JFP **21** (2): 159–207, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000134 First published online 22 September 2010

Roles, stacks, histories: A triple for Hoare

JOHANNES BORGSTRÖM, ANDREW D. GORDON
Microsoft Research, Cambridge, United Kingdom
(*e-mail:* adg@microsoft.com)

RICCARDO PUCELLA
Northeastern University, College of Computer and Information Science, Boston, Massachusetts, USA

Abstract

Behavioral type and effect systems regulate properties such as adherence to object and communication protocols, dynamic security policies, avoidance of race conditions, and many others. Typically, each system is based on some specific syntax of constraints, and is checked with an *ad hoc* solver. Instead, we advocate types refined with first-order logic formulas as a basis for behavioral type systems, and general purpose automated theorem provers as an effective means of checking programs. To illustrate this approach, we define a triple of security-related type systems: for role-based access control, for stack inspection, and for history-based access control. The three are all instances of a refined state monad. Our semantics allows a precise comparison of the similarities and differences of these mechanisms. In our examples, the benefit of behavioral type-checking is to rule out the possibility of unexpected security exceptions, a common problem with code-based access control.

1 Introduction

1.1 Behavioral type systems

Type-checkers for behavioral type systems are an effective programming language technology aimed at verifying various classes of program properties. We consider type and effect systems, typestate analyses, and various security analyses as being within the class of behavioral type systems. A few examples include memory management (Gifford & Lucassen 1986), adherence to object and communication protocols (Strom & Yemini 1986; DeLine & Fähndrich 2001), dynamic security policies (Pistoia *et al.* 2007b), authentication properties of security protocols (Gordon & Jeffrey 2003), avoidance of race conditions (Flanagan & Abadi 1999), and many more.

While the proliferation of behavioral type systems is a good thing—evidence of their applicability to a wide range of properties—it leads to the problem of fragmentation of both theory and implementation techniques. Theories of different behavioral type systems are based on a diverse range of formalisms, such as calculi of objects, classes, processes, functions, and so on. Checkers for behavioral type systems often make use of specialized proof engines for *ad hoc* constraint languages. The fragmentation into multiple theories and implementations hinders both the comparison of different systems, and also the sharing of proof engines between implementations.

We address this fragmentation. We show three examples of security-related behavioral type systems that are unified within a single logic-based framework. Moreover, they may be checked by invoking the current generation of automated theorem provers, rather than by building *ad hoc* solvers.

1.2 Refinement types and automated theorem proving

The basis for our work is the recent development of automatic type-checkers for pure functional languages equipped with refinement types. A *refinement type* $\{x : T \mid C\}$ consists of the values x of type T such that the formula C holds. Since values may occur within the formula, refinement types are a particular form of dependent type. Variants of this construction are referred to as refinement types in the setting of ML-like languages (Freeman & Pfenning 1991; Xi & Pfenning 1999; Flanagan 2006), but also as *subset types* (Nordström *et al.* 1990) or *set types* (Constable *et al.* 1986) in the context of constructive type theory, and *predicate subtypes* in the setting of the interactive theorem prover PVS (Rushby *et al.* 1998).

In principle, type-checking with refinement types may generate logical verification conditions requiring arbitrarily sophisticated proof. In PVS, for example, some verification conditions are implicitly discharged via automated reasoning, but often the user needs to suggest an explicit proof tactic.

Still, some recent type-checkers for these types use external solvers to discharge automatically the proof obligations associated with refinement formulas. These solvers take as input a formula in the syntax of first-order logic, including equality and linear arithmetic, and attempt to show that the formula is satisfiable. This general problem is known as *satisfiability modulo theories* (SMT) (Ranise & Tinelli 2006); it is undecidable, and hence the solvers are incomplete, but remarkable progress is being made.

Three examples of type-checkers for refinement types are SAGE (Flanagan 2006; Gronski *et al.* 2006), F7 (Bengtson *et al.* 2008), and Dsolve (Rondon *et al.* 2008). These type-checkers rely on the SMT solvers Simplify (Detlefs *et al.* 2005), Z3 (de Moura & Bjørner 2008), and Yices (Dutertre & de Moura 2006).

Our implementation experiments are based on the F7 type-checker, which checks programs in a subset of the Objective Caml and F# dialects of ML against a type system enhanced with refinements. The theoretical foundation for F7 and its type system is RCF, which is the standard Fixpoint Calculus (FPC, a typed call-by-value λ-calculus with sums, pairs, and iso-recursive types) (Plotkin 1985; Gunter 1992) augmented with message-passing concurrency and refinement types with formulas in first-order logic.

1.3 RIF: refinement types meet the state monad

Moggi (1991) pioneered the *state monad* as a basis for the semantics of imperative programming. Wadler (1992) advocated its use to obtain imperative effects within pure functional programming, as in Haskell, for instance. The state monad can be written as the following function type, parametric in a type state, of global imperative state.

$$\mathcal{M}(T) \triangleq \mathsf{state} \rightarrow (T \times \mathsf{state})$$

The idea is that $\mathcal{M}(T)$ is the type of a computation that, if it terminates on a given input state, returns an answer of type T, paired with an output state.

With the goal of full verification of imperative computations, various authors, including Filliâtre (1999) and Nanevski *et al.* (2006), consider the state monad of the form below, where P and Q are assertions about state. (We elide some details of variable binding.)

$$\mathcal{M}_{P,Q}(T) \triangleq (\mathsf{state} \mid P) \rightarrow (T \times (\mathsf{state} \mid Q))$$

The idea here is that $\mathcal{M}_{P,Q}(T)$ is the type of a computation returning T, with precondition P and postcondition Q. More precisely, it is a computation that, if it terminates on an input state satisfying the precondition P, returns an answer of type T, paired with an output state satisfying the postcondition Q. Hence, one can build frameworks for Hoare-style reasoning about imperative programs (Filliâtre & Marché 2004; Nanevski *et al.* 2008), where $\mathcal{M}_{P,Q}(T)$ is interpreted so that $(\mathsf{state} \mid P)$ and $(\mathsf{state} \mid Q)$ are dependent pairs consisting of a state together with proofs of P and Q. (The recent paper by Régis-Gianas & Pottier (2008) on Hoare logic reasoning for pure functional programs has a comprehensive literature survey on formalizations of Hoare logic.)

In this paper, we consider an alternative reading: let the *refined state monad* be the interpretation of $\mathcal{M}_{P,Q}(T)$, where $(\mathsf{state} \mid P)$ and $(\mathsf{state} \mid Q)$ are refinement types populated by states known to satisfy P and Q. In this reading, $\mathcal{M}_{P,Q}(T)$ is simply a computation that accepts a state known to satisfy P and returns a state known to satisfy Q, as opposed to a computation that passes around states paired with proof objects for the predicates P and Q.

This paper introduces and studies an imperative calculus in which computations are modeled as Fixpoint Calculus expressions in the refined state monad $\mathcal{M}_{P,Q}(T)$. More precisely, our calculus, which we refer to as *Refined Imperative FPC*, or RIF for short, is a generalization of FPC with dependent types, subtyping, global state accessed by get and set operations, and computation types refined with preconditions and postconditions. To specify correctness properties, we include assumptions and assertions as expressions. The expression **assume**$(s)C$ adds the formula $C\{^{M}\!/_{s}\}$, where M is the current state, to the *log*, a collection of formulas assumed to hold. The expression **assert**$(s)C$ always returns at once, but we say it *succeeds* when the formula $C\{^{M}\!/_{s}\}$, where M is the current state, follows from the log, and otherwise it *fails*. We define the syntax, operational semantics, and type system for RIF, and give a safety result, Theorem 1 (Safety), which asserts that safety (the lack of all assertion failures) follows by type-checking. This theorem follows from a direct encoding of RIF within RCF, together with appeal to a safety theorem for RCF itself. The appendices include the direct encoding of our calculus RIF within the existing calculus RCF.

Our calculus is similar in spirit to Hoare Type Theory (Nanevski *et al.* 2006) and YNot (Nanevski *et al.* 2008), although we use refinement types for states instead of dependent pairs, and we use formulas in classical first-order logic suitable for direct proof with SMT solvers, instead of constructive higher-order logic. Another difference is that RIF has a subtype relation, which may be applied to computation types to, for example, strengthen preconditions or weaken postconditions. A third difference is that we are not pursuing full program verification, which typically requires some human interaction, but instead view RIF as a foundation for automatic type-checkers for behavioral type systems.

If we ignore variable binding, both our refined type $\mathcal{M}_{P,Q}(T)$ and the constructive types in the work of Filliâtre & Marché (2004) and Nanevski *et al.* (2008) are instances of Atkey's (Atkey 2009) parameterized state monad, where the parameterization is over the formulas concerning the type state. When variable binding is included, the type $M_{P,Q}(T)$ is no longer a parameterized monad, since the preconditions and postconditions are of different types as the postcondition can mention the initial state.

1.4 Unifying behavioral types for roles, stacks, and histories

Our purpose in introducing RIF is to show that the refined state monad can unify and extend several automatically checked behavioral type systems. RIF is parametric in the choice of the type of imperative state. We show that by making suitable choices of the type state, and by deriving suitable programming interfaces, we recover several existing behavioral type systems, and uncover some new ones.

We focus on security-related examples where runtime security mechanisms—based on roles, stacks, and histories—are used by trusted library code to protect themselves against less trusted callers. Unwarranted access requests result in security exceptions.

First, we consider role-based access control (RBAC) (Ferraiolo & Kuhn 1992; Sandhu *et al.* 1996), where the current state is a set of activated roles. Each activated role confers access rights to particular objects.

Second, we consider permission-based access control, where the current state includes a set of permissions available to running code. We examine two standard variants: stack-based access control (SBAC) (Gong 1999; Wallach *et al.* 2000; Fournet & Gordon 2003) and history-based access control (HBAC) (Abadi & Fournet 2003). We implement each of the three access control mechanisms as an application programming interface (API) within RIF.

We show how the APIs for the RBAC, SBAC, and HBAC mechanisms support the writing of both trusted libraries and untrusted application code. In each case, checking application code against the API amounts to behavioral typing, and ensures that application code causes no security exceptions. For trusted library code, which controls the state representing the access control permissions, static checking prevents accidental programming errors that can lead to security exceptions. For untrusted callers that may invoke the trusted library, and which do not have direct access to the state representing the access control permissions, static checking prevents both programming errors and subversion of the security mechanism. (Of course, subversion cannot be prevented in trusted library code, or in untrusted callers when the trusted library provides direct access to the access control permissions.)

Our results show the theoretical feasibility of our approach. We have type-checked all of the example code in this paper by first running a tool that implements the encoding of RIF into RCF described in the appendices, and then type-checking the translated code with F7 and Z3.

The contents of the paper are as follows. Section 2 considers access control with roles. Section 3 considers access control with permissions, based either on stack inspection or a history variable. We use our typed calculus in these sections but postpone the formal definition to Section 4. Finally, Section 5 discusses related work and Section 6 offers some conclusions, and a dedication.

Appendix A recalls the definition of RCF (Bengtson *et al.* 2008). Appendix B provides a semantics of the calculus of this paper in RCF.

An earlier, abridged version of this article will appear in the proceedings of a meeting at Microsoft Research, Cambridge, on April 16–17, 2009, to celebrate the 75th birthday of Tony Hoare. A draft of the present article appears as a technical report (Borgström *et al.* 2009).

2 Types for role-based access control

In general, access control policies regulate access to resources based on information about both the resource and the entity requesting access to the resource, as well as information about the context of the request. In particular, RBAC policies base their decisions on the actions that an entity is allowed to perform within an organization—their role. Without loss of generality, we can identify resources with operations to access these resources, and therefore RBAC decisions concern whether a user can perform a given operation based on the role that the user plays. Thus, roles are a device for indirection: instead of assigning access rights directly to users, we assign roles to users, and access rights to roles.

In this section, we illustrate the use of our calculus by showing how to express RBAC policies, and demonstrate the usefulness of refinements on state by showing how to statically enforce that the appropriate permissions are in place before controlled operations are invoked. This appears to be the first type system for RBAC properties—most existing studies on verifying RBAC properties in the literature use logic programming to reason about policies independently from code (Li *et al.* 2002; Becker & Sewell 2004; Becker & Nanz 2007). We build on the typeful approach to access control introduced by Fournet *et al.* (2005), where the access policy is expressed as a set of logical assumptions; relative to that work, the main innovation is the possibility of de-activating as well as activating access rights.

2.1 Simple RBAC: File system permissions

As we mentioned in the introduction, our calculus is a generalization of FPC with dependent types and subtyping. As such, we will use an ML-like syntax for expressions in the calculus. The calculus also uses a global state to track security information, and computation types refined with preconditions and postconditions to express properties of that global state. The security information recorded in the global state may vary depending on the kind of security guarantees we want to provide. Therefore, our calculus is parameterized by the security information recorded in the global state and the operations that manipulate that information.

To use our calculus, we need to *instantiate* it with an extension API module that implements the security information tracked in the global state, and the operations to manipulate that information. The extension API needs to define a concrete state type that captures the information recorded in the global state. Functions in the extension API are the only functions that can explicitly manipulate the state via the primitives **get**() and **set**(). Moreover, the extension API defines predicates by assuming logical formulas; this is the only place where assumptions are allowed.

We present an extension API for RBAC. In the simplest form of RBAC, permissions are associated with roles, and therefore we assume a type role representing the class of roles. The model we have in mind is that roles can be active or not. To be able to use the permissions associated with a role, that role must be active. Therefore, the security information to be tracked during computation is the set of roles that are currently active.

RBAC API:

```
type state = role list

val activate: r:role → {(s)True} unit {(s')Add(s',s,r)}
val deactivate: r:role → {(s)True} unit {(s')Rem(s',s,r)}

assume ∀ts,x. Mem(x,ts) ⇔ (∃y, vs. ts = y::vs ∧ (x = y ∨ Mem(x,vs)))
assume ∀rs,ts,x. Add(rs,ts,x) ⇔ (∀y. Mem(y,rs) ⇔ (Mem(y,ts) ∨ x=y))
assume ∀rs,ts,x. Rem(rs,ts,x) ⇔ (∀y. Mem(y,rs) ⇔ (Mem(y,ts) ∧ ¬(x = y)))
assume ∀s,r. Active(r,s) ⇔ Mem(r,s)
```

An extension API supplies three kinds of information. First, it fixes a type for the global state. Based on the discussion above, the global state of a computation is the set of roles that are active, hence we take **type** state = role list, where role is the type for roles, which is a parameter to the API.

Second, an extension API gives functions to manipulate the global state. The extension API for primitive RBAC has two functions only: activate to add a role to the state of active roles, and deactivate to remove a role from the state of active roles.

We use **val** f : T to give a type to a function in an API. Expressions get *computation types* of the form $\{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\}$, where the scope of s_0 is C_0, T and C_1 and the scope of x and s_1 is C_1. Such a computation type is interpreted semantically using the refined state monad mentioned in Section 1.3, where it corresponds to the type $\mathcal{M}_{(s_0)C_0,(s_1)C_1}(T)$. In particular, a computation type states that an expression starts its evaluation with a state satisfying C_0 (in which s_0 is bound to that state in C_0) and yields a value of type T and a final state satisfying C_1 (in which s_0 is bound to the initial state of the computation in C_1, s_1 is bound to the final state of the computation, and x is bound to the value returned by the computation). Thus, for instance, activate is a function that takes role r as input and computes a value of type unit. That computation takes an unconstrained state (that is, satisfying True), and returns a state that is the union of the initial state and the newly activated role r—recall that a state here is a list of roles. Similarly, deactivate is a function that takes a role as input and computes a unit value in the presence of an unconstrained state and producing a final state that is simply the initial state minus the deactivated role.

The third kind of information contained in an API are logical axioms. Observe that the postconditions for activate and deactivate use predicates such as Add and Rem. We define such predicates using *assumptions*, which let us assume arbitrary formulas in our assertion logic, formulas that will be taken to be valid in any code using the API. Ideally, these assumed formulas would be proved sound in some external proof assistant, in terms of some suitable model, but here we follow an axiomatic approach. For the purposes of RBAC, we assume some set-theoretic predicates (using lists as a representation for sets) and a predicate Active true exactly when a given role is currently active. Most predicates are parameterized by the global state, that is, the current set of active roles.

RBAC API implementation:

```
// Set-theoretic operations (provided by a library)
val add: l:α list → e:α → {(s)True} r:α list {(s')s=s' ∧ Add(r,l,e)}
val remove: l:α list → e:α → {(s)True} r:α list {(s')s=s' ∧ Rem(r,l,e)}

let activate r = let rs = get() in let rs' = add rs r in set(rs')
let deactivate r = let rs = get() in let rs' = remove rs r in set(rs')
```

The implementation of activate and deactivate use primitive operations **get**() and **set**() to, respectively, get and set the state of the computation. It is important that **get**() and **set**() only be used in the implementation of API functions; in particular, user code calling into the API cannot use those operations to arbitrarily manipulate the state. The API functions are meant to encapsulate all state manipulation. In practice, we enforce such a restriction through a module system that keeps **get**() and **set**() local to the module implementing the API. For the purpose of this paper, we shall simply assume that **get**() and **set**() are only available in API functions implementations. Beyond the use of **get**() and **set**(), the implementation of the API functions above also uses set-theoretic operations add and remove to manipulate the content of the state. We only give the types of these operations—their implementations are the standard list-based implementations.

We associate permissions to roles via an access control policy expressed as logical assumptions. We illustrate this with a simple example, that of modeling access control in a primitive file system. We assume two kinds of roles: the superuser, and friends of normal users (represented by their login names):

```
type role = SuperUser | FriendOf of string
```

In this scenario, permissions concern which users can read which files. For simplicity, we consider a policy where a superuser can read all files, while other users can access specific files, as expressed in the policy. A predicate CanRead(f,s) expresses the "file f can be read in global state s". Here is a simple policy in line with this description: permission.

```
assume ∀file,s. Active(SuperUser,s) ⇒ CanRead(file,s)
assume ∀s.Active(FriendOf("Andy"),s) ⇒ CanRead("andy.log",s)
```

This policy, aside from stating that the superuser can read all files, also states that if the specific role FriendOf("Andy") is active, then the file andy.log can be read. For simplicity, we consider only read permissions here. It is straightforward to extend the example to include write permissions or execute permissions.

The main function we seek to restrict access to is readFile, which intuitively requires that the currently active roles suffice to derive that the file to be read can in fact be read.

```
val readFile: file:string → {(rs) CanRead(file,rs)} string {(s)s=rs}
let readFile file = assert (rs)(CanRead(file,rs)); primReadFile file
```

We express this requirement by writing an assertion in the code of readFile, before the call to the underlying system call primReadFile. The **assert** expression checks that the current state (bound to variable rs) proves that CanRead(file,rs) holds. Such an assertion *succeeds* if the formula is provable, and *fails* otherwise. The main property of our language is given by a *safety theorem*: if a program type-checks, then all assertions succeed. In other words, if a program that uses readFile type-checks, then we are assured that by the time we call primReadFile, we are permitted to read file, at least according to the access control policy. The type system, somewhat naturally, forces the precondition of readFile to ensure that the state can derive CanRead for the file under consideration.

Intuitively, the following expression type-checks:

```
activate(SuperUser); readFile "andy.log"
```

The expression first adds role SuperUser to the state, and the postcondition of activate notes that the resulting state is the union of the initial state (of which nothing is know) with SuperUser. When readFile is invoked, the precondition states that the current state rs must be able to prove the permission CanRead("andy.log",rs). Because SuperUser is active and Active(SuperUser,s) implies CanRead(file,s) for any file and global state s, we get CanRead("andy.log",rs), and we can invoke readFile. The following examples type-check for similar reasons, since Active(FriendOf "Andy",rs) can prove the formula CanRead("andy.log",rs):

```
activate(FriendOf "Andy"); readFile "andy.log"

activate(FriendOf "Andy"); deactivate(FriendOf "Jobo"); readFile "andy.log"
```

In contrast, the following example fails to activate any role that gives a CanRead permission on file "andy.log", and therefore fails to type-check:

```
activate(FriendOf "Ric"); readFile "andy.log" // Does not type-check
```

After activating FriendOf "Ric", the postcondition of activate expresses that the state contains whatever was in the initial state along with role FriendOf "Ric". When invoking readFile, the type system tries to establish the precondition, but it only knows that Active(FriendOf "Ric",rs), where rs is the current global state, and the policy cannot derive the formula CanRead("andy.log",rs) from it. Therefore, the type system fails to satisfy the precondition of readFile "andy.log", and reports a type error.

The access control policy need not be limited to a statically known set of files. Having a full predicate logic at hand affords us much flexibility. To express, for instance, that any file with extension .txt can be read by anyone, we can use a predicate Match:

```
assume ∀file,s.Match(file,"*.txt") ⇒ CanRead(file,s)
```

Rather than axiomatizing the Match predicate, we rely on a function glob that does a dynamic check to see if a file name matches the provided pattern, and in its postcondition fixes the truth value of the Match predicate on those arguments:

```
val glob: file:string → pat:string →
  {(rs) True} r:bool {(rs') rs=rs' ∧ (r=true ⇔ Match(file,pat))}
let glob file pat = if (* ... code for globbing ... *)
                    then assume Match(file,path); true
                   else assume ¬Match(file,path); false
```

The following code therefore type-checks, even when all the activated roles do not by themselves suffice to give a CanRead permission. When checking the then-branch of the conditional, our system recalls the post-condition of the call to glob, and also that its result is true. These facts imply that Match(f, "*.txt") holds, and hence that CanRead(f, s) holds for any s, which suffices to establish the precondition of the call to readFile.

```
activate(FriendOf "Ric");
let f = "log.txt" in
  if (glob f "*.txt") then readFile f else "skipped"
```

Similarly, not only can we specify which roles give CanRead permissions for which files by saying so explicitly in the policy (as above), we can also dynamically check that a friend of some user can read a file by querying the physical file system through a primitive function primReadFSPerm(f,u) that checks whether a given user u (and therefore their friends) can access a given file f, and reflects the result of that dynamic check into the type system:

```
val hasFSReadPermission: f:string → u:string → {(rs) True}
    r:bool {(rs') rs=rs' ∧ (r=True ⇒ (∀s.Active(FriendOf(u),s) ⇒ CanRead(f,s)))}
let hasFSReadPermission f u =
        if primReadFSPerm (f,u)
          then assume (s) Active(FriendOf(u),s) ⇒ CanRead(f,s); true
          else false
```

Above, the assumption in primReadFSPerm is dependent on the current state, bound to the variable s in the formula Active(FriendOf(u),s)⇒ CanRead(f,s). The following code now type-checks:

```
activate(FriendOf "Andy");
if (hasFSReadPermission "somefile" "Andy")
  then readFile "somefile"
  else "cannot read file"
```

The code first activates the role FriendOf "Andy", and then dynamically checks, by querying the physical file system, that user "Andy" (and therefore his friends) can in fact read file "somefile". The type of hasFSReadPermission is such that if the result of the check is true, the new formula ∀s.Active(FriendOf("Andy"),s)⇒ CanRead ("somefile",s) can be used in subsequent expressions—in particular, when calling readFile "somefile". At that point, FriendOf "Andy" is active in global state rs, and therefore CanRead("somefile",rs) holds.

To what extent is our encoding of RBAC as an API in RIF correct? Unfortunately, we are not aware of any work that presents an independently-motivated semantics for RBAC as a type system for a calculus or programming language against which we could compare our encoding. The best we can do is observe that we can capture with our type system the kind of policies that are investigated in specialized logics for access control. This is somewhat unsatisfying because those logics are rarely embedded in a programming language, making a direct comparison difficult. The next section provides an example of the kind of comparison we can perform.

2.2 Extended RBAC: Health care policies

To evaluate how suitable our language is for modeling complex RBAC scenarios, we embed an example by Becker & Nanz (2007, Section 4), inspired by policies in electronic health care.

To model this example, we build on top of the insights gained in Section 2.1, using a more extensive state. Not only do we have roles such as patient, clinician and administrator, as before, but we introduce a notion of users that activate and deactivate those roles, and a database of facts to record information that can be queried by the policies. In particular, the database will record information such as which users can activate which roles.

We assume a type id of identities, a type role of roles, and a type fact of facts (see example below). We take states to have **type** state = id * role list * fact list. The API is now somewhat richer, including operations to deal with identities and the database of facts. To simplify the presentation of pre- and postconditions, we assume that free variables that occur in a pre- or postcondition are existentially quantified in that formula, e.g., pre- or postcondition (s')s=(i,rs,db) ∧ s'=(u,[],db) in the scope of u and s is a shorthand for (s')∃i,rs,db. s=(i,rs,db) ∧ s'=(u,[],db).

Extended RBAC API:

```
type state = id * role list * fact list

val switch_user: u:id → {(s)True} unit {(s') s=(i,rs,db) ∧ s'=(u,[],db)}
val activate: r:role → {(s) CanActivate(r,s)} unit {(s')s=(i,rs,db) ∧ s'=(i,rs',db) ∧
      Add(rs',rs,r)}
val deactivate: r:role → {(s)True} unit {(s')s=(i,rs,db) ∧ s'=(i,rs',db) ∧ Rem(rs',rs,r)}
val record: f:fact → {(s)True} unit {(s')s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Add(db',db,f)}
val remove: f:fact → {(s)True} unit {(s')s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Rem(db',db,f)}

assume ∀s,id,rs,db.s=(id,rs,db) ⇒ Ident(id,s)
assume ∀s,id,rs,db.s=(id,rs,db) ⇒ (∀r.Active(r,s) ⇔ Mem(r,rs))
assume ∀s,id,rs,db.s=(id,rs,db) ⇒ (∀f.Fact(f,s) ⇔ Mem(f,db))
```

Extended RBAC API implementation:

```
let switch_user u = let (i,rs,db) = get() in set(u,[],db)
let activate r = assert (s)(CanActivate(r,s));
                 let (i,rs,db) = get() in let rs' = add rs r in set(i,rs',db)
let deactivate r = let (i,rs,db) = get() in let rs' = remove rs r in set(i,rs',db)
let record f = let (i,rs,db) = get() in let db' = add db f in set(i,rs,db')
let remove f = let (i,rs,db) = get() in let db' = remove db f in set(i,rs,db')
```

The main difference here is that activate now has precondition stating that the role can be activated by the current user. The predicates Ident, Active, and Fact, respectively, capture the current identity, the currently active roles, and the currently registered facts recorded in the global state. The predicate CanActivate is defined by the access control policy, as we shall see below.

The API above is very general, and in particular does not impose any restriction on who can record and remove facts from the database. In a library built using this API, as we shall see below, we should specialize the types of record and remove for the various kinds of facts at hand, and we should prevent user code from accessing the underlying record and remove functions. Again, this is achieved in practice using a module system, but here we simply assume the underlying functions are hidden.

Identities are simply names, roles include patient, clinician, and administrator, and the database of facts records information such as which identities can activate which roles, as well as consents that have been requested and granted.

```
type id = Name of string
type role = Patient | Clinician | Admin
type fact = IsMember of id * role | HasConsented of id * id
         | HasRequestedConsent of id * id
```

In order for a user to activate a role, we need to ensure that the current identity is either allowed to activate the role according to a static policy, or is a member of the requested role as recorded in the database. In addition, we impose a separation-of-duty restriction, ensuring that one cannot activate both the Clinician role and the Administrator role at the same time.

```
assume ∀r,s.CanActivate(r,s) ⇔ (UserCanActivate(r,s) ∧ NoConflict(r,s))
assume ∀r,s,id.(Ident(id,s) ∧ Fact(IsMember(id,r)),s) ⇒ UserCanActivate(r,s)
assume ∀s.Ident(Name("Andy"),s) ⇒ UserCanActivate(Admin,s)
assume ∀r,s.NoConflict(r,s) ⇔ ((r=Clinician ⇒ ¬Active(Admin,s)) ∧
                               (r=Admin ⇒ ¬Active(Clinician,s)))
```

Since role activation may depend on membership information kept in the database, we define the following primitives for registering and unregistering membership information, subject to the requirement that they can only be invoked when the Admin role is active—an assertion to that effect enforces this requirement.

```
val register: u:id → r:role → {(s) Active(Admin,s)} unit
    {(s')s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Add(db',db,IsMember(u,r))}
let register u r = assert (s)(Active(Admin,s)); record(IsMember(u,r))

val unregister: u:id → r:role → {(s) Active(Admin,s)} unit
    {(s')s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Rem(db',db,IsMember(u,r))}
let unregister u r = remove(IsMember(u,r))
```

Consider the following policy for reading electronic health records (EHR): users can read their own EHR, and clinicians can read any EHR for which they have received consent.

```
assume ∀u,s.Ident(u,s) ⇒ CanReadEHR(u,s)
assume ∀u,s.Active(Clinician,s) ∧ Consented(u,s) ⇒ CanReadEHR(u,s)
```

The main function is readEHR, which reads an EHR. It asserts that the current user can in fact read the EHR, based on the currently active roles.

```
val readEHR: file:string → {(s) CanReadEHR(Name(file),s)} string {(t)s=t}
let readEHR file =
   assert (s)(CanReadEHR(Name(file),s));
   (* ... read record ... *)
```

There remains the issue of consent. A patient can give consent to an individual to read their EHR, as long as that individual first requested consent from the patient. We record who requested consent and who consented (and to whom in both cases) in the database. We provide functions requestConsent and giveConsent that refine the type of record and ensure that the right identity is used in the consent, and that the right roles are active:

```
assume ∀u,v,s.Consented(u,s) ⇔ (Ident(v,s) ∧ Fact(HasConsented(u,v),s))

val requestConsent: u:id → v:id →
  {(s) Ident(u,s) ∧ Active(Clinician,s)} unit
  {(s') s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Add(db',db,HasRequestedConsent(u,v))}
let requestConsent u v = record(HasRequestedConsent(u,v))

val giveConsent: u:id → v:id →
   {(s) Ident(u,s) ∧ Active(Patient,s) ∧ Fact(HasRequestedConsent(v,u,s))}
       unit
   {(s')s=(i,rs,db) ∧ s'=(i,rs,db') ∧ Add(db',db,HasConsented(u,v))}
let giveConsent u v =
       assert (s)(Ident(u,s) ∧ Active(Patient,s) ∧ Fact(HasRequestedConsent(v,u,s)));
       record(HasConsented(u,v))
```

How can we use the above interface? One application is to type-check access control properties of workflows for interacting with a medical records server. Roughly speaking, a workflow is a description of the steps that a group of users can follow to achieve an objective. In the setting of this section, it is probably more natural to consider workflows as models of a system of users attempting to achieve an objective, thereby using RIF as a modeling language. This is similar in spirit to how we can use the logic of Becker & Nanz (2007, Section 4) which inspired our example. It is also possible to implement workflows as actual executable artifacts, for instance, on a machine such as a web server, or a smartcard. The key primitive that needs to be implemented there is the switch_user *u* primitive, which corresponds to a context switch between users, and may be implemented by having the server present a login window requiring password-based authentication, with the primitive returning only if user *u* correctly logs in.

As an example, we can verify that the following workflow is well typed against the above interface. It takes users *pat* and *doc* as arguments—where *doc* is assumed to be a clinician—registers *pat* as a patient, and lets *doc* read *pat*'s medical file.

```
val workflow: pat:id → doc:id → {(s) Fact(IsMember(doc,Clinician),s)} string {(s')
     True}
let workflow pat doc =
  switch_user (Name "Andy"); activate Admin; register pat Patient;
  switch_user doc; activate Clinician; requestConsent doc pat;
  switch_user pat; activate Patient; giveConsent pat doc;
  switch_user doc; activate Clinician; readEHR pat
```

This workflow prescribes that an administrator (here, `Name "Andy"`) must first log in to enable the registration of the patient, and then the doctor must log in to request consent from the patient, followed by the patient logging in to give consent, at which point the doctor can log in again to read the medical file. The point here is to force the author of the workflow to put in sufficient input validation that there will be no policy-driven error messages at runtime.

3 Types for permission-based access control

The RBAC systems of the previous section are most applicable in an interactive setting, where principals inhabiting different roles can influence the computation as it is running. Without interaction, we can instead work with a static division of the program code based on its provenance. We assume that each function is assigned a set of static permissions that enable it to perform certain side effects, such as file system IO. A classical problem in this setting is the Confused Deputy (Hardy 1988), where untrusted code performs unauthorized side effects through exploiting a trusted API. This problem has been addressed through various mechanisms. In this section, we consider SBAC (Gong 1999; Wallach *et al.* 2000) and HBAC (Abadi & Fournet 2003).

The purpose of SBAC is to protect trusted functions from untrusted callers. Unless explicitly requested, a permission only holds at run-time if all callers on the call stack statically hold the permission.

HBAC also intends to protect trusted code from the untrusted code it may call, by ensuring that the run-time permissions depend on the static permissions of every function *called so far in the entire program.* In particular, when a function returns, the current run-time permissions can never be greater than the static permissions of that function. HBAC can be seen as a refinement of SBAC, in the sense that the run-time permissions at any point when using the HBAC calling conventions are less than those when using SBAC.

In this section, we show how the RIF calculus supports type-checking of both SBAC and HBAC policies. There are several formalizations of SBAC, some of which include type systems. Previous type systems for SBAC took a rather simple view of permissions. To quote Pottier *et al.* (2005): “In our model, privileges are identifiers, and expressions cannot compute privileges. It would be desirable to extend the static framework to at least handle first-class parameters of privileges, so for example, a Java FilePermission, which takes a parameter that is a specific file, could be modeled.” Having both computation types and dependent types in our imperative calculus lets us treat not only parameters to privileges, but also have a general theory of partially ordered privileges. We can also type-check code that computes privileges, crucially including the privilege-manipulating API functions defined in Section 3.2.

As a side effect, we can also investigate the differences between SBAC and HBAC as implemented in our framework. We show one (previously known) example, where switching from SBAC to HBAC resolves a security hole by throwing a run-time exception; additionally, static type-checking discovers that the code is not safe to run under HBAC.

The use of type-checkers allows the authors of trusted code to statically exclude run-time security exceptions relating to lack of privileges. As discussed above, we provide a more sensitive analysis than previous work, which facilitates the use of the principle of least privilege. Type-checking can also be applied to untrusted code before loading it, ensuring the lack of run-time security exceptions.

3.1 A lattice of permission sets

As a running example, we introduce the following permissions. The ScreenIO permission is atomic. A FileIO permission is a tuple of an access right of type RW and a file scope of type Wildcard. The access rights are partially ordered: the owner of a file can both read and write it. The scope Any extends to any file in the system.

Partially ordered permissions:

```
type α Wildcard = Any | Just of α
type RW = Read | Write | Owns
type Permission = ScreenIO | FileIO of RW * (string Wildcard)
type Perms = Permission list
```

When generalizing HBAC and SBAC to the setting, where permissions are partially ordered, we run into a problem. Both HBAC and SBAC are built on taking unions and intersections of sets of atomic permissions. In our setting permissions are not atomic, but are built from partially ordered components, which makes set-theoretic union and (especially) intersection unsuitable. As an example, the greatest permission implied by both FileIO(Owns,Just(logFile)) and FileIO(Read,Any) is FileIO(Read,Just(logFile)), rather than the empty permission.

For this reason, we need to generalize the usual model of working directly with the lattice of (finite) subsets of an atomic permission set.

Note the contrast to RBAC, where the set of roles is partially ordered and the permissions of a lesser role automatically accrue to all greater roles. Structured permissions should still be useful in RBAC, since computations on permission sets could be used to define and check constraints.

To represent and calculate with structured permissions such as wildcards, we use a simple theory of lattices. Generalizing from the example above, we start with a partially ordered set $(P, \leqslant)$ of permissions, where $p \leqslant q$ iff holding permission q implies that we also hold p. In this setting, the permissions at any point when running a program form a downward closed subset of P. Since such sets can be infinite, we represent them by the cochain of their maximal elements, which we require to be finite.

Definition 1 (*Finite Lower Bounds*)

If $ps \subseteq P$, we write $\downarrow ps \triangleq \{p \mid \exists q \in ps.\ p \leqslant q\}$ for the *downward closure* of ps. If $ps = \downarrow ps$, we say that ps is *downward closed*. The *maximal elements* of a set $ps \subseteq P$ is $maxs(ps) \triangleq \{q \in ps \mid \forall p \in ps.\ q \leqslant p \Rightarrow q = p\}$. ps is a *cochain* (of P) if $ps = maxs(ps)$. The *maximal lower bounds* of $q, r \in P$ is $mlbs(q,r) \triangleq maxs(\{p \in P \mid p \leqslant q \wedge p \leqslant r\})$. P has *finite lower bounds* (FLB) iff $mlbs(q,r)$ is finite for all for all $q, r \in P$.

In the following, we assume that P has FLB. We let $\mathcal{O}_{\text{fin}}(P)$ be the set of finite cochains of P, and define a lattice structure on $\mathcal{O}_{\text{fin}}(P)$ as follows. The greatest lower bound of ps and qs is $ps \sqcup qs \triangleq maxs((\downarrow ps) \cup (\downarrow qs))$, and the least upper bound of ps and qs is $ps \sqcap qs \triangleq maxs((\downarrow ps) \cap (\downarrow qs))$. We write $ps \sqsubseteq qs$ iff $\downarrow ps \subseteq \downarrow qs$, "*ps is subsumed by qs*".

If all cochains in a poset P are finite, then P trivially has FLB. In the example above, string Wildcard has FLB, but infinite cochains. As a common special case, if $(P, \leqslant)$ forms a forest with the maximal elements at the roots then P has FLB and $mlbs(p,q)$ is the smaller of p and q (with respect to $\leqslant$), or empty if they are incomparable. Returning to the running example, where the permissions form a forest, we have that $mlbs(\mathsf{Owns}, \mathsf{Read}) = \{\mathsf{Read}\}$ and $mlbs(\mathsf{Any}, \mathsf{Just}(\mathsf{logFile})) = \{\mathsf{Just}(\mathsf{logFile})\}$.

The greatest lower bound (glb) of two permission sets ps and qs subsumes precisely those sets subsumed by both ps and qs. Dually, the least upper bound (lub) of two permission sets ps and qs is the smallest set subsuming both ps and qs. We can compute the results of these lattice operations as follows.

Proposition 2 (*Computing lattice operations*)

If P has FLB and $ps, qs \in \mathcal{O}_{\text{fin}}(P)$ then $ps \sqcup qs = maxs(ps \cup qs)$ and $ps \sqcap qs = maxs(\bigcup\{mlbs(p,q) \mid p \in ps, q \in qs\})$. Moreover, $ps \sqsubseteq qs$ iff $ps \subseteq \downarrow qs$.

Assume that P and Q both have FLB. Then $P \times Q$ has FLB, with $mlbs((p_1,q_1),(p_2,q_2)) = mlbs(p_1,p_2) \times mlbs(q_1,q_2)$. Furthermore, if P and Q are disjoint then $P \cup Q$ also has FLB, with $\mathcal{O}_{\text{fin}}(P \cup Q)$ isomorphic to $\mathcal{O}_{\text{fin}}(P) \times \mathcal{O}_{\text{fin}}(Q)$.

Proof

Note that $maxs$ and $\downarrow$ are idempotent, $maxs(\downarrow ps) = maxs(ps)$ and $\downarrow maxs(ps) = \downarrow ps$. The intersection of two downward closed sets is itself downward closed, and $\downarrow$ distributes over $\cup$ and $\times$. If ps and qs are both cochains and $\downarrow ps = \downarrow qs$, then $ps = qs$.

- $ps \sqcup qs = maxs((\downarrow ps) \cup (\downarrow qs)) = maxs(\downarrow(ps \cup qs)) = maxs(ps \cup qs)$.
- $\downarrow(ps \sqcap qs) = \downarrow maxs((\downarrow ps) \cap (\downarrow qs)) = \downarrow((\downarrow ps) \cap (\downarrow qs)) = (\downarrow ps) \cap (\downarrow qs)$ since $(\downarrow ps) \cap (\downarrow qs)$ is downward closed. Then $r \in (\downarrow ps) \cap (\downarrow qs)$ iff $\exists q \in ps, r \in qs$ such that $r \leqslant p$ and $r \leqslant q$; that is, iff $r \in \downarrow mlbs(p,q)$. Thus $\downarrow(ps \sqcap qs) = \bigcup\{\downarrow mlbs(p,q) \mid$

$p \in ps, rq \in qs\} = \downarrow\bigcup\{mlbs(p,q) \mid p \in ps, q \in qs\}$. $maxs\downarrow(ps \sqcap qs) = ps \sqcap qs$ since $ps \sqcap qs$ is a cochain, so $ps \sqcap qs = maxs(\bigcup\{mlbs(p,q) \mid p \in ps, q \in qs\})$.

- $\downarrow mlbs((p_1,p_2),(q_1,q_2)) = \{(r_1,r_2) \mid r_i \leqslant p_i \wedge r_i \leqslant q_i$ for $i = 1,2\} = \{(r_1,r_2) \mid r_i \in \downarrow mlbs(p_i,q_i)$ for $i = 1,2\} = (\downarrow mlbs(p_1,q_1)) \times (\downarrow mlbs(p_2,q_2)) = \downarrow(mlbs(p_1,q_1) \times mlbs(p_2,q_2))$. $mlbs(p_1,q_1) \times mlbs(p_2,q_2)$ is a cochain since $mlbs(p_1,q_1)$ and $mlbs(p_2,q_2)$ are. Thus $mlbs((p_1,p_2),(q_1,q_2)) = mlbs(p_1,q_1) \times mlbs(p_2,q_2)$.
- Since P and Q are disjoint we have that p and q are incomparable if $p \in P$ and $q \in Q$. Thus, if $ps \subseteq P$ and $qs \subseteq Q$ then $\downarrow ps \cap \downarrow qs = \emptyset$ and $maxs(ps \cup qs) = maxs(ps) \cup maxs(qs)$.
  We first show that $P \cup Q$ has FLB. Take $p,q \in P \cup Q$. By symmetry we may assume that $p \in P$. If $q \in P$ then $mlbs(p,q)$ is finite by assumption. If $q \in Q$ then $mlbs(p,q) = maxs(\downarrow\{p\} \cap \downarrow\{q\}) = maxs(\emptyset) = \emptyset$, which is finite.
  The function $f : \mathcal{O}_{\text{fin}}(P) \times \mathcal{O}_{\text{fin}}(Q) \to \mathcal{O}_{\text{fin}}(P \cup Q)$, where $f(ps,qs) \triangleq ps \cup qs$ is an isomorphism iff f is a bijection that preserves least upper and greatest lower bounds. Since P and Q are disjoint, f is injective. Moreover, every finite cochain $rs \subseteq (P \cup Q)$ can be written as $f(ps,qs)$, where $ps = rs \cap P$ and $qs = rs \cap Q$ are finite cochains (in P resp. Q), so f is surjective.
  Fix $ps, ps' \in \mathcal{O}_{\text{fin}}(P)$ and $qs, qs' \in \mathcal{O}_{\text{fin}}(Q)$. We have that $f(ps \sqcup ps', qs \sqcup qs') = ps \sqcup ps' \cup qs \sqcup qs' = maxs(\downarrow ps \cup \downarrow ps') \cup maxs(\downarrow qs \cup \downarrow qs') = maxs(\downarrow ps \cup \downarrow ps' \cup \downarrow qs \cup \downarrow qs') = maxs(\downarrow(ps \cup qs) \cup \downarrow(ps' \cup qs')) = f(ps,qs) \sqcup f(ps',qs')$. Finally, $f(ps \sqcap ps', qs \sqcap qs') = ps \sqcap ps' \cup qs \sqcap qs' = maxs(\downarrow ps \cap \downarrow ps') \cup maxs(\downarrow qs \cap \downarrow qs') = maxs((\downarrow ps \cap \downarrow ps') \cup (\downarrow qs \cap \downarrow qs')) = maxs((\downarrow ps \cup \downarrow qs) \cap (\downarrow ps' \cup \downarrow qs')) = maxs(\downarrow(ps \cup qs) \cap \downarrow(ps' \cup qs')) = f(ps,qs) \sqcap f(ps',qs')$. □

We can now compute that

$$\{\text{FileIO(Owns,Just(logFile))}\} \sqcap \{\text{FileIO(Read,Any)}\} = \{\text{FileIO(Read,Just(logFile))}\},$$

as desired.

We encode the partial order on permissions as a predicate Holds(p,ps) that checks if a permission p is in the downward closure of the permission set ps. We define the predicate Subsumed in term of Holds.

Predicate symbols and their definitions:

assume ∀x,y,xs. Holds(FileIO(Owns,y),xs) ⇒ Holds(FileIO(x,y),xs)
assume ∀x,y,xs. Holds(FileIO(x,Any),xs) ⇒ Holds(FileIO(x,Just(y)),xs)
assume ∀x,xs. Holds(x,x::xs)
assume ∀x,y,xs. Holds(x,xs) ⇒ Holds(x,y::xs)
assume ∀xs. Subsumed(xs,xs) ∧ Subsumed([],xs)
assume ∀x,xs,ys. Holds(x,ys) ∧ Subsumed(xs,ys) ⇒ Subsumed(x::xs,ys)

We also define predicates for Lub and Glb, with the intended meaning that Lub(x,y,z) holds iff x is the least upper bound of y and z (and idem. for Glb). As an incomplete axiomatization, we assume standard lattice axioms relating these predicates to each other and to Subsumed.

Lattice axioms:

```
// Glb is the greatest lower bound ; Lub is the least upper bound.
assume (∀x,y,z. Glb(x,y,z) ⇒ Subsumed(x,y))
assume (∀x,y,z. Lub(x,y,z) ⇒ Subsumed(y,x))
assume (∀x,y,z,t. Subsumed(t,x) ∧ Subsumed(t,y) ∧ Glb(z,x,y) ⇒ Subsumed(t,z))
assume (∀x,y,z,t. Subsumed(x,t) ∧ Subsumed(y,t) ∧ Lub(z,x,y) ⇒ Subsumed(z,t))
// Glb/Lub of comparable elements
assume (∀x,y. Subsumed(y,x) ⇒ Glb(y,x,y))
assume (∀x,y. Subsumed(y,x) ⇒ Lub(x,x,y))
// Associativity and Transitivity
assume (∀x,y,z. Glb(x,y,z) ⇒ Glb(x,z,y))
assume (∀x,y,z. Lub(x,y,z) ⇒ Lub(x,z,y))
assume (∀x,y,z. Subsumed(x,y) ∧ Subsumed(y,z) ⇒ Subsumed(x,z))
```

We then assume functions lub, glb and subsumed that compute the corresponding operations for the permission language defined above, with the following types.

Types for lattice operations:

```
val lub: ps:Perms → qs:Perms → {(s) True} res:Perms {(t) s=t ∧ Lub(res,ps,qs)}
val glb: ps:Perms → qs:Perms → {(s) True} res:Perms {(t) s=t ∧ Glb(res,ps,qs)}
val subsumed: ps:Perms → qs:Perms →
                {(s) True} x:bool {(t) s=t ∧ (x=True ⇔ Subsumed(ps,qs))}
```

3.2 Stack-based access control

In order to compare HBAC and SBAC in the same framework, we begin by implementing API functions for requesting and testing permissions. We let state be a record type with two fields: **type** state = {ast:Perms; dy:Perms}. The ast field contains the current static permissions, which are used only when requesting additional dynamic permissions (see request below). The dy field contains the current dynamically requested permissions. Computations have type (α ;req) SBACcomp, for some return type α and required initial dynamic permissions req. An SBACthunk wraps a computation in a function with unit argument type (a thunk); this abbreviation is introduced to simplify some latter examples.

The API functions have the following types and implementations. The become function is used (notionally by the run-time system) when calling a function that may have different static permissions from its caller. It first sets the static permissions to those of the called code. Then, since the called function may be untrusted, it reduces the dynamic permissions to the greatest lower bound of the current dynamic permissions and the static permissions of the called function. Dually, upon return the run-time system calls sbacReturn with the original permissions returned by become, restoring them. The request function augments the dynamic permissions, after checking that the static context (Subsumed(ps,st)) permits it. We check that the permissions ps dynamically hold using the function demand; it has type ps:Perms → (unit;ps) SBACcomp.

SBAC API and calling convention:

```
type (α ;req:Perms) SBACcomp = {(s) Subsumed(req,s.dy)} α {(t) s=t}
type (α ;req:Perms) SBACthunk = unit → (α ;req) SBACcomp
val become: ps:Perms→ {(s)True}s':State{(t) s=s' ∧ t.ast = ps ∧ Glb(t.dy,ps,s.dy)}
val sbacReturn: olds:State → {(s) True} unit {(t) t=olds}
val permitOnly: ps:Perms→ {(s) True}unit{(t) s.ast = t.ast ∧ Glb(t.dy,ps,s.dy)}
val request: ps:Perms →
              {(s) Subsumed(ps,s.ast)} unit {(t) s.ast = t.ast ∧ Lub(t.dy,ps,s.dy)}
val demand: ps:Perms → (unit;ps) SBACcomp
```

The postcondition of an SBACcomp is that the state is unchanged. In order to recover formulas that hold about the state, we use subtyping. As usual, a subtype of a function type may return a subtype of the original computation type. In a subtype G of a computation type F, we can strengthen the precondition. The postcondition of G must also be weaker than (implied by) the precondition of G together with the postcondition of F. As an example, $\{(s)C\}\alpha\{(t)C\{^t/_s\}\}$ is a subtype of (α ;[])SBACcomp for every C, since $\vdash C \Rightarrow$ Subsumed([],s.dy) and $\vdash (C \wedge s = t) \Rightarrow C\{^t/_s\}$. Subtyping is used to ensure that pre- and postconditions match up when sequencing computations using **let**. We also use subtyping to propagate assumptions that do not mention the state, such as the definitions of predicates.

This API is used for code-based access control by inserting a become P call at the start of every function with provenance P, and a call to sbacReturn where the function returns. In the implementations of request and demand below, we **assert** that subsumed returns **true**. The safety rheorem for RIF states that in a well-typed program, assertions never fail at runtime. In the case of request, this corresponds to the condition that no function ever requests permissions that its provenance does not allow. In the case of demand, this corresponds to the condition that the necessary permissions ps have been requested earlier in the call stack, and that all later stack frames have static permissions at least ps.

SBAC API implementation:

```
let sbacReturn s = set s
let become ps =
   let {ast=st;dy=dy} = get() in let dz = glb ps dy in
   set {ast=ps;dy=dz}; {ast=st;dy=dy}
let permitOnly ps =
  let {ast=st;dy=dy} = get() in let dz = glb ps dy in set ({ast=st;dy=dz})
let request ps =
   let {ast=st;dy=dy} = get() in let x = subsumed ps st in
   if x then let dz = lub ps dy in set {ast=st; dy=dz}
   else assert False ; failwith "SecurityException: request"
let demand ps =
   let {ast=_; dy=dy} = get() in let x = subsumed ps dy in
   if x then () else assert False; failwith "SecurityException: demand"
```

To exercise this framework, we work in a setting with two principals. Applet is untrusted, and can perform screen IO, read a version file and owns a temporary file. System can read and write every file. We define three trusted functions, that either run primitive (non-refined) functions or run as System. Function readFile demands that the read permission for its argument holds dynamically. Similarly deleteFile requires a write permission. Finally, cleanupSBAC takes a function returning a filename, and then deletes the file returned by the function.

```
let Applet = [ScreenIO;FileIO(Read,Just(version));FileIO(Owns,Just(tempFile))]
let System = [ScreenIO;FileIO(Write,Any);FileIO(Read,Any)]

val readFile: a:string → (string;[FileIO(Read,Just(a))]) SBACcomp
let readFile n = let olds = become System in demand [FileIO(Read,Just(n))] ;
  let res = "Content of "^n in sbacReturn olds; res

val deleteFile: a:string → (string;[FileIO(Write,Just(a))]) SBACcomp
let deleteFile n = let olds = become System in demand [FileIO(Write,Just(n))] ;
  let res = primitiveDelete n in sbacReturn olds; res

val cleanupSBAC: (string;[]) SBACthunk → (unit;[]) SBACcomp
let cleanupSBAC f = let olds = become System in request [FileIO(Write,Any)];
  let s = f () in let res = deleteFile s in sbacReturn olds; res
```

We now give some examples of untrusted code using these trusted functions and the SBAC calling conventions. In SBAC1, an applet attempts to read the version file. Since Applet has the necessary permission, this function is well typed at type (unit;[]) SBACcomp.

```
let SBAC1: (unit;[]) SBACthunk = fun () → let olds = become Applet in
  request [FileIO(Read,Just(version))]; readFile version; sbacReturn olds
```

In SBAC2, the applet attempts to delete a password file. Since the applet does not have the necessary permissions, a runtime exception is thrown when executing the code—and we cannot type the function SBAC2 at type (unit;[]) SBACcomp.

```
 //Does not typecheck
let SBAC2 = fun () → let olds = become Applet in
  request [FileIO(Read,Just("passwd"))]; deleteFile "passwd"; sbacReturn olds
```

However, in SBAC3, the SBAC abstraction fails to protect the password file. Here the applet instead passes an untrusted function to cleanup. Since the permissions are reset after returning from the untrusted function, the cleanup function deletes the password file. This is a variant of the *confused deputy problem* (Hardy 1988) that is common to SBAC mechanisms. Moreover, SBAC3 type-checks.

```
let aFunSBAC: (string;[]) SBACthunk = fun () → let olds = become Applet in
  let res = "passwd" in sbacReturn olds; res
let SBAC3: (unit;[]) SBACthunk = fun () → let olds = become Applet in
  cleanupSBAC aFunSBAC; sbacReturn olds
```

3.3 History-based access control

The HBAC calling convention was defined (Abadi & Fournet 2003) to protect against the kind of attack that SBAC fails to prevent in SBAC3 above. To protect callers from untrusted functions, HBAC reduces the dynamic permissions after calling an untrusted function. This is in contrast to the SBAC mechanism, where the dynamic permissions are restored. Moreover, extensions of HBAC can be used to enforce history-dependent properties such as Chinese Walls (a program must not access both of two conflicting sets of data). Abadi and Fournet argue that a continuation-passing transform would allow a HBAC policy to be checked in an SBAC setting; intutively, this works because programs in CPS never return from a function, so the dynamic permissions are never restored. (We do not attempt to prove this result.)

In our formalization, a computation in HBAC of type (α ;req,pres) HBACcomp returning type α preserves the static permissions and does not increase the dynamic permissions. It also requires permissions req and preserves permissions pres. As above, a HBACthunk is a function from unit returning an HBACcomp. Here (α ;req) SBACcomp is a subtype of (α ;req,pres) HBACcomp for every pres. The HBAC calling convention is implemented by the function hbacReturn, which resets the static permissions and reduces the dynamic permissions to at most the initial ones.

HBAC API and calling convention (Part 1):

```
type (α ;req:Perms,pres:Perms) HBACcomp = {(s) Subsumed(req,s.dy) } α
    {(t) s.ast = t.ast ∧ Subsumed(t.dy,s.dy)
        ∧ (∀qs. Subsumed(qs,pres) ∧ Subsumed(qs,s.dy) ⇒ Subsumed(qs,t.dy))}
type (α ;req:Perms,pres:Perms) HBACthunk = unit → (α ;req,pres) HBACcomp
val hbacReturn: os:State → {(s) True} unit {(t) t.ast=os.ast ∧ Glb(t.dy,s.dy,os.dy)}
```

The HBAC API includes the functions become, permitOnly, request, and demand from the SBAC API, but also two functions for structured control of permissions, grant and accept, which can be seen as scoped versions of request. We use grant to run a subcomputation with augmented permissions. The second argument to grant ps is a (α ;ps,[]) HBACthunk, which may assume that the permissions ps hold upon entry. We can only call grant itself if the current static permissions subsume ps. Dually, accept allows us to recover permissions that might have been lost when running a subcomputation. A call accept ps takes an HBACthunk that does not require any dynamic permissions, and guarantees that at least the glb (intersection) between ps and the initial dynamic permissions holds upon exit. As before, we can only call accept if the current static permissions subsume ps.

HBAC API and calling convention (Part 2):

```
val grant: ps:Perms → (α ;ps,[]) HBACthunk → {(s) Subsumed(ps,s.ast)} α
   {(t) t.ast=s.ast ∧ Subsumed(t.dy,s.dy)}
val accept: ps:Perms → (α ;[],[]) HBACthunk → {(s) Subsumed(ps,s.ast)} α
   {(t) s.ast = t.ast ∧ Subsumed(t.dy,s.dy) ∧
        (∀qs. Subsumed(qs,ps) ∧ Subsumed(qs,s.dy) ⇒ Subsumed(qs,t.dy))}
```

Note that grant and accept are higher-order functions. As given above, their types are conservative in that they assume that their thunk arguments neither preserve nor, in the case of accept, require any permissions. For more precise typing, we could parameterize these functions with these permissions; we have not developed this approach, in part because these parameters would be needed only for the purpose of type-checking.

HBAC API implementation:

```
let hbacReturn s = let {ast=oldst; dy=oldy} = s in let {ast=st;dy=dy} = get() in
    let dz = glb dy oldy in set {ast=oldst;dy=dz}
let grant ps a = let {ast=_;dy=dy} = get() in request ps; let res = a () in
      permitOnly dy; res
let accept ps a = let {ast=_;dy=dy} = get() in let res = a () in request ps;
      permitOnly dy; res
```

As seen above, SBAC computations have no effect on the dynamic permissions after they have returned. In the case where System is the maximal permission, we have that the type (string;[]) SBACthunk (the argument type of cleanupSBAC) corresponds to (string;[],System) HBACthunk. Below, we repeat the successful attack on SBAC in the stricter history-based setting. Here, we cannot type-check the attack code, and the attack fails because of insufficient dynamic permissions at the call to deleteFile.

```
type cleanupArg: (string;[],System) HBACthunk
val cleanupHBAC: cleanupArg → (unit;[],System) HBACthunk

let cleanupHBAC f = let olds = become System in
request [FileIO(Write,Any)]; let s = f () in deleteFile s ; hbacReturn olds

let aFunHBAC: (string;[],[]) HBACthunk = fun () → let olds = become Applet in
  let res = "passwd" in hbacReturn olds; res

//Does not type-check, since aFunHBAC is not a cleanupArg
let HBAC1 = fun () → let olds = become Applet in
    cleanupHBAC aFunHBAC ; hbacReturn olds
```

On the other hand, cleanupHBAC will delete the given file if the function it calls preserves the relevant write permission. This can cause a vulnerability. For instance, assume a library function expand that (notionally) expands environment variables in its argument. Such a library function would be statically trusted, and passing it to cleanup_HBAC will result in the sensitive file being deleted. Moreover, we can type-check expand at type string →cleanupArg, where a cleanupArg preserves all System permissions, including FileIO(Write,Just("passwd")), when run.

```
let expand:string →cleanupArg = fun n →fun () →
 let olds = become System in let res = n in hbacReturn olds ; n

let HBAC2:(unit;[],[]) HBACthunk = fun () →let olds = become Applet in
   cleanup_HBAC (expand "passwd") ; hbacReturn olds
```

In HBAC, the functions grant and accept are preferred over request. As an example, we define a function cleanup_grant. This function prudently checks the return value of its untrusted argument, and uses grant to give precisely the required permission to deleteFile. If the check fails, we instead give an error message (not to be confused with a security exception). For this reason, HBAC3 type-checks.

```
let cleanup_grant: (string;[],[]) HBACthunk →(unit;[],[]) HBACcomp =
  fun f →let olds = become System; let s = f () in
  (if (s = tempFile) then let h = deleteFile s in grant [FileIO(Write,Just(s))] h
   else print "Check of untrusted return value failed.");
  hbacReturn olds

let aFunHBAC: (string;[],[]) HBACthunk = fun () →
   let olds = become Applet in let res = "passwd" in hbacReturn olds ; res

let HBAC3: (unit;[],[]) HBACthunk = fun () →
   let olds = become Applet in cleanup_grant aFunHBAC ; hbacReturn olds
```

Here HBAC provides a middle ground when compared to SBAC on the one hand and taint-tracking systems on the other, in regards to accuracy and complexity.

In the examples above, well-typed code does not depend on the actual state in which it is run. Indeed, we could dispense with the state-passing entirely, keeping the state as a “ghost” or specification variable. However, we can also introduce a function which lets us check if we hold certain run-time permissions. When this function is part of the API, we need to keep a concrete permission state (in the general case).

API function for checking run-time permissions:

```
val check: ps:Perms →{(s) True} b:bool {(t) s=t ∧(b=true ⇒Subsumed(ps,t.dy))}
let check ps = let {ast=_;dy=dy} = get() in subsumed ps dy
```

We can use this function in the following (type-safe) way.

```
let HBAC4:(unit;[],[]) HBACthunk = fun () → let olds = become Applet in
  (if check [FileIO(Write,Just("passwd"))]
   then deleteFile "passwd"
   else print "Not enough permissions: giving up.");
  hbacReturn olds
```

In future work, it would be interesting to verify the correctness of our encoding of SBAC and HBAC. One direction would be to establish a relationship between code written using the SBAC API and the calculus of Pottier *et al.* (2005). More precisely, we conjecture that it is possible to translate programs in the latter calculus into RIF programs using the SBAC API in such a way that both the operational meaning of the program is preserved and the original program type-checks if and only if its translation type-checks in RIF. An analogous result may apply for HBAC, but to state it we need first to devise a formal calculus of HBAC.

4 A Calculus for the refined state monad

In this section, we present the formal definition of RIF, the calculus we have been using to model security mechanisms based on roles, stacks and histories. We begin with its assertion logic, syntax and operational semantics in Sections 4.1, 4.2 and 4.3. Section 4.4 describes the type system of RIF and its soundness with respect to the operational semantics. Finally, Section 4.5 describes how the calculus may be instantiated by suitable choice of the state type.

4.1 Logic

Formally, RIF, like RCF (Bengtson *et al.* 2008), is parameterized by an *authorization logic*, which is specified by a set of formulas C, and a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas. For the purposes of this paper, we assume the logic is FOL/FO, which is first-order logic together with axiom schemes for the disjointness and injectivity of the syntactic constructors of closed first-order RIF values (that is, values that do not contain functions).

Syntax of formulas:

p	predicate symbol
$C ::=$	formula
$\quad p(M_1,\dots,M_n)$	atomic formula, M_i first order
$\quad M = M'$	equation, M and M' first order
$\quad C \wedge C'$	conjunction
$\quad C \vee C'$	disjunction
$\quad \neg C$	negation
$\quad \forall x.C$	universal quantification
$\quad \exists x.C$	existential quantification

$\mathsf{True} \triangleq () = ()$ $\quad$ $\mathsf{False} \triangleq \neg\mathsf{True}$ $\quad$ $M \neq M' \triangleq \neg(M = M')$ $\quad$ $(C \Rightarrow C') \triangleq (\neg C \vee C')$
$(C \Leftrightarrow C') \triangleq (C \Rightarrow C') \wedge (C' \Rightarrow C)$

We identify all phrases of syntax up to the consistent renaming of bound variables. In general, we write $\phi\{\psi/x\}$ for the outcome of substituting the phrase ψ for each free occurrence of the variable x in the phrase ϕ. We write $\mathrm{fv}(\phi)$ for the set of variables occurring free in the phrase ϕ.

In general, an authorization logic must at least satisfy the following rules. The choice of rules is driven by the type safety proof of RCF, as discussed by Bengtson *et al.* (2008). The rules are mostly standard. The rules (Eq) and (Ineq) require that the logic can deduce equations and inequations between ground values. The rule (Ineq Cons) additionally states that if M is closed and its outer constructor is not h, then $\forall x. h\, x \neq M$ is deducible.

Minimal properties of deducibility of an authorization logic: $S \vdash C$

S, C stands for $S \cup \{C\}$; in (Subst), σ ranges over substitutions of values for variables and permutations of names.

(Axiom) $$\frac{}{C \vdash C}$$

(Mon) $$\frac{S \vdash C}{S, C' \vdash C}$$

(Subst) $$\frac{S \vdash C}{S\sigma \vdash C\sigma}$$

(Cut) $$\frac{S \vdash C \quad S, C \vdash C'}{S \vdash C'}$$

(And Intro) $$\frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1}$$

(And Elim) $$\frac{S \vdash C_0 \wedge C_1}{S \vdash C_i}$$

(Or Intro) $$\frac{S \vdash C_i}{S \vdash C_0 \vee C_1}\ i = 0, 1$$

(Exists Intro) $$\frac{S \vdash C\{M/x\}}{S \vdash \exists x. C}$$

(Exists Elim) $$\frac{S \vdash \exists x. C \quad S, C \vdash C' \quad x \notin \mathrm{fv}(S, C')}{S \vdash C'}$$

(Eq) $$\frac{}{\varnothing \vdash M = M}$$

(Ineq) $$\frac{M \neq N \quad \mathrm{fv}(M, N) = \varnothing}{\varnothing \vdash M \neq N}$$

(Ineq Cons) $$\frac{h\ N = M \text{ for no } N \quad \mathrm{fv}(M) = \varnothing}{\varnothing \vdash \forall x. h\, x \neq M}$$

FOL/FO consists of the standard inference rules of first-order logic together with the following additional axiom schemas. The intended universe of FOL/FO is the free algebra of the constructors of closed first-order values. A *syntactic* function symbol is one used to represent such a value as a logical term. The rules below state that these functions are injective and that they have disjoint ranges.

Additional rules for FOL/FO:

(F Disjoint) $$\frac{f \neq f' \text{ syntactic}}{S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) \neq f'(\vec{y})}$$

(F Injective) $$\frac{f \text{ syntactic}}{S \vdash \forall \vec{x}. \forall \vec{y}. f(\vec{x}) = f(\vec{y}) \Rightarrow \vec{x} = \vec{y}}$$

For a more detailed discussion of authorization logics and the particular logic FOL/FO, see Bengtson *et al.* (2008). (Our work in this paper relies on the typechecker F7, and implementation of RCF for F#. F7 uses the Z3 SMT solver, after loading a background theory containing instances of the rules (F Disjoint) and

(F Injective) as universally quantified formulas. Future versions of F7 may exploit Z3's primitives for algebraic types.)

4.2 Syntax

Our starting point is the Fixpoint Calculus (FPC) (Plotkin 1985; Gunter 1992), a deterministic call-by-value λ-calculus with sums, pairs and iso-recursive data structures.

Syntax of the core Fixpoint Calculus:

s, x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction (an application of a value constructor)
$A, B ::=$	expression
M	value
$M\ N$	application
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)
let $(x, y) = M$ **in** A	pair split (scope of x, y is A)
match M **with** $h\ x \to A$ **else** B	constructor match (scope of x is A)

A value may be a variable x, the unit value $()$, a function **fun** $x \to A$, a pair (M, N), or a construction. The constructions inl M and inr M are the two sorts of value of sum type, while the construction fold M is a value of an iso-recursive type. A *first-order* value is any value not containing any instance of **fun** $x \to A$.

In our formulation of FPC, the syntax of expressions is in a reduced form in the style of A-normal form (Sabry & Felleisen 1993), where sequential composition of redexes is achieved by inserting suitable let-expressions. The other expressions are function application $M\ N$, equality $M = N$ (which tests whether the values M and N are syntactically identical), pair splitting **let** $(x, y) = M$ **in** A, and constructor matching **match** M **with** $h\ x \to A$ **else** B.

To complete our calculus, we augment FPC with the following operations for manipulating and writing assertions about a global state. The state is implicit and is simply a value of the calculus.

Completing the syntax: Adding global state to the fixpoint calculus

$A, B ::=$	expression
$\cdots$	expressions of the Fixpoint Calculus
get()	get current state
set(M)	set current state
assume $(s)C$	assumption of formula C (scope of s is C)
assert $(s)C$	assertion of formula C (scope of s is C)

In programs, formulas C may *a priori* contain function values; this will be excluded by typing. A formula C is of first order if and only if it only contains first-order values. A collection S is of first order if and only if it only contains first-order formulas.

The expression **get**() returns the current state as its value. The expression **set**(M) updates the current state with the value M and returns the unit value ().

We specify intended properties of programs by embedding assertions, which are formulas expected to hold with respect to the *log*, a finite multiset of assumed formulas. The assumption expression **assume** $(s)C$ adds the formula $C\{^M/_s\}$ to the logged formulas, where M is the current state, and returns (). The assertion expression **assert** $(s)C$ immediately returns (); we say the assertion *succeeds* if the formula $C\{^M/_s\}$ is deducible from the logged formulas, and otherwise that it *fails*. This style of embedding assumptions and assertions within expressions is in the spirit of the pioneering work of Floyd, Hoare, and Dijkstra on imperative programs; the formal details are an imperative extension of assumptions and assertions in RCF (Bengtson *et al.* 2008).

We use some syntactic sugar to make it easier to write and understand examples. We write $A; B$ for **let** _ $= A$ **in** B. We define Boolean values as **false** $\triangleq$ inl () and **true** $\triangleq$ inr (). Conditional statements can then be defined as **if** M **then** A **else** $B \triangleq$ **match** M **with** $\mathsf{inr}\ x \to A$ **else** B. We write **let rec** $f\ x = A$ **in** B as an abbreviation for defining a recursive function f, where the scope of f is A and B, and the scope of x is A. When s does not occur in C, we simply write C for $(s)C$. In our examples, we often use a more ML-like syntax, lessening the A-normal form restrictions of our calculus. In particular, we use **let** $f\ x = A$ for **let** $f =$ **fun** $x \to A$, **if** A **then** B_1 **else** B_2 for **let** $x = A$ **in if** x **then** B_1 **else** B_2 (where $x \notin \mathrm{fv}(B_1, B_2)$), **let** $(x, y) = A$ **in** B for **let** $z = A$ **in let** $(x, y) = z$ **in** B (where $z \notin \mathrm{fv}(B)$), and so on. See Bengtson *et al.* (2008), for example, for a discussion of how to recover standard functional programming syntax and data types like Booleans and lists within the core Fixpoint Calculus.

4.3 Semantics

We formalize the semantics of our calculus as a small-step reduction relation on configurations, each of which is a triple (A, N, S) consisting of a closed expression A, a state N, and a log S, which is a multiset of formulas generated by assumptions. A configuration (A, N, S) is of first order if and only if N, S and all formulas occurring in A are of first order.

We present the rules for reduction in two groups. The rules in the first group are independent of the current state, and correspond to the semantics of core FPC.

Reductions for the core calculus: $(A, N, S) \longrightarrow (A', N', S')$

$\mathcal{R} ::= [\,] \mid \mathbf{let}\ x = \mathcal{R}\ \mathbf{in}\ A$	evaluation context
$(\mathcal{R}[A], N, S) \longrightarrow (\mathcal{R}[A'], N', S')$ if $(A, N, S) \longrightarrow (A', N', S')$	(Red Ctx)
$((\mathbf{fun}\ x \to A)\ M, N, S) \longrightarrow (A\{^M/_x\}, N, S)$	(Red Fun)
$(M_1 = M_2, N, S) \longrightarrow (\mathbf{true}, N, S)$ if $M_1 = M_2$	(Red Eq)
$(M_1 = M_2, N, S) \longrightarrow (\mathbf{false}, N, S)$ if $M_1 \neq M_2$	(Red Neq)
$(\mathbf{let}\ x = M\ \mathbf{in}\ A, N, S) \longrightarrow (A\{^M/_x\}, N, S)$	(Red Let)
$(\mathbf{let}\ (x, y) = (M_1, M_2)\ \mathbf{in}\ A, N, S) \longrightarrow (A\{^{M_1}/_x\}\{^{M_2}/_y\}, N, S)$	(Red Split)
$(\mathbf{match}\ (h\ M)\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B, N, S) \longrightarrow (A\{^M/_x\}, N, S)$	(Red Match)
$(\mathbf{match}\ (h'\ M)\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B, N, S) \longrightarrow (B, N, S)$ if $h \neq h'$	(Red Mismatch)

The second group of rules formalizes the semantics of assumptions, assertions and the get and set operators, described informally in the previous section.

Reductions related to state: $(A, N, S) \longrightarrow (A', N', S')$

$(\mathbf{get}(), N, S) \longrightarrow (N, N, S)$	(Red Get)
$(\mathbf{set}(M), N, S) \longrightarrow ((), M, S)$	(Red Set)
$(\mathbf{assume}\ (s)C, N, S) \longrightarrow ((), N, S \cup \{C\{^N/_s\}\})$	(Red Assume)
$(\mathbf{assert}\ (s)C, N, S) \longrightarrow ((), N, S)$	(Red Assert)

Intuitively, we say an expression is safe if none of its assertions may fail at runtime. Recall that an assertion fails when its formula is not deducible from the logged formulas. An assertion failure has no effect during execution. This may appear strange, but first note that our type system guarantees that such assertion failures will never occur during execution. Second, because our operational semantics is meant to reflect our implementation behavior, this means that we do not need to establish the deducibility of formulas during execution—which is undecidable for the logic we are considering.

To define safety formally, we say that a configuration (A, N, S) has *failed* when $A = \mathcal{R}[\mathbf{assert}\ (s)C]$ for some evaluation context $\mathcal{R}$, where $S \cup \{C\{^N/_s\}\}$ is not of first order or we cannot derive $S \vdash C\{^N/_s\}$. A configuration (A, N, S) is *safe* if and only if there is no failed configuration reachable from (A, N, S), that is, for all (A', N', S'), if $(A, N, S) \longrightarrow^* (A', N', S')$ then (A', N', S') has not failed. The safety of a (first-order) configuration can always be assured by carefully chosen assumptions (for example, **assume** (s)False). For this reason, user code should use assumptions with prudence (and possibly not at all).

The purpose of the type system in the next section is to establish safety by typing.

4.4 Types

There are two categories of type: *value types* characterize values, while *computation types* characterize the imperative computations denoted by expressions. Computation types resemble Hoare triples, with preconditions and postconditions.

Syntax of value types and computation types:

$T, U, V ::=$	(value) type
α	type variable
unit	unit type
$\Pi x : T.\ F$	dependent function type (scope of x is F)
$\Sigma x : T.\ U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
$F, G ::=$	computation type
$\{(s_0)C_0\}\ x{:}T\ \{(s_1)C_1\}$	(scope of s_0 is C_0, T, C_1, and scope of s_1, x is C_1)

Value types are based on the types of the Fixpoint Calculus, except that function types $\Pi x : T.\ F$ and pair types $\Sigma x : T.\ U$ are dependent. In our examples we use the F7-style notations $x : T \rightarrow F$ and $x : T * U$ instead of $\Pi x : T.\ F$ and $\Sigma x : T.\ U$. If the bound variable x is not used, these types degenerate to simple types. In particular, if x is not free in U, we write $T * U$ for $x : T * U$, and if x is not free in F, we write $T \rightarrow F$ for $x : T \rightarrow F$. A value type T is of *first order* if and only if T contains no occurrences of $\Pi x : U.\ F$ (and hence contains no computation types). (It follows that the values of a first-order type are themselves first-order values, that is, they contain no function values.) For the type $\Pi x : T.\ F$ to be well formed, we require that either T is a first-order type or that x is not free in F. Similarly, for the type $\Sigma x : T.\ U$ to be well formed, we require that either T is a first-order type or that x is not free in U.

A computation type $\{(s_0)C_0\}\ x{:}T\ \{(s_1)C_1\}$ means the following: if an expression has this type and it is started in an initial state s_0 satisfying the precondition C_0, and it terminates in final state s_1 with an answer x, then postcondition C_1 holds. As above, we write $\{(s_0)C_0\}\ T\ \{(s_1)C_1\}$ for $\{(s_0)C_0\}\ x{:}T\ \{(s_1)C_1\}$ if x is not free in C_1. If T is not of first order, we require that x is not free in C_1.

When we write a type T in a context where a computation type is expected, we intend T as a shorthand for the computation type $\{(s_0)\mathsf{True}\}\ T\ \{(s_1)s_1 = s_0\}$. This is convenient for writing curried functions. Thus, the curried function type $x : T \rightarrow y : U \rightarrow F$ stands for $\Pi x : T.\ \{(s_0')\mathsf{True}\}\ \Pi y : U.\ F\ \{(s_1')s_1' = s_0'\}$.

Our calculus is parameterized by a type state representing the type of data in the state threaded through a computation, and which we take to be an abbreviation for a closed RIF type not involving function types – that is, a closed first-order type.

Our typing rules are specified with respect to *typing environments*, given as follows, which contain value types of variables, temporary subtyping assumptions for iso-recursive types and the names of the state variables in scope.

Syntax of typing environments:

$\mu ::=$	environment entry
$\quad \alpha <: \alpha'$	$\quad$ subtype ($\alpha \neq \alpha'$)
$\quad s$	$\quad$ state variable
$\quad x : T$	$\quad$ variable
$E ::= \varnothing \mid E, \mu$	environment

$$\mathrm{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\} \qquad \mathrm{dom}(s) = \{s\} \qquad \mathrm{dom}(x : T) = \{x\}$$
$$\mathrm{dom}(E, \mu) = \mathrm{dom}(E) \cup \mathrm{dom}(\mu) \qquad \mathrm{dom}(\varnothing) = \varnothing$$
$$\mathrm{fov}(E) = \{s \in E\} \cup \{x \in \mathrm{dom}(E) \mid (x : T) \in E,\ T \text{ is first order}\}$$

Our type system consists of several inductively defined judgments.

Judgments:

$E \vdash \diamond$	E is syntactically well formed
$E \vdash T$	in E, type T is syntactically well formed
$E \vdash F$	in E, type F is syntactically well formed
$E \vdash C$ fo	in E, formula C is of first order
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash F <: G$	in E, type F is a subtype of type G
$E \vdash M : T$	in E, value M has type T
$E \vdash A : F$	in E, expression A has computation type F

The rules defining these judgments are displayed in a series of groups. First, we describe the rules defining when environments, formulas, and value and computation types are well formed. An environment is well formed if its entries have pairwise disjoint domains. A formula is well formed if all its free variables have first-order type in the environment. A type is well formed if its free variables have first-order type in the environment.

Rules of well formedness:

(Env Empty)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env Entry)
$$\frac{E \vdash \diamond \quad \mathrm{fv}(\mu) \subseteq \mathrm{fov}(E) \quad \mathrm{dom}(\mu) \cap \mathrm{dom}(E) = \varnothing}{E, \mu \vdash \diamond}$$

(Form)
$$\frac{E \vdash \diamond \quad C \text{ is of first order} \quad \mathrm{fv}(C) \subseteq \mathrm{fov}(E)}{E \vdash C \text{ fo}}$$

(Env Type)
$$\frac{E \vdash \diamond \quad \mathrm{fv}(T) \subseteq \mathrm{fov}(E)}{E \vdash T}$$

First-order values may occur in types, but only within formulas; since our logic is untyped, these well-formedness rules need not constrain values occurring within types to be themselves well typed. We do constrain variables occurring in formulas to have first-order types, to ensure that substitution is defined.

Notice that the assumptions of the rule (Env Entry) are such that if $E, x : T \vdash \diamond$ is derivable with (Env Entry) then we also have $E \vdash T$ derivable with (Env Type). Hence, each type in a well-formed environment is itself well formed.

So as to have the same rules for those types that are of first order and those that are not, we define the notation (C if T fo) to mean the formula C, if T is of first order, and otherwise to mean the formula True.

General rules for expressions:

(Exp Return)

$$\frac{E, s_0 \vdash M : T \qquad E, s_0, x{:}T, s_1 \vdash ((x = M \text{ if } T \text{ fo}) \land s_0 = s_1) \text{ fo}}{E \vdash M : \{(s_0)\mathsf{True}\}\, x : T\, \{(s_1)(x = M \text{ if } T \text{ fo}) \land s_0 = s_1\}}$$

(Stateful Exp Let)

$$\frac{\begin{array}{l} E \vdash A : \{(s_0)C_0\}\, x_1{:}T_1\, \{(s_1)C_1\} \\ E, s_0, x_1 : T_1 \vdash B : \{(s_1)C_1\}\, x_2{:}T_2\, \{(s_2)C_2\} \\ \{s_1, x_1\} \cap \mathrm{fv}(T_2, C_2) = \varnothing \end{array}}{E \vdash \mathbf{let}\ x_1 = A\ \mathbf{in}\ B : \{(s_0)C_0\}\, x_2{:}T_2\, \{(s_2)C_2\}}$$

(Exp Eq)

$$\frac{\begin{array}{l} E \vdash M : T \quad E \vdash N : U \quad x \notin \mathrm{fv}(M, N) \quad E, s_0, s_1 \vdash C \text{ fo} \\ C = (s_0 = s_1) \land (x = \mathbf{true} \Leftrightarrow M = N \text{ if } \Sigma_{\_} : T.\ U \text{ fo}) \end{array}}{E \vdash M = N : \{(s_0)\mathsf{True}\}\, x{:}\mathsf{bool}\, \{(s_1)C\}}$$

In (Exp Return), when returning a value from a computation, the state is unchanged. If the type is a first-order type, we additionally record the returned value. In (Exp Eq), the return value of an equality test is refined with the logical formula expressing the test. The rule (Stateful Exp Let) glues together two computation types if the postcondition of the first matches the precondition of the second.

Assumptions and assertions:

(Exp Assume)

$$\frac{E, s_0, s_1 \vdash \diamond \qquad E, s_0 \vdash C \text{ fo}}{E \vdash \mathbf{assume}\ (s_0)C : \{(s_0)\mathsf{True}\}\, \mathsf{unit}\, \{(s_1)((s_0 = s_1) \land C)\}}$$

(Exp Assert)

$$\frac{E, s_0, s_1 \vdash \diamond \qquad E, s_0 \vdash C \text{ fo}}{E \vdash \mathbf{assert}\ (s_0)C : \{(s_0)C\}\, \mathsf{unit}\, \{(s_1)s_0 = s_1\}}$$

In (Exp Assume), an assumption **assume** $(s)C$ has C as postcondition, and does not modify the state. Dually, in (Exp Assert), an assertion **assert** $(s)C$ has C as precondition.

Rules for state manipulation:

(Stateful Get)

$$\frac{E, s_0, x_1 : \mathsf{state}, s_1 \vdash \diamond}{E \vdash \mathbf{get}() : \{(s_0)\mathsf{True}\}\, x_1{:}\mathsf{state}\, \{(s_1)x_1 = s_0 \wedge s_1 = s_0\}}$$

(Stateful Set)

$$\frac{E \vdash M : \mathsf{state} \quad E, s_0, s_1 \vdash \diamond}{E \vdash \mathbf{set}(M) : \{(s_0)\mathsf{True}\}\, \mathsf{unit}\, \{(s_1)s_1 = M\}}$$

In (Stateful Get), the type of **get**() records that the value read is the current state. In (Stateful Set), the postcondition of **set**(M) states that M is the new state. The postcondition of **set**(M) does not mention the initial state. We can recover this information through subtyping (see below).

Subtyping for computations:

(Sub Comp)

$$\frac{\begin{array}{l} E, s_0 \vdash C_0 \text{ fo} \quad \mathrm{E}, \mathrm{s}_0 \vdash \mathrm{C}'_0 \text{ fo} \\ E, s_0, x{:}T, s_1 \vdash C_1 \text{ fo} \quad \mathrm{E}, \mathrm{s}_0, \mathrm{x}{:}\mathrm{T}', \mathrm{s}_1 \vdash \mathrm{C}'_1 \text{ fo} \\ C'_0 \vdash C_0 \quad E, s_0 \vdash T <: T' \quad (C'_0 \wedge C_1) \vdash C'_1 \end{array}}{E \vdash \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\} <: \{(s_0)C'_0\}\, x{:}T'\, \{(s_1)C'_1\}}$$

(Exp Subsum)

$$\frac{E \vdash A : F \quad E \vdash F <: F'}{E \vdash A : F'}$$

In (Sub Comp), when computing the supertype of a computation type, we may strengthen the precondition, and weaken the postcondition relative to the strengthened precondition. For example, since $(C_0 \wedge C_1) \vdash (C_0 \wedge C_1)$, we have:

$$E \vdash \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\} <: \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_0 \wedge C_1\}$$

Next, we present rules grouped by the different forms of value type. When type-checking values, we may gain information about their structure. We record this information by adding it to the precondition of the computation that uses the data. To do so, we define the notation $C \leadsto F$ below, where C is a formula, and F is a computation type.

Augmenting the precondition of a computation type:

$C \leadsto F \triangleq \{(s_0)C \wedge C_0\}\, x{:}T\, \{(s_1)C_1\}$ where $F = \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\}$ and $s_0 \notin \mathrm{fv}(C)$

Rules for unit and variables:

(Val Unit)

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

(Val Var)

$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

The unit type has only one inhabitant (). The rule (Val Var) looks up the type of a variable in the environment.

Rules for Pairs:

(VAL PAIR)
$$\frac{E \vdash M : T \quad E \vdash N : U\{^{M}/_{x}\}}{E \vdash (M,N) : (\Sigma x : T.\ U)}$$

(STATEFUL EXP SPLIT)
$$\frac{E \vdash M : (\Sigma x : T.\ U) \quad E, x : T, y : U \vdash A : ((x,y) = M \text{ if } \Sigma x : T.\ U \text{ fo}) \rightsquigarrow F \quad \{x,y\} \cap \mathrm{fv}(F) = \varnothing}{E \vdash \textbf{let } (x,y) = M \textbf{ in } A : F}$$

In (STATEFUL EXP SPLIT), when splitting a pair, we strengthen the precondition of the computation with the equation $(x, y) = M$ derived from the pair split, if the pair type is of first order.

Rules for sums and recursive types:

$\mathsf{inl}{:}(T, T{+}U) \quad \mathsf{inr}{:}(U, T{+}U) \quad \mathsf{fold}{:}(T\{^{\mu\alpha.T}/_{\alpha}\}, \mu\alpha.T)$

(VAL INL INR FOLD)
$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U}$$

(STATEFUL EXP MATCH INL INR FOLD)
$$\frac{E \vdash M : T \quad h : (U,T) \quad x \notin \mathrm{fv}(F) \quad E, x : U \vdash A : (h\ x = M \text{ if } T \text{ fo}) \rightsquigarrow F \quad E \vdash B : (\forall x.h\ x \neq M \text{ if } T \text{ fo}) \rightsquigarrow F}{E \vdash \textbf{match } M \textbf{ with } h\ x \to A \textbf{ else } B : F}$$

The rules for constructions $h\ M$ depend on an auxiliary relation $h : (T, U)$ that gives the argument T and result U of each constructor h. The rule (STATEFUL EXP MATCH INL INR FOLD) strengthens the preconditions of the different branches with information derived from the branching condition. In particular, provided the type T is of first order, we remember the equation $h\ x = M$ when checking the positive branch A, and the inequation $\forall x.h\ x \neq M$ when checking the negative branch B. Strengthening these preconditions in this rule and in the rule (STATEFUL EXP SPLIT) is essential for context-dependent type-checking.

The following typing rules for dependent functions are standard.

Rules for functions:

(STATEFUL VAL FUN)
$$\frac{E, x : T \vdash A : F}{E \vdash \textbf{fun } x \to A : (\Pi x : T.\ F)}$$

(STATEFUL EXP APPL)
$$\frac{E \vdash M : (\Pi x : T.\ F) \quad E \vdash N : T}{E \vdash M\ N : F\{^{N}/_{x}\}}$$

We complete the system with the following rules of subtyping for value types.

Subtyping for value types:

(SUB UNIT)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}}$$

(SUB SUM)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

(Stateful Sub Fun)
$$\frac{E \vdash T' <: T \quad E, x : T' \vdash F <: F'}{E \vdash (\Pi x : T.\ F) <: (\Pi x : T'.\ F')}$$

(Sub Pair)
$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')}$$

(Sub Var)
$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(Sub Rec)
$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathrm{fv}(T') \quad \alpha' \notin \mathrm{fv}(T)}{E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')}$$

These rules are essentially standard (Cardelli 1986; Pierce & Sangiorgi 1996; Aspinall & Compagnoni 2001). In (Sub Rec), when checking subtyping of recursive types, we use the environment to keep track of assumptions introduced when unfolding the types.

The main result of this section is that a well-typed expression run in a state satisfying its precondition is *safe*, that is, no assertions fail. Using this result, we can implement different type systems for reasoning about stateful computation in the calculus.

Theorem 4.1 (*Safety*)
If $\varnothing \vdash A : \{(s)C\}\ \_ : T\ \{(s')\mathsf{True}\}$, $\varnothing \vdash C\{^{M}\!/\!_{s}\}$ and $\varnothing \vdash M : \mathsf{state}$ then configuration $(A, M, \varnothing)$ is safe.

The proof of this theorem uses a state-passing translation of RIF into RCF. In particular, a computation type $\{(s_0)C_0\}\ x{:}T\ \{(s_1)C_1\}$ is translated to the refined state monad $\mathcal{M}_{C_0,C_1}([\![T]\!])$ described in the introduction, where $[\![T]\!]$ is the translation of the value type T. We prove that the translation preserves types, allowing us to appeal to the safety theorem for well-typed RCF programs. We give this translation and proof of the safety theorem in the appendices.

4.5 Pragmatics

We find it useful to organize our code into modules. Rather than formalize modules in the syntax, we follow the conventions of Bengtson *et al.* (2008). A module consists of a set of function names $f_1, \ldots, f_k$ with corresponding implementations $M_1, \ldots, M_k$ and associated types $T_1, \ldots, T_k$. It may also include predicate symbols p and an assumption **assume** $(s)C$. In our examples, such top-level assumptions are used to introduce formulas defining the meaning of logical predicates, and we allow the parameter s, which refers to the initial state of the whole program, to be omitted. In fact, our examples never rely on s to constrain the initial state. (Without loss of generality, we suppose there is a single such **assume** expression, but clearly multiple assume expressions can be reduced to a single **assume** expression with a conjunction of the assumed formulas.) A module is *well formed* if the functions type-check at the declared function types, under the given assumptions, that is, if for all $i \in [1..k]$: $f_1 : T_1, \ldots, f_k : T_k \vdash \mathbf{let}\ \_ = \mathbf{assume}\ (s)C\ \mathbf{in}\ M_i : T_i$. All modules used in this paper are well formed. We use **let** $f = M$ to define the implementation of a function in a module, and **val** $f : T$ for its associated type. We sometimes also use **let** $f : T = M$ to capture the same information.

Type-checking a computation A (at type F) in the context of a module with functions $f_1,\dots,f_k$ with implementations $M_1,\dots,M_k$ and types $T_1,\dots,T_k$ corresponds to type-checking $f_1 : T_1,\dots,f_k : T_k \vdash \textbf{let}\ \_ = \textbf{assume}\ (s)C\ \textbf{in}\ A : F$ and executing A in the context of that module corresponds to executing the expression $\textbf{assume}\ (s)C; \textbf{let}\ f_1 = M_1\ \textbf{in}\ \dots \textbf{let}\ f_n = M_n\ \textbf{in}\ A$.

As illustrated in previous sections, to use our calculus, we first *instantiate* it with an *extension API module* that embodies the behavioral type system that we want to capture. In particular, functions in an extension API module perform all the required state manipulations. These extension API functions are written in the internal language described earlier, using the state-manipulation primitives **get**() and **set**(). Moreover, the extension API defines a concrete state type.

4.6 Implementation

There are several challenges that remain pertaining to the implementation of the type system described above. For instance, the typing rule (STATEFUL EXP LET) requires pre- and postconditions of two expressions to match, which means in practice that typing a **let** expression requires a use of (SUB COMP) to make the pre- and post-conditions match. Turning such a nondeterministic rule into a deterministic type-checker is usually achieved by computing either weakest preconditions or strongest postconditions for expressions. We have not yet studied the question of type inference for our type system.

As we noted in the introduction, we have a prototype implementation, Stateful F7, that we have used to type-check all the examples in this paper. This prototype implementation avoids the challenges we highlighted above by using the translation of RIF into RCF given in the appendices—the one used to prove Theorem 4.1—and doing the actual type-checking in RCF. As we establish in the appendices, the translation preserves both semantics and typeability, and the result of the translation is safe only if the original program is safe. As a consequence, our implementation may type-check programs that RIF rejects (but only such programs that are safe, per Corollary 3).

Our implementation translates programs written in a subset of Objective Caml and F# extended with RIF-based computation types into the language of the F7 type-checker, which implements the RCF type system, and we perform type-checking in F7, which itself calls Z3 to discharge proof obligations. The RBAC API and examples yielded 53 verification conditions, while the HBAC and SBAC APIs and examples yielded 80 conditions; the total run time of Z3 on all of these conditions is under a second. Error messages inform the user if Z3 cannot prove these formulas. We unfortunately have no fallback on interactive proof when Z3 is unable to prove a valid formula; it is sometimes possible to guide Z3's proof search by asserting and then assuming suitable intermediate formulas.

5 Related work

We discuss related work on type systems for access control. Pottier *et al.* (2005) develop a type and effect system for SBAC. As in our work, the goal is to prevent security exceptions. Our work is intended to show that their type system may be generalized so that effects are represented as formulas. Hence, our work is more flexible in that we can deal with an arbitrary lattice of dependent permissions; their system is limited to a finite set of permissions.

Besson *et al.* (2004) develop a static analysis for .NET libraries, to discover anomalies in the security policy implemented by stack inspection. The tool depends on a flow analysis rather than a type system.

A separate line of work investigates the information flow properties of SBAC and HBAC (Banerjee & Naumann 2005a, 2005b; Pistoia *et al.* 2007a). We believe our type system could be adapted to check information flow, but this remains future work. Another line of future investigation is type inference; ideas from the study of refinement types may be helpful (Knowles & Flanagan 2007; Rondon *et al.* 2008).

Abadi *et al.* (1993) initiated the study of logic for access control in distributed systems; they propose a propositional logic with a says-modality to indicate the intentions of different principals. This logic is used by Wallach *et al.* (2000) to provide a logical semantics of stack inspection. Abadi (2006) develops an approach to access control in which the formulas of a constructive version of the logic are interpreted as types. AURA (Jia *et al.* 2008) is a language that is based, in part, on this idea.

Fournet *et al.* (2005) introduced the idea of type-checking code to ensure conformance to a logic-based authorization policy. A series of papers develops the idea for distributed systems modeled with process calculi (Fournet *et al.* 2007; Maffeis *et al.* 2008). In this line of work, access rights may be granted but not retracted. Our approach in Section 2 is different in that we deal with roles that may be activated and deactivated.

6 Conclusion

We described a higher-order imperative language whose semantics is based on the state monad, refined with preconditions and postconditions. By making different choices for the underlying state type, and supplying suitable primitive functions, we gave semantics for standard access control mechanisms based on stacks, histories, and roles. Type-checking ensures the absence of security exceptions, a common problem for code-based access control.

This work is dedicated to Tony Hoare, in part in gratitude for his useful feedback over the years on various behavioral type systems for process calculi. Some of those calculi had a great deal of innovative syntax. So we hope he will endorse our general conclusion that it is better to design behavioral type systems using types refined with logical formulas, than to invent still more syntax.

Acknowledgment

We are grateful to Martín Abadi, Robert Atkey, Anindya Banerjee, Moritz Becker, Cliff Jones, David Naumann, Nikhil Swamy, and Wouter Swierstra for comments and discussions.

References

Abadi, M. (2006) Access control in a core calculus of dependency. In *International Conference on Functional Programming (ICFP'06)*, pp. 263–273.

Abadi, M., Burrows, M., Lampson, B. & Plotkin, G. (1993) A calculus for access control in distributed systems, *ACM Trans. Program. Lang. Syst.*, 15(4): 706–734.

Abadi, M. & Fournet, C. (2003) Access control based on execution history. In *Network and Distributed System Security Symposium (NDSS'03)*, M. Reiter & V. Gligor (eds). Reston, VA: The Internet Society, pp. 107–121.

Aspinall, D. & Compagnoni, A. (2001) Subtyping dependent types, *Theor. Comput. Sci.*, 266 (1–2): 273–309.

Atkey, R. (2009) Parameterized notions of computation, *J. Funct. Program.*, 19: 355–376.

Banerjee, A. & Naumann, D. (2005a) History-based access control and secure information flow. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2004)*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet & T. Muntean (eds), Lecture Notes in Computer Science, vol. 3362. Berlin Heidelberg, Germany: Springer, pp. 27–48.

Banerjee, A. & Naumann, D. (2005b) Stack-based access control and secure information flow, *J. Funct. Program.*, 15(2): 131–177.

Becker, M. Y. & Nanz, S. (2007) A logic for state-modifying authorization policies. In *European Symposium on Research in Computer Security (ESORICS'07)*, J. Biskup & J. Lόpez (eds), Lecture Notes in Computer Science, vol. 4734. Berlin Heidelberg, Germany: Springer, pp. 203–218.

Becker, M. Y. & Sewell, P. (2004) Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop (CSFW'04)*, pp. 139–154.

Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D. & Maffeis, S. 2008 *Refinement Types for Secure Implementations*. Technical Report MSR–TR–2008–118, Microsoft Research (a preliminary, abridged version appears in the proceedings of Computer Security Foundations Symposium 2008).

Besson, F., Blanc, T, Fournet, C. & Gordon, A. D. (2004) From stack inspection to access control: A security analysis for libraries. In *IEEE Computer Security Foundations Workshop (CSFW'04)*, pp. 61–77.

Borgström, J., Gordon, A. D. & Pucella, R. (2009) *Roles, Stacks, Histories: A Triple for Hoare*. Technical Report MSR–TR–2009–97, Microsoft Research.

Cardelli, L. (1986) Typechecking dependent types and subtypes. In *Foundations of Logic and Functional Programming*, M. Boscarol, L. C. Aiello & G. Levi (eds), Lecture Notes in Computer Science, vol. 306. Berlin Heidelberg, Germany: Springer, pp. 45–57.

Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T. & Smith, S. F. (1986) *Implementing Mathematics with the Nuprl Proof Development system*. Hemel Hampstead, England: Prentice-Hall.

DeLine, R. & Fähndrich, M. (2001) Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI'01)*, pp. 59–69.

de Moura, L. & Bjørner, N. (2008) Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, C. R. Ramakrishnan & J. Rehof (eds), Lecture Notes in Computer Science, vol. 4963. Berlin Heidelberg, Germany: Springer, pp. 337–340.

Detlefs, D., Nelson, G. & Saxe, J. B. (2005) Simplify: A theorem prover for program checking, *J. ACM*, 52(3):365–473.

Dutertre, B. & de Moura, L. (2006) The YICES SMT solver [online]. Accessed August 13, 2010. Available at: `http://yices.csl.sri.com/tool-paper.pdf`

Ferraiolo, D. F. & Kuhn, D. R. (1992) Role based access control. In *National Computer Security Conference*, pp. 554–563.

Filliâtre, J. & C. Marché, C. (2004) Multi-prover verification of C Programs. In *International Conference on Formal Engineering Methods (ICFEM 2004)*, J. Davies, W. Schulte & M.

Barnett (eds), Lecture Notes in Computer Science, vol. 3308. Berlin Heidelberg, Germany: Springer, pp. 15–29.

Filliâtre, J.-C. (1999) Proof of imperative programs in type theory. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES '98)*, vol. 1657. Berlin Heidelberg, Germany: Springer, pp. 78–92.

Flanagan, C. (2006) Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, pp. 245–256.

Flanagan, C. & Abadi, M. (1999) Types for safe locking. In *European Symposium on Programming (ESOP'99)*, S. Doaitse Swierstra (ed), Lecture Notes in Computer Science, vol. 1576. Berlin Heidelberg, Germany: Springer, pp. 91–108.

Fournet, C. & Gordon, A. D. (2003) Stack inspection: Theory and variants, *ACM Trans. Program. Lang. Syst.*, 25 (3): 360–399.

Fournet, C., Gordon, A. D. & Maffeis, S. (2005) A type discipline for authorization policies. In *European Symposium on Programming (ESOP'05)*, M. Sagiv (ed), Lecture Notes in Computer Science, vol. 3444. Berlin Heidelberg, Germany: Springer, pp. 141–156.

Fournet, C., Gordon, A. D. & Maffeis, S. (2007) A type discipline for authorization policies in distributed systems. In *IEEE Computer Security Foundation Symposium (CSF'07)*, pp. 31–45.

Freeman, T. & Pfenning, F. (1991) Refinement types for ML. In *Programming Language Design and Implementation (PLDI'91)*. ACM Press, pp. 268–277.

Gifford, D. & Lucassen, J. (1986) Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pp. 28–38.

Gong, L. (1999) *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley.

Gordon, A. D. & Fournet, C. (2009) *Principles and Applications of Refinement Types.* Technical Report MSR–TR–2009–147, Microsoft Research.

Gordon, A. D. & Jeffrey, A. S. A. (2003) Authenticity by typing for security protocols, *J. Comput. Secur.*, 11 (4): 451–521.

Gronski, J., Knowles, K., Tomb, A., Freund, S. N. & Flanagan, C. (2006) Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, Findler, R. (ed), pp. 93–104.

Gunter, C. (1992) *Semantics of Programming Languages.* MIT Press.

Hardy, N. (1988) The confused deputy (or why capabilities might have been invented), *ACM SIGOPS Oper. Syst. Rev.*, 22: 36–38.

Jia, L., Vaughan, J. A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J. & Zdancewic, S. (2008) *AURA: Preliminary Technical Results.* Technical Report MS-CIS-08-10, University of Pennsylvania.

Knowles, K. W. & Flanagan, C. (2007) Type reconstruction for general refinement types. In *European Symposium on Programming (ESOP'07)*, R. De Nicola (ed), Lecture Notes in Computer Science, vol. 4421. Berlin Heidelberg, Germany: Springer, pp. 505–519.

Li, N., Mitchell, J. C. & Winsborough, W. H. (2002) Design of a role-based trust management framework. In *IEEE Security and Privacy*, pp. 114–130.

Maffeis, S., Abadi, M., Fournet, C. & Gordon, A. D. (2008) Code-carrying authorization. In *European Symposium On Research In Computer Security (ESORICS'08)*, pp. 563–579.

Moggi, E. (1991) Notions of computations and monads, *Inf. Comput.*, 93: 55–92.

Nanevski, A., Morrisett, G. & Birkedal, L. (2006) Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming (ICFP'06)*, pp. 62–73.

Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P. & Birkedal, L. (2008) Ynot: Dependent types for imperative programs. In *International Conference on Functional Programming (ICFP'08)*, pp. 229–240.

Nordström, B., Petersson, K. & Smith, J. (1990) *Programming in Martin-Löf's type Theory.* Clarendon Press, Oxford.

Pierce, B. & Sangiorgi, D. (1996) Typing and subtyping for mobile processes, *Math. Struct. Comput. Sci.*, 6 (5): 409–454.

Pistoia, M., Banerjee, A. & Naumann, D. (2007a) Beyond stack inspection: A unified access-control and information-flow security model. In *IEEE Security and Privacy*, pp. 149–163.

Pistoia, M., Chandra, S., Fink, S. J. & Yahav, E. (2007b) A survey of static analysis methods for identifying security vulnerabilities in software systems, *IBM Syst. J.*, 46 (2): 265–288.

Plotkin, G. D. (1985) *Denotational Semantics with Partial Functions*. Unpublished lecture notes, CSLI, Stanford University.

Pottier, F., Skalka, C. & Smith, S. (2005) A systematic approach to static access control, *ACM Trans. Program. Lang. Syst.*, 27(2): 344–382.

Ranise, S. & Tinelli, C. (2006) *The SMT-LIB Standard: Version 1.2*. [online]. Accessed August 13, 2010. Available at: http://goedel.cs.uiowa.edu/smtlib/papers.html

Régis-Gianas, Y. & Pottier, F. (2008) A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction (MPC'08)*, P. Adebaud & C. Paulin-Mohring (eds), Lecture Notes in Computer Science, vol. 5133. Berlin Heidelberg, Germany: Springer, pp. 305–335.

Rondon, P., Kawaguchi, M. & Jhala, R. (2008) Liquid types. In *Programming Language Design and Implementation (PLDI'08)*. ACM, pp. 159–169.

Rushby, J., Owre, S. & Shankar, N. (1998) Subtypes for specifications: Predicate subtyping in PVS, *IEEE Trans. Softw. Eng.*, 24 (9): 709–720.

Sabry, A. & Felleisen, M. (1993) Reasoning about programs in continuation-passing style, *LISP Symb. Comput.*, 6 (3–4): 289–360.

Sandhu, R., Coyne, E. J., Feinstein, H. L. & Youman, C. E. (1996) Role-based access control models, *IEEE Comput.*, 29 (2): 38–47.

Strom, R. E. & Yemini, S. (1986) Typestate: A programming language concept for enhancing software reliability, *IEEE Trans. Softw. Eng.*, 12: 157–171.

Wadler, P. (1992) Comprehending monads, *Math. Struct. Comput. Sci.*, 2: 461–493.

Wallach, D. S., Appel, A. W. & Felten, E. W. (2000) SAFKASI: A security mechanism for language-based systems, *ACM Trans. Softw. Eng. Methodol.*, 9(4): 341–378.

Xi, H. & Pfenning, F. (1999) Dependent types in practical programming. In *Principles of Programming Languages (POPL'99)*, pp. 214–227.

Appendix A RCF: refined concurrent FPC

Our theory of RIF is based on refined concurrent FPC, a typed concurrent λ-calculus. This section gives the formal definitions of the calculus, and states the properties relied on in this paper. For fuller explanations, please consult the original report on RCF (Bengtson *et al.* 2008) or some recent tutorial notes (Gordon & Fournet 2009).

The expressions and values of RCF are as follows. It consists of the core Fixpoint Calculus, together with constructs for communication and concurrency from the π-calculus, and assumptions and assertions from Dijkstra's guarded command language. The calculus is parameterized on a choice of authorization logic, as introduced in Section 4.1. The choice of logic determines the exact syntax of formulas C and the deduction relation $\{C_1, \dots, C_n\} \vdash C$. In this paper, both RIF and RCF are parameterized with the variant FOL/FO of first-order logic defined in Section 4.1.

Syntax of RCF values and expressions:

a, b, c	name
x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type

inr	right constructor of sum type
fold	constructor of recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction
$A, B ::=$	expression
M	value
$M\ N$	application
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)
let $(x, y) = M$ **in** A	pair split (scope of x, y is A)
match M **with** $h\ x \to A$ **else** B	constructor match (scope of x is A)
$(\nu a)A$	restriction (scope of a is A)
$A \Rsh B$	fork
$a!M$	transmission of M on channel a
$a?$	receive message off channel
assume C	assumption of formula C
assert C	assertion of formula C

false $\triangleq$ inl $()$ **true** $\triangleq$ inr $()$

RCF follows similar syntactic conventions as in RIF but additionally its syntax includes names as well as variables. We say a value is of *first order* if it contains no functions **fun** $x \to A$. We write $\text{fn}(\phi)$ for the set of names occurring free in ϕ, and also $\text{fnfv}(\phi) = \text{fv}(\phi) \cup \text{fn}(\phi)$, the set of both names and variables occurring free in ϕ. In the logic, each RCF name is a constant, that is, a nullary syntactic function symbol.

Expressions represent run-time configurations as well as source code. Structures **S** are normal forms for expressions, and formalize the idea that a configuration has three parts: (1) the *log*, a multiset $\prod_{i \in 1..m}$ **assume** C_i of assumed formulas; (2) a series of messages M_j sent on channels but not yet received; and (3) a series of elementary expressions e_k being evaluated in parallel contexts.

We give structures below, together with a notion of *static safety*, which means that all active assertions in a structure are deducible in the logic from the active assumptions. We additionally require all formulas to be of first order, since the logic only speaks about first-order values.

Structures and static safety:

$e ::= M \mid M\ N \mid M = N \mid$ **let** $(x, y) = M$ **in** $A \mid$
match M **with** $h\ x \to A$ **else** $B \mid a? \mid$ **assert** C

$\prod_{i \in 1..n} A_i \triangleq () \Rsh A_1 \Rsh \ldots \Rsh A_n$

$\mathcal{L} ::= \{\} \mid ($**let** $x = \mathcal{L}$ **in** $B)$

$\mathbf{S} ::= (\nu a_1)\ldots(\nu a_\ell)(\prod_{i\in 1\ldots m}\mathbf{assume}\ C_i) \Rsh (\prod_{j\in 1\ldots n} c_j!M_j) \Rsh (\prod_{k\in 1\ldots o} \mathcal{L}_k\{e_k\}))$

Let structure $\mathbf{S}$ be *statically safe* if and only if, for all $p \in 1\ldots o$ and C, if $e_p = \mathbf{assert}\ C$ then $C_1,\ldots,C_m$ and C are first order, and $\{C_1,\ldots,C_m\} \vdash C$.

Next, we present the *heating relation*, $A \Rrightarrow A'$, which relates expression up to various structural re-arrangements. In particular, every expression can be related to a structure via heating.

Heating: $A \Rrightarrow A'$

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$	(Heat Refl)
$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$	(Heat Trans)
$A \Rrightarrow A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \Rrightarrow \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(Heat Let)
$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$	(Heat Res)
$A \Rrightarrow A' \Rightarrow (A \Rsh B) \Rrightarrow (A' \Rsh B)$	(Heat Fork 1)
$A \Rrightarrow A' \Rightarrow (B \Rsh A) \Rrightarrow (B \Rsh A')$	(Heat Fork 2)
$() \Rsh A \equiv A$	(Heat Fork ())
$a!M \Rrightarrow a!M \Rsh ()$	(Heat Msg ())
$\mathbf{assume}\ C \Rrightarrow \mathbf{assume}\ C \Rsh ()$	(Heat Assume ())
$a \notin \mathrm{fn}(\mathrm{A}') \Rightarrow \mathrm{A}' \Rsh ((\nu a)\mathrm{A}) \Rrightarrow (\nu a)(\mathrm{A}' \Rsh \mathrm{A})$	(Heat Res Fork 1)
$a \notin \mathrm{fn}(\mathrm{A}') \Rightarrow ((\nu a)\mathrm{A}) \Rsh \mathrm{A}' \Rrightarrow (\nu a)(\mathrm{A} \Rsh \mathrm{A}')$	(Heat Res Fork 2)
$a \notin \mathrm{fn}(\mathrm{B}) \Rightarrow$ $\mathbf{let}\ x = (\nu a)A\ \mathbf{in}\ B \Rrightarrow (\nu a)\mathbf{let}\ x = A\ \mathbf{in}\ B$	(Heat Res Let)
$(A \Rsh A') \Rsh A'' \equiv A \Rsh (A' \Rsh A'')$	(Heat Fork Assoc)
$(A \Rsh A') \Rsh A'' \Rrightarrow (A' \Rsh A) \Rsh A''$	(Heat Fork Comm)
$\mathbf{let}\ x = (A \Rsh A')\ \mathbf{in}\ B \equiv$ $A \Rsh (\mathbf{let}\ x = A'\ \mathbf{in}\ B)$	(Heat Fork Let)

The *reduction relation*, $A \to A'$, is the operational semantics of RCF.

Reduction: $A \to A'$

$(\mathbf{fun}\ x \to A)\ N \to A\{N/x\}$	(R Red Fun)
$(\mathbf{let}\ (x_1,x_2) = (N_1,N_2)\ \mathbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\}$	(R Red Split)
$(\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B) \to \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$	(R Red Match)
$M = N \to \begin{cases} \mathbf{true} & \text{if } M = N \\ \mathbf{false} & \text{otherwise} \end{cases}$	(R Red Eq)
$a!M \Rsh a? \to M$	(R Red Comm)
$\mathbf{assert}\ C \to ()$	(R Red Assert)
$\mathbf{let}\ x = M\ \mathbf{in}\ A \to A\{M/x\}$	(R Red Let Val)
$A \to A' \Rightarrow \mathbf{let}\ x = A\ \mathbf{in}\ B \to \mathbf{let}\ x = A'\ \mathbf{in}\ B$	(R Red Let)
$A \to A' \Rightarrow (\nu a)A \to (\nu a)A'$	(R Red Res)

$A \to A' \Rightarrow (A \upharpoonright B) \to (A' \upharpoonright B)$	(R Red Fork 1)
$A \to A' \Rightarrow (B \upharpoonright A) \to (B \upharpoonright A')$	(R Red Fork 2)
$A \to A' \quad \text{if } A \Rrightarrow B, B \to B', B' \Rrightarrow A'$	(R Red Heat)

We define expression safety as follows. A closed expression A is *safe* if and only if, in all evaluations of A, all assertions succeed.

Expression safety:

An expression A is *safe* if and only if, for all A' and $\mathbf{S}$, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then $\mathbf{S}$ is statically safe.

The purpose of the system of refinement types for RCF is to verify by typing that an expression is safe. The types of RCF are as follows. The starting point is the system of unit, function, pair, sum, and iso-recursive types of FPC, to which we add refinement types $\{x : T \mid C\}$, while making function and pair types dependent.

Syntax of types:

$H, T, U, V ::=$	type
α	type variable
unit	unit type
$\Pi x : T.\ U$	dependent function type (scope of x is U)
$\Sigma x : T.\ U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
$\{x : T \mid C\}$	refinement type (scope of x is C)

A type T is of first order if and only if T contains no occurrences of $\Pi x : T.\ U$. For a type $\Pi x : T.\ F$ or $\Sigma x : T.\ U$ or $\{x : T \mid C\}$ to be well formed, we require that either T is a first-order type or that x is not free in F, U, or C, respectively. The type system relies on the following derivable types. An *ok-type* $\{C\}$ is a unit token conveying that the formula C holds. We use ok-types to record logical assumptions in our typing environments.

Some derivable types:

$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$	ok-type
$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$	Boolean type

Below, we define the syntax of typing environments, E, for tracking the identifiers (type variables, names, and (value) variables) in scope during type-checking.

Syntax of typing environments:

$\mu ::=$	environment entry
$\alpha <: \alpha'$	subtype ($\alpha \neq \alpha'$)
$a \updownarrow T$	name of a typed channel
$x : T$	variable
$E ::= \mu_1, \ldots, \mu_n$	environment

$\mathrm{dom}(\alpha <: \alpha') = \{\alpha, \alpha'\}$	$\mathrm{fnfv}(\alpha <: \alpha') = \varnothing$
$\mathrm{dom}(a \updownarrow T) = \{a\}$	$\mathrm{fnfv}(a \updownarrow T) = \mathrm{fnfv}(T)$
$\mathrm{dom}(x : T) = \{x\}$	$\mathrm{fnfv}(x : T) = \mathrm{fnfv}(T)$

$\mathrm{dom}(\mu_1, \ldots, \mu_n) = \mathrm{dom}(\mu_1) \cup \ldots \cup \mathrm{dom}(\mu_n)$
$\mathrm{fov}(E) = \{x \in \mathrm{dom}(E) \mid (x : T) \in E \text{ with } T \text{ first order}\}$
$\mathrm{recvar}(E) = \{\alpha \mid \alpha \in \mathrm{dom}(E)\}$

The type system consists of five inductively defined judgments.

Judgments ($E \vdash \mathcal{J}$):

$E \vdash \diamond$	E is syntactically well formed
$E \vdash T$	in E, type T is syntactically well formed
$E \vdash C$ fo	formula C is first order in E
$E \vdash C$	formula C is derivable from E
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

The judgments $E \vdash \diamond$, $E \vdash T$, and $E \vdash C$ are inductively defined by the rules in the following table. The function $\mathsf{forms}(E)$, also given below, returns the set of formulas occurring in refinement types in the environment.

Rules of well formedness and deduction:

(R Env Empty)
$$\frac{}{\varnothing \vdash \diamond}$$

(R Env Entry)
$$\frac{E \vdash \diamond \quad \mathrm{fnfv}(\mu) \subseteq \mathrm{dom}(E) \quad \mathrm{dom}(\mu) \cap \mathrm{dom}(E) = \varnothing}{E, \mu \vdash \diamond}$$

(R Type)
$$\frac{E \vdash \diamond \quad \mathrm{fnfv}(T) \subseteq \mathrm{dom}(E)}{E \vdash T}$$

(R Derive)
$$\frac{E \vdash C \text{ fo} \quad \mathsf{forms}(E) \vdash C}{E \vdash C}$$

(R Form)
$$\frac{E \vdash \diamond \quad C \text{ is first order} \quad \mathrm{fn}(C) \subseteq \mathrm{dom}(E) \quad \mathrm{fv}(C) \subseteq \mathrm{fov}(E)}{E \vdash C \text{ fo}}$$

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{^y/_x\}\} \cup \mathsf{forms}(y : T) & \text{if } E = (y : \{x : T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

The judgment $E \vdash T <: T'$ is inductively defined by the following rules.

Rules for subtyping:

(R Sub Refl)
$$\frac{E \vdash T \quad \mathrm{recvar}(E) \cap \mathrm{fnfv}(T) = \varnothing}{E \vdash T <: T}$$

(R Sub Unit)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}}$$

(R Sub Fun)
$$\frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\ U) <: (\Pi x : T'.\ U')}$$

(R Sub Pair)
$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')}$$

(R Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + T') <: (U + U')}$$

(R Sub Var)
$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'}$$

(R Sub Rec)
$$\frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin \mathrm{fnfv}(T') \quad \alpha' \notin \mathrm{fnfv}(T)}{E \vdash (\mu\alpha.T) <: (\mu\alpha'.T')}$$

(R Sub Refine Left)
$$\frac{E \vdash \{x : T \mid C\} \quad E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'}$$

(R Sub Refine Right)
$$\frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

The judgment $E \vdash A : T$ is inductively defined by the rules in the following table. We use the notation $E + C$ defined as follows: when $E \vdash C$ fo holds, we let $E + C \triangleq E, \_ : \{C\}$; otherwise, we let $E + C \triangleq E$.

Rules for typing expressions:

(R Val Var)
$$\frac{E \vdash \diamond \quad (x : T) \in E}{E \vdash x : T}$$

(R Exp Subsum)
$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

(R Exp Eq)
$$\frac{E \vdash M : T \quad E \vdash N : U \quad x \notin \mathrm{fv}(M, N)}{E \vdash M = N : \{x : \mathsf{bool} \mid x = \mathbf{true} \Leftrightarrow M = N\}}$$

(R Exp Assume)
$$\frac{E \vdash C \text{ fo}}{E \vdash \mathbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}}$$

(R Exp Assert)
$$\frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathsf{unit}}$$

(R Exp Let)
$$\frac{E \vdash A : T \quad E, x : T \vdash B : U \quad x \notin \mathrm{fv}(U)}{E \vdash \mathbf{let}\ x = A\ \mathbf{in}\ B : U}$$

(R Val Unit)
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}}$$

(R Val Fun)
$$\frac{E, x : T \vdash A : U}{E \vdash \mathbf{fun}\ x \to A : (\Pi x : T.\ U)}$$

(R Exp Appl)
$$\frac{E \vdash M : (\Pi x : T.\ U) \quad E \vdash N : T}{E \vdash M\ N : U\{^{N}\!/_{x}\}}$$

(R Val Pair)

$$\frac{E \vdash M : T \quad E \vdash N : U\{{}^{M}\!/_{x}\}}{E \vdash (M, N) : (\Sigma x : T.\ U)}$$

(R Exp Split)

$$\frac{\begin{array}{l} E \vdash M : (\Sigma x : T.\ U) \\ E, x : T, y : U + ((x, y) = M) \vdash A : V \\ \{x, y\} \cap \mathrm{fv}(V) = \varnothing \end{array}}{E \vdash \textbf{let}\ (x, y) = M\ \textbf{in}\ A : V}$$

$\mathsf{inl}{:}(T, T{+}U) \quad \mathsf{inr}{:}(U, T{+}U) \quad \mathsf{fold}{:}(T\{{}^{\mu\alpha.T}\!/_{\alpha}\}, \mu\alpha.T)$

(R Val Inl Inr Fold)

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U}$$

(R Exp Match Inl Inr Fold)

$$\frac{\begin{array}{l} E \vdash M : T \quad h : (U, T) \\ E, x : H + (h\ x = M) \vdash A : U \quad x \notin \mathrm{fv}(U) \\ E + (\forall x.h\ x \neq M) \vdash B : U \end{array}}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

(R Val Refine)

$$\frac{E \vdash M : T \quad E \vdash C\{{}^{M}\!/_{x}\}}{E \vdash M : \{x : T \mid C\}}$$

(R Exp Res)

$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin \mathrm{fn(U)}}{E \vdash (\nu a)A : U}$$

(R Exp Send)

$$\frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}}$$

(R Exp Recv)

$$\frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

(R Exp Fork)

$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \Rsh A_2) : T_2}$$

$\overline{(\nu a)A} \triangleq (\exists a.\overline{A}) \quad \overline{A_1 \Rsh A_2} \triangleq (\overline{A_1} \wedge \overline{A_2}) \quad \overline{\textbf{let}\ x = A_1\ \textbf{in}\ A_2} \triangleq \overline{A_1} \quad \overline{\textbf{assume}\ C} \triangleq C$

$\overline{A} \triangleq \mathsf{True}$ if A matches no other rule

To state the following general properties of all the judgments of our system, we let $\mathfrak{J}$ range over $\{\diamond, T, C, C\ \mathrm{fo}, \mathrm{T} <: \mathrm{T}', \mathrm{A} : \mathrm{T}\}$.

Admissible rules:

(R Bound Weakening)

$$\frac{E \vdash T' <: T \quad E, x : T, E' \vdash \mathfrak{J}}{E, x : T', E' \vdash \mathfrak{J}}$$

(R Bound Unrefine)

$$\frac{E, x : T, E' \vdash \mathfrak{J}}{E, x : \{x : T \mid C\}, E' \vdash \mathfrak{J}}$$

(R Weakening)

$$\frac{E, E' \vdash \mathfrak{J}}{E, x : T, E' \vdash \mathfrak{J}}$$

(R Exchange)

$$\frac{E, \mu_1, \mu_2, E' \vdash \mathfrak{J} \quad \mathrm{dom}(\mu_1) \cap \mathrm{fnfv}(\mu_2) = \varnothing}{E, \mu_2, \mu_1, E' \vdash \mathfrak{J}}$$

(R Sub Refine)

$$\frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \vdash C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}}$$

(R Sub Refine Left Refl)

$$\frac{E \vdash \{x : T \mid C\}}{E \vdash \{x : T \mid C\} <: T}$$

The primary soundness results about RCF, proved elsewhere (Bengtson *et al.* 2008), are as follows. Let E be *executable* if and only if recvar(E) $= \varnothing$.

Lemma 3 (Static Safety)
If $\varnothing \vdash \mathbf{S} : T$ then $\mathbf{S}$ is statically safe.

Proposition 4 ($\Rrightarrow$ Preserves Types)
If E is executable and $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.

Proposition 5 ($\rightarrow$ Preserves Types)
If E is executable, $\mathrm{fv}(A) = \varnothing$, and $E \vdash A : T$ and $A \rightarrow A'$ then $E \vdash A' : T$.

Theorem 2 (Safety of RCF)
If $\varnothing \vdash A : T$ then A is safe.

Appendix B: Semantics of RIF by translation to RCF

We give a semantics to RIF through translation to the sequential fragment of RCF. For our purposes, this fragment is virtually identical to the source language, except for the lack of binders on **assert** C and **assume** C. The translation of values and formulas is homomorphic.

$\mathcal{V}[\![M]\!]$ is the translation of the value M in the source language to a value in RCF.

Translation of values: $\mathcal{V}[\![M]\!]$

$$\begin{aligned}
&\mathcal{V}[\![x]\!] \triangleq x\\
&\mathcal{V}[\![()]\!] \triangleq ()\\
&\mathcal{V}[\![\mathbf{fun}\ x \rightarrow A]\!] \triangleq \mathbf{fun}\ x \rightarrow \mathcal{E}[\![A]\!]\\
&\mathcal{V}[\![(M,N)]\!] \triangleq (\mathcal{V}[\![M]\!], \mathcal{V}[\![N]\!])\\
&\mathcal{V}[\![h\ M]\!] \triangleq h\ (\mathcal{V}[\![M]\!])
\end{aligned}$$

Let $\mathcal{E}[\![A]\!]$ be the translation of an expression A in the source language to a function in RCF. This function receives the current state and returns a pair, consisting of the original return value and the resulting state. Intuitively, the translation threads a state through the original computation. We rely on an auxiliary translation $\mathcal{E}_c[\![A]\!]s$ on expressions to reduce the number of administrative redexes.

Translation of expressions: $\mathcal{E}[\![A]\!]$ **and** $\mathcal{E}_c[\![A]\!]s$

$$\mathcal{E}[\![A]\!] \triangleq \mathbf{fun}\ s \rightarrow \mathcal{E}_c[\![A]\!]s \qquad\qquad (s \notin \mathrm{fv}(A))$$

$$\begin{aligned}
&\mathcal{E}_c[\![M]\!]s \triangleq (\mathcal{V}[\![M]\!], s)\\
&\mathcal{E}_c[\![M\ N]\!]s \triangleq \mathbf{let}\ f = \mathcal{V}[\![M]\!]\ \mathcal{V}[\![N]\!]\ \mathbf{in}\ f\ s\\
&\mathcal{E}_c[\![M = N]\!]s \triangleq \mathbf{let}\ b = (\mathcal{V}[\![M]\!] = \mathcal{V}[\![N]\!])\ \mathbf{in}\ (b, s)\\
&\mathcal{E}_c[\![\mathbf{let}\ x = A\ \mathbf{in}\ B]\!]s \triangleq \mathbf{let}\ y = \mathcal{E}_c[\![A]\!]s\ \mathbf{in}\ \mathbf{let}\ (x, s') = y\ \mathbf{in}\ \mathcal{E}_c[\![B]\!]s' \qquad (y, s' \notin \mathrm{fv}(B))\\
&\mathcal{E}_c[\![\mathbf{let}\ (x, y) = M\ \mathbf{in}\ A]\!]s \triangleq \mathbf{let}\ (x, y) = \mathcal{V}[\![M]\!]\ \mathbf{in}\ \mathcal{E}_c[\![A]\!]s\\
&\mathcal{E}_c[\![\mathbf{match}\ M\ \mathbf{with}\ h\ x{\rightarrow}A\ \mathbf{else}\ B]\!]s \triangleq \mathbf{match}\ \mathcal{V}[\![M]\!]\ \mathbf{with}\ h\ x{\rightarrow}\mathcal{E}_c[\![A]\!]s\ \mathbf{else}\ \mathcal{E}_c[\![B]\!]s\\
&\mathcal{E}_c[\![\mathbf{assume}\ (s)C]\!]s \triangleq \mathbf{let}\ \_ = \mathbf{assume}\ [\![C]\!]\ \mathbf{in}\ ((), s)\\
&\mathcal{E}_c[\![\mathbf{assert}\ (s)C]\!]s \triangleq \mathbf{let}\ \_ = \mathbf{assert}\ [\![C]\!]\ \mathbf{in}\ ((), s)
\end{aligned}$$

$\mathcal{E}_c[\![\mathbf{get}()]\!]s \triangleq (s, s)$
$\mathcal{E}_c[\![\mathbf{set}(M)]\!]s \triangleq ((), \mathcal{V}[\![M]\!])$

We want to work in a single, unified authorization logic for both the source and the target of the translation. We prove that (first-order) formulas are mapped to themselves by formula translation; equiprovability trivially follows. Let $[\![C]\!]$ be the translation of the formula C.

Translation of formulas: $[\![C]\!]$

$[\![p(M_1, \dots, M_n)]\!] \triangleq p(\mathcal{V}[\![M_1]\!], \dots, \mathcal{V}[\![M_n]\!])$
$[\![M = M']\!] \triangleq \mathcal{V}[\![M]\!] = \mathcal{V}[\![M']\!]$
$[\![C \wedge C']\!] \triangleq [\![C]\!] \wedge [\![C']\!]$
$[\![C \vee C']\!] \triangleq [\![C]\!] \vee [\![C']\!]$
$[\![\neg C]\!] \triangleq \neg [\![C]\!]$
$[\![\forall x.C]\!] \triangleq \forall x.[\![C]\!]$
$[\![\exists x.C]\!] \triangleq \exists x.[\![C]\!]$

A formula C is of *first order* if it contains only first-order values.

Lemma 6 (*Translating first-order phrases*)
If M is a first-order value then $[\![M]\!] = M$, and if C is a first-order formula then $C = [\![C]\!]$.

Proof
By inductions on the structure of M and C. □

Corollary 1 (*Equiprovability*)

- If C is a first-order formula, then $\vdash C$ iff $\vdash [\![C]\!]$.
- If C and all formulas in S are of first order, then $S \vdash C$ iff $[\![S]\!] \vdash [\![C]\!]$.

The translation of types is straightforward. We translate computation types into a function type that takes a refinement of the state type and returns a pair of the original return type and a refined state.

Translation of types: $\mathcal{T}[\![T]\!], \mathcal{T}[\![F]\!]$

$\mathcal{T}[\![\alpha]\!] \triangleq \alpha$
$\mathcal{T}[\![\mathsf{unit}]\!] \triangleq \mathsf{unit}$
$\mathcal{T}[\![\Pi x : T.\ F]\!] \triangleq \Pi x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![F]\!]$
$\mathcal{T}[\![\Sigma x : T.\ U]\!] \triangleq \Sigma x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![U]\!]$
$\mathcal{T}[\![T + U]\!] \triangleq \mathcal{T}[\![T]\!] + \mathcal{T}[\![U]\!]$
$\mathcal{T}[\![\mu\alpha.T]\!] \triangleq \mu\alpha.\mathcal{T}[\![T]\!]$

$\mathcal{T}[\![\{(s_0)C_0\}\ x_1{:}T_1\ \{(s_1)C_1\}]\!] \triangleq \Pi s_0 : \{s_0 : \mathsf{state} \mid [\![C_0]\!]\}.\ \Sigma x_1 : \mathcal{T}[\![T_1]\!].\ \{s_1 : \mathsf{state} \mid [\![C_1]\!]\}$

Closed first-order types are translated to themselves, so $\mathcal{T}[\![\mathsf{state}]\!] = \mathsf{state}$. The translation of environments is also straightforward; we make explicit that state variables are of type state.

Translation of environments: $[\![E]\!]$

$[\![\varnothing]\!] \triangleq \varnothing$
$[\![E, \mu]\!] \triangleq [\![E]\!], [\![\mu]\!]$
$[\![\alpha <: \alpha']\!] \triangleq (\alpha <: \alpha')$
$[\![s]\!] \triangleq (s : \mathsf{state})$
$[\![x : T]\!] \triangleq x : \mathcal{T}[\![T]\!]$

The state-passing translation of expressions and values is type-preserving.

Proposition 7 (*Static Adequacy*)

(1) If $E \vdash \diamond$ then $[\![E]\!] \vdash \diamond$.
(2) If $E \vdash T$ then $[\![E]\!] \vdash \mathcal{T}[\![T]\!]$.
(3) If $E \vdash F$ then $[\![E]\!] \vdash \mathcal{T}[\![F]\!]$.
(4) If $E \vdash C$ fo then $[\![E]\!] \vdash [\![C]\!]$ fo.
(5) If $E \vdash T <: U$ then $[\![E]\!] \vdash \mathcal{T}[\![T]\!] <: \mathcal{T}[\![U]\!]$.
(6) If $E \vdash F <: G$ then $[\![E]\!] \vdash \mathcal{T}[\![F]\!] <: \mathcal{T}[\![G]\!]$.
(7) If $E \vdash M : T$ then $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$.
(8) If $E \vdash A : F$ then $[\![E]\!] \vdash \mathcal{E}[\![A]\!] : \mathcal{T}[\![F]\!]$ using (R Val Fun) as the top-level rule of the derivation.

The proof of the proposition is by induction on the derivation, and relies on several lemmas, as follows.

Lemma 8 (*Free Variable Preservation*)

(1) $\mathrm{fnfv}(\mathcal{V}[\![M]\!]) = \mathrm{fv}(M)$, $\mathrm{fnfv}(\mathcal{E}[\![A]\!]) = \mathrm{fv}(A)$ and $\mathrm{fnfv}(\mathcal{E}_c[\![A]\!]s) = \mathrm{fv}(A) \cup \{s\}$.
(2) $\mathrm{fnfv}(\mathcal{T}[\![T]\!]) = \mathrm{fv}(T)$ and $\mathrm{fnfv}(\mathcal{T}[\![F]\!]) = \mathrm{fv}(F)$.
(3) $\mathrm{fnfv}([\![\mu]\!]) = \mathrm{fv}(\mu)$ and $\mathrm{fnfv}([\![E]\!]) = \mathrm{fv}(E)$.

Proof
By structural inductions. □

Lemma 9 (*Environment Translation*)

(1) $\mathrm{dom}([\![\mu]\!]) = \mathrm{dom}(\mu)$ and $\mathrm{dom}([\![\mathrm{E}]\!]) = \mathrm{dom}(\mathrm{E})$.
(2) If $\mu \in E$ then $[\![\mu]\!] \in [\![E]\!]$.

Proof
By induction on E. □

Lemma 10 (*Substitutivity*)

(1) $\mathcal{V}[\![N\{^{M}/_{x}\}]\!] = \mathcal{V}[\![N]\!]\{^{\mathcal{V}[\![M]\!]}/_{x}\}$.
(2) $\mathcal{E}[\![A\{^{M}/_{x}\}]\!] = \mathcal{E}[\![A]\!]\{^{\mathcal{V}[\![M]\!]}/_{x}\}$.
(3) $\mathcal{E}_c[\![A\{^{M}/_{x}\}]\!]s = \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_{x}\}$ if $s \notin \mathrm{fv}(M, x)$.
(4) $\mathcal{T}[\![T\{^{M}/_{x}\}]\!] = \mathcal{T}[\![T]\!]\{^{\mathcal{V}[\![M]\!]}/_{x}\}$.
(5) $\mathcal{T}[\![F\{^{M}/_{x}\}]\!] = \mathcal{T}[\![F]\!]\{^{\mathcal{V}[\![M]\!]}/_{x}\}$.
(6) $[\![C\{^{M}/_{x}\}]\!] = [\![C]\!]\{^{\mathcal{V}[\![M]\!]}/_{x}\}$.
(7) $\mathcal{T}[\![T\{^{U}/_{\alpha}\}]\!] = \mathcal{T}[\![T]\!]\{^{\mathcal{T}[\![U]\!]}/_{\alpha}\}$.
(8) $\mathcal{T}[\![F\{^{U}/_{\alpha}\}]\!] = \mathcal{T}[\![F]\!]\{^{\mathcal{T}[\![U]\!]}/_{\alpha}\}$.

Proof
By structural inductions. □

B.1 Safety

We define $[\![\mathcal{R}]\!]$ by $[\![\mathbf{let}\ x = \mathcal{R}\ \mathbf{in}\ B]\!] \triangleq \mathbf{let}\ y = [\![\mathcal{R}]\!]\ \mathbf{in}\ \mathbf{let}\ (x, s') = y\ \mathbf{in}\ \mathcal{E}_{\mathrm{c}}[\![B]\!]s'$ and $[\![[]]\!] \triangleq []$. When S is the multiset $\{C_1, \ldots, C_n\}$, we write $[\![S]\!] \triangleq \{[\![C_1]\!], \ldots, [\![C_n]\!]\}$ and $[\![\mathbf{assume}\ S]\!] \triangleq \mathbf{assume}\ [\![C_1]\!] \Rsh \cdots \Rsh \mathbf{assume}\ [\![C_n]\!] \Rsh ()$.

Lemma 11 (*Static Safety*)
If $[\![\mathbf{assume}\ S]\!] \Rsh \mathcal{E}_{\mathrm{c}}[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_{s}\}$ is statically safe, then (A, M, S) has not failed.

Proof
The configuration (A, M, S) has failed iff $A = \mathcal{R}[\mathbf{assert}\ (s)C]$ and we cannot derive $S \vdash C\{^{M}/_{s}\}$. If $A = \mathcal{R}[\mathbf{assert}\ (s)C]$ then $B \triangleq \mathcal{E}_{\mathrm{c}}[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_{s}\} = [\![\mathcal{R}]\!][\mathbf{let}\ \_ = \mathbf{assert}\ C\{^{\mathcal{V}[\![M]\!]}/_{s}\}\ \mathbf{in}\ ((), s)]$. If $[\![\mathbf{assume}\ S]\!] \Rsh B$ is statically safe, then $[\![S]\!] \vdash C\{^{\mathcal{V}[\![M]\!]}/_{s}\}$. Thus $[\![S]\!]$ and $C\{^{\mathcal{V}[\![M]\!]}/_{s}\}$ are of first order. The translations of formulas send embedded functions to functions, so it follows that S and $C\{^{M}/_{s}\}$ must also be of first order. By Corollary 1, $S \vdash C\{^{\mathcal{V}[\![M]\!]}/_{s}\}$. □

Lemma 12 (*Simulation*)
If $(A_0, N_0, S_0) \rightarrow (A_1, N_1, S_1)$ then $[\![\mathbf{assume}\ S_0]\!] \Rsh \mathcal{E}_{\mathrm{c}}[\![A_0]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_{s}\} \rightarrow^{*} [\![\mathbf{assume}\ S_1]\!] \Rsh \mathcal{E}_{\mathrm{c}}[\![A_1]\!]s\{^{\mathcal{V}[\![N_1]\!]}/_{s}\}$

Proof
By induction on the derivation of the transition. □

Corollary 2 (*Safety Preservation*)
If $\mathcal{E}[\![A]\!]\mathcal{V}[\![M]\!]$ is safe then $(A, M, \varnothing)$ is safe.

Proof
By Lemma A.1, Lemma 12 and the definition of safety. □

Corollary 3 (*Safety by Well-typed Translation*)
If $\varnothing \vdash \mathcal{E}[\![A]\!]\mathcal{V}[\![M]\!] : T$ then $(A, M, \varnothing)$ is safe.

Proof
By Corollary 2 and Theorem 2 (Safety of RCF). □

Theorem 3 (*Theorem 1* (*Safety*))
If $\varnothing \vdash A : \{(s)C\}\ \_ : T\ \{(s')\mathsf{True}\}$, $\varnothing \vdash C\{^{M}/_{s}\}$ and $\varnothing \vdash M : \mathsf{state}$ then configuration $(A, M, \varnothing)$ is safe.

Proof
By Corollary 3, it suffices to prove $\varnothing \vdash \mathcal{E}[\![A]\!]\ \mathcal{V}[\![M]\!] : U$ for some U. By Proposition 7 (Static Adequacy), we have $\varnothing \vdash \mathcal{E}[\![A]\!] : \Pi s : \{s : \mathcal{T}[\![\mathsf{state}]\!] \mid [\![C]\!]\}.\ \Sigma\_ : T.\ \{s' : \mathcal{T}[\![\mathsf{state}]\!] \mid \mathsf{True}\}$ (*) and $\varnothing \vdash M : \mathcal{T}[\![\mathsf{state}]\!]$. By (R Val Refine) $\varnothing \vdash M : \{s : \mathcal{T}[\![\mathsf{state}]\!] \mid [\![C]\!]\}$ (**). Finally, by (R Exp Appl) (*, **) $\varnothing \vdash \mathcal{E}[\![A]\!]\ \mathcal{V}[\![M]\!] : U$ with $U \triangleq \Sigma\_ : T.\ \{s' : \mathcal{T}[\![\mathsf{state}]\!] \mid \mathsf{True}\}$. □

By Corollary 3, we can prove the safety of RIF configurations through type-checking their RCF translation with the refinement type-checker F7.