesses ($e.i$), pointer creations ($\&e$), pointer dereferences ($*e$), and existential packages ($\mathbf{pack}\ \tau', e\ \mathbf{as}\ \exists^\phi \alpha.\tau$). In the last form, τ' is the witness type; its explicit mention is just a technical convenience. We distinguish locations ("lvalues") from values ("rvalues"). Locations have the form ℓp where ℓ is a label (an address) for a heap

[2] Hence we could allow casts between $\tau_1 \times (\tau_2 \times \tau_3)$ and $(\tau_1 \times \tau_2) \times \tau_3$, but we have no reason to add this feature.

record and p is a path that identifies a subrecord. Locations do not appear in source programs, but we use them in the dynamic semantics, as described below.

Statements include doing nothing (**skip**), assignment—altering the contents of a heap record ($e_1 := e_2$), and local bindings—extending the heap (**let** $x : \tau = e$ **in** s). Because memory management is not our present concern, the dynamic semantics never contracts the heap. There are two forms for destructing existential packages. The form **open** e **as** α, x **in** s binds x to a *copy* of the contents of the evaluation of e, whereas **open** e **as** $\alpha, *x$ **in** s binds x to a *pointer* to the contents of the evaluation of e. The latter form corresponds to the previous section's reference patterns, but for simplicity it produces a pointer to the entire contents, not a particular field.

3.2 Dynamic Semantics

A heap (H) maps labels (ℓ) to values. We write $H[\ell \mapsto v]$ for the heap that is like H except that ℓ maps to v, and we write $\cdot$ for the empty heap. Because values may be pairs or packages, we use paths (p) to specify parts of values. A path is just a sequence of 0, 1, and $\mathtt{u}$ (explained below) where $\cdot$ denotes the empty sequence. We write $p_1 p_2$ for the sequence that is p_1 followed by p_2. We blur the distinction between sequences and sequence elements as convenient. So $0p$ means the path beginning with 0 and continuing with p and $p0$ means the path ending with 0 after p.

The get relation defines the use of paths to destruct values. As examples, $\mathrm{get}((v_0, v_1), 1, v_1)$ and $\mathrm{get}(\mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\phi}\alpha.\tau,\ \mathtt{u},\ v)$. That is, we use $\mathtt{u}$ to get a package's contents. The set relation defines the use of paths to update parts of values: $\mathrm{set}(v_1, p, v_2, v_3)$ means updating the part of v_1 corresponding to p with v_2 produces v_3. For example, $\mathrm{set}((v_1, ((v_2, v_3), v_4)), 10, (v_5, v_6), (v_1, ((v_5, v_6), v_4)))$. Figure 2 defines both relations.

Unlike C, expression evaluation in our core language has no side effects, so we have chosen a large-step semantics. Given a heap, we use the $\Downarrow_{\mathrm{L}}$ relation to evaluate expressions to locations and the $\Downarrow_{\mathrm{R}}$ relation to evaluate expressions to values. The two relations are interdependent (see the $\Downarrow_{\mathrm{L}}$ rule for $*e$ and the $\Downarrow_{\mathrm{R}}$ rule for $\&e$). For many expressions there are no H, ℓ, and p such that $H \vdash e \Downarrow_{\mathrm{L}} \ell p$ (for example, $e = (e_1, e_2)$). Only a few rules merit further discussion: The $\Downarrow_{\mathrm{L}}$ rule for projection puts the field number on the right of a path. The $\Downarrow_{\mathrm{R}}$ rules for ℓp and $*e$ use the heap H and the get relation. Figure 3 defines $\Downarrow_{\mathrm{L}}$ and $\Downarrow_{\mathrm{R}}$.

Statements operate on heaps; we define a small-step semantics in which $(H, s) \to (H', s')$ means, "under heap H, statement s produces H' and becomes s'." The meaning of a program s is H where $(\cdot, s) \to^* (H, \mathbf{skip})$ (where $\to^*$ is the reflexive transitive closure of $\to$). We write $s\{\ell p/x\}$ for the substitution of ℓp for x in s and $s\{\tau/\alpha\}$ for the capture-avoiding substitution of τ for α in s. We omit the straightforward but tedious definitions of substitution.

We now describe the interesting rules for evaluating statements. (All rules are in Figure 4.) The interesting part for assignment is in the set judgment. For **let**, we map a fresh label ℓ in the heap to (a copy of) the value and substitute that location (that is, the label and path $\cdot$) for the binding variable in the body

$$\frac{}{\mathrm{get}(v, \cdot, v)} \qquad \frac{\mathrm{get}(v_0, p, v)}{\mathrm{get}((v_0, v_1), 0p, v)} \qquad \frac{\mathrm{get}(v_1, p, v)}{\mathrm{get}((v_0, v_1), 1p, v)}$$

$$\frac{\mathrm{get}(v_1, p, v)}{\mathrm{get}(\mathbf{pack}\ \tau', v_1\ \mathbf{as}\ \exists^{\&}\alpha.\tau, \mathtt{u}p, v)}$$

$$\frac{}{\mathrm{set}(v_{old}, \cdot, v, v)} \qquad \frac{\mathrm{set}(v_0, p, v, v')}{\mathrm{set}((v_0, v_1), 0p, v, (v', v_1))} \qquad \frac{\mathrm{set}(v_1, p, v, v')}{\mathrm{set}((v_0, v_1), 1p, v, (v_0, v'))}$$

$$\frac{\mathrm{set}(v_1, p, v, v')}{\mathrm{set}(\mathbf{pack}\ \tau', v_1\ \mathbf{as}\ \exists^{\phi}\alpha.\tau,\ \mathtt{u}p,\ v,\ \mathbf{pack}\ \tau', v'\ \mathbf{as}\ \exists^{\phi}\alpha.\tau)}$$

Fig. 2. Dynamic Semantics: Heap Objects

$$\frac{}{H \vdash \ell p \Downarrow_{\mathrm{L}} \ell p} \qquad \frac{H \vdash e \Downarrow_{\mathrm{L}} \ell p}{H \vdash e.i \Downarrow_{\mathrm{L}} \ell p i} \qquad \frac{H \vdash e \Downarrow_{\mathrm{R}} \&\ell p}{H \vdash *e \Downarrow_{\mathrm{L}} \ell p}$$

$$\frac{}{H \vdash c \Downarrow_{\mathrm{R}} c} \qquad \frac{\mathrm{get}(H(\ell), p, v)}{H \vdash \ell p \Downarrow_{\mathrm{R}} v} \qquad \frac{H \vdash e \Downarrow_{\mathrm{R}} (v_0, v_1)}{H \vdash e.0 \Downarrow_{\mathrm{R}} v_0} \qquad \frac{H \vdash e \Downarrow_{\mathrm{R}} (v_0, v_1)}{H \vdash e.1 \Downarrow_{\mathrm{R}} v_1}$$

$$\frac{H \vdash e_0 \Downarrow_{\mathrm{R}} v_0 \quad H \vdash e_1 \Downarrow_{\mathrm{R}} v_1}{H \vdash (e_0, e_1) \Downarrow_{\mathrm{R}} (v_0, v_1)} \qquad \frac{H \vdash e \Downarrow_{\mathrm{L}} \ell p}{H \vdash \&e \Downarrow_{\mathrm{R}} \&\ell p} \qquad \frac{H \vdash e \Downarrow_{\mathrm{R}} \&\ell p \quad \mathrm{get}(H(\ell), p, v)}{H \vdash *e \Downarrow_{\mathrm{R}} v}$$

$$\frac{H \vdash e \Downarrow_{\mathrm{R}} v}{H \vdash \mathbf{pack}\ \tau', e\ \mathbf{as}\ \exists^{\phi}\alpha.\tau \Downarrow_{\mathrm{R}} \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\phi}\alpha.\tau}$$

Fig. 3. Dynamic Semantics: Expressions

$$\frac{H \vdash e_1 \Downarrow_{\mathrm{L}} \ell p \quad H \vdash e_2 \Downarrow_{\mathrm{R}} v \quad \mathrm{set}(H(\ell), p, v, v')}{(H, e_1 := e_2) \to (H[\ell \mapsto v'], \mathbf{skip})}$$

$$\frac{}{(H, \mathbf{skip}; s) \to (H, s)} \qquad \frac{(H, s_1) \to (H', s_1')}{(H, s_1; s_2) \to (H', s_1'; s_2)}$$

$$\frac{H \vdash e \Downarrow_{\mathrm{R}} v \quad \ell \notin \mathrm{dom}(H)}{(H, \mathbf{let}\ x : \tau = e\ \mathbf{in}\ s) \to (H[\ell \mapsto v], s\{\ell\cdot/x\})}$$

$$\frac{H \vdash e \Downarrow_{\mathrm{R}} \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\phi}\alpha.\tau \quad \ell \notin \mathrm{dom}(H)}{(H, \mathbf{open}\ e\ \mathbf{as}\ \alpha, x\ \mathbf{in}\ s) \to (H[\ell \mapsto v], s\{\tau'/\alpha\}\{\ell\cdot/x\})}$$

$$\frac{H \vdash e \Downarrow_{\mathrm{L}} \ell' p \quad \mathrm{get}(H(\ell'), p, \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\phi}\alpha.\tau) \quad \ell \notin \mathrm{dom}(H)}{(H, \mathbf{open}\ e\ \mathbf{as}\ \alpha, *x\ \mathbf{in}\ s) \to (H[\ell \mapsto \&\ell' p\mathtt{u}], s\{\tau'/\alpha\}\{\ell\cdot/x\})}$$

Fig. 4. Dynamic Semantics: Statements

of the statement. The rule for **open** e **as** α, x **in** s is similar; we also substitute τ' for α, where τ' is the witness type in the package to which e evaluates. The rule for **open** e **as** $\alpha, *x$ **in** s is like the rule for **open** e **as** α, x **in** s except that we use $\Downarrow_{\mathrm{L}}$ to evaluate e to a location $\ell' p$ and we map ℓ to $\&\ell' p\mathtt{u}$.

As an example, here is a variation of the previous unsoundness example. Instead of using function pointers, we use assignment, but the idea is the same. For now, we do not specify the style of the existential types.

(1) **let** *zero* : $\mathsf{int} = 0$ **in**
(2) **let** *pzero* : $\mathsf{int}* = \&zero$ **in**
(3) **let** *pkg* : $\exists^{\phi}\alpha.\alpha* \times \alpha =$ **pack** $\mathsf{int}*, (\&pzero, pzero)$ **as** $\exists^{\phi}\alpha.\alpha* \times \alpha$ **in**
(4) **open** *pkg* **as** $\beta, *pr$ **in**
(5) **let** *fst* : $\beta* = (*pr).0$ **in**
(6) *pkg* := **pack** $\mathsf{int}, (pzero, zero)$ **as** $\exists^{\phi}\alpha.\alpha* \times \alpha$;
(7) $*fst := (*pr).1$;
(8) $*pzero := zero$

In describing the example, we assume that when binding a variable x, we choose ℓ_x as the fresh location. Hence line (3) substitutes $\ell_{pkg}\cdot$ for *pkg* and line (4) substitutes $\mathsf{int}*$ for β and $\ell_{pr}\cdot$ for pr. Furthermore, after line (4), ℓ_{pkg} contains **pack** $\mathsf{int}*, (\&\ell_{pzero}\cdot, \&\ell_{zero}\cdot)$ **as** $\exists^{\phi}\alpha.\alpha* \times \alpha$ and ℓ_{pr} contains $\&\ell_{pkg}\mathtt{u}$. After line (6), ℓ_{fst} contains $\&\ell_{pzero}\cdot$ and ℓ_{pkg} contains **pack** $\mathsf{int}, (pzero, 0)$ **as** $\exists^{\phi}\alpha.\alpha* \times \alpha$. Hence line (7) assigns 0 to ℓ_{pzero}, which causes line (8) to be stuck because there is no ℓp to which $\Downarrow_{\mathrm{L}}$ can evaluate $*0$.

To complete the example, we need to choose δ or & for each ϕ. Fortunately, as the next section explains, no choice produces a well-typed program.

Note that the type information associated with packages and paths is just to keep type-checking syntax-directed. We can define an erasure function over heaps that replaces **pack** τ', v **as** $\exists^{\phi}\alpha.\tau$ with v and removes $\mathtt{u}$ from paths. Although we have not formally done so, it should be straightforward to prove that erasure and evaluation commute. That is, we do not need type information at run time.

3.3 Static Semantics

We now present a type system for source (label-free) programs. Section 4 extends the system to heaps and locations in order to prove type safety.

We use two auxiliary judgments on types. We allow **pack** τ', v **as** $\exists^{\phi}\alpha.\tau$ only if $\vdash \tau'$ packable. Assuming constants and pointers have the same run-time representation, this restriction ensures that code manipulating a package need not depend on the witness type, as it would if $\tau_1 \times \tau_2$ could be a witness type. We allow $e_1 := e_2$ only if e_1 has a type τ such that $\vdash \tau$ assignable. This judgment requires that any type of the form $\exists^{\&}\alpha.\tau'$ occurring in τ occurs in a pointer type. As a result, the witness type of a location holding **pack** τ_1, v **as** $\exists^{\&}\alpha.\tau_2$ never changes. A judgment on expressions, $\vdash e$ lval, defines the terms that are sensible for $\&e$ and $e := e'$. Figure 5 defines these auxiliary judgments.

With these auxiliary judgments, the rules for expressions and statements (Figure 6) are mostly straightforward. The context includes the type variables

$$\frac{}{\vdash \mathsf{int}\ \text{packable}} \qquad \frac{}{\vdash \alpha\ \text{packable}} \qquad \frac{}{\vdash \tau*\ \text{packable}}$$

$$\frac{\vdash \tau\ \text{packable}}{\vdash \tau\ \text{assignable}} \qquad \frac{\vdash \tau_0\ \text{assignable} \quad \vdash \tau_1\ \text{assignable}}{\vdash \tau_0 \times \tau_1\ \text{assignable}} \qquad \frac{\vdash \tau\ \text{assignable}}{\vdash \exists^{\delta}\alpha.\tau\ \text{assignable}}$$

$$\frac{}{\vdash x\ \text{lval}} \qquad \frac{}{\vdash \ell p\ \text{lval}} \qquad \frac{}{\vdash *e\ \text{lval}} \qquad \frac{\vdash e\ \text{lval}}{\vdash e.i\ \text{lval}}$$

Fig. 5. Static Semantics: Auxiliary Judgments

and term variables that are in scope. Term variables map to types. The rule for package expressions requires that the witness type is packable. The rules for **let** and **open** extend the context appropriately. The rule for assignment requires that the expressions' type is assignable. Most importantly, the rule for **open** e **as** $\alpha, *x$ **in** s requires that e has the form $\exists^{\&}\alpha.\tau$. (The repetition of α is not a restriction because $\exists^{\&}\alpha.\tau$ is α-convertible.) In other words, you cannot use this statement form to get an alias to a value of type $\exists^{\delta}\alpha.\tau$.

$$\Delta \subset \text{Tyvar} \qquad \Gamma : \text{Var} \rightharpoonup \text{Type} \qquad \frac{\tau \text{ is closed under } \Delta}{\Delta \vdash \tau} \qquad \frac{\text{for all } \tau \in \text{rng}(\Gamma),\ \Delta \vdash \tau}{\Delta \vdash \Gamma}$$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash c : \mathsf{int}} \qquad \frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta;\Gamma \vdash e : \tau_0 \times \tau_1}{\Delta;\Gamma \vdash e_i : \tau_i} \qquad \frac{\Delta;\Gamma \vdash e_0 : \tau_0 \quad \Delta;\Gamma \vdash e_1 : \tau_1}{\Delta;\Gamma \vdash (e_0, e_1) : \tau_0 \times \tau_1}$$

$$\frac{\Delta;\Gamma \vdash e : \tau\{\tau'/\alpha\} \quad \vdash \tau'\ \text{packable}}{\Delta;\Gamma \vdash \mathbf{pack}\ \tau', e\ \mathbf{as}\ \exists^{\phi}\alpha.\tau : \exists^{\phi}\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \tau \quad \vdash e\ \text{lval}}{\Delta;\Gamma \vdash \&e : \tau*} \qquad \frac{\Delta;\Gamma \vdash e : \tau*}{\Delta;\Gamma \vdash *e : \tau}$$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash \mathbf{skip}} \qquad \frac{\Delta;\Gamma \vdash s_1 \quad \Delta;\Gamma \vdash s_2}{\Delta;\Gamma \vdash s_1; s_2} \qquad \frac{\Delta;\Gamma[x \mapsto \tau] \vdash s \quad \Delta;\Gamma \vdash e : \tau \quad x \notin \text{dom}(\Gamma)}{\Delta;\Gamma \vdash \mathbf{let}\ x : \tau = e\ \mathbf{in}\ s}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \quad \Delta;\Gamma \vdash e_2 : \tau \quad \vdash e_1\ \text{lval} \quad \vdash \tau\ \text{assignable}}{\Delta;\Gamma \vdash e_1 := e_2}$$

$$\frac{\Delta, \alpha;\Gamma[x \mapsto \tau] \vdash s \quad \Delta;\Gamma \vdash e : \exists^{\phi}\alpha.\tau \quad \alpha \notin \Delta \quad x \notin \text{dom}(\Gamma)}{\Delta;\Gamma \vdash \mathbf{open}\ e\ \mathbf{as}\ \alpha, x\ \mathbf{in}\ s}$$

$$\frac{\Delta, \alpha;\Gamma[x \mapsto \tau*] \vdash s \quad \Delta;\Gamma \vdash e : \exists^{\&}\alpha.\tau \quad \alpha \notin \Delta \quad x \notin \text{dom}(\Gamma)}{\Delta;\Gamma \vdash \mathbf{open}\ e\ \mathbf{as}\ \alpha, *x\ \mathbf{in}\ s}$$

Fig. 6. Static Semantics: Source Programs

In short, the static semantics ensures that "δ-packages" are not aliased except with existential types and that "&-packages" are not mutated. The next section shows that these restrictions suffice for type soundness.

Returning to the example from Section 3.2, we can show why this program is not well-typed: First, the rules for packages and assignment ensure that the

three ϕ in the program (lines 3 and 6) must be the same. If they are δ, then line 4 is not well-typed because *pkg*'s type does not have the form $\exists^{\&}\alpha.\tau$. If they are &, then line 6 is not well-typed because we cannot derive $\vdash \exists^{\&}\alpha.\alpha* \times \alpha$ assignable.

4 Soundness Proof

In this section, we show how to extend the type system from the previous section in order to attain a syntactic [19] proof of type soundness. We then describe the proof, the details of which we relegate to a technical report [4].

For the most part, the extensions are the conventional ones for a heap and references [5], with several tedious complications that paths introduce. The basic idea is to prove that the types of labels are invariant. (That is, the value to which the heap maps a label may change, but only to a value of the same type.) However, we also need to prove an additional heap invariant for packages that have been opened with the **open** e **as** $\alpha, *x$ **in** s form. Such a package has the form **pack** τ', v **as** $\exists^{\&}\alpha.\tau$; its type does not mention τ', but τ' must not change. As explained below, we prove this invariant by explicitly preserving a partial map from locations to the witness types of "&-packages" at those locations.

4.1 Heap Static Semantics

For heap well-formedness, we introduce the judgment $H \vdash \Psi; \Upsilon$, where Ψ maps labels to types and Υ maps locations (labels and paths) to types. Intuitively, Ψ gives the type of each heap location, whereas Υ gives the witness types of the "&-packages" at some locations. Ψ is a conventional device; the use of Υ is novel. Every label in the heap must be in the domain of Ψ, but not every location containing an "&-package" needs to be in the domain of Υ. The location ℓp must be in the domain of Υ only if there is a value in the heap or program of the form $\ell p \mathtt{u} p'$. We say that Ψ' *extends* Ψ (similarly, Υ' *extends* Υ) if the domain of Ψ' contains the domain of Ψ and the two maps agree on their intersection.

The rule for type-checking locations, described below, needs a Ψ and Υ in its context. Hence we add a Ψ and Υ to the context for type-checking expressions and statements. Each rule in Figure 6 must be modified accordingly. A program state is well-typed if its heap and statement are well-typed and type-closed using the same Ψ and Υ. All of the above considerations are summarized in Figure 7.

What remains is type-checking expressions of the form ℓp. Intuitively, we start with the type $\Psi(\ell)$ and destruct it using p. However, when $p = \mathtt{u}p'$ we require that the type has the form $\exists^{\&}\alpha.\tau$ *and* the current location is in the domain of Υ. The resulting type uses Υ to substitute the witness type for α in τ. This operation is very similar to the way the dynamic semantics for **open** substitutes the witness type for the binding type variable. Υ has the correct witness type because $\vdash H : \Psi; \Upsilon$. We formalize these considerations with the auxiliary gettype relation in Figure 8. Using this relation, the rule for type-checking locations is:

$$\frac{\Upsilon; \ell \vdash \text{gettype}(\cdot, \Psi(\ell), p, \tau) \quad \Delta \vdash \Gamma}{\Psi; \Upsilon; \Delta; \Gamma \vdash \ell p : \tau}$$

$$\Psi : \mathrm{Lab} \rightharpoonup \mathrm{Type} \qquad \Upsilon : \mathrm{Lab} \times \mathrm{Path} \rightharpoonup \mathrm{Type}$$

$$\frac{\vdash H : \Psi; \Upsilon \qquad \Psi; \Upsilon; \emptyset; \emptyset \vdash s \qquad \text{for all } \tau \in \mathrm{rng}(\Psi) \cup \mathrm{rng}(\Upsilon),\ \tau \text{ is closed}}{\vdash (H, s)}$$

$$\frac{\mathrm{dom}(H) = \mathrm{dom}(\Psi) \qquad \text{for all } \ell \in \mathrm{dom}(H),\ \Psi; \Upsilon; \emptyset; \emptyset \vdash H(\ell) : \Psi(\ell) \qquad \text{for all } (\ell, p) \in \mathrm{dom}(\Upsilon),\ \mathrm{get}(H(\ell), p, \mathbf{pack}\ \Upsilon(\ell, p), v\ \mathbf{as}\ \exists^{\&}\alpha.\tau)}{\vdash H : \Psi; \Upsilon}$$

Fig. 7. Static Semantics: States and Heaps

$$\frac{}{\Upsilon; \ell \vdash \mathrm{gettype}(p, \tau, \cdot, \tau)} \qquad \frac{\Upsilon; \ell \vdash \mathrm{gettype}(p\mathtt{u}, \tau'\{\Upsilon(\ell, p)/\alpha\}, p', \tau)}{\Upsilon; \ell \vdash \mathrm{gettype}(p, \exists^{\&}\alpha.\tau', \mathtt{u}p', \tau)}$$

$$\frac{\Upsilon; \ell \vdash \mathrm{gettype}(p0, \tau_0, p', \tau)}{\Upsilon; \ell \vdash \mathrm{gettype}(p, \tau_0 \times \tau_1, 0p', \tau)} \qquad \frac{\Upsilon; \ell \vdash \mathrm{gettype}(p1, \tau_1, p', \tau)}{\Upsilon; \ell \vdash \mathrm{gettype}(p, \tau_0 \times \tau_1, 1p', \tau)}$$

Fig. 8. Static Semantics: Heap Objects

4.2 Proving Type Safety

We now summarize our proof of type safety. As usual with syntactic approaches, our formulation indicates that a well-typed program state (a heap and a statement) is either a terminal configuration (the statement is **skip**) or there is a step permitted by the dynamic semantics and all such steps produce well-typed program states.[3] Type safety is a corollary of preservation (subject-reduction) and progress lemmas, which we formally state as follows:

Lemma 1 (Preservation). *If* $\vdash (H, s)$ *and* $(H, s) \to (H', s')$*, then* $\vdash (H', s')$*.*

Lemma 2 (Progress). *If* $\vdash (H, s)$*, then either* $s = \mathbf{skip}$ *or there exist* H' *and* s' *such that* $(H, s) \to (H', s')$*.*

Proving these lemmas requires analogous lemmas for expressions:

Lemma 3 (Expression Preservation). *Suppose* $\vdash H : \Psi; \Upsilon$ *and* $\Psi; \Upsilon; \emptyset; \emptyset \vdash e : \tau$*. If* $H \vdash e \Downarrow_{\mathrm{L}} \ell p$*, then* $\Psi; \Upsilon; \emptyset; \emptyset \vdash \ell p : \tau$*. If* $H \vdash e \Downarrow_{\mathrm{R}} v$*, then* $\Psi; \Upsilon; \emptyset; \emptyset \vdash v : \tau$*.*

Lemma 4 (Expression Progress). *If* $\vdash H : \Psi; \Upsilon$ *and* $\Psi; \Upsilon; \emptyset; \emptyset \vdash e : \tau$*, then there exists a* v *such that* $H \vdash e \Downarrow_{\mathrm{R}} v$*. If we further assume* $\vdash e\ \mathrm{lval}$*, then there exist* ℓ *and* p *such that* $H \vdash e \Downarrow_{\mathrm{L}} \ell p$*.*

We prove the progress lemmas by induction on the structure of terms, using the preservation lemmas and a canonical-forms lemma (which describes the form of values of particular types) as necessary.

[3] For our formal language, it is also the case that the dynamic semantics cannot diverge, but we do not use this fact.

For example, if $s = \textbf{open}\ e\ \textbf{as}\ \alpha, x\ \textbf{in}\ s'$, we argue as follows: By assumption, there must be a Ψ and Υ such that $\Psi; \Upsilon; \emptyset; \emptyset \vdash s$, which means there is a τ such that $\Psi; \Upsilon; \emptyset; \emptyset \vdash e : \exists^{\phi}\alpha.\tau$. So by the expression-progress lemma, there is a v such that $H \vdash e \Downarrow_{\mathrm{R}} v$. By the expression-preservation lemma, $\Psi; \Upsilon; \emptyset; \emptyset \vdash v : \exists^{\phi}\alpha.\tau$. By the canonical-forms lemma, $v = \textbf{pack}\ \tau', v'\ \textbf{as}\ \exists^{\phi}\alpha.\tau$ for some τ' and v'. So for any $\ell \notin \mathrm{dom}(H)$, we can derive $(H, \textbf{open}\ e\ \textbf{as}\ \alpha, x\ \textbf{in}\ s) \rightarrow (H[\ell \mapsto v], s\{\tau'/\alpha\}\{\ell\cdot/x\})$

The proof cases involving explicit heap references (such as $s = e_1 := e_2$ because $H \vdash e_1 \Downarrow_{\mathrm{L}} \ell p$) require a lemma relating the get, set, and gettype relations:

Lemma 5 (Heap-Record Type Safety). *Suppose* $\vdash H : \Psi; \Upsilon$ *and* $\Upsilon; \ell \vdash \mathrm{gettype}(\cdot, \Psi(\ell), p, \tau)$. *Then there is a* v' *such that* $\mathrm{get}(H(\ell), p, v')$ *and* $\Psi; \Upsilon; \emptyset; \emptyset \vdash v' : \tau$. *Also, for all* v_1 *there is a* v_2 *such that* $\mathrm{set}(v, p, v_1, v_2)$.

This lemma's proof requires a stronger hypothesis: We assume $\mathrm{get}(H(\ell), p_1, v'')$, $\Upsilon; \ell \vdash \mathrm{gettype}(p_1, \tau'', p_2, \tau)$, and $\Psi; \Upsilon; \emptyset; \emptyset \vdash v'' : \tau''$ to prove $\mathrm{get}(H(\ell), p_1p_2, v')$ (and analogues of the other conclusions), by induction on the length of p_2. The interesting case is when $p_2 = \mathtt{u}p'$ because its proof requires the heap invariant that the map Υ imposes. In this case, the gettype assumption implies that $(\ell, p_1) \in \mathrm{dom}(\Upsilon)$, which means we know $\mathrm{get}(H(\ell), p_1, \textbf{pack}\ \Upsilon(\ell, p_1), v_p\ \textbf{as}\ \exists^{\&}\alpha.\tau')$. Without Υ and heap well-formedness, a type other than $\Upsilon(\ell, p_1)$ could be in the package. We then could not invoke the induction hypothesis—there would be no suitable τ'' when using v_p for v''.

We prove the preservation lemmas by induction on the evaluation (dynamic semantics) derivation, proceeding by cases on the last rule used in the derivation. We need auxiliary lemmas for substitution and heap extension:

Lemma 6 (Substitution).

- *If* $\Psi; \Upsilon; \Delta, \alpha; \Gamma \vdash s$, $\Delta \vdash \tau'$, *and* $\vdash \tau'$ packable, *then* $\Psi; \Upsilon; \Delta; \Gamma\{\tau'/\alpha\} \vdash s\{\tau'/\alpha\}$.
- *If* $\Psi; \Upsilon; \Delta; \Gamma[x \mapsto \tau'] \vdash s$ *and* $\Psi; \Upsilon; \Delta; \Gamma \vdash \ell p : \tau'$, *then* $\Psi; \Upsilon; \Delta; \Gamma \vdash s\{\ell p/x\}$.

Lemma 7 (Heap Extension). *Suppose* Υ' *and* Ψ' *are well-formed extensions of well-formed* Υ *and* Ψ, *respectively.*

- *If* $\Psi; \Upsilon; \Delta; \Gamma \vdash e : \tau$, *then* $\Psi'; \Upsilon'; \Delta; \Gamma \vdash e : \tau$.
- *If* $\Psi; \Upsilon; \Delta; \Gamma \vdash s$, *then* $\Psi'; \Upsilon'; \Delta; \Gamma \vdash s$.

With these lemmas, most of the proof cases are straightforward arguments: Rules using substitution use the substitution lemma. Rules extending the heap use the heap-extension lemma for the unchanged heap values and the resulting statement. Rules using the get relation use the heap-record type-safety lemma.

The most interesting cases are for $e_1 := e_2$ (proving the invariant Υ imposes still holds) and $\textbf{open}\ e\ \textbf{as}\ \alpha, *x\ \textbf{in}\ s'$ (extending Υ in the necessary way).[4]

[4] The $H \vdash e.i \Downarrow_{\mathrm{L}} \ell pi$ case is also "interesting" in that it extends the path on the *right* whereas the gettype relation destructs paths from the *left*. The $\textbf{open}\ e\ \textbf{as}\ \alpha, *x\ \textbf{in}\ s'$ case has an analogous complication because it adds $\mathtt{u}$ on the right.

We prove the case $e_1 := e_2$ as follows: By the assumed derivations and the expression-preservation lemma, there are ℓ, p, v, v', and τ such that we have $\Psi;\Upsilon;\emptyset;\emptyset \vdash \ell p : \tau$, $\Psi;\Upsilon;\emptyset;\emptyset \vdash v : \tau$, $\vdash \tau$ assignable, and $\mathrm{set}(H(\ell), p, v, v')$. Letting $\Psi' = \Psi$ and $\Upsilon' = \Upsilon$, we need to show $\vdash H[\ell \mapsto v'] : \Psi;\Upsilon$. The technical difficulties are showing $\Psi;\Upsilon;\emptyset;\emptyset \vdash v' : \Psi(\ell)$ and for all p such that $(\ell, p) \in \mathrm{dom}(\Upsilon)$, $\mathrm{get}(v', p, \mathbf{pack}\ \Upsilon(\ell, p), v''\ \mathbf{as}\ \exists^{\&}\alpha.\tau)$. The proofs are quite technical, but the intuition is straightforward: The typing judgment holds because v has type τ. The other requirement holds because the part of the old $H(\ell)$ replaced by v has no terms of the form $\mathbf{pack}\ \Upsilon(\ell, p), v\ \mathbf{as}\ \exists^{\&}\alpha.\tau$ (because $\vdash \tau$ assignable) and the rest of the old $H(\ell)$ is unchanged.

We prove the case $\mathbf{open}\ e\ \mathbf{as}\ \alpha, *x\ \mathbf{in}\ s'$ as follows: By the assumed derivations, the expression-preservation lemma, and the induction hypothesis, there are ℓ', p, τ', v, and τ such that $H \vdash e \Downarrow_{\mathrm{L}} \ell' p$, $\mathrm{get}(H(\ell'), p, \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\&}\alpha.\tau)$, and $\Psi;\Upsilon;\alpha;[x \mapsto \tau*] \vdash s'$. Heap well-formedness ensures $\vdash \tau'$ packable. Letting $\Psi' = \Psi[\ell \mapsto \tau\{\tau'/\alpha\}*]$ and $\Upsilon' = \Upsilon[(\ell', p) \mapsto \tau']$, we must show $\vdash H[\ell \mapsto \&\ell' p\mathtt{u}]$ and $\Psi;\Upsilon;\emptyset;\emptyset \vdash s\{\tau'/\alpha\}\{\ell\cdot/x\}$.[5] The latter follows from the heap-extension and substitution lemmas. For the former, we need a technical lemma to conclude that $\Upsilon';\ell' \vdash \mathrm{gettype}(\cdot, \Psi(\ell'), p\mathtt{u}, \tau\{\tau'/\alpha\})$. The difficulty comes from appending $\mathtt{u}$ to the right of the path. From this fact, we can derive $\Psi';\Upsilon';\emptyset;\emptyset \vdash \&\ell' p\mathtt{u} : \tau\{\tau'/\alpha\}*$. The heap-extension lemma suffices for showing the rest of the heap is well-typed.

5 Related Work

It does not appear that previous work has combined existential types with C-style aliasing and assignment.

Mitchell and Plotkin's original work [11] used existential types to model "second-class" abstraction constructs, so mutation of existential packages was impossible. Similarly, encodings using existentials, such as objects [1] and closures [10], have not needed to mutate a witness type. Current Haskell implementations [16, 15] include existential types for "first-class" values, as suggested by Läufer [7]. Of course, these systems' existential packages are also immutable.

More relevant are low-level languages with existential types. For example, Typed Assembly Language [12] does not allow aliases at the opened type. There is also no way to change the type of a value in the heap—assigning to an existential package means making a *pointer* refer to a different heap record. Xanadu [20], a C-like language with compile-time reasoning about integer values, also does not have aliases at the opened type. Roughly, int is elaborated to $\exists\alpha \in \mathcal{Z}.\alpha$ and uses of int values are wrapped by the necessary **open** expressions. This expression *copies* the value, so aliasing is not a problem. It appears that witnesses can change: this change would happen when an int in the heap is mutated.

Languages with *linear* existential types can provide a solution different than the ones presented in this work. In these systems, there is only one reference to an existential package, so *a fortiori* there are no aliases at the opened type.

[5] Note that $\mathrm{get}(H(\ell'), p, \mathbf{pack}\ \tau', v\ \mathbf{as}\ \exists^{\&}\alpha.\tau)$ ensures Υ' is an extension of Υ.

Walker and Morrisett [18] exploit this invariant to define a version of **open** that does not introduce any new bindings. Instead, it mutates the location holding the package to hold the package's contents. The Vault system [3] also has linear existential types. Formally, opening a Vault existential package introduces a new binding. In practice, the Vault type-checker infers where to put **open** statements and how to rewrite terms using the bindings that these statements introduce.

Smith and Volpano [13, 14] describe an integration of *universal* types into C. Their technical development is somewhat similar to ours, but they leave the treatment of structures to future work. It is precisely structures that motivate existential types and our treatment of them.

The well-studied problem of polymorphic references in ML [5, 19, 17] also results from quantified types, aliasing, and mutation, so it is natural to suppose the work presented here is simply the logical dual of the same problem. We have not found the correspondence between the two issues particularly illuminating, but we nonetheless point out similarities that may suggest a duality.

In this work's notation, if `NULL` can have any pointer type, the polymorphic-reference problem is that a naive type system might permit the following:

(1) **let** $x : \forall\alpha.(\alpha*) = \texttt{NULL}$ **in**
(2) **let** $zero : \mathsf{int} = 0$ **in**
(3) **let** $pzero : \mathsf{int}* = \&zero$ **in**
(4) $x := \&pzero$;
(5) $*x := 0$;
(6) $*pzero := zero$

The problem is giving x type $\forall\alpha.(\alpha*)$ (as opposed to $(\forall\alpha.\alpha)*$ or a monomorphic type), which allows us to treat the same location as though it had types $\mathsf{int}**$ and $\mathsf{int}*$. The example assigns to x at an *instantiated* type (line 4) and *then* instantiates x at a different type (line 5). In contrast, our unsoundness example with existential types assigns to a value at an *unopened* type only *after* creating an alias at the opened type.

The "value restriction" is a very clever way to prevent types like $\forall\alpha.(\alpha*)$ by exploiting that expressions of such types cannot be values in ML. It effectively prevents certain types for a mutable locations' *contents*. In contrast, our "no witness changes" solution prevents certain types for a mutation's *location*.

With the exception of linear type systems, we know of no treatment of universal types that actually permits the types of values at mutable locations to change, as our "no aliases at the opened type" solution does. It is unclear what an invariant along these lines would look like for polymorphic references.

6 Acknowledgments

I am grateful for relevant discussions with Rob DeLine, Manuel Fähndrich, Greg Morrisett, David Walker, Kevin Watkins, Yanling Wang, and Steve Zdancewic.

References

1. Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.
2. *Cyclone User's Manual*, 2001. http://www.cs.cornell.edu/projects/cyclone.
3. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, June 2001.
4. Dan Grossman. Existential types for imperative languages: Technical results. Technical Report 2001-1854, Cornell University Computer Science, October 2001.
5. Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994.
6. Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002. To appear.
7. Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.
8. B. Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1984.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.
11. J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version in 12th ACM Symposium on Principles of Programming Languages, 1985.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
13. Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 341–355, Linköping, Sweden, April 1996. Springer-Verlag.
14. Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(2–3):49–72, 1998.
15. *The Glasgow Haskell Compiler User's Guide, Version 5.02*, 2001. http://www.haskell.org/ghc/docs/latest/set/book-users-guide.html.
16. *The Hugs 98 User Manual*, 2001. http://www.cse.ogi.edu/PacSoft/projects/Hugs/pages/hugsman/index.html.
17. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.
18. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, Montreal, Canada, September 2000. Springer-Verlag.
19. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
20. Hongwei Xi. Imperative programming with dependent types. In *15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, CA, June 2000.