# Type System Support for Dynamic Revocation

Chris Hawblitzel Thorsten von Eicken

Cornell University

**Abstract**

Safe languages provide mechanisms to share data among multiple protection domains in a protected fashion, but lack support for revocation and control over resource usage. This paper describes a type system extension, called *permits*, to support the revocation of data structures. Permits use a combination of static and run-time checks to ensure the revocability of data shared between safe language tasks, with a minimum of run-time overhead. The paper presents the syntax for this extension formally, and summarizes our experience with implementing the type system in a real language.

## 1 Introduction

Several recent projects have proposed using type-safe languages to provide security for extensible systems that run in a single address space. For example, SPIN[BSP+95] allows extensions written in Modula-3 to run in the operating system kernel, Java applets[WBD97] extend a web browser, and extensible web servers load Java servlets[Jav] into the server's address space. We'll describe such an extensible system as a collection of *tasks* all running in a single address space, where a task is simply a collection of objects, and every object in the system is owned by a particular task. For example, an extensible web server consists of one task for the server, and one task for each servlet (see the J-Kernel[HCC+98] and GVM[BTS+98] for more about safe languages tasks). Figure 1 shows two tasks, one controlled by Alice and one by Bob. Small boxes represent objects, and arrows represent references (pointers) from one object to another.

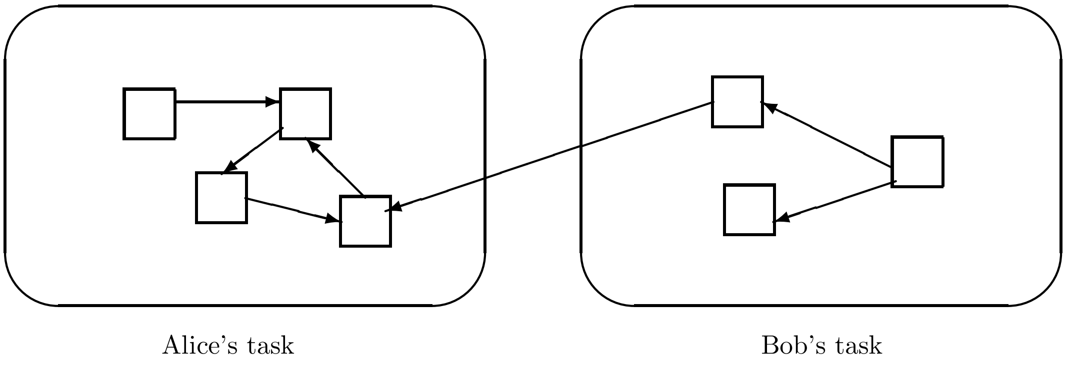


Figure 1: Tasks and Objects

One technique common to all of these language-based extensible systems is the use of unforgeable references as *capabilities*. The idea is simple: if Alice gives Bob a reference to one of her objects, then Bob can access the object by invoking methods on it; otherwise, Bob has no way to access the object. In figure 1, Bob has been given a reference to one of Alice's object, and can therefore access this object, but the language does not allow him to "forge" a reference to Alice's other objects (by trying to cast an integer to a reference, for instance).

Although capabilities have been implemented in operating systems before without relying on type-safe languages [WLH81, Lev84], such implementations typically store capabilities in the operating system kernel to ensure that they cannot be forged, which means that they can only be manipulated by special operations exported by the kernel. This slows down accesses to capabilities, and makes capability manipulation cumbersome. Using safe language object references as capabilities, rather than relying on the operating system kernel, promises to make capabilities both inexpensive, since a capability access is simply a method invocation or field access, and easy to use, since object references can be passed as arguments to method invocations and stored in data structures in a natural way. Both these factors should make capabilities more attractive than they have been in the past, enabling programmers to construct secure, extensible systems without fearing a high performance penalty.

One major problem with object references is that they are not revocable. Once Alice gives Bob a reference to her object, Bob can use the reference for as long as he wants. Revocation is important for several reasons:

- An individual user's preferences may change over time. For example, Alice might mistakenly consider Bob trustworthy at first, but later change her mind if he abuses the rights that she granted him.
- If Alice only needs to grant Bob a capability for a short time (such as the duration of a transaction), then the principle of least privilege says that it is better to give Bob access to the capability only for the necessary duration, rather than to give him access that lasts forever.
- Shutting down a task requires deallocating the task's objects. Any other tasks that had access to the deallocated objects must have their access revoked in a safe way, to prevent dangling references to the deallocated objects.

Historically, revocation has been a difficulty for many capability systems. One solution for traditional capability systems was proposed by Redell [Red74]: rather than giving away a direct pointer to a capability, give away a pointer to a pointer to the capability. The extra pointer, which we'll call an indirection pointer, contains an internal pointer to the real object, and accesses to the indirection pointer are forwarded to the real object through the internal pointer. Access to the real object can be revoked simply by setting this internal pointer to null. This scheme seems so easy to implement in a safe language that one might hope that this is all it takes to solve the revocation problem. However, this simple approach has several limitations:

1. Just because programmers can protect objects with a level of indirection doesn't mean that they will. In general, the language cannot systematically guarantee that indirection pointers are inserted where necessary. For example, pointers from one task to another must be revocable so that tasks can be shut down, but the system has no guarantee that programmers will add a level of indirection to every cross-task pointer.
2. A single indirection pointer isn't enough to protect an entire data structure. Suppose we want to use indirection to protect a simple Java list class: `class List {int i; List next;}`. An indirection pointer can only point to one object, so traversing a list would require allocating a separate indirection pointer for each element, and then ensuring that all the indirection pointers can be revoked together. This wastes quite a bit of space, and tracking all the indirection pointers is messy if the list adds and removes elements dynamically.
3. In some cases, the cost of performing a run-time check on each access to an object is prohibitively expensive. In particular, systems that use arrays of bytes to handle performance-critical data such as network packets would suffer enormously from a run-time check for each byte that is accessed; it would be cheaper in this case to simply copy the data. As a result, systems like SPIN share byte arrays directly with no way to perform revocation.

This paper proposes a type system extension to provide revocation in a way that addresses all three of these issues. The idea is to allow direct sharing of objects between tasks, but to use the type system to track this sharing. The type system systematically ensures that all shared objects are revocable (solving problem #1 above), easily allows direct sharing and revocation of complex data structures, not just individual objects (solving problem #2 above), and cooperates with a specialized run-time system to lift run-time revocation checks out of loops when possible (addressing problem #3 above). Section 2 introduces the type system informally, section 3 formalizes the type system, section 4 discusses extensions to the type system to deal with selective revocation, and section 5 describes our current implementation of the type system and plans for future implementations.

## 2 Permits (presented informally)

This section presents a type system supporting revocation using an informal Java-like syntax. The central idea is to represent a revocable object reference as a pair of a normal object reference and a *permit* that controls access to the object. We'll write the type of such an object as A˜p, where A is the type of the object and p is a permit variable. In order to make revocation dynamic, permits are created at run-time and can be revoked on demand as the program runs. Because permits are dynamically revoked, accessing an object of type A˜p requires a run-time check of p, and the check raises an exception if it discovers that p was revoked.

For example, the following function takes a Pair object protected by a permit p, and returns the sum of the pair's fields:

```
class Pair {int x; int y;}
int sum(permit p, Pair~p pair) {return pair.x + pair.y;}
```

The `pair` argument points directly to a Pair object, without requiring any indirection. Only the type of `pair` indicates that access to it is controlled by the permit p. Based on this type information, the compiler automatically inserts run-time checks before the accesses to `pair.x` and `pair.y`.

Every object in the system is allocated under the control of some permit, using the syntax "`new A~p(...))`" to allocate an object of type A˜p. The type system statically ensures that such an object always has the type A˜p, where p is the permit that was used to allocate the object; the object cannot be cast to a type annotated with a different permit. When the permit p is revoked, the object is deallocated. This deallocation is safe, because an object of type A˜p is inaccessible after p is revoked. Because permits allow safe deallocation, they provide a clean way to shut down tasks safely: each task is given its own permit with which to allocate objects, and when this permit is revoked, the task is shut down and all of the task's objects are deallocated.

The depiction of tasks in Figure 1 shows that references in a task either point to local objects, or to objects in another task. We'll call the former *local* references, and the latter *remote* references. Rather than require the types of local references to explicitly name the task's own permit, a local reference to an object of type A simply has the type A, with no permit annotation. In this way, the type system clearly distinguishes between ordinary, local object references, whose types are unannotated, and remote references, whose types are always annotated with permits. This clear distinction has two benefits:

- No run-time revocation check is needed for accesses to local objects, because objects local to a task aren't revoked until the task itself is shut down. This means that while the task is running, it's local objects are guaranteed to be accessible. Because local objects are distinguished by the type system, the compiler can easily determine which object accesses need run-time revocation checks. This ensures that "ordinary" code which doesn't explicitly use permits is not slowed down.
- The distinction helps programmers track which objects are local and which are remote. Programmers need to be aware of which objects are remote for two reasons. First, they must be prepared to handle failure, if a remote object is revoked. Second, tasks may not trust each other, so programmers must be careful not to pass sensitive information or privileges to remote objects accidentally. Flagging remote objects in the type system helps to mark the boundaries of trust in the system.

A single permit can be applied to multiple objects: if x has type List˜p and y has type Tree˜p, then the access to both x and y is controlled by the same permit, p. If p is revoked, both x and y become inaccessible. Furthermore, entire data structures can be protected by a single permit. The following function takes a permit p and a List guarded by the permit p as arguments, and traverses the list:

```
class List {int i; List next;}
int sum(permit p, List~p list) {
   int n = 0;
   while(list != null) {n += list.i; list = list.next;}
   return n;
}
```

The list of type List~p may have many elements, and all the elements should be protected by p. The protection provided by p would be nearly useless if one could obtain a unprotected pointer to the tail of the list by reading the `next` field of the list:

```
List tail = list.next; // oops!  tail should have type List~p, not List
```

To prevent this, we slightly alter the meaning of the "`List next;`" declaration, so that accessing the `next` field yields an object of type List~p. Let every object have an implicit permit variable called "this_permit" that holds the permit with which the object is protected, in analogy to Java's "this" pointer. Further, let the unannotated type "List" be shorthand for "List~this_permit". So if the variable `list` has type List~p, then inside the list object, this_permit is equal to p, so that the `list.next` field has type List~p. Recursively, `list.list.next` also has type List~p, and so on, so that the whole data structure is protected by p. Note that this change does not affect the run-time *implementation* of a List, only the type-checking rules. A List object is still represented at run-time as an pair of an integer and a direct pointer to the next element of the list.

The sum function above performs a run-time revocation check for each access to list.next and list.i, since list is protected by a permit. Rather than having two run-time checks per iteration, it would be preferable to perform this check once at the start of the loop and then let the loop run with no further checks. However, if p is revoked by another thread while the loop is still running, then the loop must be interrupted. It would be unacceptable to delay the revocation of p until the loop finishes, especially if there is no guarantee that the loop terminates. The programmer can express the scope in which the program can be interrupted with an "open(p) ..." block:

```
int n = 0;
open(p) {while(list != null) {n += list.i; list = list.next;}}
return n;
```

When a thread enters the open statement, the run-time system registers the fact that the thread has p open. If p is revoked, the run-time system finds all threads that have p open and raises exceptions in them asynchronously (this is similar to Java's Thread.stop() method).

The "open" statement demonstrates a combination of dynamic and static checks: the example above depends both on dynamic support from the run-time system (a run-time check when p is opened, and a possible asynchronous exception if p is revoked), and on the type system's static guarantee that all the elements of the list are guarded with the same permit.

# 3 Formalization

This section formalizes the ideas of the previous section. The formalization is based on the typed lambda calculus written in continuation passing style [App92]. Memory allocation is explicit: as in Tofte et al. [TT94], memory consists of a set of regions, each of which contains a set of heap values. The syntax is shown in Figure 2.

Type judgments for expressions have the form $\Psi; \Delta; \Theta; \Gamma \vdash e$, which means that the expression $e$ typechecks under the assumptions that memory has type $\Psi$, the permit variables listed in $\Delta$ are in scope, the value variables listed in $\Gamma$ are in scope, and the permits listed in $\Theta$ are open (more on this below). The operational semantics describe how a running program steps from a state $(M, \Theta, e)$ to $(M', \Theta', e')$, where $M$ is the state of the heap, and $\Theta$ is a set of open permits. The operational semantics only support one thread of execution, which means that revocation does not take place asynchronously. Even without asynchronous revocation, we still have to worry about the thread revoking one of its own open permits (this isn't detectable statically, because of aliasing), which is why $\Theta$ is used to track which permits the thread has open at run-time. There isn't space here to present the complete typechecking rules and operational semantics, but we have specified these formally, and we have proved that the resulting system is sound (every well-typed program state either halts or steps to another well-typed state).

The types shown in figure 2 are fairly standard except that function types $(\vec{\tau} \to 0) \triangleleft \vec{p}$ and tuple types $\langle\vec{\tau}\rangle \triangleleft p$ are annotated with permits, where a permit $p$ is either a permit variable $\rho$ or a region name $r$. The region names are used only during the evaluation of a program, though (as the program runs, permit variables are replaced with region names), so from the programmer's point of view, a permit is simply a permit variable $\rho$. Functions can have types $\forall\rho.\tau$ parameterized over permit variables, and values can be packed with existential types $\exists\rho.\tau$ [MP88] so that permits can be stored in data structures and later retrieved.

$$
\begin{array}{rcll}
\textit{types}: & \tau & = & \text{int} \mid (\vec{\tau} \to 0) \triangleleft \vec{p} \mid \langle\vec{\tau}\rangle \triangleleft p \mid \forall\rho.\tau \mid \exists\rho.\tau \mid p \text{ handle} \\
\textit{permits}: & p & = & \rho \mid r \\
\textit{values}: & v & = & x \mid i \mid r.l \mid \text{handle}(r) \mid \text{pack } p, v \text{ as } \exists\rho.\tau \\
\textit{heap values}: & h & = & \langle\vec{v}\rangle \mid \mu f.\Lambda\vec{\rho}.\lambda\overrightarrow{x{:}\tau}.(e \triangleleft \vec{p}) \\
\textit{expressions}: & e & = & \text{halt}[\tau]v \mid \text{if0 } v \text{ then } e_1 \text{ else } e_2 \mid v_0(\vec{p})(\vec{v}) \\
 & & \mid & \text{let } \rho, x = \text{unpack } v \text{ in } e \mid \text{let } x = v \text{ in } e \mid \text{let } x = \#n\ v \text{ in } e \\
 & & \mid & \text{let } x = h \text{ at } v \text{ in } e \mid \text{let } \rho, x = \text{newpermit in e} \\
 & & \mid & \text{setopen } \vec{v} \text{ in } e_1 \text{ catch } e_2 \mid \text{revoke } v \text{ in } e_1 \text{ catch } e_2 \\
\textit{heaps}: & M & = & \{r_1 \mapsto R_1, \ldots, r_n \mapsto R_n\} \\
\textit{regions}: & R & = & \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} \mid \text{revoked} \\
\textit{open permit sets}: & \Theta & = & \{p_1, \ldots, p_n\} \\
\textit{heap types}: & \Psi & = & \{r_1 \mapsto \Upsilon_1, \ldots, r_n \mapsto \Upsilon_n\} \\
\textit{region types}: & \Upsilon & = & \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\} \\
\textit{permit environments}: & \Delta & = & \{\rho_1, \ldots, \rho_n\} \\
\textit{value environments}: & \Gamma & = & \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\}
\end{array}
$$

Figure 2: Typed lambda calculus extended with permits

The "let $\rho, x$ = newpermit in e" expression creates a new permit and binds it to a variable $\rho$. In addition, it creates a *handle* for the new permit of type "$\rho$ handle", which the program uses to control the region at run-time, and binds this handle to the variable $x$. At run-time, the permit and permit handle are represented by a region $r$ and a region handle "handle($r$)", respectively, where $r$ is a new region of memory allocated in the heap $M$. When newpermit creates $r$, it initializes $M(r)$ to the empty set (containing no heap values). Subsequent instructions manipulate the region by using the region handle:

- A program can allocate heap values (tuples and functions) with the "let $x = h$ at $v$ in $e$" expression, where $h$ is a heap value and $v$ is a handle to some permit $p$. When this executes, $h$ is added to $M(p)$, a pointer $p.l$ is created to name the location of the heap object, and this pointer is bound to the variable $x$. The type of the heap value is annotated by $p$, ensuring that the value is not accessible after $p$ is revoked.

- A program can revoke the permit with the "revoke $v$ in $e_1$ catch $e_2$" expression, where $v$ is a handle to some permit $p$. When executed, this expression deallocates the region $p$, replacing $M(p)$ with the value "revoked". Note that this may leave dangling pointers to the heap values that used to be in $M(p)$. However, the types of these values are annotated with $p$, and the type system prevents the pointers from being traversed after $p$ is revoked.

- A program can open a set of permits with the "setopen $\vec{v}$ in $e_1$ catch $e_2$" command, specifying a list of permit handles to open. At run-time, the program maintains a set $\Theta$ of open permits; setopen simply replaces the old $\Theta$ with a new set of permits:

$$(M, \Theta, \text{setopen (handle}(r_1), \ldots, \text{handle}(r_n)) \text{ in } e_1 \text{ catch } e_2) \longmapsto (M, \{r_1, \ldots, r_n\}, e_1)\ \ if\ \forall i.M(r_i) \neq \text{revoked}$$

The setopen expression must perform a run-time check to make sure that none of the permits have been revoked. If it discovers a revoked permit, then it transfers control to an exception handler $e_2$ rather than to $e_1$, and sets $\Theta$ to the empty set. Similarly, after performing a revocation, the revoke expression transfers control to an exception handler if the program revokes a permit that is a member of $\Theta$:

$$\begin{array}{ll}(M, \Theta, \text{revoke handle}(r) \text{ in } e_1 \text{ catch } e_2) \longmapsto (M\{r \mapsto \text{revoked}\}, \Theta, e_1) & if\ r \notin \Theta \\ (M, \Theta, \text{revoke handle}(r) \text{ in } e_1 \text{ catch } e_2) \longmapsto (M\{r \mapsto \text{revoked}\}, \{\}, e_2) & if\ r \in \Theta\end{array}$$

These run-time checks enforce the invariant that at all times, $\Theta$ only contains non-revoked permits.

We chose the semantics of setopen to simplify the type system, since this one instruction can both open and close permits. However, setopen performs superfluous run-time checks of permits that are already open. To avoid these checks, setopen could be broken up into different instructions for opening and closing permits. In addition, we chose to use CPS conversion to simplify the operational semantics of setopen: without CPS conversion, the operational semantics would need to keep a stack of $\Theta$ sets, so that setopen can restore the old $\Theta$ when it finishes execution.

In order to use a value whose type is annotated with permits, the program must first open the permits listed in the annotation using the setopen expression. Once the permits are open, the heap objects may be accessed with no run-time checks. To statically track which permits are open, the typechecking rules keep an environment $\Theta$ of open permits. The setopen expression introduces permits into this environment:

$$\frac{\begin{array}{l}(\Psi; \Delta; \Gamma \vdash v_1 : p_1 \text{ handle}) \ldots (\Psi; \Delta; \Gamma \vdash v_n : p_n \text{ handle}) \\ \Psi; \Delta; \{p_1, \ldots, p_n\}; \Gamma \vdash e_1 \\ \Psi; \Delta; \{\}; \Gamma \vdash e_2\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash \text{setopen } v_1, \ldots, v_n \text{ in } e_1 \text{ catch } e_2}$$

Accesses to heap values can only be typechecked if the proper permit is open. For example, projecting the i$^{\text{th}}$ field of a tuple of type $\langle\vec{\tau}\rangle\triangleleft\, p$ requires p to be open:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \langle\tau_1, \ldots, \tau_n\rangle\triangleleft\, p \quad \Psi; \Delta; \Theta; \Gamma\{x \mapsto \tau_i\} \vdash e \quad p \in \Theta}{\Psi; \Delta; \Theta; \Gamma \vdash \text{let } x = \#i\ v \text{ in } e}$$

Functions are introduced with a recursive function definition $\mu f.\Lambda\vec{\rho}.\lambda\overline{x\!:\!\tau}.(e\triangleleft\vec{p})$, which defines a recursive function $f$ that takes permit parameters $\vec{\rho}$, value parameters $\overline{x\!:\!\tau}$, and has a body $e$. Loops are expressed with recursion, so in order to avoid performing run-time checks on every recursive call, a function can specify a list of permits $\vec{p}$ that must already be opened before the function may be called; the function body $e$ can then use these permits without reopening them. For example, a variant on of the sum function from the last section can be declared as follows:

$$\begin{array}{c}\mu f_{sum}.\Lambda\rho_1, \rho_2.\lambda x\!:\!\langle\text{int}, \text{int}\rangle\triangleleft\ \rho_1, y\!:\!(\text{int} \to 0) \triangleleft \rho_2.(e \triangleleft \rho_1, \rho_2) \\ where\ e = (\text{let } z_1 = \#1\ x \text{ in let } z_2 = \#2\ x \text{ in let } z_{sum} = z_1 + z_2 \text{ in } y(z_{sum}))\end{array}$$

For simplicity, this sum function $f_{sum}$ takes a pair of integers rather than a list of integers. The $x$ argument holds the pair, guarded by permit $\rho_1$, and the $y$ argument holds a continuation, guarded by permit $\rho_2$, which receives the integer return value of the sum function (the return value $z_{sum}$ is the sum of the two integers in the pair). The sum function requires that both $\rho_1$ and $\rho_2$ be open when it is called, so that it can access $x$ and $y$. Alternately, the sum function could take handles to $\rho_1$ and $\rho_2$, and then open the handles itself with setopen.

Since functions are heap values, the sum function must be allocated in a region using the "let x=h at v in e" expression. If allocated in some region $\rho_3$, the sum function would have type:

$$\forall\rho_1.\forall\rho_2.(\langle\text{int}, \text{int}\rangle\triangleleft\ \rho_1, (\text{int} \to 0) \triangleleft \rho_2 \to 0) \triangleleft \rho_1, \rho_2, \rho_3$$

# 4 Selective Access Control

Using object references as capabilities makes it easy to selectively grant access to a resource: if Alice creates an object and gives the object reference to Bob, but not to Carl, then Bob will have access to the object and Carl will not. However, the revocation mechanism presented in the previous section cannot revoke access selectively; if Alice revokes anyone's access to her object, by revoking the object's permit, then she revokes everyone's access, including her own. Ideally, Alice would be able to revoke Bob's access without revoking her own access. This section discusses how to extend the system in the previous section with selective revocation, although these extensions have not been formalized yet.

Suppose Alice's object has type $\langle\vec{\tau}\rangle\triangleleft\ \rho_1$. Alice wants to give this object to Bob in a more restricted form, so that it can be revoked before $\rho_1$ is revoked. To do this, Alice gives Bob her object with type $\langle\vec{\tau}\rangle\triangleleft\ \rho_2$, so that Bob sees the object protected by $\rho_2$ rather than $\rho_1$. Alice can revoke Bob's access by revoking $\rho_2$, while she continues to view the object with type $\langle\vec{\tau}\rangle\triangleleft\ \rho_1$, which isn't affected by revoking $\rho_2$. For this approach to be sound, $\rho_2$ must be revoked before $\rho_1$. Otherwise, Bob could use $\rho_2$ to access the object after $\rho_1$ is revoked and the object is deallocated. We'll write the constraint that $\rho_2$ does not outlive $\rho_1$ as "$\rho_2 \subseteq \rho_1$".

There are two ways to express this approach to selective revocation. One is to make $\langle\vec{\tau}\rangle\triangleleft\ \rho_1$ a subtype of $\langle\vec{\tau}\rangle\triangleleft\ \rho_2$, so that Alice's object can be coerced to type $\langle\vec{\tau}\rangle\triangleleft\ \rho_2$:

$$\frac{\tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n' \quad \rho_2 \subseteq \rho_1}{\langle\vec{\tau}\rangle\triangleleft\ \rho_1 <: \langle\vec{\tau'}\rangle\triangleleft\ \rho_2}$$

Subtyping is fairly complex, though, so rather than giving Bob an object with a different type, a second way to implement selective revocation is to give Bob an object of the original type $\langle\vec{\tau}\rangle\triangleleft\ \rho_1$, but to deny him the handle for $\rho_1$. Instead, Alice gives Bob the handle for $\rho_2$, with the guarantee that $\rho_2 \subseteq \rho_1$. Now we can say that if Bob opens $\rho_2$, he can access $\langle\vec{\tau}\rangle\triangleleft\ \rho_1$. This is still sound, because $\rho_2$ cannot outlive $\rho_1$, and it implements the desired selective revocation policy: Bob can only use the object until $\rho_2$ is revoked, while Alice can continue to use the object because she has the handle to $\rho_1$. The main difference between this approach and subtyping is that Bob has to work with two permit variables, rather than just one (although he only needs one handle); for example, he might write a function of the following form:

$$\Lambda\rho_1.\Lambda\rho_2 \subseteq \rho_1.\lambda x : \langle\vec{\tau}\rangle\triangleleft\ \rho_1, y : \rho_2\ \text{handle}.(\ldots)$$

Both schemes above assume that there is some way to prove that $\rho_2 \subseteq \rho_1$ in the first place. This guarantee can be made if permits are organized hierarchically, so that when a parent permit is revoked, all its children are automatically revoked. To do this, the "newpermit" expression must be altered so that it creates a child permit, given a handle to a parent permit.

This hierarchical arrangement allows for multiple levels of sublifetimes: if Alice gives Bob a $\rho_2$ such that $\rho_2 \subseteq \rho_1$, Bob can create a $\rho_3$ so that $\rho_3 \subseteq \rho_2$, and pass $\rho_3$ to Carl. Both Alice and Bob can revoke Carl's access to $\rho_3$ at this point. This idea is similar to Redell's idea of using multiple levels of indirection to a single object (a pointer to a pointer to a pointer)[Red74], but has the advantage that accessing an object can be performed in constant time, without traversing a chain of pointers.

# 5 Implementation

To demonstrate the use of revocation in a real language, we have implemented a language called Passport, which is an extension to a safe subset of C. It features permits, an "open" statement, the use of subtyping for selective revocation, pointers to functions, preemptive user-level threads, tasks, and primitive module and exception handling facilities. It lacks type polymorphism, object orientation, higher-order functions, and a garbage collector. To maintain consistency with the syntax of section 2 (and because a Java-like syntax is easier to read than Passport's C-like syntax), all the examples of Passport in this section have been translated into a Java-based syntax. This section does not attempt to describe Passport in detail; instead, it tries to fill in some implementation details left out of the previous sections, and to list some lessons that we learned from trying to move the ideas of the previous sections into a "real-world" language.

Functions in Passport can take permits as arguments, and datatypes (which are introduced by the keyword `struct` rather than `class`, since Passport doesn't support object orientation) can be annotated with permits. The following function takes a pair of integers guarded by a permit, and returns the sum of the integers:

```
struct Pair {int z1, z2;}
int sum(permit p, Pair~p x) {
   open(p) {return x.z1 + x.z2;}
}
```

Permits can also be stored in data structures, which corresponds to the existential types from section 3. In the following datatype, each node in the data structure has its own permit, rather than constraining the whole data structure to be protected by just one permit:

```
struct HeteroList {
   int i;
   permit p;
   HeteroList~p next;
}
```

Unlike the region variables of Tofte et al.[TT94], which are used statically but aren't needed at run-time, permits need a run-time implementation so that revocation can be checked for at run-time. In Passport, a permit is implemented as a pointer to a data structure containing a flag that indicates whether the permit has been revoked, pointers to the child and parent permit data structures, and a reference count of the permits pointing to it, so that it can be freed when it is no longer needed. Creating a new permit and performing a revocation check on a permit are constant time operations; revoking a permit frees the memory blocks allocated under the permit and revokes all of the permit's children, so that revocation takes time proportional to the resources that must be freed.

Although Passport did demonstrate the implementation of dynamic revocation in a language, in retrospect we made many mistakes in its design. Here is a summary of some of the things we learned in implementing and using Passport:

- Passport focused excessively on user-defined memory management. Programmers were expected to use permits to explicitly deallocate objects, rather than relying garbage collection. While this meant that Passport could safely deallocate objects without a garbage collector, it put a large burden on the programmer. For the language to be easy to use, automatic memory management is essential.

- Passport used subtyping for selective revocation. This did not fit well into an imperative language, because programmers had to flag variables as being "covariant" and/or immutable in order to take advantage of subtyping. The second approach described in section 4 is more appropriate for an imperative language.

- Passport did not support the "this_permit" idiom described in section 2. In order to declare the List type from section 2, programmers had to write a struct that was explicitly polymorphic over permits (the syntax `<permit p>` introduces a struct that is polymorphic over a permit):

  ```
  struct<permit p> List2 {
     int i;
     List2<p>~p next;
  }
  ```

  This requires p to be mentioned twice in "`List2<p>~p`". Adding "this_permit" to the language makes sharing data structures between tasks more natural.

- The declaration `int sum(permit p, A~p x)` uses dependent types [XP98]: x's type depends on the variable p. Passport allows permits to be stored in structs, which means that it has to handle declarations like `int f(B x, A~(x.p) y)`. Dependent types complicate the type checker implementation.

A better design would separate the permit variables from the value variables, both in function and struct declarations, in the same way that languages supporting polymorphism separate type variables from value variables.

In addition, Passport lacks object orientation and type polymorphism. Based on our experience with Passport, our current efforts are focused on integrating permits into a higher-level, easier to use language based on a subset of Java.

# 6 Related Work

Our experience with Java showed that while Java has strong protection mechanisms in the core of the language (the unforgeability of pointers and access specifiers such as private and public), it fails to clearly mark the boundary between different protection domains. Other languages have also been used to supplement or replace hardware protection in various systems (Mesa in Pilot[RDH80], Modula-3 in SPIN[BSP+95], Oberon in the Oberon operating system[WG89]). Like Java, these systems take a "single region approach", where all protection domains' objects live in a single pool, and their is no notion of one protection domain owning an object. This prevents them from precisely controlling the resource usage of different protection domains.

One attempt to address these problems is the J-Kernel [HCC+98], an operating system layer built on top of Java that provides a strong notion of tasks and revocation. However, to ensure that all shared objects can be revoked, the J-Kernel sacrifices the direct sharing of data structures between tasks. The only objects that can be shared between tasks are remote procedure call stubs called capabilities that use indirection to implement revocation. All other types of objects can only be passed between tasks by copy. Permit annotations allow tasks to share data structures directly, without requiring copies, and without losing track of who owns the data structures. Furthermore, programmers can still copy a remote object into a local object if they want, by writing their own copy code, so that they can express J-Kernel style remote procedure calls.

Virtual memory-based operating systems such as L4 [Lie95] have an advantage over language based protection in that every memory access goes through a level of indirection, as specified in a process' page table. This makes revocation and resource reclamation straightforward, as revoking a process' access to memory simply requires modifying the page table and invalidating the TLB. The advantages of virtual memory inspired the design of permits. For example, the use of open permits to lift run-time checks out of loops is analogous to the TLB's caching of page table entries. At a higher level, the use of permits to build safe language "tasks" was motivated by the tasks and processes found in traditional operating systems.

Permit annotations are based on the region annotations described in Tofte et al.[TT94]. In this system, annotations are used a memory management technique that can supplement or replace garbage collection. This approach uses lexical scoping to ensure that data structures are never accessed after their region goes out of scope. This lexical scoping allows the typechecker to implement a form of revocation entirely statically, with no run-time checks. However, it also limits the structure of sharing in the program: regions cannot be shared among threads or stored in data structures for later use. In addition, revocation cannot be performed on demand. In order to revoke access to a region, one has to wait for the evaluation inside a letregion to complete, which may never happen. For these reasons, permits use run-time as well as static checks.

A version of regions for CPS converted programs was described by Crary et al.[CWM99]. Like the approach of Tofte et al., it relies entirely on static mechanisms to implement revocation. The main challenge overcome by this system is tracking the aliasing that occurs when two region variables refer to the same region. Interestingly, selective revocation is quite easy in this system, since it is possible to pass an object to a function, and statically check that the function relinquishes access to the object when it finishes execution.

## 7 Conclusion

While safe language object references provide a way to grant access to resources, languages lack a corresponding way to revoke access. Permit annotations extend a standard type system with dynamic revocation, systematically ensuring that access to data structures can be revoked on demand. This approach retains the initial advantages of language based protection — object references can still be used as capabilities, and the language's own mechanisms, such as method invocation, are used to transfer capabilities from one task to another in a protected way. Although the theoretical description of permits is fairly precise at this point, the integration of permits into a practical language is still an ongoing endeavor. Our experience with Passport demonstrates that such a practical language is feasible, but that more work is necessary to make the language easy to use.

## References


[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System.* 15th ACM Symposium on Operating Systems Principles, p. 267-284, Copper Mountain, CO, December 1995.

[BTS+98] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. *Java Operating Systems: Design and Implementation.* University of Utah Rechnical Teport UUCS-98-015, August 1998.

[CWM99] K. Crary, D. Walker, and G. Morrisett. *Typed Memory Management in a Calculus of Capabilities.* In 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p. 262-275, San Antonio, TX, January 1999.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java.* 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.

[Jav] JavaSoft. *Java Servlet API.* http://java.sun.com.

[Lev84] H. M. Levy. Capability-Based Computer Systems. Digital Press, Bedford, Massachusetts, 1984.

[Lie95] J. Liedtke. *On $\mu$-kernel Construction.* 15th ACM Symposium on Operating Systems Principles, p. 237-250, Copper Mountain, CO, December 1995.

[MP88] J. C. Mitchell and G. D. Plotkin. *Abstract Types Have Existential Type.* ACM Transactions on Programming Languages and Systems, 10(3):470-502, July 1988.

[Red74] D. D. Redell. *Naming and Protection in Extendible Operating Systems.* Technical Report 140, Project MAC, MIT 1974.

[RDH80] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. *Pilot: An operating system for a personal computer.* Communications of the ACM, Volume 23, Number 2, p. 81-92, February 1980.

[TT94] M. Tofte and J.P. Talpin. *Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions.* 21st ACM Symposium on Principles of Programming Languages, p. 188-201, Portland, OR, January 1994.

[WG89] N. Wirth, J. Gutknecht. *The Oberon System.* Software Practice and Experience, Volume 19, Number 9, p. 857-893, 1989.

[WBD97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116-128, Saint-Malo, France, October 1997.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbsion. *Hydra/C.mmp:An Experimental Computer System.* McGraw-Hill, 1981.

[XP98] H Xi and F. Pfenning. *Eliminating Array Bound Checking Through Dependent Types.* 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, 17-19 June 1998.