Permission-Based Programming Languages (NIER Track)*

Jonathan Aldrich† Ronald Garcia† Mark Hahnenberg† Manuel Mohr†⋆ Karl Naden† Darpan Saini† Sven Stork† Joshua Sunshine† Éric Tanter‡ Roger Wolff†
†School of Computer Science, Carnegie Mellon University ⋆Karlsruhe Institute of Technology
‡PLEIAD Lab, Computer Science Dept (DCC), University of Chile
jonathan.aldrich@cs.cmu.edu (contact author)

ABSTRACT

Linear *permissions* have been proposed as a lightweight way to specify how an object may be aliased, and whether those aliases allow mutation. Prior work has demonstrated the value of permissions for addressing many software engineering concerns, including information hiding, protocol checking, concurrency, security, and memory management.

We propose the concept of a *permission-based programming language*—a language whose object model, type system, and runtime are all co-designed with permissions in mind. This approach supports an object model in which the structure of an object can change over time, a type system that tracks changing structure in addition to addressing the other concerns above, and a runtime system that can dynamically check permission assertions and leverage permissions to parallelize code. We sketch the design of the permission-based programming language Plaid, and argue that the approach may provide significant software engineering benefits.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Representation*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Permissions*

General Terms

Design, Documentation, Human Factors, Languages, Performance, Reliability, Security, Theory, Verification

Keywords

types, permissions, programming languages

*partially funded by Fondecyt project 1110051, NSF grant CCF-0811592, the NSF CIFellows grant #0937060 to the CRA, and CMU|Portugal grant CMU-PT/SE/0038/2008. We thank the anonymous reviewers for helpful feedback.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

Permissions are annotations on pointer variables such as "unique" [2] or "read-only" [11] that specify how an object may be aliased, and which aliases may read or write to the object [7]. Permissions systems have been proposed to address a diversity of software engineering concerns, including encapsulation [12], protocol checking [8, 4], safe concurrency [6], security [5], and memory management [17, 10].

The broad range of these concerns demonstrates the general utility of understanding the aliasing and mutation characteristics of programs. However, each existing permission system above focuses on a particular issue, and differs slightly from all the others, so adding permissions to code for one concern does not aid in checking any other concern. In addition, previous permission systems lack key characteristics that make modern type systems practical, such as the ability to carry out dynamic checks (typically as casts) when a static checking system is overly conservative.

In this paper, we propose the concept of a *permission-based programming language*—a language whose object model, type system, and runtime are all co-designed with permissions in mind. In such a language, every pointer variable is associated not just with a type describing the object pointed to, but a permission describing aliasing, mutability, and other aspects of how the object may be used.

Such a language would generalize existing concern-specific permission systems to provide a single set of permissions that can check multiple software engineering concerns in a unified way. A language-based approach could go beyond mere checking, however, to enable completely new capabilities in the runtime system and object model of the language.

For example, just as the runtime system of object-oriented languages tracks the type of objects, the runtime system in a permission-based language can track permissions. Thus, if a programmer has a pointer with a `shared` permission (indicating aliases to the pointer are possible), and casts it to `unique` (indicating there are no aliases to the pointer), the runtime system will allow the cast to succeed only if there are no other pointers to that object. The runtime system can also execute different operations concurrently whenever permissions show that the operations are independent.

Permissions also support interesting new object models–for example, the ability to change the interface, representation, and behavior of an existing object at run time, as in typestate-oriented programming [1]. Permission-based languages thus enable new design and modeling capabilities, verification of diverse properties, and a level of practicality that prior permissions systems have lacked.

After a brief survey of prior work on language designs with permissions in section 2, section 3 sketches the design of the Plaid permission-based programming language. Although Plaid is still under development, section 4 briefly examines how evidence from prior permission systems supports the idea of integrating permissions into programming languages.

2. RELATED WORK

Most earlier work on permissions focuses on annotations added to existing languages, and supports purely static checking—although Boyland et al. define the dynamic semantics of their permission assertions (without implementing them) [7]. There has been considerable work on theoretical languages with permission-like type systems based on linear logic, starting with Wadler's let! construct [16].

A few languages that support permissions have been implemented. Vault is a systems-programming language that uses permissions to verify interface protocols [8]. Cyclone uses unique permissions to support safe manual memory management [10], building on an earlier calculus of capabilities [17]. Clean used permissions to integrate state into a purely functional language [13]. Concurrent with Plaid, the Alms functional programming language was developed with a permission-like type system based on linear logic that can enforce correct protocol use [15]. Each of these languages uses permissions for a single narrow concern; none considers run-time checks of permissions or the ways in which permissions can enable changes to a language's object model.

Earlier papers on Plaid described its typestate [1] and concurrency [14] features; here we focus on the concept of a permission-based language and its consequences for software engineering.

3. A PERMISSION-BASED LANGUAGE

Plaid is a general-purpose high-level permission-based programming language. It is designed to investigate the design space of permission-based languages, including their potential applications, costs, and benefits.

In order to evaluate the tradeoffs of permission-based languages, Plaid is intended to be practical; that is, it should support the construction of large software systems at a reasonable cost. For example, in our prototype, Plaid compiles to Java bytecode, allowing Plaid programs to easily interoperate with existing Java libraries. While a rigorous evaluation of "reasonable cost" has yet to be done, we have reimplemented the Plaid compiler in Plaid, demonstrating that reasonably-sized programs can be written in the language.

Plaid's permission system is driven by the software engineering properties we wish to analyze using those permissions. These properties include safe parallelism, protocol checking, encapsulation, and security properties that derive from these. Since Plaid, like Java, is intended for applications programming, we provide a garbage collector and thus do not use permissions for memory management.[1]

Permissions for Parallelism. Modern multicore processors require the use of parallelism to achieve high performance. However, parallel programming is difficult and error-prone in today's programming models, with one of the major reasons being the potential for race conditions when multiple threads access shared mutable data structures.

```
1  method void initialize(unique Model m);
2  method ... summarize(immutable Model m)
3  method ... analyze(immutable Model m)
4  ...
5  val unique Model m = new Model;
6  initialize(m);
7  // split unique into immutable
8  val s = summarize(m);
9  val a = analyze(m);
```

Listing 1: Implicit Parallelism with Permissions

Listing 1 demonstrates in the context of Plaid how permissions can be used to safely and automatically parallelize an application. In the example code, a `Model` object is created, yielding a `unique` object. The `initialize` method requires a `unique Model` because it is going to update the model during initialization. Once initialized, though, the other operations to be performed will not change the object; Plaid's type system infers that the `unique` permission can be safely split into two `immutable` permissions. Because it is safe to share any immutable data structure between threads, Plaid's runtime can execute the `summarize` and `analyze` methods in parallel.

This automatic parallelization goes beyond earlier work that used permissions to check explicitly parallel programs, and demonstrates the benefits of co-designing the type system and runtime of a permission-based language. While functional programming also allows automatic parallelization, permissions allow the functional part of the program to be parallelized even when other parts are imperative. Furthermore, permissions allow data structures to be mutated during construction, then made immutable, a common pattern that is poorly supported by mainstream functional languages.

Permissions for Protocol Checking. Building on our earlier permission-based protocol checking work, we leverage permissions to check the correct usage of stateful abstractions. For example, consider the example of a pipe that can be open or closed. We can model this with Plaid's `state` construct, declaring a global `Pipe` state and substates for `OpenPipe` and `ClosedPipe`, as shown in listing 2. Note that the `OpenPipe` state has an internal buffer as well as methods for reading, writing, and closing the pipe, while `ClosedPipe` has none of these methods, as they are not applicable to closed pipes.

The signatures of the methods in `OpenPipe` show what permissions these methods require and whether they change the state of the pipe. Permissions to the receiver object (the pipe in this case) are given in square [brackets] after the method arguments. Because a pipe is often used to communicate between two processes, and this communication involves mutating the pipe's internal buffer, we cannot use a `unique` or `immutable` permission for the read and write operations. Instead, we use `shared OpenPipe`, which indicates there may be multiple pointers to the object, and while each may be used to mutate the object's state, all clients have agreed to maintain a *state guarantee* [4] that the pipe remains in the `OpenPipe` state. Because of the shared permission, we must synchronize using `atomic` when reading from the buffer to avoid races between threads.

[1] A permission-based systems programming language might leverage ideas from Cyclone for this purpose [10]

```
1  state Pipe { }
2  state OpenPipe case of Pipe {
3      val unique Buffer buf;
4      method int read() [shared OpenPipe] {
5          atomic { /* read from the buffer */ }
6      }
7      method void write(int) [shared OpenPipe];
8      method void close()
9          [unique OpenPipe >> unique ClosedPipe] {
10         this <- ClosedPipe;
11 } }
12 state ClosedPipe case of Pipe { }
13
14 method unique OpenPipe createPipe();
15 method void forkWriter(shared OpenPipe >> none p);
16 method void readAndClose(shared OpenPipe
17                           >> unique ClosedPipe p) {
18     val data = p.read();
19     // done reading, assert unique
20     val closeablePipe = (unique OpenPipe) p;
21     closeablePipe.close();
22 }
23
24 // client code
25 val unique OpenPipe p = createPipe();
26 // split unique permission into shared permissions
27 forkWriter(p);
28 readAndClose(p);
```

Listing 2: State Change with Permissions

The `close` method's signature shows an example of a method that changes the state of the receiver: two permissions are given, one for the precondition state `OpenPipe` and one for the postcondition state `ClosedPipe`. We require a `unique` permission because we are changing the state of the object in a way that would interfere with any other clients trying to use the pipe.

A key benefit of building permissions into the language is that we can model states as a run-time construct: in Plaid, the `OpenPipe` and `ClosedPipe` states are essentially different classes, each with a different interface, behavior, and representation. The state change primitive on line 12 demonstrates this, changing the receiver's state from `OpenPipe` to `ClosedPipe`. Contrast this to previous typestate checkers [8, 4] in which typestates were used for static checking but ignored at run time. Because of runtime support for states, we can test the state of an object at run time if we lose track of it. We can also make our code more true to the abstract model, and thereby simplify reasoning about correctness. For example, an open pipe has a buffer, but a closed pipe does not, in the abstract. In a Java pipe class, we would still have the buffer field around when the pipe was closed, and we would have to set it to null and reason about the correlation between the state of the pipe and the nullness of the field. In Plaid, since states are a run-time construct, we can declare that the buffer field only exists in the `OpenPipe` state, and there is never any question about what it contains in the `ClosedPipe` state (it doesn't exist) or when it is null (it never is—Plaid types do not allow the null value by default).

Dynamic Permission Tests. A second advantage of building permissions into the language is that the runtime system can keep track of permissions, enabling dynamic checks that a permission is valid. This is important because of the conservative nature of type systems: any (decidable) type system will inevitably statically reject some valid programs, so popular languages support casts as escape hatches that replace a static check with a dynamic one.

To illustrate the value of casts in permission-based languages, consider the client code in listing 2. After creating a pipe, its `unique` permission is split into two `shared` permissions, one of which is passed to a writer thread, and the other of which goes to the reader. After reading the data and getting an end of file, the reader knows that the writer is done with the pipe. This knowledge is difficult to encode in a decidable type system, but we can simply assert that the pipe reference that we have is `unique` (line 24). When this cast executes, the runtime system will check that there are no other pointers to the pipe—i.e., the writer thread is done. At this point, it is safe to close the pipe, as we have the required permission.

Implementing casts efficiently and with appropriate semantics is a challenge. Our initial plan is to use reference counts, which can be made efficient, but requires programmer care in the case of cycles. Casts to `unique` could also be allowed to succeed, checking lazily if the object is later accessed by another reference; this avoids errors from references that exist but will never be used, at the cost of delayed notification of the cast failure. Adding type system features could reduce the need for casts; for example Cyclone and Clean included parameterization for `unique` references [10, 13], while recent work in our group extended the ideas to other permission forms [3].

Encapsulation and Security. Final concerns that may benefit from permissions include encapsulation and, partly as a consequence, security. Bokowski and Vitek cite an example Java security hole in which an internal array holding the list of principals that have signed a class was leaked from a public member function [5]. Malicious applets could get the list of signers and then mutate it to make an untrusted class appear to be trusted. Bokowski and Vitek provide a type system that allows the designer of a secure program to specify that certain data should be confined. If the designer neglects that particular specification, however, the program may still be insecure—an omission may result in a security vulnerability.

Building similar permissions into the language provides stronger protection, because in a permission-based language, *every* reference must be given a permission. When considering the list of signers of a class, the designer must explicitly consider which permission is appropriate; choosing `unique`, `immutable`, or `full` (i.e. unique write) are all sufficient to ensure security. The designer can still make an error, for example, by specifying a `shared` permission, which allows the data to leak. However, this is an error of commission, not of omission, and thus ought to be less likely.

Implementation Status. Appropriately for NIER, he Plaid language is still emerging. We have implemented a prototype compiler for the language, first in Java and now in Plaid, but work on the typechecker is still ongoing as of this writing. The runtime system includes support for unoptimized parallelization of Plaid code, as well as basic support for checking casts to `unique`.

4. POTENTIAL IMPACT

While true validation is not feasible given the emerging nature of the permission-based language concept, cautious extrapolation from experience with related permission systems can provide insight into the possible benefits and costs of the approach.

Many of the benefits of permission-based programming languages come from the synergy between the many high-impact applications of permissions. This impact has been documented in a number of existing projects. With respect to concurrency, the Fluid tool [9] leverages permissions such as uniqueness to verify correct thread usage; Fluid has been used in 100kloc+ case studies at several top software vendors, identifying race conditions that resulted in patches to shipping code. In the area of typestate, the Plural [4] tool has been applied to multiple open source programs totaling 125kloc+, assuring hundreds of protocol uses and finding many protocol errors and race conditions in well-tested code [3]. A recent study of protocols in Java suggests that almost three times as many types define protocols as define type parameters, suggesting that significant benefits may accrue from modeling protocols explicitly in the language and type system [3]. Designing a language around permissions allows programmers to gain the benefits of checking both concurrency and typestate, while specifying the relevant permissions only once. In addition, native language support allows the expression of permissions in a lower-overhead way than in annotation-based approaches.

A second benefit of permission-based languages stems from the ability to support permissions in the runtime system as well as in the type system. While permission-based static checkers are good, every static checker yields either false positives or false negatives. In a permission-based language, we can use a sound checking approach (which has no false negatives) and use casts to address false positives, as is done in static type systems. The resulting system shares the benefits of static checkers in providing assurance that checked code is correct, but provides a dynamic check as a backup when the programmer believes a permission warning is a false positive. Run-time support for permissions can also automatically parallelize the program, as discusssed before.

Permission-based languages have costs as well. In principle, every type in the program must now include a permission, and supporting concerns like typestate requires permissions in some places (e.g. the receiver of a method) where existing languages do not require types. We plan to mitigate this cost through the sensible use of defaults: many types are designed to be immutable (e.g. purely functional data structures), or are typically unaliased (e.g. iterators, which are mostly used on the stack), and so we can put a default permission on the type declaration rather than at every use of the type. Perhaps more important than the syntactic cost is the semantic cost: programmers must consider how each object might be aliased. Considering the problems that are caused by aliased, mutable state in modern software systems, however, a case can be made that when the permission is obvious, a default is likely applicable, and when the permission is not obvious, it should be documented anyway. Perhaps the greatest value of types is not in the checking they perform, but rather in the provision of automatically-checked, up-to-date documentation that can be used by programmers, IDEs, and tools alike: permission-based languages extend this documentation to a new class of important design constraints.

In summary, a permission-based language integrates information about aliasing and mutation within both the type system and the runtime of the program. This permission information can then be leveraged for checking a wide range of software engineering concerns, and can also provide powerful new modeling capabilities in the object model of the language. The combination of static and dynamic checking facilitated by a language-based approach increases both the power and the practicality of the checking. Although much research remains to be done, permission-based programming languages show the potential to significantly improve the engineering of modern software systems.

5. REFERENCES

[1] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-Oriented Programming. In *Proc. Onward!*, 2009.

[2] H. G. Baker. 'Use-once' variables and linear objects—storage management, reflection, and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995.

[3] N. Beckman. *Types for Correct Concurrent API Usage*. PhD thesis, Carnegie Mellon University, 2010.

[4] K. Bierhoff and J. Aldrich. Modular typestate verification of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[5] B. Bokowski and J. Vitek. Confined Types. In *Object-Oriented Programming, Systems, Languages, and Applications*, November 1999.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications*, November 2002.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *European Conference on Object-Oriented Programming*, 2001.

[8] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*, 2001.

[9] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *International Conference on Software Engineering*, 2002.

[10] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Programming Language Design and Implementation*, 2002.

[11] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 1991.

[12] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.

[13] R. Plasmeijer and M. van Eekelen. Keep it Clean: A unique approach to functional programming. In *ACM Sigplan Notices*, 1999.

[14] S. Stork, P. Marques, and J. Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Proc. Onward!*, 2009.

[15] J. A. Tov and R. Pucella. Practical affine types. In *Principles of Programming Languages*, 2011.

[16] P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, 1990.

[17] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Program. Lang. Syst.*, 22(4):701–771, 2000.