Quantified Types in an Imperative Language

DAN GROSSMAN
University of Washington

We describe universal types, existential types, and type constructors in Cyclone, a strongly typed C-like language. We show how the language naturally supports first-class polymorphism and polymorphic recursion while requiring an acceptable amount of explicit type information. More importantly, we consider the soundness of type variables in the presence of C-style mutation and the address-of operator. For polymorphic references, we describe a solution more natural for the C level than the ML-style "value restriction." For existential types, we discover and subsequently avoid a subtle unsoundness issue resulting from the address-of operator. We develop a formal abstract machine and type-safety proof that capture the essence of type variables at the C level.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Cyclone, existential types, polymorphism, type variables

1. INTRODUCTION

Strongly typed programming languages prevent certain programming errors and provide a way for users to enforce abstractions (e.g., at interface boundaries). Inevitably, strong typing prohibits some correct programs, but an expressive type system reduces the limitations. In particular, universal (i.e., polymorphic) types let code operate uniformly over many types of data, and data-hiding constructs (such as closures and objects) let users give the same type to data with some components of different types.

High-level languages such as ML [Milner et al. 1997; Chailloux et al. 2000; Leroy 2002a], Haskell [Jones and Hughes 1999], and GJ (a Java extension) [Bracha et al. 1998] have sound type systems with universal types. Essentially every strongly typed high-level language has a powerful data-hiding mechanism. Language implementations must balance performance-critical tradeoffs

Author's address: Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350; email: djg@cs.washington.edu.

that polymorphism and data hiding introduce: implementing polymorphic functions requires some combination of indirection (passing pointers to data), run-time type information (passing type tags as extra arguments), or code duplication (generating different code for data of different types). Implementing closures or objects requires data structures for storing free variables or object fields.

In lower-level languages, users are not at the mercy of the implementation for efficient data representation. Conventionally, this flexibility has required sacrificing the benefits of strong typing. Recent work on languages such as Typed Assembly Language [Morrisett et al. 1999b, 2002] has applied ideas from strongly typed high-level languages to lower-level settings. Typed Assembly Language distinguishes type variables that represent types of different sizes and uses *existential types* [Mitchell and Plotkin 1988] to encode data-hiding constructs [Minamide et al. 1996; Bruce et al. 1999].

In this work, we describe the essence of universal and existential types in Cyclone [Cyclone 2001; Jim et al. 2002], a safe language that is very much like C [ISO 1999]. Our implementation provides a level of abstraction equal to conventional C implementations, which is squarely between high-level and assembly languages. Cyclone does not add levels of indirection or hidden data fields, but it uses native calling conventions and lets values have platform-specific sizes. As in C, mutation and aliasing run rampant in Cyclone programs.

1.1 Contributions

This work describes Cyclone's approach to quantified types, novel technical problems and solutions that arise at the C level, and a formal machine suitable for proving that the solutions are sound. Despite some interconnection, we can enumerate more specific contributions:

(1) A complete design, implementation, and extensive use of a polymorphic type system for a safe C-level language. Unlike ML or Haskell, Cyclone (and C) distinguish "left expressions" from "right expressions" and evaluate them differently. For example, the address-of operator (i.e., the `&` in `&e`) makes a right expression from a left expression. Even object-oriented languages with quantified types typically do not have `&`, which has interesting semantic and soundness ramifications.

(2) A solution to the polymorphic-reference problem different from the "value restriction" or "weak type variables." In Cyclone, it suffices to restrict universal quantification to (immutable) functions, but the formal language reveals that a much weaker restriction suffices: We need only ensure that type instantiations are not left expressions. Section 5.2 discusses why the stronger restriction is not a burden in practice.

(3) A new soundness issue that arises from existential types, mutation, and the address-of operator. We present the problem, identify its source, present two orthogonal solutions, and prove both solutions sound.

(4) A simple kind system for distinguishing pointer types from types of unknown size. While crucial for implementing Cyclone efficiently and

modularly, it turns out restricting the use of unknown-size types is not a soundness issue: the formal type-soundness proof never relies on the restrictions because the formal dynamic semantics can implicitly "copy" values of arbitrary size whereas such an operation is difficult to compile.

These results are crucial for safe C-like languages that rely on universal types for code reuse and existential types for data hiding. Moreover, the results serve as warning and guidance for language designers combining mutation and polymorphism at any abstraction level.

1.2 Overview

Section 2 places this work in context by describing Cyclone's goals, applications to date, and relevant features. It motivates C-level quantified types, but it is not necessary for understanding them. Section 3 then describes how we use quantified types in Cyclone to describe polymorphic code, data hiding, and container types. This section adapts well-known ideas to a C-like language; readers familiar with quantified types and more interested in technical issues than programming-language design might skip this section. Section 4 informally presents the complications that arise at the C level and how Cyclone addresses them. In particular, it highlights the technical insights of this work. Section 5 discusses the type system's limitations. Section 6 presents a formal language, abstract machine, and type system suitable for arguing that Cyclone is type-safe, which is particularly important in light of the interaction with mutation. Section 7 discusses related work. Section 8 concludes the article. The Appendix proves type safety for the formal language.

This work is presented more thoroughly in the author's dissertation [Grossman 2003a]. The unsound interaction with existential types is the subject of a previous conference publication [Grossman 2002]. The Cyclone compiler is publicly available.[1]

2. CYCLONE CONTEXT

Because this work extracts the essence of a type system from a full programming language, it is useful to describe briefly the language's goals and features. In particular, type-level variables in Cyclone enforce several invariants, but in this work we consider only invariants regarding abstraction of conventional types.

2.1 Goals and Applications

Cyclone is a type-safe language that, except for safety, is very much like C. The language uses many techniques to avoid the ways a C program can thwart safety (dangling-pointer dereferences, incorrect type casts, array-bounds violations, etc.). Remaining at the C level makes it easier to interoperate with low-level systems, port legacy systems incrementally, and write applications where low-level data-representation and memory-management decisions are essential. The ultimate goal is to provide a safe, convenient alternative suitable for implementing most of a large software system such as an operating system.

[1] Go online to `http://www.research.att.com/projects/cyclone`.

Cyclone has enjoyed moderate success as a safe language for research projects developing low-level extensible systems. Example systems using Cyclone include MediaNet [Hicks et al. 2003] (a multimedia overlay network), the Open Kernel Environment [Bos and Samwel 2002] (a kernel allowing partially trusted extensions), RBClick [Patel and Lepreau 2003] (an active network), and STP [Patel et al. 2003] (an extensible transport protocol). These systems exploit Cyclone's safety guarantee, user-controlled performance, and low-level interoperability. In addition, the Cyclone compiler and extensive libraries are written in Cyclone, demonstrating Cyclone is effective for large programs (compiling over 100,000 lines of code in about 30s on commodity hardware). Generic libraries, such as for lists and closures, require quantified types and are used often in practice. We have studied the run-time performance and source-program changes necessary to port C benchmarks to Cyclone [Jim et al. 2002; Grossman et al. 2002; Hicks et al. 2004].

2.2 Achieving Safety

To avoid needing C's unsafe features, Cyclone employs complementary and synergistic techniques including run-time checking, intraprocedural flow analysis, and a powerful type system. Preventing array-bounds violations is an example that demonstrates how programmers can choose among the techniques: the simplest approach uses "fat pointers," which like arrays in high-level languages carry bounds information and raise a run-time exception upon an out-of-bounds access. This approach does not detect errors until run-time and imposes space and time overhead compared to C arrays. Therefore, programmers can use the type system to specify an array's size is known statically or the size is stored in a user-specified location. In either case, the flow analysis ensures the user checks the bound before an access. Leaving the check under user control allows users to control cost by, for example, hoisting checks out of loops. When the intraprocedural analysis is too weak, users can use the type system to express interprocedural and data-structure invariants.

As an example, a programmer could define a `struct` type describing values in which there exists an integer α such that the `sz` field holds α, and the `arr1` and `arr2` fields point to arrays of size α.[2] The two arrays share a size and the flow analysis ensures users of the arrays consult `sz` appropriately. On a technical level, this example uses type-level variables ranging over constant integers and existential quantification to describe a safety-critical invariant.

In fact, Cyclone type variables and quantified types are essential to the entire type system. We use them to describe array lengths, object lifetimes [Grossman et al. 2002], locking disciplines [Grossman 2003b], and polymorphic functions. Indeed, this unified approach to seemingly disparate safety threats is the essence of Cyclone and the key way we keep the language's complexity manageable. In practice, it means most Cyclone functions are polymorphic in one way or another, so a sound and convenient approach to type variables is an absolute necessity.

[2] In Cyclone, we write `struct T { <‘i> tag_t<‘i> sz; int*{‘i} arr1; int*{‘i} arr2;};`

Therefore, a careful study of type variables for a safe, C-level language provides an important intellectual foundation. For this purpose, the work here considers only "conventional" type variables abstracting types. Doing so lets us focus on the interactions among type variables, mutation, and the address-of operator without complications (e.g., an effects system) that other kinds of type variables introduce.

3. BASIC CONSTRUCTS

This section shows how we adapt the well-known theory of quantified types to Cyclone, deferring complications such as nonuniform sizes of data objects to Section 4.

One form of type in Cyclone is a *type variable* (written α, β, etc.). (The actual ASCII syntax is `'a`, `'b`, etc.). Certain constructs introduce type variables in a particular scope. The type variable describes values of an unknown type. The power of type variables is that a type variable always describes the *same* unknown type, within some scope. We present each of the constructs that introduce type variables and explain their usefulness. We then present techniques that render optional much of the cumbersome notation in the explanations.

3.1 Universal Quantification

The simplest example of universal quantification is this function:

```
α id<α>(α x) { return x;}
```

The function `id` has a type parameter represented by the type variable α in angle brackets; `id` takes an argument `x` of type α and returns a value of type α. The function is *polymorphic* because callers can *instantiate* α with different types to use the function for values of different types.

In general, a function can introduce *universally bound* type variables $\alpha_1, \alpha_2, \ldots$ by writing <$\alpha_1, \alpha_2, \ldots$> after the function name. The type variables' scope is the parameters, return type, and function body. The type of the function is a *universal type*. For example, the type of `id` is `α id<α>(α)`, pronounced, "for all α, `id` takes an α and returns an α." Using more conventional notation for universal types and function types, we would write $\forall\alpha.\alpha \rightarrow \alpha$. As Section 4 explains, `id` cannot actually be used for all types.

To use a polymorphic function (i.e., a value of universal type), we *instantiate* the type variables with types. As examples, `id<int>` has type `int id(int)` and if `y` has type `int*`, then `id<int*>(y)` has type `int*`.

C programmers typically use `void*` to compensate for the lack of type variables. Doing so is more flexible, error-prone, and cumbersome. For example, `void* id(void*)` expresses no connection between the argument and return type. If they were different, clients might still think they were the same. In any case, clients must use an unchecked cast on the result before using it.

More interesting polymorphic functions take function-pointer arguments, such as this function, which applies a function to the elements of a 10-element array[3]:

[3] Where convenient, we exploit C's arcane rules regarding function pointers and their types. The first argument to `app10` is an implicit pointer that is implicitly dereferenced in `f(env,arr[i])`.

```
void app10<α,β>(void f(β, α), β env, α arr[10]) {
  for(int i=0; i < 10; ++i)
    f(env,arr[i]);
}
```

The function call type-checks because the arguments have the type the function expects, namely, β and α. We pass env explicitly because Cyclone functions must be closed. (Function closures are encodable, but not built-in.) To show that the code is reusable, we instantiate it two different ways:

```
int i; // global variable that functions modify
void add_int(int *p,  int  x)  { *p += x;  }
void add_ptr(int *p1, int *p2) { *p1 += *p2; }
void add_intarr(int  arr[10]) { app10<int*,int >(add_int,&i,arr); }
void add_ptrarr(int* arr[10]) { app10<int*,int*>(add_ptr,&i,arr); }
```

In short, universally quantified type variables are a powerful way to encode idioms in which code does not need to know certain types, but does need to relate the types of multiple arguments or arguments and results. In C, we conflate all such types with void*, sacrificing the ability to detect inconsistencies with the type system.

In Cyclone, the refined information from polymorphism induces no run-time cost; type instantiation is a compile-time operation. The compiler does not duplicate code; there is one compiled version of app10 regardless of the types with which the program instantiates it. Similarly, instantiation does not require the function body, so we can compile app10's uses separately from app10's implementation. (This approach assumes all pointers have the same representation and calling convention. The C standard disallows this assumption, though it holds on many architectures. Compiling Cyclone under weaker assumptions is beyond the scope of this work.)

We also do not use run-time type information: we pass app10 the same arguments we would in C. There are no "secret arguments" describing the type instantiation, which is important for two reasons. First, it meets our goal of "acting like C" and not introducing extra data and run-time cost. We do not want to penalize reusable code. Second, it becomes complicated to compile polymorphic code differently than monomorphic code, as this example suggests:

```
α id<α>(α x) { return x; }
int f(int x) { return x+1; }
void g(bool b) {
  int (*h)(int) = (b ? id<int> : f);
  // use h
}
```

Because id<int> and f have the same type, we need to support (indirect) function calls where we do not know until run-time which we are calling. To do so without extra run-time cost, the two functions must have the same calling convention.

Cyclone also supports *first-class polymorphism* and *polymorphic recursion*. The former means universal types can appear anywhere function types appear, not just in the types of top-level functions. This small example requires this feature:

```
void f(void g<α>(α), int x, int *y) {
   g<int> (x);
   g<int*>(y);
}
```

Polymorphic recursion lets recursive function calls instantiate type variables differently than the outer call. Without this feature, within a function f quantifying over $\alpha_1, \alpha_2, \ldots$, all instantiations of f must be f<$\alpha_1, \alpha_2, \ldots$>. This small example uses polymorphic recursion:

```
α slow_id<α>(α x, int n) {
  if(n >= 0)
    return *slow_id<α*>(&x, n-1);
  return x;
}
```

First-class polymorphism and polymorphic recursion are natural features. We emphasize them because languages (most notably ML) often lack them because they usually make full type inference undecidable [Wells 1999; Henglein 1993; Kfoury et al. 1993]. Cyclone provides convenient mechanisms for eliding type information without supporting full inference. As such, it easily supports these more expressive features, which become more important when using type variables for safe memory management [Grossman et al. 2002] and multithreading [Grossman 2003b].

3.2 Existential Quantification

Cyclone struct types can *existentially quantify* over types, as in this example:

```
struct T { <α>
  α env;
  int (*f)(α);
};
```

In English, this reads as "given a value of type struct T, *there exists* a type α such that the env field has type α and the f field is a function expecting an argument of type α." The scope of α is the field definitions. A common use of such types is a library interface that lets clients register *call-backs* to execute when some event occurs. Different clients can register call-backs that use different types for α, which is more flexible than the library writer choosing a type that all call-backs process. When the library calls the f field of a struct T value, the only argument it can use is the env field of the *same* struct because it is the only value known to have the type the function expects. In short, we have a much stronger interface than using void* for the type of env and the argument type of f.

Existential types describe *first-class abstract types* [Mitchell and Plotkin 1988]. For example, we can describe a simple abstraction for integer sets with this type:

```
struct IntSet { <α>
  α elts;
  void (*add)(α,int);
  void (*remove)(α,int);
  bool (*is_member)(α,int);
  struct IntSet* (*filter)(α, bool(*f)(int));
};
```

The `elts` field stores the data necessary for implementing the operations. Existential quantification ensures clients do not assume any particular type for `elts`, so the implementation of each set remains abstract. For example, we can create a set that stores its elements in a list and another set that stores its elements in an array to which `elts` points. The abstract types are *first-class*: we can choose among set implementations at run-time. Moreover, we can put sets using lists and sets using arrays together, such as in an array with elements of type `struct IntSet`.

Most programming languages do not have existential types per se. Rather, they have first-class function closures or first-class objects (in the sense of object-oriented programming). These features have well-known similarities with existential types. They all have types that do not constrain private state (fields of existentially bound types, free variables of a first-class function, private fields of an object). Indeed, a language without any first-class data hiding construct is impoverished, but any one suffices for encoding simple forms of the others. For example, we can use existential types to encode closures [Minamide et al. 1996] and some forms of objects [Bruce et al. 1999]. Many of the difficult complications in Cyclone arise from existential types (we have to modify the examples of this section to support manual memory management [Grossman et al. 2002] or thread-local data [Grossman 2003b]), but the problems would remain if we replaced them with another data-hiding feature. In essence, all such features allow the type of reachable data (via a field or free variable) not to appear in the type of the enclosing construct (closure, object, or package). This hiding means we cannot use the syntax of the enclosing construct's type to restrict the reachable data unless we "leak" more information in some way, such as a kind system or an effect system.

Cyclone provides existential types rather than closures or objects because they give programmers more control over data representation. Compiling closures or objects efficiently involves space and time tradeoffs that can depend on the program [Appel 1992; Abadi and Cardelli 1996]. At the C level, we prefer a powerful type system in which programmers can make such tradeoffs.

We now present the term-level constructs for creating and using values of existential types. We call such values *existential packages*. When creating an existential package, we must choose types for the existentially bound type variables, and the fields must have the right types for our choice. We call the types the *witness types* for the existential package. They serve a similar purpose to the types used to instantiate a polymorphic function. Witness types do not exist at run-time.

To simplify checking that packages are created correctly, users must create them via *constructor expressions*, as in this example, which uses `struct T` defined above:

```
int deref(int * x) { return *x;  }
int twice(int   x) { return 2*x; }
int i;
struct T makeT(bool b) {
  if(b)
    return T{<int*> .env=&i, .f=deref};
  return T{<int> .env=i, .f=twice};
}
```

If the code executes the body of the if-statement, we use `int*` for the witness type of the returned value, else we use `int`. The return type is just `struct T` (a "flat" multiword object); the witness type is not part of it. We never allow inconsistent fields: For the given types of `i` and `deref`, there is no τ such that `T{<`τ`> .env=i, .f=deref}` is well typed.

To use an existential package, Cyclone provides *pattern matching* to *unpack* (often called *open*) the package, as in this example:

```
int useT(struct T pkg) {
  let T{<β> .env=e, .f=fn} = pkg;
  return fn(e);
}
```

The pattern binds `e` and `fn` to (copies of) the `env` and `f` fields of `pkg`. It also introduces the type variable β. The scope of β, `e`, and `fn` is the rest of the code block (in the example, the rest of the function). The types of `e` and `fn` are β and `int (*)(`β`)` (a function from β to `int`), respectively, so the call `fn(e)` type-checks. Within its scope, we can use β like any other type. For example, we could write β `x = id<`β`>(e);`.

We require reading the fields of a package with pattern matching (instead of individual field projections), in the same way as we require building a package all at once. For the most part, not allowing the "`.`" and "`->`" operators for existential types simplifies type-checking. When creating a package, we can check for the correct witness types. When using a package, it clearly defines the types of the fields and the scope of the introduced type variables. We can unpack a package more than once, but the unpacked values will have types using different type variables (the type system properly distinguishes each binding occurrence), so we could not use, for example, the function pointer from one unpack with the environment from the other.

3.3 Type Constructors

Type constructors with *type parameters* let us concisely describe families of types. Applying a type constructor to a list of types produces a type. For example, we can use this type constructor to describe linked lists:

```
struct List<α> {
  α hd;
  struct List<α> * tl;
};
```

The type constructor `struct List` is a type-level function: Given a type, it produces a type. So the types `struct List<int>`, `struct List<int*>`, and `struct`

List<struct List<int>*> are different. The type α is the formal parameter; its scope is the field definitions. Because the tl field has type struct List<α>*, all types that struct List produces describe homogeneous lists (i.e., all elements have the same type).

Type constructors can encode more sophisticated idioms. We can use this type constructor to describe lists where the elements alternate between two types:

```
struct ListAlt<α,β> {
  α hd;
  struct ListAlt<β,α> * tl;
};
```

Building and using values of types that type constructors produce is straightforward. For example, to make a struct List<int>, we put an int in the hd field and a struct List<int>* in the tl field. If x has type struct List<int>, then x.hd and x.tl have types int and struct List<int>*, respectively.

The conventional use of type constructors is to describe a "container type" and then write a library of polymorphic functions for the type. For example, these prototypes describe generic routines for linked lists[4]:

```
int length<α>(struct List<α>*);
bool cmp<α,β>(bool f(α,β), struct List<α>*, struct List<β>*);
struct List<α>* append<α>(struct List<α>*, struct List<α>*);
struct List<β>* map<α,β>(β f(α), struct List<α>*);
```

Type constructors and existential quantification also interact well. For example, struct Fn is a type constructor for encoding function closures:

```
struct Fn<α,β> { <γ>
  β (*f)(γ, α);
  γ env;
};
```

This constructor describes functions from α to β with an environment of some abstract type γ. Different values of type struct Fn<τ_1,τ_2> can have environments of different types. A library can provide polymorphic functions for operations on closures, such as application, composition, currying, and uncurrying.

Parameters for typedef provide a related convenience. The parameters to a typedef are bound in the type definition. We must apply such a typedef to produce a type, as in this example:

```
typedef struct List<α> * list_t<α>;
```

The rightmost α is the binding occurrence. As in C, typedef is transparent: a use is equivalent to its definition. So list_t<int> is just an abbreviation for struct List<int>*.

[4] In practice, we also provide versions of cmp and map taking environment arguments, as in our earlier app10 example.

3.4 Default Annotations

We have added universal quantification, existential quantification, and type constructors so that programmers can encode a large class of idioms for reusable code without resorting to unchecked casts. So far, we have focused on the type system's expressiveness without describing features that reduce the burden on programmers. We now present these techniques.

First, in function definitions and function prototypes at the top level (i.e., not within a function body or type definition), the outermost function implicitly universally quantifies over any free type variables. So instead of writing

```
α id<α>(α x);
list_t<β> map<α,β>(β f(α), list_t<α>);
```

we can write

```
α id(α x);
list_t<β> map(β f(α), list_t<α>);
```

Because instantiation of polymorphic functions is typically implicit, the order of the type variables rarely matters. If explicit instantiation is necessary, explicit quantification is too. Explicit quantification is also necessary for first-class polymorphism:

```
void f(void g<α>(α), int x, int *y);
```

Omitting the quantification would make `f` polymorphic instead of g.

Second, instantiation of polymorphic functions and selection of witness types can be implicit. The type-checker infers the correct instantiation or witness from the types of the arguments or field initializers, respectively. Some examples are

```
struct T { <α> α env; int (*f)(α); };
struct T f(list_t<α> lst) {
  id(7);
  map(id,lst);
  return T{.env=7, .f=id};
}
```

Polymorphic recursion poses no problem because function types are explicit. Inference does not require immediately applying a function, as this example shows:

```
void f() {
  int (*idint)(int) = id;
  idint(7);
}
```

In fact, type inference uses unification within function bodies such that all explicit type annotations are optional. Although the implemented inference procedure is incomplete (a function may typecheck only if some explicit types are present), in practice we can omit almost all explicit types in function bodies. Every occurrence of a polymorphic function is implicitly instantiated; delaying the instantiation requires explicit syntax, as in this example:

```
void f(int x) {
  α (*idvar)<α>(α) = id<>; // do not instantiate yet
  idvar(x);  // instantiate with int
  idvar(&x); // instantiate with int*
}
```

Third, unpacking does not require explicit type variables. The type-checker can create the correct number of type variables and gives terms the appropriate types. We can write

```
int useT(struct T pkg) {
  let T{.env=e, .f=fn} = pkg;
  return fn(e);
}
```

The type-checker creates a type variable β with the same scope that a user-provided type variable would have.

Fourth, we can omit explicit applications of type constructors or apply them to too few types. In function bodies, unification infers omitted arguments. In other cases (function prototypes, function argument types, etc.) the type-checker fills in omitted arguments with fresh type variables. So these declarations are equivalent:

```
int length(list_t<α>);
int length(list_t);
```

In practice, we need explicit type variables only to indicate that multiple terms have the same unknown type. There is no reason for the programmer to create type variables for types that occur only once, such as the element type for `length`, so the type-checker creates names and fills them in. We do not mean that type constructors are types, just that application can be implicit.

None of the rules for omitting explicit type annotations require the type-checker to perform interprocedural analysis. Every function has a complete type determined only from its prototype, not its body, so the type-checker can process each function body without reference to any other. Even earlier declarations of the function are relevant only in that the types must be equivalent.[5]

4. COMPLICATIONS

This section considers features of Cyclone (largely inherited from C) that complicate sound, efficient implementation of quantified types. The features include nonuniform data sizes (Section 4.1), multiple calling conventions (Section 4.2), mutable data (Sections 4.3 and 4.4), and static variables (Section 4.5).

4.1 Size

Different values in C and Cyclone can have different sizes. For example, we expect a `struct` with three `int` fields to be larger than a `struct` with two `int` fields. Conventionally, all values of the same type have the same size, and we call the size of values of a type the *size of the type*. C implementations have latitude

[5] We do not allow a prototype to be less general than the corresponding definition.

in choosing type sizes (to accommodate architecture restrictions like native-word size and alignment constraints), but sizes are compile-time constants.

However, not all sizes are known because C has *abstract* `struct` declarations (also known as *incomplete structs*), such as "`struct T;`". To enable efficient code generation, C greatly restricts where such types can appear. For example, if `struct T` is abstract, C forbids this declaration:

```
struct T2 {
  struct T x;
  int y;
};
```

The implementation would not know how much space to allocate for a variable of type `struct T2` (or `struct T`). If `s` has type `struct T2*`, there is no simple, efficient way to compile `s->y`. In short, because the size of abstract types is unknown, C permits only pointers to them.

In Cyclone, type variables are abstract types, so we confront the same problems. Cyclone provides two solutions, which we explain after introducing the *kind* system that describes them. Kinds classify types just like types classify terms. We consider only two kinds, `A` (for "any") and `B` (for "boxed"). Every type has kind `A`. Pointer types and `int` also have kind `B`. We consistently assume, unlike C, that `int` has the same size and calling convention as `void*`. Saying `int` is just more concise than saying, "an integral type represented like a pointer."

The two solutions for type variables correspond to type variables of kind `B` and type variables of kind `A`. A type variable's binding occurrence usually specifies its kind with α:`B` or α:`A`, and the default is `B`.[6] All the examples in Section 3 used type variables of kind `B`. Simple rules dictate the restrictions kinds impose:

- A universally quantified type variable of kind `B` can be instantiated only with a type of kind `B`.
- An existentially quantified type variable of kind `B` can have witness types only of kind `B`.
- If α has kind `A`, then α is subject to the same restrictions as abstract `struct` types in C. It must occur under pointers and one cannot dereference pointers of type α`*`.
- The type variables introduced in an existential unpack do not specify kinds. Instead, the ith type variable has the same kind as the ith existentially quantified type variable in the type of the package unpacked.

Less formally, type variables of kind `B` classify types that we can convert to `void*` in C. We forbid instantiating such a type variable α with a `struct` type for the same reasons C forbids casting a `struct` type to `void*`. Type variables of kind `A` are less common because of the restrictions on their use, but here is an example:

[6] In Cyclone, the default kind is sometimes `A`, depending on how the type variable is used, but we use simpler default rules here.

```
struct T1<α:A> { α *x; α *y; };
void swap<α:A>(struct T1<α> *p) {
  α * tmp = p->x;
  p->x = p->y;
  p->y = tmp;
}
```

Because swap quantifies over a type of kind A, we can instantiate swap with any type.

A final addition makes type variables of kind A more useful. The unary type constructor sizeof_t describes the size of a type: the only value of type sizeof_t<τ> is sizeof(τ). As in C, we allow sizeof(τ) only where the compiler knows the size of τ (i.e., all abstract types are under pointers).

One purpose of sizeof_t is to give Cyclone types to primitive library routines we can write in C, such as this function for copying memory:

```
void mem_copy<α:A>(α* dest, α* src, sizeof_t<α> sz);
```

Disappointingly, it is impossible to implement mem_copy in Cyclone, but we can provide a safe interface to a C implementation. Implementing mem_copy requires a loop that copies an α value in pieces and Cyclone's type system does not permit such operations, nor could it verify that the result is a value of type α.

4.2 Calling Convention

We do not give float kind B, which deserves explanation because we could assume that float has the same size as void*, as we did with int. Many architectures use a different calling convention for floating-point arguments. If float had kind B, then we could not have one implementation of a polymorphic function while using native calling conventions, as this example demonstrates:

```
float f1(float x) { return x; }
α f2(α x) { return x; }
void f3(bool b) {
  float (*f)(float) = b ? f1 : f2;
  f(0.0);
}
```

As Section 7 describes, the ML community has explored many solutions for giving float kind B. None preserve data representation (a float being just a floating-point number) without secret function arguments or a possibly exponential increase in the amount of compiled code. In Cyclone, we prefer to expose this problem to programmers; they can encode the solutions manually.

4.3 Polymorphic References

Safety demands that the expressiveness gained via type variables does not let a program view data at the wrong type. Mutable locations are a notorious source of errors in safe-language design, and most locations in Cyclone are mutable. In this section and the next, we describe potential pitfalls and how Cyclone avoids them.

Cyclone does not have *polymorphic references*, which allow programs like this:

```
void bad(int *p) {
  (∀α.(α*)) x = NULL<>; // not legal Cyclone, but could be
  x<int*> = &p;         // not legal Cyclone, and should not be
  *(x<int>) = -1;
  *p = 123;             // violates safety
}
```

We can give the Cyclone keyword NULL any pointer type (i.e., any type of the form $\tau*$), so it is tempting to give it the polymorphic type $\forall\alpha.(\alpha*)$. By not instantiating it, we can give x the same type. But by assigning to an instantiation of x (i.e., x<int*> = &p), we put a nonpolymorphic value in x. Hence, the second instantiation (x<int>) is wrong and leads to a safety violation.

Note that the second and third statements both instantiate polymorphic types, but the second statement uses type instantiation as a left expression (the first subexpression of assignment) and the third as a right expression (the only subexpression of pointer dereference). So unless we allow e<t> to be a left expression (if e is a left expression), bad fails to typecheck.[7] In fact, Section 6 proves that a language with distinct left and right expression evaluation is sound if type instantiation is not a left expression. That is, this restriction is sufficient.

The simplicity and directness of this approach is possible exactly because left and right expressions are distinct, unlike in ML where the fact that a location is mutated is hidden behind a function type (specifically := which has type 'a ref * 'a -> unit). This approach also has a theoretical justification: models of the polymorphic lambda calculus can treat type application as subtyping [Mitchell 1988] and writing to a location requires contravariance in the location's type.

In practice, the actual Cyclone language enforces a more restrictive, trivial-to-check property than forbidding type application as a left expression: the only polymorphic values are functions and functions are immutable. Hence any type instantiation is a reference to immutable code, so a fortiori no type instantiation could be the left subexpression of an assignment. Our formalism therefore justifies Cyclone's soundness while proving a much more general result.

Section 7 discusses how other languages avoid polymorphic references.

4.4 Mutable Existential Packages

Prior work has not carefully studied the interaction of existential types with features like mutation and C's address-of (&) operator. Orthogonality suggests that existential types in Cyclone 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 {<α> α x; α y;}; and a polymorphic function void swap(β*, β*) for swapping two locations' contents, we would like to permit a call like swap(&v.x, &v.y). Unfortunately, these features create a subtle unsoundness.

[7] Removing the second statement makes bad attempt to dereference a NULL pointer. Cyclone has types for compile-time NULL-checking, but the type shown causes a checked run-time exception.

The first feature—mutating a location holding a package to hold a different package with a different witness type—is supported naturally. After all, if `p1` and `p2` both have type `struct T`, then, as in C, `p1=p2` copies the fields of `p1` into the fields of `p2`. Note that the assignment can *change* `p2`'s witness type, as in this example:

```
struct T {<α> void (*f)(int, α); α env;};
void ignore(int x, int y) {}
void assign(int x, int *y) { *y = x; }
void f(int* ptr) {
  struct T p1 = T{<int>  .f=ignore, .env=-1};
  struct T p2 = T{<int*> .f=assign  .env=ptr};
  p2 = p1;
}
```

Because we forbid access to existential-package fields with the "`.`" or "`->`" operators, we do not yet have a way to acquire the address of a package field. We need this feature for the `swap` example above. To use pattern matching to acquire field addresses, Cyclone provides *reference patterns*: The pattern `*id` matches any location and binds `id` to the location's address.[8] Continuing our example, we could use a reference pattern pointlessly:

```
  let T{<β> .f=g, .env=*arg} = p2;
  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 created existential packages, used assignment to modify memory that has an existential type, and used reference patterns to get aliases of 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{<int>  .f=ignore, .env=-1};
  struct T p2 = T{<int*> .f=assign  .env=ptr};
  let T{<β> .f=g, .env=*arg} = p2;
  p2 = p1;
  g(37,*arg);
}
```

The call `g(37,*arg)` executes `assign` with `37` and `-1`—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

[8] Reference patterns also allow mutating fields of discriminated-union variants, which is why we originally added them to Cyclone.

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. The 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 {<α> void f(int, α); α* env;};` and the pattern `T2{<β> .f=g,.env=arg}`, 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. Third, it is well known that the typing rule for opening an existential package must forbid the introduced type variable from occurring in the type assigned to the term in which the type variable 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.

Multithreading introduces a similar problem: the existential unpack is unsound if the witness can change in between the binding of `g` and `arg`. We must exclude a witness change while binding a package's fields [Grossman 2003b].

4.5 Static Variables

In C and Cyclone, a static variable is essentially a global variable that is only in one function's scope. Although such variables are syntactically declared in a function body, their initializers cannot refer to the enclosing function's arguments. In Cyclone, we similarly disallow a static variable's type to refer to the function's type parameters. In other words, the type must not contain free type variables. It is nonsensical and unsound to do otherwise.

5. LIMITATIONS

Type variables have proven incredibly useful in Cyclone. Providing compile-time equalities of unknown types gives us a safe, flexible way to describe generic code, first-class abstract types, and container types. Kind restrictions that make perfect sense for a C-level language avoid run-time overhead. Nonetheless, any sound, decidable system must prevent certain safe idioms. This section describes the most noticeable limitations, their impact, and how future extensions could relax them.

5.1 Kind Distinction

Cyclone's kind distinction is no more burdensome than in C, where abstract types must occur under pointers and `struct` types cannot be converted to `void*`. Some of the inconvenience is inherent to exposing data representation; it is infeasible to support polymorphism over types of different sizes and calling conventions without imposing run-time cost or duplicating code. Nonetheless, C provides little support for abstract types, so it is fairly easy to be "as good as C."

One cannot, for example, write a function that works for all arrays with elements that are eight bytes. Adding more descriptive kinds is straightforward. For example, A8 could describe all types τ such that sizeof(τ)==8 so long as the types have the same alignment constraints, calling convention, etc. Subkinding would make A8 a subkind of A. However, sizeof(τ) is implementation-dependent, so portable code cannot assume its value. Typed assembly languages can have such kinds because all sizes are known [Morrisett et al. 1999a; Crary 2003].

We could relax the rules about where abstract types appear by duplicating code for every type at which it is instantiated. This approach is closer to C++ templates [Stroustrup 2000]. It is a valuable alternative for widely used, performance-critical libraries, such as hashtables, where a level of indirection can prove costly. However, it is difficult to maintain separate compilation. Polymorphic recursion is also a problem if we cannot bound the amount of generated code. For example, this program needs an amount of code that cannot be bounded at compile-time:

```
struct T<α:A> {α x; α y; }; // not legal Cyclone
void f<α:A>(struct T<α> t) {
  struct T<struct T<α>> bigger = T{.x=t, .y=t};
  if(flip_a_coin()) f(bigger);
}
```

In ongoing work, Cyclone's designers are relaxing the type system to allow struct types where the last field has unknown size. Doing so avoids well-known shortcomings of the C type system, but makes it difficult to compile Cyclone to C while supporting operations such as getting the address of the last field. For now, we relegate this important extension to future work.

5.2 Quantified-Type Formation

Restricting where programmers can introduce type quantifiers (universal quantification only on function types and existential quantification only on struct types as in the Cyclone implementation) is usually not restrictive. To see why, consider this small formal grammar for types that is less restrictive:

$$\tau ::= \alpha \mid \text{int} \mid \tau \to \tau \mid \tau \times \tau \mid \tau * \mid \exists\alpha.\tau \mid \forall\alpha.\tau$$

Types can be type variables, int, function types, pair types (i.e., anonymous struct types), pointer types, existential types, or universal types. Unlike the Cyclone implementation, this grammar does not restrict the form of quantified types. We argue informally why the generality is not very useful:

—$\forall\alpha.\alpha$ should not describe any value; no value should have every type. $\exists\alpha.\alpha$ could describe any value (ignoring kind distinctions), but expressions of this type are unusable. For $\forall\alpha.\beta$ and $\exists\alpha.\beta$, we can just use β.

—For $\forall\alpha.$int and $\exists\alpha.$int, we can just use int.

—Cyclone provides $\forall\alpha.\tau_1 \to \tau_2$. For $\exists\alpha.\tau_1 \to \tau_2$, if α appears in τ_1, expressions of this type are unusable because we cannot call the function. Otherwise, we can just use $\tau_1 \to \exists\alpha.\tau_2$.

—Cyclone provides the analogue of $\exists\alpha.\tau_1 \times \tau_2$. For $\forall\alpha.\tau_1 \times \tau_2$, a similar value of type $(\forall\alpha.\tau_1) \times (\forall\alpha.\tau_2)$ is strictly more useful. Constructing such a similar value is easy because type-checking expressions of type τ_1 (respectively τ_2) does not exploit the type of τ_2 (respectively τ_1). Furthermore, by using a type constructor (e.g., $T = \lambda\alpha.(\tau_1 \times \tau_2)$) one can introduce a type equality and quantify over the constructor's argument type (e.g., the β in $T\texttt{<}\beta\texttt{>}$).

—For $\forall\alpha.(\tau*)$ and $\exists\alpha.(\tau*)$, we can just use $(\forall\alpha.\tau)*$ and $(\exists\alpha.\tau)*$, respectively. Note that $\forall\alpha.(\tau*)$ should not describe pointers to mutable values.

—For $\exists\alpha.\forall\beta.\tau$, if $\exists\alpha.\tau$ is not useful then neither is $\exists\alpha.\forall\beta.\tau$. If $\exists\alpha.\tau$ is useful (i.e., τ is a product type), then $\forall\beta.\tau$ is not useful, so we can replace $\exists\alpha.\forall\beta.\tau$ with $\exists\alpha.\tau$. The argument for $\forall\beta.\exists\alpha.\tau$ is analogous.

However, it would be useful to allow $\forall\alpha.\tau_1 + \tau_2$, where $\tau_1 + \tau_2$ is a (disjoint) *sum type*, especially in conjunction with abstract types. That way, if a value v has a closed type τ_1 and inl injects a value into a sum type, then we could give $\mathsf{inl}(v)$ type $\forall\alpha.\tau_1 + \tau_2$, allowing different instantiations of the quantified type to "share" $\mathsf{inl}(v)$. However, for the reasons in Section 4.3, we cannot give a mutable location the type $\forall\alpha.\tau_1 + \tau_2$.

Finally, types of the form $\exists\alpha.\tau_1 + \tau_2$ are less useful than $(\exists\alpha.\tau_1) + (\exists\alpha.\tau_2)$ for reasons dual to the situation for $\forall\alpha.\tau_1 \times \tau_2$.

5.3 Bounded Quantification

Though not discussed in this article, Cyclone has sound forms of subtyping, such as letting a pointer to a `struct` be subsumed to a pointer to the first field of the `struct`.[9] Therefore, *bounded quantification* would make the type system more expressive by allowing type constraints on polymorphic functions, as in this example:

```
α f(void g(τ), α x : α<τ) {
  g(x);
  return x;
}
```

The constraint $\alpha\texttt{<}\tau$ requires any instantiation of `f`'s type to use a subtype of τ. In the body of `f`, we can soundly assume the constraint holds, so we use subsumption to type-check the function call. Without bounded quantification, the most permissive type for `f` would give `x` and the result the type τ. But then callers of `f` using a strict subtype of τ could not assume that the result of the call had the subtype.

Although Cyclone has constraints for other issues such as memory lifetimes, we have not had enough practical need for bounded quantification to incorporate it. Section 7 discusses some known problems with bounded quantification.

[9] If the `struct` is an existential type, this subsumption is not allowed.

5.4 Uncontrolled Aliasing

The inability of the Cyclone type system to express restrictions on aliases to locations causes Cyclone to forbid some safe programs. For example, given a pointer to a value of type α, it is safe to store a value of type β at the pointed-to location "temporarily," *provided that no code expecting an α reads the location before it again holds an α* (and provided `sizeof(`β`)`$\leq$`sizeof(`α`)`). If no aliases to the location exist, this property is much easier to check statically. As another example, we can allow reference patterns for fields of mutable existential packages, provided no (witness-changing) mutation occurs before the variable bound with the reference pattern is dereferenced.

Recent extensions to Cyclone describe unaliased data [Hicks et al. 2004], but these extensions have been exploited only for memory deallocation, not relaxing restrictions related to type variables.

More generally, systems using linear of affine types offer complementary benefits: they allow programs that "change" the type of data, but they restrict aliasing relationships among variables. Both approaches are extremely useful; this article essentially investigates the expressiveness of a polymorphic type system that does not restrict aliasing.

5.5 Parametricity

The well-known concept of *parametricity* [Strachey 1967; Reynolds 1983; Wadler 1989] ensures the behavior of code cannot depend on the instantiation of type variables. As a simple example, in the polymorphic lambda calculus, a term with the type $\forall\alpha\forall\beta.(\alpha \times \beta) \rightarrow (\beta \times \alpha)$ must behave like the function that given (e_0, e_1) returns (e_1, e_0). However, Pierce and Sangiorgi [2000] presented a very clever trick showing that languages with mutable references (such as ML) can violate parametricity. Unsurprisingly, a variant of the trick works in Cyclone. In previous work [Grossman et al. 2000], we argued that the true source of the trick is aliasing of values at more and less abstract types (e.g., a value available at types α`*` and `int*`). Clients of polymorphic functions may be able to avoid such aliasing, but the Cyclone type system provides no support in checking they have done so. As such, the benefits of strong typing in Cyclone follow primarily from memory safety and abstract interfaces, not parametricity.

5.6 Package-Field Access

Forbidding direct access to existential-package fields is inconvenient. Perhaps a simple flow analysis could infer the unpacking implicit in field access without violating soundness. This convenience is particularly important when porting C code.

5.7 Partial Instantiation

Partial instantiation of type constructors and polymorphic functions is sometimes less convenient than we have suggested. The instantiation is in order, which means the type-constructor and function creator determines what partial applications are allowed. (The same shortcoming exists at the term level in functional languages with currying.) Moreover, the partial applications shown earlier are just shorthand for implicit full applications. But sometimes it is necessary to instantiate a universal type partially and delay the rest of the instantiation. The Cyclone implementation actually supports such a true partial instantiation via special syntax.

6. FORMALISM

To investigate the soundness of the features presented, especially in the presence of the complications described in Section 4.4, we develop a formal abstract machine and a type system for it. This machine defines programs that manipulate a heap of mutable locations holding integers or pointers. The machine gets “stuck” if a program tries to dereference an integer. The type system has universal and existential types (with both solutions from Section 4.4). To keep the model tractable, we omit type constructors and memory management. The theorem in Section 6.4 ensures well-typed programs never lead to stuck machines.

Before proceeding, we emphasize the most novel aspects of the formalism:

(1) Like Cyclone and C, we distinguish left expressions and right expressions. The definitions for these classes of terms are mutually inductive. For simplicity, we treat statements as right expressions and a function implicitly returns the value to which its body evaluates. See the author’s dissertation [Grossman 2003a] for a model with distinct statements including a nonlocal return.

(2) We allow aliasing of mutable fields (e.g., &x.i.j) and assignment to aggregate values (e.g., x.i=e where x.i is itself an aggregate). This feature complicates the rules for accessing, mutating, and type-checking aggregates.

(3) We classify types with kinds B and A. The type system prohibits programs that would need to know the size of a type variable of kind A.

(4) To support both our solutions for mutable existential packages, the syntax distinguishes two styles of existential types. The type system defines the set of “assignable” types to disallow some witness changes. Moreover, the type-safety proof requires the type system to maintain the witness types for packages used in reference patterns. Otherwise, the induction hypothesis would not be strong enough to show that evaluation preserves typing.

Section 6.1, 6.2, and 6.3 present the abstract machine’s syntax, evaluation rules, and type system, respectively. In practice, we need to type-check only source programs, but proving type safety requires extending the type system to type-check program states. Table I summarizes all judgments defined in subsequent sections.

6.1 Syntax

Figure 1 presents the language’s syntax. We model execution with a state comprising a heap for data and an expression for control. We represent heap addresses with variables, so the heap maps variables to values. We write $\cdot$ for the

Table I. Summary of Judgments

$H;e \xrightarrow{\mathrm{l}} H';e'$	One-step left-expression evaluation
$H;e \xrightarrow{\mathrm{r}} H';e'$	One-step right-expression evaluation
$e[\tau/\alpha] \quad \tau'[\tau/\alpha]$	Type-substitution (through expressions and types)
$\mathrm{get}(v_1, p, v_2)$	Path-getting: at path p in v_1 is v_2
$\mathrm{set}(v_1, p, v_2, v_3)$	Path-setting: putting v_2 at path p in v_1 makes v_3
$\Delta \vdash_{\mathrm{k}} \tau : \kappa$	Kinding of types, disallowing types of unknown size
$\Delta \vdash_{\mathrm{ak}} \tau : \kappa$	Kinding of types, allowing types of unknown size
$\Delta \vdash_{\mathrm{asgn}} \tau$	"Assignable" types
$\Delta \vdash_{\mathrm{wf}} \Gamma \quad \vdash_{\mathrm{wf}} \Upsilon \quad \vdash_{\mathrm{wf}} C$	Well-formedness (disallows free type variables)
$C \vdash_{\mathrm{t}} e : \tau$	Typing of expressions
$\vdash_{\mathrm{lval}} e$	"Valid" left expressions
$\Upsilon; xp' \vdash \mathrm{gettype}(\tau_1, p, \tau_2)$	Path-getting: at path p in τ_1 is τ_2, using $\Upsilon(xp'p_1)$ if $p = p_1\mathrm{u}p_2$
$\Upsilon; \Gamma \vdash_{\mathrm{h}} H : \Gamma'$	Typing of heaps
$H \vdash_{\mathrm{refp}} \Upsilon$	"Υ-checking": H has packages with the witness types that Υ claims
$\vdash_{\mathrm{prog}} H;e$	State typing: implies heap- and expression-typing, and Υ-checking

$$
\begin{array}{rrcl}
\text{kinds} & \kappa & ::= & \mathtt{B} \mid \mathtt{A} \\
\text{types} & \tau & ::= & \alpha \mid \mathrm{int} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau* \mid \forall\alpha{:}\kappa.\tau \mid \exists^{\phi}\alpha{:}\kappa.\tau \\
 & \phi & ::= & \delta \mid \& \\
\text{expressions} & e & ::= & xp \mid i \mid e{=}e \mid \&e \mid *e \mid (e,e) \mid e.i \mid \lambda x:\tau.\ e \mid e(e) \\
 & & \mid & e;e \mid \mathsf{if}\ e\ e\ e \mid \mathsf{while}\ e\ e \mid \mathsf{let}\ x = e;\ e \\
 & & \mid & \Lambda\alpha{:}\kappa.e \mid e[\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ e \mid \mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ e \\
 & p & ::= & \cdot \mid ip \mid \mathtt{u}p \\
\text{values} & v & ::= & i \mid \&xp \mid (v,v) \mid \lambda x:\tau.\ e \mid \Lambda\alpha{:}\kappa.e \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \\
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \\
\text{states} & P & ::= & H;e \\
\text{contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
 & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
 & \Upsilon & ::= & \cdot \mid \Upsilon, xp{:}\tau \\
 & C & ::= & \Delta; \Upsilon; \Gamma
\end{array}
$$

Fig. 1. Formal syntax.

empty heap and assume implicit reordering of elements (so heaps act as partial maps).

Expressions include integers (i), function definitions ($\lambda x : \tau_1.\ e$), universal quantification ($\Lambda\alpha{:}\kappa.e$), pointer creations ($\&e$), pointer dereferences ($*e$), pairs ((e_1, e_2)), field accesses ($e.i$), assignments ($e_1 = e_2$), function calls ($e_1(e_2)$), type instantiations ($e[\tau]$), and existential packages ($\mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau$). In this package creation, τ' is the witness type.

Expressions also include statement-like forms for sequential composition ($e_1; e_2$), conditionals ($\mathsf{if}\, e_1\, e_2\, e_3$), and loops ($\mathsf{while}\, e_1\, e_2$). A variable binding ($\mathsf{let}\, x = e_1;\ e_2$) extends the heap with a binding for x that is in scope in e_2. We can assume x is unique because the binding is α-convertible. Because memory management is not our concern, the dynamic semantics never contracts the heap. There are two forms for destructing existential packages: $\mathsf{open}\ e_1\ \mathsf{as}\ \alpha, x;\ e_2$ binds x to a *copy* of the contents of the evaluation of e_1, whereas $\mathsf{open}\ e_1\ \mathsf{as}\ \alpha, *x;\ e_2$ binds x to a *pointer* to the contents of the evaluation of e_1. The latter form corresponds to reference patterns. For simplicity, it produces a pointer to the entire contents, not a particular field. In both forms, x is bound in e_2.

Finally, instead of variables (x), we write variables with *paths* (p), so the expression form is xp. If p is the empty path ($\cdot$), then xp is like a variable x, and we write x as short-hand for $x\cdot$. There is no need for nonempty paths in source programs. Because values may be pairs or packages, we use paths to refer to parts of values. A path is just a sequence of 0, 1, and u. As defined in the next section, 0 and 1 refer to pair components and u refers to the value inside an existential package. 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 valid left expressions are a subset of the valid right expressions. The type system enforces the restriction. The expression under the & operator, the left side of an assignment, and e_1 in open e_1 as $\alpha, *x;\ e_2$ must be valid left expressions.

Types include type variables (α), a base type (int), products ($\tau_1 \times \tau_2$), pointers ($\tau*$), existentials ($\exists^{\phi}\alpha{:}\kappa.\tau$), and universals ($\forall\alpha{:}\kappa.\tau$). Quantified types are equal up to systematic renaming of the bound type variable (α-conversion). Compared to Cyclone, we have replaced `struct` types with "anonymous" product types (pairs) and eliminated user-defined type constructors. Type-variable bindings include an explicit kind, κ. Because aliasing is relevant, all pointers are explicit. In particular, a value of a product type is a record, not a pointer to a record. To distinguish our two approaches to existential types, we annotate $\exists$ with δ (allowing witness changes) or & (allowing aliases at the opened type).

As technical points, we treat the parts of a typing context (Δ, Γ, and Υ) as implicitly reorderable (and as partial maps) where convenient. When we write $\Gamma, x{:}\tau$, we assume $x \notin \mathrm{Dom}(\Gamma)$. We write $\Gamma\Gamma'$ (and similarly for Δ and Υ) for the union of two contexts with disjoint domains, implicitly assuming disjointedness.

Some (less interesting) inference rules for the dynamic and static semantics include multiple conclusions, which is just a convenient abbreviation for multiple rules with the same hypotheses. That is, the notation

$$\frac{\mathcal{P}_1}{\begin{matrix}\mathcal{P}_2\\ \mathcal{P}_3\end{matrix}} \quad \text{is shorthand for} \quad \frac{\mathcal{P}_1}{\mathcal{P}_2} \quad \text{and} \quad \frac{\mathcal{P}_1}{\mathcal{P}_3}.$$

6.2 Dynamic Semantics

Several deterministic relations define the (small-step, operational) dynamic semantics. A program state $H; e$ becomes $H'; e'$ if the rules in Figure 2 establish $H; e \xrightarrow{\mathrm{r}} H'; e'$. The relations $H; e \xrightarrow{\mathrm{r}} H'; e'$ and $H; e \xrightarrow{\mathrm{l}} H'; e'$ are interdependent because left and right expressions can contain each other. The relations in Figure 3 describe how paths direct the access and mutation of values. Type substitution gives operational meaning to $e[\tau]$ and open. Types play no essential run-time role, so we can view substitution as an effectless operation useful for proving type preservation. We now describe the definitions in more detail.

$$\frac{\mathrm{get}(H(x), p, v)}{H; xp \xrightarrow{\mathrm{r}} H; v}\ \mathrm{DR1} \qquad \frac{\mathrm{set}(v', p, v, v'')}{H, x \mapsto v', H'; xp{=}v \xrightarrow{\mathrm{r}} H, x \mapsto v'', H'; v}\ \mathrm{DR2}$$

$$\frac{}{H; {*}\&xp \xrightarrow{\mathrm{r}} H; xp}\ \mathrm{DR3} \qquad \frac{}{H; (v_0, v_1).i \xrightarrow{\mathrm{r}} H; v_i}\ \mathrm{DR4}$$

$$\frac{}{H; (\lambda x : \tau.\ e)(v) \xrightarrow{\mathrm{r}} H; (\mathsf{let}\ x = v;\ e)}\ \mathrm{DR5} \qquad \frac{}{H; (\Lambda\alpha{:}\kappa.e)[\tau] \xrightarrow{\mathrm{r}} H; e[\tau/\alpha]}\ \mathrm{DR6}$$

$$\frac{x \notin \mathrm{Dom}(H)}{H; (\mathsf{let}\ x = v;\ e) \xrightarrow{\mathrm{r}} H, x \mapsto v; e}\ \mathrm{DR7} \qquad \frac{}{H; \mathsf{while}\ e_1\ e_2 \xrightarrow{\mathrm{r}} H; (\mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ 0)}\ \mathrm{DR8}$$

$$\frac{}{H; (v; e) \xrightarrow{\mathrm{r}} H; e}\ \mathrm{DR9} \qquad \frac{}{H; \mathsf{if}\ 0\ e_1\ e_2 \xrightarrow{\mathrm{r}} H; e_2}\ \mathrm{DR10} \qquad \frac{i \neq 0}{H; \mathsf{if}\ i\ e_1\ e_2 \xrightarrow{\mathrm{r}} H; e_1}\ \mathrm{DR11}$$

$$\frac{}{H; (\mathsf{open}\ (\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)\ \mathsf{as}\ \alpha, x;\ e) \xrightarrow{\mathrm{r}} H; (\mathsf{let}\ x = v;\ e[\tau'/\alpha])}\ \mathrm{DR12}$$

$$\frac{\mathrm{get}(H(x), p, \mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)}{H; (\mathsf{open}\ xp\ \mathsf{as}\ \alpha, {*}x';\ e) \xrightarrow{\mathrm{r}} H; (\mathsf{let}\ x' = \&xp\mathsf{u};\ e[\tau'/\alpha])}\ \mathrm{DR13}$$

$$\frac{}{H; (xp).i \xrightarrow{\mathrm{l}} H; xpi}\ \mathrm{DL1} \qquad \frac{}{H; {*}\&xp \xrightarrow{\mathrm{l}} H; xp}\ \mathrm{DL2}$$

$$\frac{H; e \xrightarrow{\mathrm{l}} H'; e'}{\begin{array}{c} H; \&e \xrightarrow{\mathrm{r}} H'; \&e' \\ H; e{=}e_2 \xrightarrow{\mathrm{r}} H'; e'{=}e_2 \\ H; (\mathsf{open}\ e\ \mathsf{as}\ \alpha, {*}x;\ e_2) \xrightarrow{\mathrm{r}} H'; (\mathsf{open}\ e'\ \mathsf{as}\ \alpha, {*}x;\ e_2) \end{array}}\ \mathrm{DR14} \qquad \frac{H; e \xrightarrow{\mathrm{l}} H'; e'}{H; e.i \xrightarrow{\mathrm{l}} H'; e'.i}\ \mathrm{DL3}$$

$$\frac{H; e \xrightarrow{\mathrm{r}} H'; e'}{\begin{array}{cc} H; {*}e \xrightarrow{\mathrm{r}} H'; {*}e' & H; (e, e_2) \xrightarrow{\mathrm{r}} H'; (e', e_2) \\ H; e.i \xrightarrow{\mathrm{r}} H'; e'.i & H; (v, e) \xrightarrow{\mathrm{r}} H'; (v, e') \\ H; xp{=}e \xrightarrow{\mathrm{r}} H'; xp{=}e' & H; e(e_2) \xrightarrow{\mathrm{r}} H'; e'(e_2) \\ H; e[\tau] \xrightarrow{\mathrm{r}} H'; e'[\tau] & H; v(e) \xrightarrow{\mathrm{r}} H'; v(e') \\ H; (e; e_2) \xrightarrow{\mathrm{r}} H'; (e'; e_2) & H; \mathsf{if}\ e\ e_1\ e_2 \xrightarrow{\mathrm{r}} H'; \mathsf{if}\ e'\ e_1\ e_2 \\ \multicolumn{2}{c}{H; \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau \xrightarrow{\mathrm{r}} H'; \mathsf{pack}\ \tau', e'\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau} \\ \multicolumn{2}{c}{H; (\mathsf{let}\ x = e;\ e_2) \xrightarrow{\mathrm{r}} H'; (\mathsf{let}\ x = e';\ e_2)} \\ \multicolumn{2}{c}{H; (\mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ e_2) \xrightarrow{\mathrm{r}} H'; (\mathsf{open}\ e'\ \mathsf{as}\ \alpha, x;\ e_2)} \end{array}}\ \mathrm{DR15} \qquad \frac{H; e \xrightarrow{\mathrm{r}} H'; e'}{H; {*}e \xrightarrow{\mathrm{l}} H'; {*}e'}\ \mathrm{DL4}$$

Fig. 2. Dynamic semantics, expressions.

The get and set relations handle the details of reading and mutating locations (DR1 and DR2). Rules DR3 and DR4 eliminate pointers and pairs, respectively. Rules DR5 eliminates function calls, using let to pass the argument. Rule DR6 uses type substitution for instantiation.

Rule DR7 is the only rule that extends the heap. Because let $x = v;\ e$ is α-convertible, we can assume x does not already name a heap location. Bindings exist forever, so an expression like let $x = v;\ \&x$ is reasonable. Rules DR8–11 are unsurprising rules for reducing loops, sequences, and conditionals. Rule DR12 uses let to simplify the results of opening an existential package. In the result, α is not in scope, so we substitute the package's witness type for α. Rule DR13 also uses let, but it binds the variable to the *address of* the package's contents.

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

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

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

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

Fig. 3. Dynamic semantics, heap objects.

To keep type-checking syntax-directed, we append u to the path. That way, we refer to the package's contents, not the package. The get relation, described below, is used here only to acquire the witness type, which we substitute.

Rules DR14–15 are congruence rules for evaluating parts of larger terms. Putting multiple conclusions in one rule is just for conciseness. Evaluation order is left-to-right. Rule DR14 indicates the left expression positions. The interesting distinction is that in open e_1 as $\alpha, x;\ e_2$, the expression e_1 is a right expression, but in open e_1 as $\alpha, *x;\ e_2$, it is a left expression.

Left expressions evaluate to something of the form xp. We need few rules because the type system restricts the form of left expressions. The only interesting rule is DL1, which appends a field projection to the path. To contrast left expressions and right expressions, compare the results of DL2 and DR3. For left expressions, the result is a terminal form, but for right expressions, rule DR1 applies.

The get relation defines how we use paths to access nested values—as examples, $\mathrm{get}((v_0, v_1), 1, v_1)$ and get(pack τ', v as $\exists^{\&}\alpha{:}\kappa.\tau,\ \mathrm{u},\ v$). That is, we use u to get a package's contents, which we never do if the witness might change. 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, $\mathrm{set}((v_1, ((v_2, v_3), v_4)), 10, (v_5, v_6), (v_1, ((v_5, v_6), v_4)))$.

Type substitution (in types, terms, and contexts) is straightforward. We replace free occurrences of the type variable with the type. Binding occurrences occur in quantified types, open, and polymorphic-function definitions.

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

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

Lines (1)–(5) allocate values in the heap. After line (3), location x_{pkg} contains pack int$*$, $(\&x_{pzero}\cdot, \&x_{zero}\cdot)$ as $\exists^{\phi}\alpha{:}\mathrm{B}.\alpha* \times \alpha$. Line (4) substitutes int$*$ for β and

$$\frac{\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \tau) \quad \Delta \vdash_{\mathrm{k}} \Gamma(x) : \mathtt{A} \quad \vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} xp : \tau}\ \text{SR1} \qquad \frac{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau* \quad \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} *e : \tau}\ \text{SR2}$$

$$\frac{C \vdash_{\mathrm{t}} e : \tau_0 \times \tau_1}{C \vdash_{\mathrm{t}} e.0 : \tau_0}\ \text{SR3} \qquad \frac{C \vdash_{\mathrm{t}} e : \tau_0 \times \tau_1}{C \vdash_{\mathrm{t}} e.1 : \tau_1}\ \text{SR4} \qquad \frac{C \vdash_{\mathrm{t}} e_0 : \tau_0 \quad C \vdash_{\mathrm{t}} e_1 : \tau_1}{C \vdash_{\mathrm{t}} (e_0, e_1) : \tau_0 \times \tau_1}\ \text{SR5} \qquad \frac{\vdash_{\mathrm{wf}} C}{C \vdash_{\mathrm{t}} i : \mathrm{int}}\ \text{SR6}$$

$$\frac{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_1 : \tau \quad \vdash_{\mathrm{lval}} e_1 \quad \Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_2 : \tau \quad \Delta \vdash_{\mathrm{asgn}} \tau}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_1{=}e_2 : \tau}\ \text{SR7} \qquad \frac{C \vdash_{\mathrm{t}} e : \tau \quad \vdash_{\mathrm{lval}} e}{C \vdash_{\mathrm{t}} \&e : \tau*}\ \text{SR8}$$

$$\frac{C \vdash_{\mathrm{t}} e_1 : \tau' \to \tau \quad C \vdash_{\mathrm{t}} e_2 : \tau'}{C \vdash_{\mathrm{t}} e_1(e_2) : \tau}\ \text{SR9} \qquad \frac{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \forall\alpha{:}\kappa.\tau' \quad \Delta \vdash_{\mathrm{ak}} \tau : \kappa}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e[\tau] : \tau'[\tau/\alpha]}\ \text{SR10}$$

$$\frac{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau[\tau'/\alpha] \quad \Delta \vdash_{\mathrm{ak}} \tau' : \kappa \quad \Delta \vdash_{\mathrm{k}} \exists^{\phi}\alpha{:}\kappa.\tau : \mathtt{A}}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau : \exists^{\phi}\alpha{:}\kappa.\tau}\ \text{SR11}$$

$$\frac{\Delta; \Upsilon; \Gamma, x{:}\tau \vdash_{\mathrm{t}} e : \tau' \quad x \notin \mathrm{Dom}(\Gamma)}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \lambda x : \tau.\ e : \tau \to \tau'}\ \text{SR12} \qquad \frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau \quad \vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma \quad \alpha \notin \mathrm{Dom}(\Delta)}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\tau}\ \text{SR13}$$

$$\frac{C \vdash_{\mathrm{t}} e_1 : \tau_1 \quad C \vdash_{\mathrm{t}} e_2 : \tau_2}{C \vdash_{\mathrm{t}} e_1; e_2 : \tau_2}\ \text{SR14} \qquad \frac{C \vdash_{\mathrm{t}} e : \mathrm{int} \quad C \vdash_{\mathrm{t}} e_1 : \tau \quad C \vdash_{\mathrm{t}} e_2 : \tau}{C \vdash_{\mathrm{t}} \mathsf{if}\ e\ e_1\ e_2 : \tau}\ \text{SR15}$$

$$\frac{C \vdash_{\mathrm{t}} e_1 : \mathrm{int} \quad C \vdash_{\mathrm{t}} e_2 : \tau}{C \vdash_{\mathrm{t}} \mathsf{while}\ e_1\ e_2 : \mathrm{int}}\ \text{SR16} \qquad \frac{\Delta; \Upsilon; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e_2 : \tau \quad \Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_1 : \tau' \quad x \notin \mathrm{Dom}(\Gamma)}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{let}\ x = e_1;\ e_2 : \tau}\ \text{SR17}$$

$$\frac{\begin{array}{l}\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_1 : \exists^{\phi}\alpha{:}\kappa.\tau' \\ \Delta, \alpha{:}\kappa; \Upsilon; \Gamma, x{:}\tau' \vdash_{\mathrm{t}} e_2 : \tau \\ \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A} \quad \alpha \notin \mathrm{Dom}(\Delta) \quad x \notin \mathrm{Dom}(\Gamma)\end{array}}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{open}\ e_1\ \mathsf{as}\ \alpha, x;\ e_2 : \tau}\ \text{SR18} \qquad \frac{\begin{array}{l}\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e_1 : \exists^{\&}\alpha{:}\kappa.\tau' \qquad \vdash_{\mathrm{lval}} e_1 \\ \Delta, \alpha{:}\kappa; \Upsilon; \Gamma, x{:}\tau'* \vdash_{\mathrm{t}} e_2 : \tau \\ \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A} \quad \alpha \notin \mathrm{Dom}(\Delta) \quad x \notin \mathrm{Dom}(\Gamma)\end{array}}{\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{open}\ e_1\ \mathsf{as}\ \alpha, *x;\ e_2 : \tau}\ \text{SR19}$$

$$\frac{}{\vdash_{\mathrm{lval}} xp}\ \text{SL1} \qquad \frac{}{\vdash_{\mathrm{lval}} *e}\ \text{SL2} \qquad \frac{\vdash_{\mathrm{lval}} e}{\vdash_{\mathrm{lval}} e.i}\ \text{SL3}$$

Fig. 4. Typing, expressions.

location x_{pr} contains $\&x_{pkg}\mathrm{u}$. After line (6), x_{fst} contains $\&x_{pzero}\cdot$ and x_{pkg} contains $\mathsf{pack}\ \mathrm{int}, (\&x_{zero}, 0)\ \mathsf{as}\ \exists^{\phi}\alpha{:}\mathtt{B}.\alpha* \times \alpha$. Hence line (7) assigns 0 to x_{pzero}, which causes line (8) to be stuck because there is no H, H', and e' for which $H; *0 \xrightarrow{1} H'; e'$.

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.

The type information associated with packages and paths keeps type-checking syntax-directed. We could define an erasure function over heaps that replaces $\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau$ with v and removes u from paths. It should be straightforward to prove that erasure and evaluation commute (treating open like let).

6.3 Static Semantics

Because program execution begins with an empty heap, a source program is just an expression e. To allow e, we require $\cdot; \cdot; \cdot \vdash_{\mathrm{t}} e : \tau$ (for some type τ), using the rules in Figure 4. These rules ensure terms are never used with inappropriate operations and never refer to undefined variables.

$$\frac{}{\Upsilon; xp \vdash \mathrm{gettype}(\tau, \cdot, \tau)} \qquad \frac{\Upsilon; xp\mathrm{u} \vdash \mathrm{gettype}(\tau'[\Upsilon(xp)/\alpha], p', \tau) \quad \cdot \vdash_{\mathrm{ak}} \Upsilon(xp) : \kappa}{\Upsilon; xp \vdash \mathrm{gettype}(\exists^{\&}\alpha{:}\kappa.\tau', \mathrm{u}p', \tau)}$$

$$\frac{\Upsilon; xp0 \vdash \mathrm{gettype}(\tau_0, p', \tau)}{\Upsilon; xp \vdash \mathrm{gettype}(\tau_0 \times \tau_1, 0p', \tau)} \qquad \frac{\Upsilon; xp1 \vdash \mathrm{gettype}(\tau_1, p', \tau)}{\Upsilon; xp \vdash \mathrm{gettype}(\tau_0 \times \tau_1, 1p', \tau)}$$

Fig. 5. Typing, heap objects.

$$\frac{}{\Delta \vdash_{\mathrm{k}} \mathrm{int} : \mathtt{B}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{B} \vdash_{\mathrm{k}} \alpha : \mathtt{B}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{k}} \alpha* : \mathtt{B}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{B}}{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\Delta \vdash_{\mathrm{k}} \tau* : \mathtt{B}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau_0 : \mathtt{A} \quad \Delta \vdash_{\mathrm{k}} \tau_1 : \mathtt{A}}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \tau_0 \times \tau_1 : \mathtt{A} \\ \Delta \vdash_{\mathrm{k}} \tau_0 \to \tau_1 : \mathtt{A}\end{array}} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau : \mathtt{A} \quad \alpha \notin \mathrm{Dom}(\Delta)}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \forall\alpha{:}\kappa.\tau : \mathtt{A} \\ \Delta \vdash_{\mathrm{k}} \exists^{\phi}\alpha{:}\kappa.\tau : \mathtt{A}\end{array}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \kappa}{\Delta \vdash_{\mathrm{ak}} \tau : \kappa} \qquad \frac{}{\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{ak}} \alpha : \mathtt{A}}$$

$$\frac{}{\Delta \vdash_{\mathrm{asgn}} \mathrm{int}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{B} \vdash_{\mathrm{asgn}} \alpha} \qquad \frac{}{\Delta \vdash_{\mathrm{asgn}} \tau*} \qquad \frac{\Delta \vdash_{\mathrm{asgn}} \tau_0 \quad \Delta \vdash_{\mathrm{asgn}} \tau_1}{\begin{array}{c}\Delta \vdash_{\mathrm{asgn}} \tau_0 \times \tau_1 \\ \Delta \vdash_{\mathrm{asgn}} \tau_0 \to \tau_1\end{array}} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{asgn}} \tau}{\begin{array}{c}\Delta \vdash_{\mathrm{asgn}} \forall\alpha{:}\kappa.\tau \\ \Delta \vdash_{\mathrm{asgn}} \exists^{\delta}\alpha{:}\kappa.\tau\end{array}}$$

$$\frac{}{\Delta \vdash_{\mathrm{wf}} \cdot} \qquad \frac{\Delta \vdash_{\mathrm{wf}} \Gamma \quad \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\Delta \vdash_{\mathrm{wf}} \Gamma, x{:}\tau} \qquad \frac{}{\vdash_{\mathrm{wf}} \cdot} \qquad \frac{\vdash_{\mathrm{wf}} \Upsilon \quad \cdot \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\vdash_{\mathrm{wf}} \Upsilon, xp{:}\tau} \qquad \frac{\Delta \vdash_{\mathrm{wf}} \Gamma \quad \vdash_{\mathrm{wf}} \Upsilon}{\vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma}$$

Fig. 6. Kinding and context well-formedness.

The strangest part of the typing judgment for expressions ($\vdash_{\mathrm{t}}$ in Figure 4) is Υ, which is irrelevant in source programs. As described below, it captures the invariant that packages used in terms of the form open e as $\alpha, *x;\ s$ are never mutated. In particular, Υ holds the witness types of packages that have been opened with open e as $\alpha, *x;\ s$. The $\vdash_{\mathrm{t}}$ rules use $\vdash_{\mathrm{lval}}$ to restrict the form of left expressions and the gettype relation (Figure 5) to type-check paths. The latter is the static analogue of the get relation.

Type-checking also restricts what types appear where, using the judgments in Figure 6. The $\vdash_{\mathrm{ak}}$ and $\vdash_{\mathrm{wf}}$ judgments primarily ensure type variables are in scope. The $\vdash_{\mathrm{k}}$ kinding judgment forbids abstract types except under pointer types. It is reassuring that $\vdash_{\mathrm{ak}}$ is easy to define in terms of $\vdash_{\mathrm{k}}$. The restriction $\vdash_{\mathrm{k}}$ prevents manipulating terms of unknown size, although formalizing this restriction is unnecessary: The dynamic semantics for the formal machine could accommodate such terms. The $\vdash_{\mathrm{asgn}}$ judgment describes types of mutable expressions.

We do not need the judgments in Figure 7 for source programs. They describe the invariant used to prove type safety. For a valid source program e, $\vdash_{\mathrm{prog}} \cdot; e$.

We now describe the judgments in more detail.

If $\Delta \vdash_{\mathrm{k}} \tau : \kappa$, then given the type variables in Δ, type τ has kind κ and its size is known. To prevent types of unknown size, we cannot derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{k}} \alpha : \kappa$, but we can derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{k}} \alpha* : \mathtt{B}$. For simplicity, we assume function types have known size, unlike in Cyclone. We can imagine implementing all function definitions with values of the same size (e.g., pointers to code), so this simplification is justifiable. Some types are not subject to the known-size

$$\frac{}{\Upsilon;\Gamma \vdash_{\mathrm{h}} \cdot : \cdot} \qquad \frac{\Upsilon;\Gamma \vdash_{\mathrm{h}} H : \Gamma' \quad \cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau}{\Upsilon;\Gamma \vdash_{\mathrm{h}} H, x \mapsto v : \Gamma', x{:}\tau}$$

$$\frac{}{H \vdash_{\mathrm{refp}} \cdot} \qquad \frac{H \vdash_{\mathrm{refp}} \Upsilon \quad \mathrm{get}(H(x), p, \textsf{pack}\ \tau', v\ \textsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau)}{H \vdash_{\mathrm{refp}} \Upsilon, xp{:}\tau'}$$

$$\frac{\Upsilon;\Gamma \vdash_{\mathrm{h}} H : \Gamma \quad H \vdash_{\mathrm{refp}} \Upsilon \quad \cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e : \tau}{\vdash_{\mathrm{prog}} H; e}$$

Fig. 7. Typing, states.

restriction, such as τ in $e[\tau]$. But we still require $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$; we can derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{ak}} \alpha : \kappa$. The types for which $\Delta \vdash_{\mathrm{asgn}} \tau$ have known size *and* any types of the form $\exists^{\&}\alpha{:}\kappa'.\tau$ occur under pointers.

We cannot give quantified types kind B, but we argued earlier that doing so is not useful. We exploit this fact in the rules for $\vdash_{\mathrm{asgn}}$: it is too lenient to allow $\Delta, \alpha{:}\mathtt{B} \vdash_{\mathrm{asgn}} \alpha$ if we might instantiate α with a type of the form $\exists^{\&}\alpha{:}\kappa'.\tau$. A more complicated kind system could distinguish assignable box kinds and unassignable box kinds (the former being a subkind of the latter).

Well-formed contexts (the $\vdash_{\mathrm{wf}}$ judgments) have only closed types of known size. Because Υ is used only to describe heaps, no Δ is necessary for $\vdash_{\mathrm{wf}} \Upsilon$.

We now describe the type-checking rules for expressions. To type-check xp, SR1 uses the gettype relation to derive a type from the type of x and the form of p. We can use u to acquire the contents of an existential package only if the package has a type of the form $\exists^{\&}\alpha{:}\kappa.\tau$. Such types are not assignable, so no mutation can interfere. Furthermore, to use u, the path to the package must be in Υ. We use Υ to remember the witness types of all packages that have been unpacked with an expression of the form open e as $\alpha, *x;\ s$. These witnesses cannot change, so it is sound to use $\Upsilon(xp)$. Before a program executes, no packages have been unpacked, so Υ is $\cdot$. In fact, there is no need for gettype at all in source programs because we can forbid nonempty paths. SR2 prevents dereferencing a pointer to a value of unknown size. SR3–6 hold no surprises. SR7 ensures that e_1 is a valid left expression and its type is assignable. SR8 requires a left expression of the address-of operator's subexpression. SR9 is the normal rule for function call. SR10–13 are conventional for quantified types and functions. We use the $\vdash_{\mathrm{ak}}$ judgment because types for instantiations and witnesses can have unknown size. Unlike C and Cyclone, we do not require that functions are closed (modulo global variables) nor do we require that they appear at top-level.

The typing rules for the statement-like forms are unsurprising, particularly SR14–SR17. Rules SR18–SR19 allow the two forms of existential unpacks. As expected, they extend Δ and Γ and the type of the bound term variable depends on the form of the unpack (τ' in SR18 and $\tau'*$ in SR19). The reuse of α in the type of e is not a restriction because existential types α-convert. The e in SR19 must be a valid left expression, so we requires $\vdash_{\mathrm{lval}} e_1$. The type of e in SR19 cannot have the form $\exists^{\delta}\alpha{:}\kappa.\tau$; this is the essence of the restriction on such types. Finally, the kinding assumption in SR18 and SR19 is a technical point to ensure that α is not free in τ, which is always possible by α-conversion of the open expression.

As in C and Cyclone, the legal left expressions (defined with $\vdash_{\text{lval}}$) are locations, dereferences, and fields of left expressions.

The judgment $\vdash_{\text{prog}} H; e$ defines the invariant that establishes type safety.

First, the heap must type-check without any free variables or any type variables. It is also this judgment that would fail to be preserved under evaluation if $e[\tau]$ could be a left expression: If we heap variable x had type $\forall\alpha.\kappa\tau_1$, then $x[\tau] = v$ could change the type of x in the heap. Note that $\Gamma; \Upsilon \vdash_{\text{h}} H : \Gamma$ allows mutually recursive functions in the heap; other cyclic data is impossible without recursive types.

Second, if $\Upsilon(xp) = \tau$, then the value in the location that xp describes has to be an existential package with witness type τ, and the package's type must indicate that the witness will not change. If the witness could change, then the mapping from xp to τ in Υ would not be preserved under evaluation, which in turn would break type preservation for expressions of the form xpu. So the Preservation Lemma in the Appendix establishes that no mapping in Υ ever changes.

Third, e must type-check under the assumptions Γ and Υ that describe the heap.

6.4 Type Safety

The appendix proves this result:

Definition 1. State $H; e$ is *stuck* if e is not a value and there are no H' and e' such that $H; e \overset{\text{r}}{\rightarrow} H'; e'$.

THEOREM TYPE SAFETY. *If* $\cdot; \cdot; \cdot \vdash_{\text{t}} e : \tau$ *and* $\cdot; e \overset{\text{r}}{\rightarrow}^{*} H'; e'$ *(where* $\overset{\text{r}}{\rightarrow}^{*}$ *is the reflexive, transitive closure of* $\overset{\text{r}}{\rightarrow}$*), then* $H'; e'$ *is not stuck.*

Informally, well-typed programs can continue evaluating until they terminate (though they may not terminate).

7. RELATED WORK

This section discussed the most closely related work on quantified types, low-level type systems, and safe C-like languages.

7.1 Universal Quantification

The seminal theoretical foundation for quantified types in programming languages is the polymorphic lambda calculus, also called System F, which Girard [1989] and Reynolds [1974] invented independently. Many general-purpose programming languages, most notably Standard ML [Milner et al. 1997], O'Caml [Chailloux et al. 2000; Leroy 2002a], Haskell [Jones and Hughes 1999], and GJ [Bracha et al. 1998], use quantified types and type constructors to allow code reuse.

Higher-level languages generally do not restrict the types that a type variable can represent. A polymorphic function can be instantiated at any type, including records and floating-point types. Simpler implementations add a level of indirection for all records and floating-point numbers to avoid code duplication.

Sophisticated analyses and compiler intermediate languages can avoid some unnecessary levels of indirection [Morrisett 1995; Tarditi 1996; Leroy 1992, 1997; Wells et al. 2002]. In the extreme, ML's lack of polymorphic recursion lets whole-program compilers *monomorphize* the code, essentially duplicating polymorphic functions for each type at which they are instantiated [Cejtin et al. 2000; Benton et al. 1998]. The amount of generated code appears tolerable in practice. C++ [Stroustrup 2000] defines template instantiation in terms of code duplication, making template functions closer to advanced macros than parametric polymorphism.

An example of a simple compromise is the current O'Caml implementation [Leroy 2002b]: records and arrays of floating-point numbers do not use indirection for the numbers. Polymorphic code accessing an array must check at run-time whether the array holds floating-point numbers or not, so run-time type information is necessary.

Without first-class polymorphism or polymorphic recursion, ML and Haskell enjoy full type inference: programs never need explicit type information. Type inference is undecidable if we add first-class polymorphism or polymorphic recursion [Wells 1999; Henglein 1993; Kfoury et al. 1993]. Haskell 98 [Jones and Hughes 1999] includes polymorphic recursion, but requires explicit types for functions that use it. Because these languages encourage using many functions, conventional wisdom considers Cyclone's approach of requiring explicit types for all function definitions intolerable. However, room for compromise between inference and more powerful type systems exists, as proposals for ML extensions and additions to Haskell implementations demonstrate [Garrigue and Rémy 1999; Botlan and Rémy 2003; Pierce and Turner 1998; The Hugs 98 User Manual 2002; The GHC Team 2003].

Section 5 described how bounded quantification for types could increase Cyclone's expressiveness. The type theory for bounded quantification has received considerable attention, particularly because of its role in encoding some object-oriented idioms [Bruce et al. 1999]. An important negative result concerns bounded quantification's interaction with subtyping: it is sound to consider $\forall\alpha \leq \tau_1.\tau_2$ a subtype of $\forall\alpha \leq \tau_3.\tau_4$ if τ_3 is a subtype of τ_1 and τ_2 is a subtype of τ_4. However, together with other conventional subtyping rules, this rule for subtyping universal types makes the subtyping question (given two types, is one a subtype of the other) undecidable [Pierce 1991]. A common compromise is to require equal bounds ($\tau_1 = \tau_3$ in our example) [Cardelli and Wegner 1985]. Another possibility is to require explicit subtyping proofs (or hints about proofs) in source programs.

The problem with polymorphic references discussed in Section 4.4 has received much attention from the ML community [Tofte 1990; Wright and Felleisen 1994; Harper 1994]. A commitment to full type inference and an advanced module system with abstract types complicate the problem. So-called "weak type variable" solutions, which make a kind distinction with respect to mutation, have fallen out of favor. Instead, a simple "value restriction" suffices. Essentially, a binding cannot receive a universal type unless it is initialized with a syntactic value, such as a variable (which is immutable) or a function. This solution interacts well with type inference and appears tolerable in practice. In Cyclone, default mutability and more explicit typing makes the solution of forbidding type instantiation in left expressions seem natural.

7.2 Existential Quantification

Explicit existential types have not been used as much in programming languages. Mitchell and Plotkin's [1988] seminal work showed how constructs for abstract types, such as the `rep` types in CLU clusters [Liskov et al. 1984] and the `abstype` declarations in Standard ML [Milner et al. 1997] are really existential types. Encodings of closures [Minamide et al. 1996] and objects [Bruce et al. 1999] using existential types suggest that the lack of explicit existential types in many languages is in some sense an issue of terminology. Current Haskell implementations [The Hugs 98 User Manual 2002; The GHC Team 2003] include existential types for "first-class" values, as suggested by Läufer [1996]. In all the above work, existential packages are immutable, so the problem from Section 4.4 is irrelevant.

Other lower-level typed languages have included existential types, but have not encountered the same unsoundness problem. For example, Typed Assembly Language [Morrisett et al. 1999b] does not have a way to create an alias of an opened type, as with Cyclone's reference patterns. 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 [Xi 2000], a C-like language with compile-time reasoning about integer values, also does not have aliases at the opened type. Roughly, `int` is short-hand for $\exists\alpha$:Int.int(α) and uses of `int` values implicitly include the necessary open expressions. Such a use *copies* the value, so aliasing is not a problem. It appears that witness types can change because mutating a heap-allocated `int` would change its witness.

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. Walker and Morrisett [2000] exploited this invariant to define open such that it does not introduce any new bindings. Instead, it mutates the location holding the package to hold the package's contents. Without run-time type information, such an open has no actual effect. The Vault system [DeLine and Fähndrich 2001] 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 and pack terms and how to rewrite terms using the bindings that open statements introduce.

7.3 Interacting with Types of Unknown Size

The Typed Assembly Language implementation [Morrisett et al. 1999a] for the IA-32 architecture has a more powerful kind system than Cyclone, though the details are not widely known. For each number i, there is a kind `M`i describing types of memory objects consuming i bytes. These kinds are subkinds of `M`, which corresponds to kind `A` in Cyclone. At the assembly level, padding and alignment are explicit, so giving types these more descriptive kinds is more appropriate. However, the fine granularity of assembly-language instructions make it difficult for the type system to allow safe use of an abstract value. For example, given a pointer to a value of type α of kind M12, we might like to push a copy of the pointed to value onto the stack. Doing so requires adjusting the stack pointer by 12 bytes and executing multiple move instructions for the parts of the abstract value. It is unclear if the details for allowing such operations were ever implemented.

The GHC Haskell implementation [The GHC Team 2003] provides alternative forms of floating-point numbers and records that do not have a level of indirection. Their uses are more restricted than in Cyclone. Not only do values of these types essentially have kind A in a language without type variables of kind A, but unboxed records can appear only in certain syntactic positions. Nonetheless, these extensions give enough control over data representation to improve performance for certain applications.

7.4 Safe C-Level Languages

There has been remarkably little work on quantified types for C-like languages. Smith and Volpano [1996, 1998] described an integration of universal types with C. Their formal development had some similarities with our work, but they did not consider `struct` types. Therefore, they had no need for existential types. Similarly, Cforall [Ditchfield 1994] supports polymorphic functions, but its aim is to remain closer to C (still permitting unsafe programs).

As discussed above, Vault [DeLine and Fähndrich 2001] has quantified types, but the emphasis is on controlling aliasing to permit type-state changes. Although Vault is intended for low-level systems where manual resource management is important, its approach to data representation is like that of high-level languages (roughly, everything is a pointer).

Other approaches to achieving C-level safety have not used quantified types. By implementing C as though it was a high-level language (e.g., by tagging each data object with its type), it is possible to detect illegal type casts (or subsequent dereferences) when they occur [Austin et al. 1994; Necula et al. 2002; Loginov et al. 2001]. This technique makes it difficult to interoperate with legacy systems, but it can be done [Jones and Kelly 1997; Condit et al. 2003]. For improved performance, static analysis can eliminate the need for most tags, but polymorphic libraries still suffer in performance. A more draconian solution can simply forbid casts between different pointer types, essentially prohibiting generic code [Kowshik et al. 2002], which may be reasonable in some application domains. Constraint-based inference techniques can analyze casts in C programs to determine whether the source and destination types are proper subtypes [Chandra and Reps 1999]. Cyclone takes a more explicit approach, requiring some explicit annotations (other than just `void*`) and rejecting programs not in the type system.

The author's dissertation [Grossman 2003a] has a much more extensive discussion of techniques and tools designed to find safety violations in C programs.

8. CONCLUSIONS

Adapting quantified types to the C level has proven extremely useful, but it has required careful design to restrict types of unknown size, prevent polymorphic references, and allow mutable data hiding constructs (via existential types). A formal language that combines C's distinction between left expressions and right expressions with quantified types has increased our confidence that Cyclone is sound. While any language designer mixing mutation and polymorphism should proceed with utmost caution, this work should provide important warnings while still indicating that the result is worth the effort.

APPENDIX

This appendix proves the theorem of Section 6.4. The proof follows the syntactic approach that Wright and Felleisen [1994] advocate. The key lemmas are

- —*Preservation*. If $\vdash_{\text{prog}} H; e$ and $H; e \xrightarrow{\text{r}} H'; e'$, then $\vdash_{\text{prog}} H'; e'$.
- —*Progress*. If $\vdash_{\text{prog}} H; e$, then e is a value or there exists H' and e' such that $H; e \xrightarrow{\text{r}} H'; e'$.

Given these lemmas (which we strengthen to prove them inductively due to left expressions), the type-safety proof is straightforward: By induction on the number of steps n taken to reach $H'; e'$, we know $\vdash_{\text{prog}} H'; e'$. (For $n = 0$, we can prove $\vdash_{\text{prog}} \cdot; e$ given the theorem's assumptions with $\Gamma = \cdot$ and $\Upsilon = \cdot$. For $n > 0$, induction and preservation suffice.) Hence progress ensures $H'; e'$ is not stuck.

Proving these lemmas requires several auxiliary lemmas. We state the lemmas and prove them "bottom-up" (presenting and proving lemmas before using them) after giving a "top-down" overview of the proof.

Preservation follows from the Preservation Lemma (terms can type-check after taking a step) and progress follows from the Progress Lemma. The Substitution Lemmas provide the usual results that appropriate type substitutions preserve the necessary properties of terms (and types contained in them), which we need for the cases of the Preservation Lemma that employ substitution. The Canonical Forms Lemma provides the usual arguments for the Progress Lemma when we must determine the form of a value given its type.

Because the judgments for terms rely on judgments for heap objects (namely get, set, and gettype), the proofs of Preservation and Progress require corresponding lemmas for heap objects. The Heap-Object Safety Lemmas fill this need. Lemmas 1 and 2 are obvious facts. Lemma 3 amounts to preservation and progress for the get relation (informally, if gettype indicates a value of some type is at some path, then get produces a value of the type), as well as progress for the set relation (informally, given a legal path, we can change what value is at the end of it). We prove these results together because the proofs require the same reasoning about paths. Lemma 5 amounts to preservation for the set relation. The interesting part is showing that the $\vdash_{\text{asgn}}$ judgment preserves the correctness of the Υ in the context, which means no witnesses for &-style packages changed. Given set(v_1, p, v_2, v_1'), Lemma 5 proves by induction the rather obvious fact that the parts of v_1' that were in v_1 (i.e., the parts not at some path beginning with p) are compatible with Υ. Lemma 4 provides the result for the part of v_1' that is v_2 (i.e., the parts at some path beginning with p). Reference patterns significantly complicate these lemmas.

The Path Extension Lemmas let us add path elements on the right of paths. We must do so, for example, to prove case DL1 of Term Preservation. The proofs require induction because the heap-object judgments destruct paths from the left.

The remaining lemmas provide more technical results that the aforementioned lemmas need. The Typing Well-Formedness Lemmas let us conclude types and contexts are well-formed given typing derivations, which helps satisfy the assumptions of other lemmas. It is uninteresting because we can add more hypotheses to the static semantics until the lemmas hold. The Commuting Substitutions Lemma provides a needed fact for the cases of the proof of Substitution Lemma 8 that have a second type substitution. Safety proofs for polymorphic languages invariably need a Commuting Substitutions Lemma, but it is rarely stated explicitly. We need the Useless Substitutions Lemma only because we use variables for heap locations. The heap does not have free type variables, so type substitution does not change the Γ and Υ that describe it. Finally, the weakening lemmas are conventional devices used to argue that unchanged subterms (e.g., e_1 when (e_0, e_1) becomes (e_0', e_1)) have the same properties in extended contexts (e.g., in the context of a larger heap).

LEMMA WEAKENING. *Suppose* $\vdash_{\text{wf}} \Delta; \Upsilon\Upsilon'; \Gamma\Gamma'$.

(1) *If* $\Delta \vdash_{\text{k}} \tau : \kappa$*, then* $\Delta\Delta' \vdash_{\text{k}} \tau : \kappa$*. If* $\Delta \vdash_{\text{asgn}} \tau\kappa$*, then* $\Delta\Delta' \vdash_{\text{asgn}} \tau\kappa$*.*
(2) *If* $\vdash_{\text{wf}} \Upsilon$*, then* $\Delta \vdash_{\text{k}} \Upsilon(xp) : \mathtt{A}$.
(3) *If* $\Upsilon; xp \vdash \text{gettype}(\tau, p', \tau')$*, then* $\Upsilon\Upsilon'; xp \vdash \text{gettype}(\tau, p', \tau')$.
(4) *If* $\Delta; \Upsilon; \Gamma \vdash_{\text{t}} e : \tau$*, then* $\Delta; \Upsilon\Upsilon'; \Gamma\Gamma' \vdash_{\text{t}} e : \tau$*.*
(5) *If* $\Upsilon; \Gamma \vdash_{\text{h}} H : \Gamma''$*, then* $\Upsilon\Upsilon'; \Gamma\Gamma' \vdash_{\text{h}} H : \Gamma''$*.*
(6) *If* $H \vdash_{\text{refp}} \Upsilon$*, then* $HH' \vdash_{\text{refp}} \Upsilon$*.*

PROOF

(1) By induction on the assumed derivations.
(2) By induction on the assumed derivation.
(3) By induction on the assumed gettype derivation: $xp \in \text{Dom}(\Upsilon)$ implies $(\Upsilon\Upsilon')(xp) = \Upsilon(xp)$.
(4) By induction on the assumed typing derivation: cases SR12 and SR17–19 can use α-conversion to ensure that $x \notin \text{Dom}(\Gamma\Gamma')$. Case SR1 follows from the part (1) because $x \in \text{Dom}(\Gamma)$ implies $(\Gamma\Gamma')(x) = \Gamma(x)$.
(5) By induction on the heap-typing derivation, using Weakening Lemma 4.
(6) By induction on the assumed derivation: $x \in \text{Dom}(H)$ implies $(HH')(x) = H(x)$. □

LEMMA USELESS SUBSTITUTIONS. *Suppose* $\alpha \notin \text{Dom}(\Delta)$.

(1) *If* $\Delta \vdash_{\text{k}} \tau' : \kappa$*, then* $\tau'[\tau/\alpha] = \tau'$.
(2) *If* $\Delta \vdash_{\text{wf}} \Gamma$*, then* $\Gamma[\tau/\alpha] = \Gamma$.
(3) *If* $\vdash_{\text{wf}} \Upsilon$*, then* $(\Upsilon(xp))[\tau/\alpha] = \Upsilon(xp)$.

Proof. Each part is by induction on the assumed derivation; parts (2) and (3) use part (1). □

Lemma Commuting Substitutions.
If β is not free in τ_2, then $\tau_0[\tau_1/\beta][\tau_2/\alpha] = \tau_0[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$.

Proof. By induction on the structure of τ_0: if $\tau_0 = \alpha$, both substitutions produce τ_2, using the assumption that β is not free in τ_2. If $\tau_0 = \beta$, both substitutions produce $\tau_1[\tau_2/\alpha]$. If τ_0 is some other type variable or int, both substitutions are useless. All other cases follow by induction and the definition of substitution. □

Lemma Substitution. *Suppose $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$.*

(1) *If $\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau' : \kappa'$, then $\Delta \vdash_{\mathrm{k}} \tau'[\tau/\alpha] : \kappa'$.*
(2) *If $\Delta, \alpha{:}\kappa \vdash_{\mathrm{ak}} \tau' : \kappa'$, then $\Delta \vdash_{\mathrm{ak}} \tau'[\tau/\alpha] : \kappa'$.*
(3) *If $\Delta, \alpha{:}\kappa \vdash_{\mathrm{asgn}} \tau'$, then $\Delta \vdash_{\mathrm{asgn}} \tau'[\tau/\alpha]$.*
(4) *If $\Delta, \alpha{:}\kappa \vdash_{\mathrm{wf}} \Gamma$, then $\Delta \vdash_{\mathrm{wf}} \Gamma[\tau/\alpha]$.*
(5) *If $\vdash_{\mathrm{wf}} \Delta, \alpha{:}\kappa; \Upsilon; \Gamma$, then $\vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma[\tau/\alpha]$.*
(6) *If $\vdash_{\mathrm{wf}} \Upsilon$ and $\Upsilon; xp \vdash$ gettype(τ_1, p', τ_2), then $\Upsilon; xp \vdash$ gettype($\tau_1[\tau/\alpha], p', \tau_2[\tau/\alpha]$).*[10]
(7) *If $\vdash_{\mathrm{lval}} e$, then $\vdash_{\mathrm{lval}} e[\tau/\alpha]$.*
(8) *If $\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau'$, then $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\mathrm{t}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.*

Proof

(1) By induction on the assumed derivation: the nonaxiom cases are by induction. The case for $\tau' =$ int is trivial. The case where τ' is a type variable is trivial unless $\tau' = \alpha$. In that case, $\Delta(\alpha) = \mathtt{B}$, so inverting $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{B}$, as desired. Similarly, the case where τ' has the form $\beta*$ is trivial unless $\beta = \alpha$. In that case, if τ is some type variable α' where $\Delta(\alpha') = \mathtt{A}$, then $\Delta \vdash_{\mathrm{k}} \alpha'* : \mathtt{B}$ as desired. Else inverting $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\mathrm{k}} \tau : \kappa$, so $\Delta \vdash_{\mathrm{k}} \tau* : \mathtt{B}$ (using the introduction rule for pointer types and possibly the subsumption rule).
(2) By cases on the assumed derivation, using the previous lemma
(3) By induction on the assumed derivation: the nonaxiom cases are by induction. The cases for int and pointer types are trivial. The case where τ' is a type variable is trivial unless $\tau' = \alpha$. In that case, $\kappa = \mathtt{B}$, so inverting $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{B}$, as desired.
(4) By induction on the assumed derivation, using Substitution Lemma 1.
(5) Corollary to Substitution Lemma 4.
(6) By induction on the assumed derivation: case $p' = \cdot$ is trivial. Cases where p' is some $0p''$ or $1p''$ are by induction. The remaining case is a derivation

[10] This lemma is somewhat unnecessary: a state reached from a source program with nonempty paths can type-check without using the gettype judgment on open types. Put another way, SR1 could require $\Gamma(x)$ to be closed unless $p = \cdot$. Rather than prove this, we just include the lemma.

of the form:

$$\frac{\Upsilon; xp\mathrm{u} \vdash \mathrm{gettype}(\tau_0[\Upsilon(xp)/\beta], p'', \tau_2)}{\Upsilon; xp \vdash \mathrm{gettype}(\exists^{\&}\beta{:}\kappa'.\tau_0, \mathrm{u}p'', \tau_2)}.$$

So by induction, $\Upsilon; xp\mathrm{u} \vdash \mathrm{gettype}(\tau_0[\Upsilon(xp)/\beta][\tau/\alpha], p'', \tau_2[\tau/\alpha])$. By α-conversion we assume β is not free in τ (and $\beta \neq \alpha$), so the Commuting Substitutions Lemma ensures $\Upsilon; xp\mathrm{u} \vdash \mathrm{gettype}(\tau_0[\tau/\alpha][\Upsilon(xp)[\tau/\alpha]/\beta], p''$, $\tau_2[\tau/\alpha])$. Useless Substitution Lemma 3 ensures $\Upsilon(xp)[\tau/\alpha] = \Upsilon(xp)$, so $\Upsilon; xp\mathrm{u} \vdash \mathrm{gettype}(\tau_0[\tau/\alpha][\Upsilon(xp)/\beta], p'', \tau_2[\tau/\alpha])$, from which we can derive $\Upsilon; xp \vdash \mathrm{gettype}(\exists^{\&}\beta{:}\kappa'.\tau_0[\tau/\alpha], p'', \tau_2[\tau/\alpha])$, as desired.

(7) By induction on the assumed derivation

(8) By induction on the assumed derivation, proceeding by cases on the last rule in the derivation: in each case, we satisfy the hypotheses of the rule after substitution and then use the rule to derive the desired result. So for most cases, we explain just how to conclude the necessary hypotheses.

SR1. Left, middle, and right hypotheses follow from Substitution Lemmas 6, 1, and 5, respectively.

SR2. Left hypothesis follows from induction. Right hypothesis follows from Substitution Lemma 1.

SR3–5. Hypotheses follow from induction.

SR6. Hypothesis follows from Substitution Lemma 5.

SR7. First and third hypotheses follow from induction. Second hypothesis follows from Substitution Lemma 7. Fourth hypothesis follows from Substitution Lemma 3.

SR8. Left and right hypotheses follow from induction and Substitution Lemma 7, respectively.

SR9. Hypotheses follow from induction.

SR10. We have a derivation of the form

$$\frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \forall\beta{:}\kappa'.\tau_1 \quad \Delta, \alpha{:}\kappa \vdash_{\mathrm{ak}} \tau_0 : \kappa'}{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} e[\tau_0] : \tau_1[\tau_0/\beta]}.$$

The left hypothesis and induction ensure $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \forall\beta{:}\kappa'.\tau_1[\tau/\alpha]$. The right hypothesis and Substitution Lemma 2 provide $\Delta \vdash_{\mathrm{ak}} \tau_0[\tau/\alpha] : \kappa'$. So we can derive $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e[\tau/\alpha][\tau_0[\tau/\alpha]] : \tau_1[\tau/\alpha][\tau_0[\tau/\alpha]/\beta]$. The Commuting Substitutions Lemma ensures the type is what we want.

SR11. We have a derivation of the form

$$\frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau_1[\tau_0/\beta] \quad \Delta, \alpha{:}\kappa \vdash_{\mathrm{ak}} \tau_0 : \kappa' \quad \Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \exists^{\phi}\beta{:}\kappa'.\tau_1 : \mathtt{A}}{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau_0, e\ \mathsf{as}\ \exists^{\phi}\beta{:}\kappa'.\tau_1 : \exists^{\phi}\beta{:}\kappa'.\tau_1}.$$

The left hypothesis and induction ensure $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\mathrm{t}} e[\tau/\alpha] : \tau_1[\tau_0/\beta][\tau/\alpha]$, which by the Commuting Substitutions Lemma ensures $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\mathrm{t}} e[\tau/\alpha] : \tau_1[\tau/\alpha][\tau_0[\tau/\alpha]/\beta]$. The middle hypothesis and Substitution Lemma 2 ensure $\Delta \vdash_{\mathrm{ak}} \tau_0[\tau/\alpha] : \kappa'$. The right hypothesis and Substitution Lemma 1 ensure $\Delta \vdash_{\mathrm{k}} \exists^{\phi}\beta{:}\kappa'.\tau_1[\tau/\alpha] : \mathtt{A}$. So we can derive the desired result: $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau_0[\tau/\alpha], e[\tau/\alpha]\ \mathsf{as}\ \exists^{\phi}\beta{:}\kappa'.\tau_1[\tau/\alpha] : \exists^{\phi}\beta{:}\kappa'.\tau_1[\tau/\alpha]$.

SR12. Hypothesis follows from induction.

SR13. Left hypothesis follows from induction (using implicit context reordering). The well-formedness hypothesis follows from Substitution Lemma 5.

SR14–17. Hypotheses follow from induction.

SR18–19. In both cases, Substitution Lemma 1 provides the kinding hypothesis and induction (and context reordering) provides the typing hypotheses. SR19 also uses Substitution Lemma 7. □

Lemma Typing Well-Formedness

(1) *If* $\vdash_{\mathrm{wf}} \Upsilon$, $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$*, and* $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}$*, then* $\Delta \vdash_{\mathrm{k}} \tau' : \mathtt{A}$.

(2) *If* $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{t}} e : \tau$*, then* $\vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma$ *and* $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}$.

Proof

(1) By induction on the gettype derivation: the case where $p' = \cdot$ is trivial. The cases where p' starts with 0 or 1 are by induction and inversion of the kinding derivation. In the remaining case, induction applies by inverting the kinding derivation (to get $\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau_0 : \mathtt{A}$ where $\tau = \exists^{\&}\alpha{:}\kappa.\tau_0$), inverting the gettype derivation (to ensure $\cdot \vdash_{\mathrm{k}} \Upsilon(xp) : \kappa$), Weakening Lemma 2 (to get $\Delta \vdash_{\mathrm{k}} \Upsilon(xp) : \mathtt{A}$), and Substitution Lemma 1 (to get $\Delta \vdash_{\mathrm{k}} \tau_0[\Upsilon(xp)/\alpha] : \mathtt{A}$).

(2) By induction on the assumed derivation: most cases follow trivially from an explicit hypothesis or from induction and the definition of $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}$. Case SR1 uses the first lemma. Case SR10 uses Substitution Lemma 1. Case SR12 uses the definition of $\vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma$ to determine the function-argument type has kind $\mathtt{A}$. As usual with existential types, explicit hypotheses in SR18 and SR19 are necessary to avoid a type variable escaping. □

Lemma Canonical Forms. *Suppose* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v : \tau$.

—*If* $\tau = \mathrm{int}$*, then* $v = i$ *for some* i.

—*If* $\tau = \tau_0 \times \tau_1$*, then* $v = (v_0, v_1)$ *for some* v_0 *and* v_1.

—*If* $\tau = \tau_0 \to \tau_1$*, then* $v = \lambda x : \tau_0.\ s$ *for some* x *and* s.

—*If* $\tau = \tau'*$*, then* $v = \&xp$ *for some* x *and* p.

—*If* $\tau = \forall\alpha{:}\kappa.\tau'$*, then* $v = \Lambda\alpha{:}\kappa.e$ *for some* e.

—*If* $\tau = \exists^{\delta}\alpha{:}\kappa.\tau'$*, then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\delta}\alpha{:}\kappa.\tau'$ *for some* τ'' *and* v'.

—*If* $\tau = \exists^{\&}\alpha{:}\kappa.\tau'$*, then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'$ *for some* τ'' *and* v'.

Proof. By inspection of the rules for $\vdash_{\mathrm{t}}$ and the form of values □

Lemma Path Extension

(1) *Suppose* $\mathrm{get}(v, p, v')$.
If v' *has the form* (v_0, v_1)*, then* $\mathrm{get}(v, p0, v_0)$ *and* $\mathrm{get}(v, p1, v_1)$*,*
else we cannot derive $\mathrm{get}(v, pip', v'')$ *for any* i*,* p'*, and* v''.
If v' *has the form* $\mathsf{pack}\ \tau', v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau$*, then* $\mathrm{get}(v, p\mathrm{u}, v_0)$*,*
else we cannot derive $\mathrm{get}(v, p\mathrm{u}p', v'')$ *for any* p' *and* v''.

(2) *Suppose* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$.
If $\tau' = \tau_0 \times \tau_1$*, then* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'0, \tau_0)$ *and* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'1, \tau_1)$.
If $\tau' = \exists^{\&}\alpha{:}\kappa.\tau_0$ *and* $\cdot \vdash_{\mathrm{ak}} \Upsilon(xp) : \kappa$*, then* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'\mathrm{u}, \tau_0[\Upsilon(xp)/\alpha])$.

PROOF

(1) By induction on the length of p: if $p = \cdot$, then $v = v'$ and the result follows from inspection of the get relation (because $\cdot p_1 = p_1$ for all p_1). For longer p, we proceed by cases on the leftmost element of p. In each case, inversion of the $\mathrm{get}(v, p, v')$ derivation and induction suffice.

(2) By induction on the length of p': if $p' = \cdot$, then $\tau = \tau'$ and the result follows from inspection of the gettype relation (because $\cdot p_1 = p_1$ for all p_1). For longer p', we proceed by cases on the leftmost element of p'. In each case, inversion of the $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$ derivation and induction hypothesis suffice. □

LEMMA HEAP-OBJECT SAFETY

(1) *There is at most one* v_2 *such that* $\mathrm{get}(v_1, p, v_2)$.

(2) *If* $\mathrm{get}(v_0, p_1, v_1)$ *and* $\mathrm{get}(v_0, p_1 p_2, v_2)$*, then* $\mathrm{get}(v_1, p_2, v_2)$.

(3) *Suppose* $H \vdash_{\mathrm{refp}} \Upsilon$*,* $\Upsilon; \Gamma \vdash_{\mathrm{h}} H : \Gamma$*, and* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1 : \tau_1$.
If $\mathrm{get}(H(x), p_1, v_1)$ *and* $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$*, then*
—*There exists a* v_2 *such that* $\mathrm{get}(H(x), p_1 p_2, v_2)$*. Also,* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2 : \tau_2$.
—*For all* v_2'*, there exists a* v_1' *such that* $\mathrm{set}(v_1, p_2, v_2', v_1')$.
Corollary: If $H \vdash_{\mathrm{refp}} \Upsilon$*,* $\Upsilon; \Gamma \vdash_{\mathrm{h}} H : \Gamma$*, and* $\Upsilon; x\cdot \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$*, then the conclusions hold with* $p_1 = \cdot$ *and* $v_1 = H(x)$.

(4) *Suppose in addition to the previous lemma's assumptions,* $\cdot \vdash_{\mathrm{asgn}} \tau_2$*. Then for all* p'*,* $xp_1 p_2 p' \notin \mathrm{Dom}(\Upsilon)$.

(5) *Suppose in addition to the previous lemma's assumptions,* $\mathrm{set}(v_1, p_2, v_2', v_1')$ *and* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2' : \tau_2$*. Then* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1' : \tau_1$ *and if* $xp_1 p' \in \mathrm{Dom}(\Upsilon)$*, there are* v''*,* τ''*,* α*, and* κ *such that* $\mathrm{get}(v_1', p', \mathsf{pack}\ \Upsilon(xp_1 p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$.
Corollary: If $H \vdash_{\mathrm{refp}} \Upsilon$*,* $\Upsilon; \Gamma \vdash_{\mathrm{h}} H : \Gamma$*,* $\Upsilon; x\cdot \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$*,* $\cdot \vdash_{\mathrm{asgn}} \tau_2$*,* $\mathrm{set}(H(x), p_2, v_2', v_1')$*, and* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2' : \tau_2$*, then the conclusions hold with* $p_1 = \cdot$ *and* $v_1 = H(x)$.

PROOF

(1) By induction on the length of p

(2) By induction on the length of p_1

(3) By induction on the length of p_2: If $p_2 = \cdot$, the gettype relation ensures $\tau_1 = \tau_2$ and the get relation ensures $\mathrm{get}(H(x), p_1\cdot, v_1)$. So letting $v_2 = v_1$, the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1 : \tau_1$ means $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2 : \tau_2$. We can trivially derive $\mathrm{set}(v_1, \cdot, v_2', v_2')$. For longer paths, we proceed by cases on the leftmost element:
$p_2 = 0p_3$: Inverting the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, 0p_3, \tau_2)$ provides $\underline{\Upsilon; xp_1 0 \vdash \mathrm{gettype}(\tau_{10}, p_3, \tau_2)}$ where $\tau_1 = \tau_{10} \times \tau_{11}$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1 : \tau_{10} \times \tau_{11}$ provides $v_1 = (v_{10}, v_{11})$ and $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_{10} : \tau_{10}}$. Applying Path Extension Lemma 1 to the assumption $\mathrm{get}(H(x), p_1, (v_{10}, v_{11}))$

provides $\underline{\mathrm{get}(H(x), p_1 0, v_{10})}$. So induction applies to the underlined results, using $p_1 0$ for p_1, v_{10} for v_1, τ_{10} for τ_1, p_3 for p_2, and τ_2 for τ_2.

Therefore, there exists a v_2 such that $\mathrm{get}(H(x), p_1 0 p_3, v_2)$ and $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2 : \tau_2$, as desired. Moreover, for all v'_2 there exists a v'_{10} such that $\mathrm{set}(v_{10}, p_3, v'_2, v'_{10})$. So we can derive $\mathrm{set}((v_{10}, v_{11}), 0p_3, v'_2, (v'_{10}, v_{11}))$, which satisfies the desired result (letting $v'_1 = (v'_{10}, v_{11})$).

$p_2 = 1p_3$: Analogous to the previous case.

$p_2 = \mathrm{u}p_3$: Inverting the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, \mathrm{u}p_3, \tau_2)$ provides $\underline{\Upsilon; xp_1\mathrm{u} \vdash \mathrm{gettype}(\tau_3[\Upsilon(xp_1)/\alpha], p_3, \tau_2)}$ where $\tau_1 = \exists^{\&}\alpha{:}\kappa.\tau_3$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1 : \exists^{\&}\alpha{:}\kappa.\tau_3$ provides $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_3 : \tau_3[\tau_4/\alpha]}$ where $v_1 = \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$. Applying Path Extension Lemma 1 to the assumption $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$ provides $\underline{\mathrm{get}(H(x), p_1\mathrm{u}, v_3)}$. From $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Heap-Object Safety Lemma 1, and $H \vdash_{\mathrm{refp}} \Upsilon$, we know $\tau_4 = \Upsilon(xp_1)$. So induction applies to the underlined results, using $p_1\mathrm{u}$ for p_1, v_3 for v_1, $\tau_3[\Upsilon(xp_1)/\alpha]$ for τ_1, p_3 for p_2, and τ_2 for τ_2.

Therefore, there exists a v_2 such that $\mathrm{get}(H(x), p_1\mathrm{u}p_3, v_2)$ and $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2 : \tau_2$, as desired. Moreover, for all v'_2 there exists a v'_3 such that $\mathrm{set}(v_3, p_3, v'_2, v'_3)$. So we can derive $\mathrm{set}(\mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3, \mathrm{u}p_3, v'_2,$ $\mathsf{pack}\ \tau_4, v'_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$, which satisfies the desired result (letting $v'_1 = \mathsf{pack}\ \tau_4, v'_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$).

The corollary holds because $\mathrm{get}(H(x), \cdot, H(x))$ and $\Upsilon; \Gamma \vdash_{\mathrm{h}} H : \Gamma$ ensures $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} H(x) : \tau_1$.

(4) Heap-Object Safety Lemmas 1 and 3 ensure there is exactly one v_2 such that $\mathrm{get}(H(x), p_1 p_2, v_2)$. Furthermore, $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_2 : \tau_2$. We proceed by induction on the structure of τ_2.

If $\tau_2 = \mathrm{int}$, the Canonical Forms Lemma ensures $v_2 = i$ for some i. Hence Path Extension Lemma 1 ensures we cannot derive $\mathrm{get}(H(x), p_1 p_2 p', v'')$ unless $p' = \cdot$ (and therefore $v'' = i$). So $\mathrm{get}(H(x), p_1 p_2 p', \mathsf{pack}\ \tau_0, v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'_0)$ is impossible, but it is necessary for $xp_1 p_2 p' \in \mathrm{Dom}(\Upsilon)$.

The cases for $\tau_2 = \tau_3{*}$, $\tau_2 = \tau_3 \to \tau_4$, $\tau_2 = \exists^{\delta}\alpha{:}\kappa.\tau_3$, and $\tau_2 = \forall\alpha{:}\kappa.\tau_3$ are analogous to the case for int, replacing i with a different form of value.

If $\tau_2 = \alpha$ or $\tau_2 = \exists^{\&}\alpha{:}\kappa.\tau_3$, the lemma holds vacuously because we cannot derive $\cdot \vdash_{\mathrm{asgn}} \tau_2$.

If $\tau_2 = \tau_3 \times \tau_4$, the Canonical Forms Lemma ensures $v_2 = (v_3, v_4)$. Hence Path Extension Lemma 1 ensures we can derive $\mathrm{get}(H(x), p_1 p_2 p', v'')$ only if $p' = \cdot$, $p' = 0p''$, or $p' = 1p''$. If $p' = \cdot$, then $\mathrm{get}(H(x), p_1 p_2 p',$ $\mathsf{pack}\ \tau_0, v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'_0)$ is impossible, but it is necessary for $xp_1 p_2 p' \in \mathrm{Dom}(\Upsilon)$. If $p' = 0p''$, applying Path Extension Lemma 2 to the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$ provides $\underline{\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_2 0, \tau_3)}$. Inverting the assumption $\cdot \vdash_{\mathrm{asgn}} \tau_3 \times \tau_4$ provides $\underline{\cdot \vdash_{\mathrm{asgn}} \tau_3}$. With the underlined results and the assumptions $\underline{\mathrm{get}(H(x), p_1, v_1)}$ and $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_1 : \tau_1}$, induction applies (using $p_2 0$ for p_2 and τ_3 for τ_2), so $xp_1 p_2 0 p'' \notin \mathrm{Dom}(\Upsilon)$, as desired. The argument for $p' = 1p''$ is analogous.

(5) By induction on the length of p_2: If $p_2 = \cdot$, the set relation ensures $v'_1 = v'_2$. and the gettype relation ensures $\tau_2 = \tau_1$. Hence the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_2 : \tau_2$ means $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_1 : \tau_1$. Heap-Object Safety Lemma 4 ensures $xp_1 \cdot p' \notin \mathrm{Dom}(\Upsilon)$, so the second conclusion holds vacuously. For longer paths, we proceed by cases on the leftmost element:

$p_2 = 0p_3$. Inverting the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, 0p_3, \tau_2)$ ensures $\underline{\Upsilon; xp_1 0 \vdash \mathrm{gettype}(\tau_{10}, p_3, \tau_2)}$ where $\tau_1 = \tau_{10} \times \tau_{11}$.

Inverting the assumption $\mathrm{set}(v_1, 0p_3, v'_2, v'_1)$ ensures $\underline{\mathrm{set}(v_{10}, p_3, v'_2, v'_{10})}$ where $v_1 = (v_{10}, v_{11})$ and $v'_1 = (v'_{10}, v_{11})$.

Path Extension Lemma 1 and the assumption $\mathrm{get}(H(x), p_1, (v_{10}, v_{11}))$ ensure $\underline{\mathrm{get}(H(x), p_1 0, v_{10})}$.

Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} (v_{10}, v_{11}) : \tau_{10} \times \tau_{11}$ ensures $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_{10} : \tau_{10}}$.

With the underlined results and the assumptions $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_2 : \tau_2}$ and $\underline{\cdot \vdash_{\mathrm{asgn}} \tau_2}$, induction applies (using $p_1 0$ for p_1, p_3 for p_2, v_{10} for v_1, τ_{10} for τ_1, τ_2 for τ_2, v'_2 for v'_2, and v'_{10} for v'_1).

Hence $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_{10} : \tau_{10}$ and if $xp_1 0p'' \in \mathrm{Dom}(\Upsilon)$, then $\mathrm{get}(v'_{10}, p'', \mathsf{pack}\ \Upsilon(xp_1 0p''), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$.

So we can derive $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} (v'_{10}, v_{11}) : \tau_{10} \times \tau_{11}$, as desired. If $xp_1 p' \in \mathrm{Dom}(\Upsilon)$, then $H \vdash_{\mathrm{refp}} \Upsilon$ provides $\mathrm{get}(H(x), p_1 p', \mathsf{pack}\ \Upsilon(xp_1 p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. Because $\mathrm{get}(H(x), p_1, (v_{10}, v_{11}))$, Path Extension Lemma 1 ensures that p' has the form $\cdot$, $0p''$, or $1p''$. Heap-Object Safety Lemma 1 precludes $p' = \cdot$. If $p' = 0p''$, the induction provides the result we need. If $p' = 1p''$, applying Heap-Object Safety Lemma 2 provides $\mathrm{get}((v_{10}, v_{11}), 1p'', \mathsf{pack}\ \Upsilon(xp_1 p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$, which by inversion provides $\mathrm{get}(v_{11}, p'', \mathsf{pack}\ \Upsilon(xp_1 p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. So we can derive $\mathrm{get}((v'_{10}, v_{11}), 1p'', \mathsf{pack}\ \Upsilon(xp_1 p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$, as desired.

$p_2 = 1p_3$. Analogous to the previous case.

$p_2 = \mathrm{u}p_3$. Inverting the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, \mathrm{u}p_3, \tau_2)$ ensures $\underline{\Upsilon; xp_1\mathrm{u} \vdash \mathrm{gettype}(\tau_3[\Upsilon(xp_1)/\alpha], p_3, \tau_2)}$ where $\tau_1 = \exists^{\&}\alpha{:}\kappa.\tau_3$. Inverting the assumption $\mathrm{set}(v_1, \mathrm{u}p_3, v'_2, v'_1)$ ensures $\underline{\mathrm{set}(v_3, p_3, v'_2, v'_3)}$ where $v_1 = \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$ and $v'_1 = \mathsf{pack}\ \tau_4, v'_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$.

Path Extension Lemma 1 and the assumption $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$ ensure $\underline{\mathrm{get}(H(x), p_1\mathrm{u}, v_3)}$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3 : \exists^{\&}\alpha{:}\kappa.\tau_3$ ensures $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v_3 : \tau_3[\tau_4/\alpha]}$. From $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Heap-Object Safety Lemma 1, and $H \vdash_{\mathrm{refp}} \Upsilon$, we know $\tau_4 = \Upsilon(xp_1)$.

With the underlined results and the assumptions $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_2 : \tau_2}$ and $\underline{\cdot \vdash_{\mathrm{asgn}} \tau_2}$, induction applies (using $p_1\mathrm{u}$ for p_1, p_3 for p_2, v_3 for v_1, $\tau_3[\Upsilon(xp_1)/\alpha]$ for τ_1, τ_2 for τ_2, v'_2 for v'_2, and v'_3 for v'_1).

Hence $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{t}} v'_3 : \tau_3[\Upsilon(xp_1)/\alpha]$ and if $xp_1\mathrm{u}p'' \in \mathrm{Dom}(\Upsilon)$, then $\mathrm{get}(v'_3, p'', \mathsf{pack}\ \Upsilon(xp_1\mathrm{u}p''), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. So we can derive

$\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau_4, v_3'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3 : \exists^{\&}\alpha{:}\kappa.\tau_3$, as desired. If $xp_1p' \in \mathrm{Dom}(\Upsilon)$, then $H \vdash_{\mathrm{refp}} \Upsilon$ provides $\mathrm{get}(H(x), p_1p', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. Because $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Path Extension Lemma 1 ensures that p' has the form $\cdot$ or $\mathtt{u}p''$. The case $p' = \cdot$ is trivial because $\mathrm{get}(v_1', \cdot, v_1')$ (the witness type did not change, so it is not a problem for $xp_1 \in \mathrm{Dom}(\Upsilon)$). The case $\mathtt{u}p''$ follows from induction. The corollary holds because $\mathrm{get}(H(x), \cdot, H(x))$ and $\Upsilon;\Gamma \vdash_{\mathrm{h}} H : \Gamma$ ensures $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} H(x) : \tau_1$. □

Definition Extension. Γ_2 (or Υ_2) *extends* Γ_1 (or Υ_1) if there exists a Γ_3 (or Υ_3) such that $\Gamma_2 = \Gamma_1\Gamma_3$ (or $\Upsilon_2 = \Upsilon_1\Upsilon_3$).

LEMMA PRESERVATION. *Suppose* $\Upsilon;\Gamma \vdash_{\mathrm{h}} H : \Gamma$, $H \vdash_{\mathrm{refp}} \Upsilon$, *and* $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e : \tau$.

—*If* $\vdash_{\mathrm{lval}} e$ *and* $H;e \xrightarrow{\mathrm{l}} H';e'$, *then there exist* Γ' *and* Υ' *extending* Γ *and* Υ *such that* $\Upsilon';\Gamma' \vdash_{\mathrm{h}} H' : \Gamma'$, $H' \vdash_{\mathrm{refp}} \Upsilon'$, $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e' : \tau$, *and* $\vdash_{\mathrm{lval}} e'$.

—*If* $H;e \xrightarrow{\mathrm{r}} H';e'$, *then there exist* Γ' *and* Υ' *extending* Γ *and* Υ *such that* $\Upsilon';\Gamma' \vdash_{\mathrm{h}} H' : \Gamma'$, $H' \vdash_{\mathrm{refp}} \Upsilon'$, *and* $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e' : \tau$.

PROOF. The proof is by simultaneous induction on the assumed derivations that the term can take a (left or right) step, proceeding by cases on the last rule used. Except where noted, we use $H' = H$, $\Gamma' = \Gamma$, and $\Upsilon' = \Upsilon$.

DL1. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xp.i : \tau$ provides $\Upsilon;x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \tau_0 \times \tau_1)$ (where $\tau = \tau_i$), $\cdot \vdash_{\mathrm{k}} \Gamma(x) : \mathtt{A}$, and $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$. Thus Path Extension Lemma 2 provides $\Upsilon;x\cdot \vdash \mathrm{gettype}(\Gamma(x), pi, \tau_i)$, so SR1 ensures $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xpi : \tau_i$. SL1 provides $\vdash_{\mathrm{lval}} xpi$.

DL2. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} {*}\&xp : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xp : \tau$ and SL1 provides $\vdash_{\mathrm{lval}} xp$.

DL3. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e_1.i : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e_1 : \tau_0 \times \tau_1$ (where $\tau = \tau_i$). So induction applies to $H;e_1 \xrightarrow{\mathrm{l}} H';e_1'$, ensuring $\Upsilon';\Gamma' \vdash_{\mathrm{h}} H' : \Gamma'$, $H' \vdash_{\mathrm{refp}} \Upsilon'$, $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e_1' : \tau$, and $\vdash_{\mathrm{lval}} e_1'$. So SR3–4 and SL3 ensure $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e' : \tau$, and $\vdash_{\mathrm{lval}} e'$ where $e' = e_1'.i$.

DL4. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} {*}e_1 : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e_1 : \tau{*}$ and $\cdot \vdash_{\mathrm{k}} e_1 : \mathtt{A}$. So induction applies to $H;e_1 \xrightarrow{\mathrm{r}} H';e_1'$, ensuring $\Upsilon';\Gamma' \vdash_{\mathrm{h}} H' : \Gamma'$, $H' \vdash_{\mathrm{refp}} \Upsilon'$, and $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e_1' : \tau$. So SR2 and SL2 ensure $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} e' : \tau$, and $\vdash_{\mathrm{lval}} e'$ where $e' = {*}e_1'$.

DR1. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xp : \tau$ ensures $\Upsilon;x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \tau)$. So Heap-Object Safety Lemmas 1 and 3 provide $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau$.

DR2. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xp{=}v : \tau$ provides $\Upsilon;x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \tau)$, $\cdot \vdash_{\mathrm{asgn}} \tau$, and $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau$. Heap-Object Safety Lemma 5 provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v' : \Gamma(x)$ and all $xp' \in \mathrm{Dom}(\Upsilon)$ are still correct in the sense of $H \vdash_{\mathrm{refp}} \Upsilon$. So letting $H' = H[x \mapsto v'']$, $\Gamma' = \Gamma$, and $\Upsilon' = \Upsilon$, we can derive the needed results.

DR3. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} {*}\&xp : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} xp : \tau$.

DR4. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} (v_0, v_1).i : \tau_i$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v_i : \tau_i$.

DR5. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} (\lambda x : \tau_1.\, e_1)(v) : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau_1$ and $\cdot;\Upsilon;\Gamma, x{:}\tau_1 \vdash_{\mathrm{t}} e_1 : \tau$. So SR17 lets us derive $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} (\mathsf{let}\ x = v;\ s) : \tau$.

DR6. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} (\Lambda\alpha{:}\kappa.e)[\tau_1] : \tau_2[\tau_1/\alpha]$ provides $\alpha{:}\kappa;\Upsilon;\Gamma \vdash_{\mathrm{t}} e : \tau_2$ and $\cdot \vdash_{\mathrm{ak}} \tau_1 : \kappa$. So Substitution Lemma 8 provides $\cdot;\Upsilon;\Gamma[\tau_1/\alpha] \vdash_{\mathrm{t}} e[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$. Because $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$, Useless Substitution Lemma 2 ensures $\Gamma[\tau_1/\alpha] = \Gamma$. So $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$.

DR7. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{let}\ x = v;\ e_1 : \tau$ provides $\underline{\cdot;\Upsilon;\Gamma, x{:}\tau' \vdash_{\mathrm{t}} e_1 : \tau}$ and $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau'$. Let $H' = H, x \mapsto v$, $\Gamma' = \underline{\Gamma, x{:}\tau'}$, and $\Upsilon' = \Upsilon$. The Typing Well-Formedness Lemma provides $\cdot \vdash_{\mathrm{k}} \tau' : \mathtt{A}$ and $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$, so $\vdash_{\mathrm{wf}} \cdot;\Upsilon';\Gamma'$. So Weakening Lemma 5 provides $\Upsilon';\Gamma' \vdash_{\mathrm{h}} H : \Gamma$, so $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau'$ provides $\underline{\Upsilon';\Gamma' \vdash_{\mathrm{h}} H' : \Gamma'}$. Weakening Lemma 6 provides $\underline{H' \vdash_{\mathrm{refp}} \Upsilon'}$. The underlined results are our obligations.

DR8. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{while}\ e_1\ e_2 : \mathrm{int}$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e_2 : \tau$ and $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e_1 : \mathrm{int}$. Typing Well-Formedness Lemma provides $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$. With these results, SR6 and SR14–16 let us derive $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{if}\ e_1\ (e_2; \mathsf{while}\ e_1\ e_2)\ 0 : \mathrm{int}$.

DR9–11. In each case, inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e : \tau$ provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} e' : \tau$.

DR12. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{open}\ (\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)\ \mathsf{as}\ \alpha, x;\ e_1 : \tau''$ provides $\alpha{:}\kappa;\Upsilon;\Gamma, x{:}\tau \vdash_{\mathrm{t}} e_1 : \tau''$, $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau[\tau'/\alpha]$, $\cdot \vdash_{\mathrm{ak}} \tau' : \kappa$, and $\cdot \vdash_{\mathrm{k}} \tau'' : \mathtt{A}$. So Substitution Lemma 8 provides $\cdot;\Upsilon;\Gamma[\tau'/\alpha], x{:}\tau[\tau'/\alpha] \vdash_{\mathrm{t}} e_1[\tau'/\alpha] : \tau''[\tau'/\alpha]$. Applying Useless Substitution Lemmas 1 and 2 (using Typing Well-Formedness Lemma for $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$) provides $\cdot;\Upsilon;\Gamma, x{:}\tau[\tau'/\alpha] \vdash_{\mathrm{t}} e_1[\tau'/\alpha] : \tau''$. So SR17 lets us derive $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{let}\ x = v;\ e_1[\tau'/\alpha] : \tau''$, as desired.

DR13. Inverting $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{open}\ xp\ \mathsf{as}\ \alpha, {*x'};\ e_1 : \tau$ provides $\alpha{:}\kappa;\Upsilon;\Gamma, x'{:}\tau'{*} \vdash_{\mathrm{t}} e_1 : \tau$, $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau')$, and $\cdot \vdash_{\mathrm{k}} \tau : \mathtt{A}$. So inverting DR13 provides $\mathrm{get}(H(x), p, \mathsf{pack}\ \tau'', v\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau')$. So Heap-Object Safety Lemmas 1 and 3 provide $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} \mathsf{pack}\ \tau'', v\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau' : \exists^{\&}\alpha{:}\kappa.\tau'$. Inverting this result provides $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \tau'[\tau''/\alpha]$ and $\cdot \vdash_{\mathrm{ak}} \tau'' : \kappa$. So Substitution Lemma 8 provides $\cdot;\Upsilon;(\Gamma[\tau''/\alpha]), x'{:}(\tau'{*})[\tau''/\alpha] \vdash_{\mathrm{t}} e_1[\tau''/\alpha] : \tau[\tau''/\alpha]$. Applying Useless Substitution Lemmas 1 and 2 (using Typing Well-Formedness Lemma for $\vdash_{\mathrm{wf}} \cdot;\Upsilon;\Gamma$) provides $\cdot;\Upsilon;\Gamma, x{:}(\tau'{*})[\tau''/\alpha] \vdash_{\mathrm{t}} e_1[\tau''/\alpha] : \tau$.

Let $\Gamma' = \Gamma$ and $H' = H$. If $xp \in \mathrm{Dom}(\Upsilon)$, let $\Upsilon' = \Upsilon$, else let $\Upsilon' = \Upsilon, xp{:}\tau''$. If $xp \in \mathrm{Dom}(\Upsilon)$, then the hypothesis of DR13, $H \vdash_{\mathrm{refp}} \Upsilon$, and Heap-Object Safety Lemma 1 ensure $\Upsilon(xp) = \tau''$. Applying Weakening Lemma 3 to $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau')$ ensures $\Upsilon'; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau')$. Applying Path Extension Lemma 2 to this result ensures $\Upsilon'; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p\mathrm{u}, \tau'[\tau''/\alpha])$. So SR1, SR8, and SL1 let us derive $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} \&xp\mathrm{u} : \tau'[\tau''/\alpha]$. Applying Weakening Lemma 4 to $\cdot;\Upsilon;\Gamma, x{:}(\tau'{*})[\tau''/\alpha] \vdash_{\mathrm{t}} e_1[\tau''/\alpha] : \tau$ ensures $\cdot;\Upsilon';\Gamma', x{:}(\tau'{*})[\tau''/\alpha] \vdash_{\mathrm{t}} e_1[\tau''/\alpha] : \tau$. So SR17 lets us derive the desired result: $\cdot;\Upsilon';\Gamma' \vdash_{\mathrm{t}} \mathsf{let}\ x' = \&xp\mathrm{u};\ e_1[\tau''/\alpha] : \tau$.

DR14–15. The argument for each conclusion is analogous, so we describe them generally. With the hypothesis $H; e_1 \xrightarrow{\text{r}} H'; e'$ (or $H; e_1 \xrightarrow{\text{l}} H'; e'_1$) and inversion of the typing derivation for e, we know e_1 is well typed (and for DR14 also $\vdash_{\text{lval}} e_1$). So induction ensures there are extensions Γ' and Υ' such that $\Upsilon'; \Gamma' \vdash_{\text{h}} H' : \Gamma'$, $H \vdash_{\text{refp}} \Upsilon'$, and e'_1 has the same type under $\cdot; \Upsilon'; \Gamma'$ that e_1 has under $\cdot; \Upsilon; \Gamma$ (and for DR14 also $\vdash_{\text{lval}} e'_1$). So using Weakening Lemma 4 to type-check unchanged subexpressions of e under Υ' and Γ', we can derive $\cdot; \Upsilon'; \Gamma' \vdash_{\text{r}} e' : \tau$. For binding forms (let and open), α-conversion (of x) ensures $\Gamma', x{:}\tau'$ makes sense. □

Lemma Progress. *Suppose* $\Upsilon; \Gamma \vdash_{\text{h}} H : \Gamma$, $H \vdash_{\text{refp}} \Upsilon$, *and* $\cdot; \Upsilon; \Gamma \vdash_{\text{r}} e : \tau$.

—*If* $\vdash_{\text{lval}} e$, *then* e *is some* xp *or there exists an* H' *and* e' *such that* $H; e \xrightarrow{\text{l}} H'; e'$.
—*Either* e *is some value* v *or there exists an* H' *and* e' *such that* $H; e \xrightarrow{\text{r}} H'; e'$.

Proof. The proof is by induction on the assumed typing derivation, proceeding by cases on the last rule used. Except where noted, the first conclusion holds vacuously (i.e., $\nvdash_{\text{lval}} e$).

SR1. The first conclusion holds because $e = xp$. For the second, Heap-Object Safety Lemma 3 provides $\text{get}(H(x), p, v)$ for some v, so DR1 applies.

SR2. By induction, if e' (where $e = {*}e'$) is not a value, it can take a step, so we use DL4 for the first conclusion and DR15 for the second. Else e' is a value with a pointer type, so the Canonical Forms Lemma provides it has the form $\&xp$. So DL2 applies for the first conclusion and DR3 for the second.

SR3. Let $e = e'.0$. For the first conclusion, induction ensures either $e' = xp$ (so DL1 applies) or e' can take a step (so DL3 applies). For the second conclusion, induction ensures e' is a value (so the Canonical Forms Lemma ensures it has the form (v_0, v_1) and DR4 applies) or e' can take a step (so DR15 applies).

SR4. Analogous to the previous case

SR5. Let $e = (e_0, e_1)$. If e_0 is not a value, or e_0 is but e_1 is not, then induction ensures the nonvalue can take a step, so DR15 applies. Else e is a value.

SR6. e is a value.

SR7. Let $e = (e_1 {=} e_2)$. If e_1 is not some xp, then induction ensures e_1 can take a (left) step, so DR14 applies. Else if e_2 is not a value, then induction ensures e_2 can take a step, so DR15 applies. Else the typing derivation and Heap-Object Safety Lemma 3 provide the hypothesis to DR2.

SR8. By induction, if e' (where $e = \&e'$) is not some xp, it can take a (left) step, so DR14 applies. Else e is a value.

SR9. Let $e = e_1(e_2)$. By induction, if e_1 is not a value or e_1 is a value and e_2 is not, then the nonvalue can take a step and DR15 applies. Else,

e_1 is a value with a function type, so the Canonical Forms Lemma ensures DR5 applies.

SR10. Let $e = e'[\tau]$. By induction, if e' is not a value, it can take step, so DR15 applies. Else it is a value with a universal type, so the Canonical Forms Lemma ensures it is a polymorphic value. So DR6 applies.

SR11. By induction, if the expression inside the package is not a value, it can take a step, so DR15 applies. Else e is a value.

SR12–13. e is a value.

SR14. By induction, either e_1 can take a step (so DR15 applies) or e_1 is some v (so DR9 applies).

SR15. By induction, if e is not a value, it can take a step, so DR15 applies. Else e is a value of type int, so the Canonical Forms Lemma ensures DR10 or DR11 applies.

SR16. DR8 applies.

SR17. Let $e = \mathsf{let}\ x = e_1;\ e_2$. By induction, if e_1 is not a value, it can take a step, so DR15 applies. Else DR7 applies.

SR18. Let $e = \mathsf{open}\ e_1\ \mathsf{as}\ \alpha, x;\ e_2$. By induction, if e_1 is not a value, it can take a step, so DR15 applies. Else e_1 is a value with an existential type, so the Canonical Forms Lemma ensures it is an existential package. So DR12 applies.

SR19. Let $e = \mathsf{open}\ e_1\ \mathsf{as}\ \alpha, *x;\ e_2$. By induction, if e_1 is not of the form xp, it can take a (left) step, so DR14 applies. Else e_1 has the form xp and $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau')$. So Heap-Object Safety Lemma 3 provides there exists some v such that $\mathrm{get}(H(x), p, v)$ and $\cdot;\Upsilon;\Gamma \vdash_{\mathrm{t}} v : \exists^{\&}\alpha{:}\kappa.\tau'$. So the Canonical Forms Lemma provides v has the form $\mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'$. So DR13 applies. □

It is straightforward to check that the preservation and progress properties stated in the proof of the Type Safety Theorem are corollaries to the Preservation and Progress Lemmas. These lemmas apply given the hypotheses of $\vdash_{\mathrm{prog}} P$ and the conclusions of the Preservation Lemma suffice to conclude $\vdash_{\mathrm{prog}} P'$. The lemmas are stronger (e.g., the static context is an extension) because of their inductive proofs.

ACKNOWLEDGMENTS

Cyclone is joint work with many people, most notably Greg Morrisett, Trevor Jim, and Michael Hicks. In particular, Morrisett was a primary designer and implementor of quantified types for Cyclone.

REFERENCES

ABADI, M. AND CARDELLI, L. 1996. *A Theory of Objects*. Springer-Verlag, New York, NY.

APPEL, A. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, U.K.

AUSTIN, T., BREACH, S., AND SOHI, G. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Orlando, FL). 290–301.

Benton, N., Kennedy, A., and Russell, G. 1998. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd ACM International Conference on Functional Programming* (Baltimore, MD). 129–140.

Bos, H. and Samwel, B. 2002. Safe kernel programming in the OKE. In *Proceedings of the 5th IEEE International Conference on Open Architectures and Network Programming* (New York, NY). 141–152.

Botlan, D. L. and Rémy, D. 2003. MLF: Raising ML to the power of System-F. In *Proceedings of the ACM International Conference on Functional Programming* (Uppsala, Sweden). 27–38.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, B.C., Canada). 183–200.

Bruce, K., Cardelli, L., and Pierce, B. 1999. Comparing object encodings. *Inform. Computat. 155*, 108–133.

Cardelli, L. and Wegner, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4, 471–522.

Cejtin, H., Jagannathan, S., and Weeks, S. 2000. Flow-directed closure conversion for typed languages. In *9th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1782. Springer-Verlag, Berlin, Germany, 56–71.

Chailloux, E., Manoury, P., and Pagano, B. 2000. *Développement d'Applications avec Objective Caml*. O'Reilly, Paris, France. An English translation is currently freely available online at `http://caml.inria.fr/oreilly-book/`.

Chandra, S. and Reps, T. 1999. Physical type checking for C. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering* (Toulouse, France). 66–75.

Condit, J., Harren, M., McPeak, S., Necula, G., and Weimer, W. 2003. CCured in the real world. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. San Diego, CA. 232–244.

Crary, K. 2003. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages* (New Orleans, LA). 198–212.

Cyclone. 2001. Cyclone user's manual. Tech. rep. 2001-1855. Department of Computer Science, Cornell University, Ithaca, NY. The current version is available online at `http://www.cs.cornell.edu/projects/cyclone/`.

DeLine, R. and Fähndrich, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Snowbird, UT). 59–69.

Ditchfield, G. 1994. Contextual polymorphism. Ph.D. dissertation. University of Waterloo, Waterloo, Onto., Canada.

Garrigue, J. and Rémy, D. 1999. Semi-explicit first-class polymorphism for ML. *Inform. Comput. 155*, 1/2, 134–169.

Girard, J.-Y., Taylor, P., and Lafont, Y. 1989. *Proofs and Types*. Cambridge University Press, Cambridge, U.K.

Grossman, D. 2002. Existential types for imperative languages. In *11th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2305. Springer-Verlag, Berlin, Germany, 21–35.

Grossman, D. 2003a. Safe programming at the C level of abstraction. Ph.D. dissertation. Cornell University, Ithaca, NY.

Grossman, D. 2003b. Type-safe multithreading in Cyclone. In *Proceedings of the ACM International Workshop on Types in Language Design and Implementation* (New Orleans, LA). 13–25.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. 2002. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Berlin, Germany). 282–293.

Grossman, D., Zdancewic, S., and Morrisett, G. 2000. Syntactic type abstraction. *ACM Trans. Programm. Lang. Syst. 22*, 6 (Nov.), 1037–1080.

Harper, R. 1994. A simplified account of polymorphic references. *Inform. Process. Lett. 51*, 4 (Aug.), 201–206.

HENGLEIN, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Programm. Lang. Syst. 15*, 2 (Apr.), 253–289.

HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. 2004. Experience with safe manual memory-management in Cyclone. In *Proceedings of the International Symposium on Memory Management* (Vancouver, B.C., Canada).

HICKS, M., NAGARAJAN, A., AND VAN RENESSE, R. 2003. User-specified adaptive scheduling in a streaming media network. In *Proceedings of the 6th IEEE International Conference on Open Architectures and Network Programming* (San Francisco, CA). 87–96.

ISO. 1999. *ISO/IEC 9899:1999, International Standard—Programming Languages—C*. International Standards Organization, Geneva, Switzerland.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (Monterey, CA). 275–288.

JONES, R. AND KELLY, P. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the AADEBUG'97. Third International Workshop on Automatic Debugging*. Linköping Electronic Articles in Computer and Information Science, vol. 2(9). Linköping, Sweden.

JONES, S. P. AND HUGHES, J., Eds. 1999. *Haskell 98: A Non-Strict, Purely Functional Language*. Available online at `http://www.haskell.org/onlinereport/`.

KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. 1993. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Programm. Lang. Syst. 15*, 2 (Apr.), 290–311.

KOWSHIK, S., DHURJATI, D., AND ADVE, V. 2002. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Grenoble, France). 288–297.

LÄUFER, K. 1996. Type classes with existential types. *J. Funct. Programm. 6*, 3 (May), 485–517.

LEROY, X. 1992. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages* (Albuquerque, NM). 177–188.

LEROY, X. 1997. The effectiveness of type-based unboxing. In *Proceedings of the Workshop on Types in Compilation* (Amsterdam, The Netherlands). Also published as Tech. rep. BCCS-97-03. Computer Science Department, Boston College, Boston, MA.

LEROY, X. 2002a. *The Objective Caml System Release 3.05, Documentation and User's Manual*. Available online at `http://caml.inria.fr/ocaml/htmlman/index.html`.

LEROY, X. 2002b. Writing efficient numerical code in Objective Caml. Available online at `http://caml.inria.fr/ocaml/numerical.html`.

LISKOV, B. ET AL. 1984. *CLU Reference Manual*. Springer-Verlag, Berlin, Germany.

LOGINOV, A., YONG, S. H., HORWITZ, S., AND REPS, T. 2001. Debugging via run-time type checking. In *4th International Conference on Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag, Berlin, Germany, 217–232.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA.

MINAMIDE, Y., MORRISETT, G., AND HARPER, R. 1996. Typed closure conversion. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL). 271–283.

MITCHELL, J. 1988. Polymorphic type inference and containment. *Inform. Computat. 76*, 11–249.

MITCHELL, J. AND PLOTKIN, G. 1988. Abstract types have existential type. *ACM Trans. Programm. Lang. Syst. 10*, 3 (July), 470–502.

MORRISETT, G. 1995. Compiling with types. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA.

MORRISETT, G., CRARY, K., GLEW, N., GROSSMAN, D., SAMUELS, R., SMITH, F., WALKER, D., WEIRICH, S., AND ZDANCEWIC, S. 1999a. TALx86: A realistic typed assembly language. In *Proceedings of the 2nd ACM Workshop on Compiler Support for System Software* (Atlanta, GA). 25–35. Published as INRIA Tech. Rep. 0288. (March 1999), Rocquencourt, France.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 2002. Stack-based typed assembly language. *J. Funct. Programm. 12*, 1 (Jan.), 43–88.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1999b. From System F to typed assembly language. *ACM Trans. Programm. Lang. Syst. 21*, 3 (May), 528–569.

NECULA, G., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages* (Portland, OR). 128–139.

PATEL, P. AND LEPREAU, J. 2003. Hybrid resource control of active extensions. In *Proceedings of the 6th IEEE International Conference on Open Architectures and Network Programming* (San Francisco, CA). 23–31.

PATEL, P., WHITAKER, A., WETHERALL, D., LEPREAU, J., AND STACK, T. 2003. Upgrading transport protocols using untrusted mobile code. In *Proceedings of the 19th ACM Symposium on Operating System Principles* (New York, NY). 1–14.

PIERCE, B. 1991. Programming with intersection types and bounded polymorphism. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA.

PIERCE, B. AND SANGIORGI, D. 2000. Behavioral equivalence in the polymorphic pi-calculus. *J. Assoc. Comp. Mach. 47*, 3, 531–584.

PIERCE, B. AND TURNER, D. 1998. Local type inference. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages* (San Diego, CA). 252–265.

REYNOLDS, J. 1974. Towards a theory of type structure. In *Programming Symposium*. Lecture Notes in Computer Science, vol. 19. Springer-Verlag, Berlin, Germany, 408–425.

REYNOLDS, J. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83*. Elsevier Science Publishers, Amsterdam, The Netherlands, 513–523.

SMITH, G. AND VOLPANO, D. 1996. Towards an ML-style polymorphic type system for C. In *6th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1058. Springer-Verlag, Berlin, Germany, 341–355.

SMITH, G. AND VOLPANO, D. 1998. A sound polymorphic type system for a dialect of C. *Sci. Comput. Programm. 32*, 2–3, 49–72.

STRACHEY, C. 1967. Fundamental concepts in programming languages. Unpublished lecture notes, Summer School in Computer Programming, Copenhagen, Denmark. Eventually published in 2003 in *High. Ord. Symbol. Computat. 13*, 1-2, 5–6.

STROUSTRUP, B. 2000. *The C++ Programming Language, spec. ed.* Addison-Wesley, Reading, MA.

TARDITI, D. 1996. Design and implementation of code optimizations for a type-directed compiler for Standard ML. Ph.D. dissertation. Carnegie Mellon University, Pittsburgh, PA.

THE GHC TEAM. 2003. *The Glasgow Haskell Compiler User's Guide, Version 6.0*. Available online at `http://www.haskell.org/ghc`.

THE HUGS 98 USER MANUAL. 2002. *The Hugs 98 User Manual*. Available online at `http://haskell.cs.yale.edu/hugs`.

TOFTE, M. 1990. Type inference for polymorphic references. *Inform. Computat. 89*, 1–34.

WADLER, P. 1989. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, NY, 347–359.

WALKER, D. AND MORRISETT, G. 2000. Alias types for recursive data structures. In *Workshop on Types in Compilation*. Lecture Notes in Computer Science, vol. 2071. Springer-Verlag, Berlin, Germany, 177–206.

WELLS, J. 1999. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic 98*, 1–3 (June), 111–156.

WELLS, J., DIMOCK, A., MULLER, R., AND TURBAK, F. 2002. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programm. 12*, 3 (May), 183–227.

WRIGHT, A. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inform. Computat. 115*, 1, 38–94.

XI, H. 2000. Imperative programming with dependent types. In *Proceedings of the 15th IEEE Symposium on Logic in Computer Science* (Santa Barbara, CA). 375–387.

Received November 2003; revised December 2004; accepted April 2005