Roles, Stacks, Histories: A Triple for Hoare

Johannes Borgström Andrew D. Gordon
Riccardo Pucella

November 2009

Technical Report
MSR–TR–2009–97

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

Publication History

This article was written for the proceedings of a meeting at Microsoft Research, Cambridge, on April 16-17, 2009, to celebrate the 75th birthday of Tony Hoare.

Chapter 1
Roles, Stacks, Histories: A Triple for Hoare

Johannes Borgström[1], Andrew D. Gordon[1], and Riccardo Pucella[2]

Abstract Behavioural 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 behavioural 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 behavioural type-checking is to rule out the possibility of unexpected security exceptions, a common problem with code-based access control.

1.1 Introduction

1.1.1 Behavioural Type Systems

Type-checkers for behavioural 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 behavioural type systems. A few examples include memory management (Gifford and Lucassen 1986), adherence to object and communication protocols (Strom and Yemini 1986; DeLine and Fähndrich 2001), dynamic security policies (Pistoia et al. 2007b), authentication properties of security protocols (Gordon and Jeffrey 2003), avoidance of race conditions (Flanagan and Abadi 1999), and many more.

[1]Microsoft Research ·[2]Northeastern University

While the proliferation of behavioural 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 behavioural type systems are based on a diverse range of formalisms, such as calculi of objects, classes, processes, functions, and so on. Checkers for behavioural type systems often make use of specialised 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 behavioural 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.

1.1.2 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 and Pfenning 1991; Xi and 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 and 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 and Bjørner 2008), and Yices (Dutertre and de Moura 2006).

Our implementation experiments are based on the F7 typechecker, 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.

1.1.3 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.

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

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.)

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

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 and 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 and 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 modelled 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 precondi-

tions 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, 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 appendix includes the direct encoding of our calculus RIF within the existing calculus RCF.

Our calculus is similar in spirit to HTT (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 typecheckers for behavioural 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 and Marché (2004) and Nanevski et al. (2008) are instances of Atkey's (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 parameterised monad, since the preconditions and postconditions are of different types as the postcondition can mention the initial state.

1.1.4 Unifying Behavioural 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 behavioural 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 behavioural 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 and 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 and Gordon 2003) and history-based access control (HBAC) (Abadi and Fournet 2003). We implement each of the three access control mechanisms as an application programming interface (API) within RIF.

In each case, checking application code against the API amounts to behavioural typing, and ensures that application code causes no security exceptions. Hence, static checking prevents accidental programming errors in trusted code and both accidental and malicious programming errors in untrusted code.

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 Appendix, and then type-checking the translated code with F7 and Z3.

The contents of the paper are as follows. Section 1.2 considers access control with roles. Section 1.3 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 1.4. Finally, Section 1.5 discusses related work and Section 1.6 offers some conclusions, and a dedication.

Appendix 1.7 recalls the definition of RCF (Bengtson et al. 2008). Appendix 1.8 provides a semantics of the calculus of this paper to RCF.

1.2 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, role-based access control (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 role-based access control 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 role-based access control 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 and Sewell 2004; Becker and 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.

1.2.1 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 role-based access control. 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. CurrentState(s) ⇒ (∀r. Active(r) ⇔ 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 state $\triangleq$ 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 $\{(s_0)C_0\}\,x{:}T\,\{(s_1)C_1\}$. Such a computation type is interpreted

semantically using the refined state monad mentioned in Section 1.1.3, 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 returning 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 not only some set-theoretic predicates (using lists as a representation for sets), but also a predicate Active true exactly when a given role is currently active. To define Active, we rely on a predicate CurrentState, where CurrentState(s) captures the assumptions that s is the current set of active roles; Active then amounts to membership in the set of active roles. We can only reason about Active under the assumption of some CurrentState(s). We shall see that our formulas for reasoning about roles will always be of the form CurrentState(s)⇒..., where s is the current state.

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. We make the assumption that **get**() and **set**() may only be used in the implementation of API functions; in particular, user code cannot use those operations to arbitrarily manipulate the state. The API functions are meant to encapsulate all state manipulation. Beyond the use of **get**() and **set**(), the implementation of the API functions above also use 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 modelling 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) expresses the "file f can be read" permission, given the currently active roles. Here is a simple policy in line with this description:

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

This policy, aside from stating that the superuser can read all files, also states that if the 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) CurrentState(rs) ⇒ CanRead(file)} string {(s)s=rs}
let readFile file =
    assert (rs)(CurrentState(rs) ⇒ CanRead(file));
    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) holds, under the assumption that CurrentState(rs). 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 must be able to prove CanRead("andy.log"). Because SuperUser is active and Active(SuperUser) implies CanRead(file) for any file, we

get CanRead("andy.log"), and we can invoke readFile. The following examples type-check for similar reasons, since Active(FriendOf "Andy") can prove the formula CanRead("andy.log"):

```
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"), and the policy cannot derive the formula CanRead("andy.log") 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.Match(file,"*.txt") ⇒ CanRead(file)
```

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 false
```

The following code therefore type-checks, even when all the activated roles do not by themselves suffice to give a CanRead permission:

```
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 reflect 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 ⇒ (Active(FriendOf(u)) ⇒ CanRead(f)))}
let hasFSReadPermission f u =
        if primReadFSPerm (f,u)
            then assume Active(FriendOf(u)) ⇒ CanRead(f); true
        else false
```

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 Active(FriendOf(`"Andy"`))⇒ CanRead (`"somefile"`) can be used in subsequent expressions—in particular, when calling readFile `"somefile"`. At that point, FriendOf `"Andy"` is active, and therefore CanRead(`"somefile"`) holds.

1.2.2 Extended RBAC: Health Care Policies

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

To model this example, we build on top of the insights gained in Section 1.2.1, 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 $\triangleq$ id ∗ role list ∗ fact list. The API is now somewhat richer, including operations to deal with identifies and the database of facts.

Extended RBAC API:

```
val switch_user : u:id → {(s)True} unit {(s') s'=(u,[],s.store)}
val activate : r:role → {(s)CurrentState(s) ⇒ CanActivate(r)} 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)}
```

Extended RBAC API Implementation:

```
let switch_user u = let (i,rs,db) = get() in set(u,[],db)
let activate r = 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 predicate CurrentState(s) states that s is the current state, and the predicates Ident, Active, and Fact respectively capture the current identity, the currently active roles, and the currently registered facts. The predicate CanActivate is defined by the access control policy, as we shall see below.

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

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

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.

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

val unregister : u:id → r:role → {(s)CurrentState(s) ⇒ Active(Admin)} 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: users can read their own EHR, and a clinicians can read any EHR for which they have received consent.

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

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

```
val readEHR : file:string →
      {(s)CurrentState(s) ⇒ CanReadEHR(Name(file))} string {(t)s=t}
let readEHR file =
    assert (s)(CurrentState(s) ⇒ CanReadEHR(Name(file)));
    (* ... 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.Consented(u) ⇔ (Ident(v) ∧ Fact(HasConsented(u,v)))

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

How can we use the above interface? An 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. Such a workflow can be implemented, for instance, on a machine such as a web server, or a smartcard. The switch_user *u* primitive corresponds to a context switch between users, and may be implemented by having the server present a login window requiring password-based authentication, 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 a clinician—registers *pat* as a patient, and lets *doc* read *pat*'s medical file.

```
val workflow : pat:id → doc:id →
    {(s) CurrentState(s) ⇒ Fact(IsMember(doc,Clinician))} 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.

1.3 Types for Permission-Based Access Control

The role-based access control 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 stack-based access control (Gong 1999; Wallach et al. 2000) and history-based access control (Abadi and Fournet 2003).

The purpose of stack-based access control (SBAC) is to protect trusted functions from untrusted callers. Unless explicitly requested, a permission only holds at runtime if all callers on the call stack statically hold the permission.

History-based access control (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 1.3.2.

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 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.

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

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

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

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 1 (Computing lattice operations). *If P has FLB and $Q, R \in \mathcal{O}_{\text{fin}}(P)$ then $Q \sqcup R = \max(Q \cup R)$ and $Q \sqcap R = \max(\bigcup\{\text{mlbs}(q,r) \mid q \in Q, r \in R\})$.*

Assume that P_1 and P_2 both have finite lower bounds. Then $P_1 \times P_2$ has FLB, with $\text{mlbs}((p_1,p_2),(q_1,q_2)) = \text{mlbs}(p_1,q_1) \times \text{mlbs}(p_2,q_2)$. Furthermore, $P_1 \uplus P_2$ also has FLB, with $\mathcal{O}_{\text{fin}}(P_1 \uplus P_2)$ isomorphic to $\mathcal{O}_{\text{fin}}(P_1) \times \mathcal{O}_{\text{fin}}(P_2)$.

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

- $Q \sqcup R = \max((\downarrow Q) \cup (\downarrow R)) = \max(\downarrow(Q \cup R)) = \max(Q \cup R)$.
- $\downarrow(Q \sqcap R) = \downarrow\max((\downarrow Q) \cap (\downarrow R)) = \downarrow((\downarrow Q) \cap (\downarrow R)) = (\downarrow Q) \cap (\downarrow R)$ since $(\downarrow Q) \cap (\downarrow R)$ is downward closed. Then $p \in (\downarrow Q) \cap (\downarrow R)$ iff $\exists q \in Q, r \in R$ such that $p \leq q$ and $p \leq r$; that is, iff $p \in \downarrow\text{mlbs}(q,r)$. Thus $\downarrow(Q \sqcap R) = \bigcup\{\downarrow\text{mlbs}(q,r) \mid q \in Q, r \in R\} = \downarrow\bigcup\{\text{mlbs}(q,r) \mid q \in Q, r \in R\}$. $\max\downarrow(Q \sqcap R) = Q \sqcap R$ since $Q \sqcap R$ is a cochain, so $Q \sqcap R = \max(\bigcup\{\text{mlbs}(q,r) \mid q \in Q, r \in R\})$.
- $\downarrow\text{mlbs}((p_1,p_2),(q_1,q_2)) = \{(r_1,r_2) \mid r_i \leq p_i \wedge r_i \leq q_i \text{ for } i = 1,2\} = \{(r_1,r_2) \mid r_i \in \downarrow\text{mlbs}(p_i,q_i) \text{ for } i = 1,2\} = (\downarrow\text{mlbs}(p_1,q_1)) \times (\downarrow\text{mlbs}(p_2,q_2)) = \downarrow(\text{mlbs}(p_1,q_1) \times \text{mlbs}(p_2,q_2))$. $\text{mlbs}(p_1,q_1) \times \text{mlbs}(p_2,q_2)$ is a cochain since $\text{mlbs}(p_1,q_1)$ and $\text{mlbs}(p_2,q_2)$ are. Thus $\text{mlbs}((p_1,p_2),(q_1,q_2)) = \text{mlbs}(p_1,q_1) \times \text{mlbs}(p_2,q_2)$.
- The isomorphism is given by $Q \mapsto (Q \cap P_1, Q \cap P_2)$. □

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, and assume the standard lattice axioms relating these to each other and to Subsumed (not shown). 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))}
```

1.3.2 Stack-Based Access Control

In order to compare history- and stack-based access control 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: state $\triangleq$ {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.

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 $(\alpha;[])$SBACcomp for every C, since $\vdash C \Rightarrow$ True 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.

In the implementations of request and demand below, we **assert** that subsumed always returns **true**. This corresponds to requiring that the caller has sufficient permissions. Since no **assert** fails in a well-typed program, any execution of such a program always has sufficient run-time permissions.

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 assertFalse ; failwith "SecurityException: request"

let demand ps =
    let {ast=_; dy=dy} = get() in let x = subsumed ps dy in
    if x then () else assertFalse; failwith "SecurityException: demand"
```

To exercise this framework, we work in a setting with two principals. Agent 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. 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.

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. Moreover, SBAC3 type-checks.

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

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

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
```

1.3.3 History-Based Access Control

The HBAC calling convention was defined (Abadi and 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. 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. The HBAC calling convention is implemented by the function hbacReturn, whic resets the static condition and reduces the dynamic conditions to at most the initial ones.

The HBAC API extends the SBAC API with 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 ar-

gument 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. accept ps takes an arbitrary HBACthunk, 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:

```
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)}
val grant: ps:Perms →(α;ps,[]) HBACthunk →
  {(s0) Subsumed(ps,s0.ast)} α{(s3) s3.ast=s0.ast ∧ Subsumed(s3.dy,s0.dy)}
val accept: ps:Perms →(α;[],[]) HBACthunk →
  {(s) Subsumed(ps,s.ast)} α{(t)s.ast = t.ast ∧
   (∀qs. Subsumed(qs,ps) ∧ Subsumed(qs,s.dy) ⇒ Subsumed(qs,t.dy))}
```

Here (α;req) SBACcomp is a subtype of (α;req,pres) HBACcomp for every pres.

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}
private val getDy: unit →{(s) True} dy:Perms {(t) t = s ∧ t.dy = dy}
let getDy () = let {ast=_;dy=dy} = get() in dy
let grant ps a = let dy = getDy () in request ps; let res = a () in permitOnly dy; res
let accept ps a = let dy = getDy () in let res = a () in request ps; permitOnly dy; res
```

As seen above, the postcondition of an hbacComp does not set a lower bound for the dynamic permissions. Because of this, we cannot type-check the cleanup function with argument type string HBACcomp. Indeed, in this example the dynamic permissions are reduced to at most Applet, which is insufficient to delete the password file.

In example HBAC1 we instead use 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, HBAC1 type-checks.

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

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 HBAC1: (unit;[],[]) HBACthunk = fun () →
    let olds = become Applet in cleanup_grant Applet_fun ; hbacReturn olds
```

However, 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.

```
type cleanupArg: (string;[],System) HBACthunk

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

//Does not type-check, since aFunHBAC is not a cleanupArg
let HBAC2 = fun () →let olds = become Applet in
    cleanupHBAC aFunHBAC ; hbacReturn olds

let expand:string →cleanupArg = fun n →fun () →
 let olds = become System in let res = n in hbacReturn olds ; n
let HBAC3:(unit;[],[]) HBACthunk = fun () →let olds = become Applet in
    cleanup_HBAC (expand "passwd") ; 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. 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 an explicit 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 dy = getDy () 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
```

1.4 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 syntax and operational semantics in Section 1.4.1 and Section 1.4.2. Section 1.4.3 describes the type system of RIF and its soundness with respect to the operational semantics. Finally, Section 1.4.4 describes how the calculus may be instantiated by suitable choice of the state type.

1.4.1 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
$A,B ::=$	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)

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.

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

In our formulation of FPC, the syntax of expressions is in a reduced form in the style of A-normal form (Sabry and 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 $\mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$, and constructor matching $\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B$.

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. We also assume an untyped first-order logic with equality over values, equipped with a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas.

Completing the Syntax: Adding Global State to the Fixpoint Calculus

$A,B ::=$	expression
$\cdots$	expressions of the Fixpoint Calculus
$\mathbf{get}()$	get current state
$\mathbf{set}(M)$	set current state
$\mathbf{assume}\ (s)C$	assumption of formula C (scope of s is C)
$\mathbf{assert}\ (s)C$	assertion of formula C (scope of s is C)
$C ::=$	formula
$p(M_1,\ldots,M_n)$	predicate – p a predicate symbol
$M = M'$	equation
$C \wedge C'$	conjunction
$C \vee C'$	disjunction
$\neg C$	negation
$\forall x.C$	universal quantification
$\exists x.C$	existential quantification

A formula C is first-order if and only if it only contains first-order values. A collection S is first-order if and only if it only contains first-order formulas.

The expression $\mathbf{get}()$ returns the current state as its value. The expression $\mathbf{set}(M)$ 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 expression $\mathbf{assume}\ (s)C$ adds the formula $C\{^M/_s\}$ to the logged formulas, where M is the current state, and returns $()$. The expression $\mathbf{assert}\ (s)C$ 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 $\mathbf{let}\ \_ = A\ \mathbf{in}\ B$. We define boolean values as $\mathbf{false} \triangleq \mathsf{inl}\ ()$ and $\mathbf{true} \triangleq \mathsf{inr}\ ()$. Conditional statements can then be defined as $\mathbf{if}\ M\ \mathbf{then}\ A\ \mathbf{else}\ B \triangleq \mathbf{match}\ M\ \mathbf{with}\ \mathsf{inr}\ x \to A\ \mathbf{else}\ B$. We write $\mathbf{let\ rec}\ f\ x = A\ \mathbf{in}\ B$ 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 $\mathbf{let}\ f\ x = A$ for $\mathbf{let}\ f = \mathbf{fun}\, x \to A$, $\mathbf{if}\ A\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2$ for $\mathbf{let}\ x = A\ \mathbf{in}\ \mathbf{if}\ x\ \mathbf{then}\ B_1\ \mathbf{else}\ B_2$ (where $x \notin \mathrm{fv}(B_1, B_2)$), $\mathbf{let}\ (x,y) = A\ \mathbf{in}\ B$ for $\mathbf{let}\ z = A\ \mathbf{in}\ \mathbf{let}\ (x,y) = z\ \mathbf{in}\ B$ (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.

1.4.2 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 first-order if and only if N, S and all formulas occurring in A are first-order.

The 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)

$(\textbf{assume}\ (s)C, N, S) \longrightarrow ((), N, S \cup \{C\{^N/_s\}\})$	(RED ASSUME)
$(\textbf{assert}\ (s)C, N, S) \longrightarrow ((), N, S)$	(RED ASSERT)

We say an expression is safe if none of its assertions may fail at runtime. A configuration (A, N, S) has *failed* when $A = \mathcal{R}[\textbf{assert}\ (s)C]$ for some evaluation context $\mathcal{R}$, where $S \cup \{C\{^N/_s\}\}$ is not 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.

1.4.3 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
$\quad \alpha$	type variable
$\quad$ unit	unit type
$\quad \Pi x : T.\ F$	dependent function type (scope of x is F)
$\quad \Sigma x : T.\ U$	dependent pair type (scope of x is U)
$\quad T + U$	disjoint sum type
$\quad \mu\alpha.T$	iso-recursive type (scope of α is T)
$F, G ::=$	computation type
$\quad \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\}$	(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 $x : T \to F$ 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 $T \to F$ for $x : T \to F$. A value type T is first-order if and only if T contains no occurrences of $\Pi x : U.\ F$ (and hence contains no computation types). 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 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 $x : T \to y : U \to F$ 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 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 pair-wise 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:

$$\text{(Env Empty)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env Entry)}\ \frac{E \vdash \diamond \quad \mathrm{fv}(\mu) \subseteq \mathrm{fov}(E) \quad \mathrm{dom}(\mu) \cap \mathrm{dom}(E) = \varnothing}{E, \mu \vdash \diamond} \qquad \text{(Form)}\ \frac{E \vdash \diamond \quad C \text{ is first-order} \quad \mathrm{fv}(C) \subseteq \mathrm{fov}(E)}{E \vdash C \text{ fo}} \qquad \text{(Env Type)}\ \frac{E \vdash \diamond \quad \mathrm{fv}(T) \subseteq \mathrm{fov}(E)}{E \vdash T}$$

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.

General Rules for Expressions:

$$\text{(Exp Return)}\ \frac{E, s_0 \vdash M : T}{E \vdash M : \{(s_0)\mathsf{True}\}\_{:}T\,\{(s_1)s_0 = s_1\}} \qquad \text{(Stateful Exp Let)}\ \frac{\begin{array}{l}E \vdash A : \{(s_0)C_0\}\,x_1{:}T_1\,\{(s_1)C_1\} \\ E, s_0, x_1 : T_1 \vdash B : \{(s_1)C_1\}\,x_2{:}T_2\,\{(s_2)C_2\} \\ \{s_1, x_1\} \cap \mathrm{fv}(T_2, C_2) = \varnothing\end{array}}{E \vdash \mathbf{let}\ x_1 = A\ \mathbf{in}\ B : \{(s_0)C_0\}\,x_2{:}T_2\,\{(s_2)C_2\}}$$

$$\text{(Exp Eq)}\ \frac{\begin{array}{l}E \vdash M : T \quad E \vdash N : U \quad x \notin \mathrm{fv}(M,N) \quad E, s_0, s_1 \vdash C \text{ fo} \\ C = (s_0 = s_1) \wedge (x = \mathbf{true} \Leftrightarrow M = N) \quad T, U \text{ first-order}\end{array}}{E \vdash M = N : \{(s_0)\mathsf{True}\}\,x{:}\mathsf{bool}\,\{(s_1)C\}}$$

In (Exp Return), when returning a value from a computation, the state is unchanged. 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:

$$\text{(Exp Assume)}\ \frac{E, s_0, s_1 \vdash \diamond \quad E, s_0 \vdash C \text{ fo}}{E \vdash \mathbf{assume}\ (s_0)C : \{(s_0)\mathsf{True}\}\,\mathsf{unit}\,\{(s_1)((s_0 = s_1) \wedge C)\}}$$

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

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:

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

(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 $\mathbf{get}()$ records that the value read is the current state. In (STATEFUL SET), the postcondition of $\mathbf{set}(M)$ states that M is the new state. The postcondition of $\mathbf{set}(M)$ does not mention the initial state. We can recover this information through subtyping, below.

Subtyping for Computations:

(SUB COMP)
$$\frac{\begin{array}{l} E, s_0 \vdash C_0 \text{ fo} \quad E, s_0 \vdash C_0' \text{ fo} \\ E, s_0, x{:}T, s_1 \vdash C_1 \text{ fo} \quad E, s_0, x{:}T', s_1 \vdash C_1' \text{ fo} \\ C_0' \vdash C_0 \quad E, s_0 \vdash T <: T' \quad (C_0' \wedge C_1) \vdash C_1' \end{array}}{E \vdash \{(s_0)C_0\}\,x{:}T\,\{(s_1)C_1\} <: \{(s_0)C_0'\}\,x{:}T'\,\{(s_1)C_1'\}}$$

(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, but only if the value being type-checked is first-order.

Augmenting the Precondition of a Computation Type:

$$C \rightsquigarrow_T F \triangleq \{(s_1)C \wedge C_1\}\,x{:}U\,\{(s_2)C_2\} \qquad \text{if } T \text{ first-order}$$
$$C \rightsquigarrow_T F \triangleq F \qquad \text{otherwise}$$
where $F = \{(s_1)C_1\}\,x{:}U\,\{(s_2)C_2\}$ and $s_1 \notin \mathrm{fv}(C)$

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

In (STATEFUL EXP SPLIT), when splitting a pair, we strengthen the precondition of the computation with the information derived from the pair split.

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

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

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. As in (STATEFUL EXP SPLIT), the rule (STATEFUL EXP MATCH INL INR FOLD) strengthens the preconditions of the different branches with information derived from the branching condition.

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 and Sangiorgi 1996; Aspinall and 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 1 (Safety).
If $\varnothing \vdash A : \{(s)C\}\ \_ : T\ \{(s')\mathsf{True}\}$, $\varnothing \vdash C\{^M/_s\}$ *and* $\varnothing \vdash M : \mathsf{state}$ *then the 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 $\mathcal{M}_{C_0,C_1}(\llbracket T \rrbracket)$ described in the introduction, where $\llbracket T \rrbracket$ is the translation of the value type T. We prove the translation to preserve 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 Appendix.

1.4.4 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** $(s)C$. (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 $i \in [1..k]$: $f_1 : T_1, \ldots, f_k : T_k \vdash$ **let** $\_ =$ **assume** $(s)C$ **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 $f_1, \ldots, f_k$ with implementations $M_1, \ldots, M_k$ and types $T_1, \ldots, T_k$ corresponds to type-checking $f_1 : T_1, \ldots, f_k : T_k \vdash$ **let** $\_ =$ **assume** $(s)C$ **in** $A : F$ and executing A in the context of that module corresponds to executing the expression **assume** $(s)C$;**let** $f_1 = M_1$ **in** $\ldots$**let** $f_n = M_n$ **in** A.

As illustrated in previous sections, to use our calculus, we first *instantiate* it with an *extension API module* that embodies the behavioural 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.

1.5 Related Work

We discuss related work on type systems for access control. Pottier et al. (2005) develop a type and effect system for stack-based access control. 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 stack-based and history-based access control (Banerjee and Naumann 2005b,a; 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 (Rondon et al. 2008; Knowles and Flanagan 2007).

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 typechecking code to ensure conformance to a logic-based authorization policy. A series of papers develops the idea for distributed systems modelled 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 1.2 is different in that we deal with roles that may be activated and deactivated.

1.6 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 behavioural 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 behavioural type systems using types refined with logical formulas, than to invent still more syntax.

Acknowledgements 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.

Appendix

1.7 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 (**?**).

Formally, RCF 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 RCF values.

Syntax of Formulas:

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

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

Properties of Deducibility: $S \vdash C$

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

$$\text{(AXIOM)}\ \frac{}{C \vdash C} \qquad \text{(MON)}\ \frac{S \vdash C}{S,C' \vdash C} \qquad \text{(SUBST)}\ \frac{S \vdash C}{S\sigma \vdash C\sigma} \qquad \text{(CUT)}\ \frac{S \vdash C \quad S,C \vdash C'}{S \vdash C'}$$

$$\text{(AND INTRO)}\ \frac{S \vdash C_0 \quad S \vdash C_1}{S \vdash C_0 \wedge C_1} \qquad \text{(AND ELIM)}\ \frac{S \vdash C_0 \wedge C_1}{S \vdash C_i} \qquad \text{(OR INTRO)}\ \frac{S \vdash C_i}{S \vdash C_0 \vee C_1} \quad i = 0,1$$

$$\text{(EQ)}\ \frac{}{\varnothing \vdash M = M} \qquad \text{(INEQ)}\ \frac{M \neq N \quad \mathrm{fv}(M,N) = \varnothing}{\varnothing \vdash M \neq N} \qquad \text{(INEQ CONS)}\ \frac{h\,N = M \text{ for no } N \quad \mathrm{fv}(M) = \varnothing}{\varnothing \vdash \forall x.hx \neq M}$$

$$\text{(EXISTS INTRO)}\ \frac{S \vdash C\{M/x\}}{S \vdash \exists x.C} \qquad \text{(EXISTS ELIM)}\ \frac{S \vdash \exists x.C \quad S, C \vdash C' \quad x \notin \mathrm{fv}(S, C')}{S \vdash C'}$$

FOL/FO is first-order logic with the following additional axiom schemas (Bengtson et al. 2008). 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 term. Each RCF name is a constant, that is, a nullary syntactic function symbol.

Additional Rules for FOL/FO:

$$\text{(F DISJOINT)}\ \frac{f \neq f' \text{ syntactic}}{S \vdash \forall \mathbf{x}.\forall \mathbf{y}.f(\mathbf{x}) \neq f'(\mathbf{y})} \qquad \text{(F INJECTIVE)}\ \frac{f \text{ syntactic}}{S \vdash \forall \mathbf{x}.\forall \mathbf{y}.f(\mathbf{x}) = f(\mathbf{y}) \Rightarrow \mathbf{x} = \mathbf{y}}$$

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.

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 $\triangleq \mathsf{inl}\ ()$ **true** $\triangleq \mathsf{inr}\ ()$

We follow similar syntactic conventions as in RIF but additionally its syntax includes names as well as variables. We write $\mathrm{fn}(\phi)$ for the set of names occurring free in ϕ, and also $\mathrm{fnfv}(\phi) = \mathrm{fv}(\phi) \cup \mathrm{fn}(\phi)$, the set of both names and variables occurring free in ϕ.

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 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

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

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

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

Let structure **S** be *statically safe* if and only if, for all $p \in 1..o$ and C,
if $e_p =$ **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)
$A \Rrightarrow A' \Rightarrow$ **let** $x = A$ **in** $B \Rrightarrow$ **let** $x = A'$ **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 ())
$a \notin \mathrm{fn}(A') \Rightarrow A' \Rsh ((\nu a)A) \Rrightarrow (\nu a)(A' \Rsh A)$	(HEAT RES FORK 1)
$a \notin \mathrm{fn}(A') \Rightarrow ((\nu a)A) \Rsh A' \Rrightarrow (\nu a)(A \Rsh A')$	(HEAT RES FORK 2)
$a \notin \mathrm{fn}(B) \Rightarrow$ $\mathbf{let}\ x = (\nu a)A\ \mathbf{in}\ B \Rrightarrow (\nu a)\mathbf{let}\ x = A\ \mathbf{in}\ B$	(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)

Lemma 1 (Structure). *For every expression A, there is a structure $\mathbf{S}$ with $A \Rrightarrow \mathbf{S}$.*

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)
$A \to A' \Rightarrow (A \Rsh B) \to (A' \Rsh B)$	(R RED FORK 1)
$A \to A' \Rightarrow (B \Rsh A) \to (B \Rsh A')$	(R RED FORK 2)
$A \to A' \quad$ 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 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.

Some Derivable Types:

$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$ (ok-type)
$\mathsf{bool} \triangleq \mathsf{unit}+\mathsf{unit}$
$\mathsf{nat} \triangleq \mu\alpha.\mathsf{unit}+\alpha$
$(T)\mathsf{list} \triangleq \mu\alpha.\mathsf{unit}+(T\times\alpha)$
$[x_1:T_1]\{C_1\}\to U \triangleq \Pi x_1:\{x_1:T_1 \mid C_1\}.\ U$
$(x_1:T_1 * \cdots * x_n:T_n)\{C\} \triangleq \begin{cases} \Sigma x_1:T_1.\ \ldots \Sigma x_{n-1}:T_{n-1}.\ \{x_n:T_n \mid C\} & \text{if } n>0 \\ \{C\} & \text{otherwise} \end{cases}$

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

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

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

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

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.

Rules of Well-Formedness and Deduction:

(R ENV EMPTY)
$$\frac{}{\varnothing \vdash \diamond}$$

(R ENV ENTRY)
$$\frac{E \vdash \diamond \quad \mathrm{fn\,fv}(\mu) \subseteq \mathrm{dom}(E) \quad \mathrm{dom}(\mu) \cap \mathrm{dom}(E) = \varnothing}{E, \mu \vdash \diamond}$$

(R TYPE)
$$\frac{E \vdash \diamond \quad \mathrm{fn\,fv}(T) \subseteq \mathrm{dom}(E)}{E \vdash T}$$

(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)
$$\frac{E \vdash T \quad \mathrm{recvar}(E) \cap \mathrm{fn\,fv}(T) = \varnothing}{E \vdash T <: T}$$

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

(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 write $E + C \triangleq E, \_ : \{C\}$ if $E \vdash C$ fo. Otherwise, $E + C \triangleq E$.

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

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

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

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

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

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

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

Admissible Rules:

$$\text{(R BOUND WEAKENING)}\quad \frac{E \vdash T' <: T \quad E, x : T, E' \vdash \mathcal{J}}{E, x : T', E' \vdash \mathcal{J}} \qquad \text{(R BOUND UNREFINE)}\quad \frac{E, x : T, E' \vdash \mathcal{J}}{E, x : \{x : T \mid C\}, E' \vdash \mathcal{J}}$$

$$\text{(R WEAKENING)}\quad \frac{E, E' \vdash \mathcal{J}}{E, x : T, E' \vdash \mathcal{J}} \qquad \text{(R EXCHANGE)}\quad \frac{E, \mu_1, \mu_2, E' \vdash \mathcal{J} \quad \mathrm{dom}(\mu_1) \cap \mathrm{fnfv}(\mu_2) = \varnothing}{E, \mu_2, \mu_1, E' \vdash \mathcal{J}}$$

$$\text{(R SUB REFINE)}\quad \frac{E \vdash T <: T' \quad E, x : \{x : T \mid C\} \vdash C'}{E \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}} \qquad \text{(R SUB REFINE LEFT REFL)}\quad \frac{E \vdash \{x : T \mid C\}}{E \vdash \{x : T \mid C\} <: T}$$

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

Lemma 2 (Static Safety).
If $\varnothing \vdash \mathbf{S} : T$ *then* $\mathbf{S}$ *is statically safe.*

Proposition 2 ($\Rrightarrow$ Preserves Types).
If E *is executable and* $E \vdash A : T$ *and* $A \Rrightarrow A'$ *then* $E \vdash A' : T$.

Proposition 3 ($\rightarrow$ 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).
If $\varnothing \vdash A : T$ *then* A *is safe.*

1.8 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: $\mathcal{V}[\![M]\!]$

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

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: $[\![C]\!]$

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

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

Lemma 3. *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 first-order, then $S \vdash C$ iff $[\![S]\!] \vdash [\![C]\!]$.*

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** $\mathcal{E}_c[\![A]\!]s$

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

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

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 4 (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 relies on several lemmas, as follows.

Lemma 4 (Free Variable Preservation).

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

Proof. By structural inductions. □

Lemma 5 (Environment Translation).

(1) $\mathrm{dom}(\llbracket\mu\rrbracket) = \mathrm{dom}(\mu)$ *and* $\mathrm{dom}(\llbracket E\rrbracket) = \mathrm{dom}(E)$.
(2) *If* $\mu \in E$ *then* $\llbracket\mu\rrbracket \in \llbracket E\rrbracket$.

Proof. By induction on E; trivial. □

Lemma 6 (Substitutivity).

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

Proof. By structural inductions. □

Proof (Proposition 4).
The rule (R BOUND UNREFINE) is used throughout this proof, with $T = \mathsf{state}$.

(1) By induction on E. The base case $E = \varnothing$ is trivial.
For the induction case, we assume that $E \vdash \diamond$ and $E = E', \mu$. By induction $\llbracket E'\rrbracket \vdash \diamond$. By assumption $\mathrm{dom}(E') \cap \mathrm{dom}(\mu) = \varnothing$ and $\mathrm{dom}(E') \supseteq \mathrm{fv}(\mu)$. By Lemma 5 $\mathrm{dom}(E') = \mathrm{dom}(\llbracket E'\rrbracket)$ and $\mathrm{dom}(\mu) = \mathrm{dom}(\llbracket\mu\rrbracket)$. By Lemma 4 $\mathrm{fv}(\mu) = \mathrm{fn}\,\mathrm{fv}(\llbracket\mu\rrbracket)$. Thus $\mathrm{dom}(\llbracket E'\rrbracket) \cap \mathrm{dom}(\llbracket\mu\rrbracket) = \varnothing$ and $\mathrm{dom}(\llbracket E'\rrbracket) \supseteq \mathrm{fn}\,\mathrm{fv}(\llbracket\mu\rrbracket)$, so $\llbracket E\rrbracket \vdash \diamond$.
(2,3,4) Since $\mathcal{T}\llbracket T\rrbracket$ is first-order whenever T is, this follows from Lemma 4 and Lemma 5.
(5,6) By induction on the derivation. Base cases:

(SUB UNIT) derives $E \vdash \mathsf{unit} <: \mathsf{unit}$ from $E \vdash \diamond$. By 1, $\llbracket E\rrbracket \vdash \diamond$, so $\llbracket E\rrbracket \vdash \mathsf{unit} <: \mathsf{unit}$.
(SUB VAR) derives $E \vdash \alpha <: \alpha'$ from $(\alpha <: \alpha') \in E$ and $E \vdash \diamond$. By 1, $\llbracket E\rrbracket \vdash \diamond$, and by Lemma 5 we have $(\alpha <: \alpha') \in \llbracket E\rrbracket$. Thus $\llbracket E\rrbracket \vdash \alpha <: \alpha'$.

Induction cases:

(SUB SUM) derives $E \vdash (T + U) <: (T' + U')$ from $E \vdash T <: T'$ and $E \vdash U <: U'$. By induction $\llbracket E\rrbracket \vdash \mathcal{T}\llbracket T\rrbracket <: \mathcal{T}\llbracket T'\rrbracket$ and $\llbracket E\rrbracket \vdash \mathcal{T}\llbracket U\rrbracket <: \mathcal{T}\llbracket U'\rrbracket$, so $\llbracket E\rrbracket \vdash (\mathcal{T}\llbracket T\rrbracket + \mathcal{T}\llbracket U\rrbracket) <: (\mathcal{T}\llbracket T'\rrbracket + \mathcal{T}\llbracket U'\rrbracket)$. Applying the definition of $\mathcal{T}\llbracket T + U\rrbracket$, this gives $\llbracket E\rrbracket \vdash \mathcal{T}\llbracket T + U\rrbracket <: \mathcal{T}\llbracket T' + U'\rrbracket$.
(SUB PAIR) derives $E \vdash \Sigma x : T.\ U <: \Sigma x : T'.\ U'$ from $E \vdash T <: T'$ and $E, x : T \vdash U <: U'$. By induction $\llbracket E\rrbracket \vdash \mathcal{T}\llbracket T\rrbracket <: \mathcal{T}\llbracket T'\rrbracket$ and $\llbracket E, x : T\rrbracket \vdash \mathcal{T}\llbracket U\rrbracket <: \mathcal{T}\llbracket U'\rrbracket$. By definition $\llbracket E, x : T\rrbracket = \llbracket E\rrbracket, x : \mathcal{T}\llbracket T\rrbracket$, so $\llbracket E\rrbracket \vdash \Sigma x : \mathcal{T}\llbracket T\rrbracket.\ \mathcal{T}\llbracket U\rrbracket <: \Sigma x : \mathcal{T}\llbracket T'\rrbracket.\ \mathcal{T}\llbracket U'\rrbracket$. Applying the definition of $\mathcal{T}\llbracket \Sigma x : T.\ U\rrbracket$, this gives $\llbracket E\rrbracket \vdash \mathcal{T}\llbracket \Sigma x : T.\ U\rrbracket <: \mathcal{T}\llbracket \Sigma x : T'.\ U'\rrbracket$.

(SUB REC) derives $(\mu\alpha.T) <: (\mu\alpha'.T')$ from $E, \alpha <: \alpha' \vdash T <: T'$, $\alpha \notin \mathrm{fv}(T')$ and $\alpha' \notin \mathrm{fv}(T)$.
By induction $[\![E, \alpha <: \alpha']\!] \vdash \mathcal{T}[\![T]\!] <: \mathcal{T}[\![T']\!]$. By definition $[\![E, \alpha <: \alpha']\!] = [\![E]\!], \alpha <: \alpha'$. By Lemma 4 $\alpha \notin \mathrm{fv}(\mathcal{T}[\![T']\!])$ and $\alpha' \notin \mathrm{fv}(\mathcal{T}[\![T]\!])$, so $[\![E]\!] \vdash (\mu\alpha.\mathcal{T}[\![T]\!]) <: (\mu\alpha'.\mathcal{T}[\![T']\!])$. Applying the definition of $\mathcal{T}[\![\mu\beta.U]\!]$, this gives $[\![E]\!] \vdash \mathcal{T}[\![\mu\alpha.T]\!] <: \mathcal{T}[\![\mu\alpha'.T']\!]$.

(SUB COMP) derives $E \vdash F <: F'$ from $\mathrm{fv}(C_0, C_0') \subseteq \mathrm{dom}(E, s_0)$, $\mathrm{fv}(C_1, C_1') \subseteq \mathrm{dom}(E, s_0, x, s_1)$, $\mathbf{vld}(C_0' \Rightarrow C_0)$, $\mathbf{vld}((C_0' \wedge C_1) \Rightarrow C_1')$ and $E, s_0 \vdash T <: T'$, where $F = \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\}$, and the same for F'.
By induction $[\![E]\!], s_0 : \mathsf{state} \vdash \mathcal{T}[\![T]\!] <: \mathcal{T}[\![T']\!]$ (*). By Lemma 4 and Lemma 5 $\mathrm{fn}\,\mathrm{fv}([\![C_0]\!], [\![C_0']\!]) \subseteq \mathrm{dom}([\![E, s_0]\!])$ and $\mathrm{fv}([\![C_1]\!], [\![C_1']\!]) \subseteq \mathrm{dom}([\![E, s_0, x, s_1]\!])$. By Corollary 1, $\mathbf{vld}([\![C_0']\!] \Rightarrow [\![C_0]\!])$ (**) and $\mathbf{vld}(([\![C_0']\!] \wedge [\![C_1]\!]) \Rightarrow [\![C_1']\!])$ (***). By definition $\mathcal{T}[\![F]\!] = \Pi s : \{s_0 : \mathsf{state} \mid [\![C_0]\!]\}.\ \Sigma x : T.\ \{s_1 : \mathsf{state} \mid [\![C_1]\!]\}$ and the same for F'.
We derive $[\![E]\!] \vdash \mathcal{T}[\![F]\!] <: \mathcal{T}[\![F']\!]$ by (R SUB FUN) ((R SUB REFINE) ((R SUB REFL), (R DERIVE) (**)), (R SUB PAIR) ((R BOUND WEAKENING) (*), (R SUB REFINE) ((R SUB REFL), (R DERIVE) (***)))).

(7,8) By induction on the derivation. Base cases:

(VAL VAR) derives $E \vdash x : T$ from $(x : T) \in E$ and $E \vdash \diamond$. By Lemma 5 $[\![x : T]\!] \in [\![E]\!]$, where $[\![x : T]\!] = x : \mathcal{T}[\![T]\!]$ by definition. By (1) $[\![E]\!] \vdash \diamond$, so $[\![E]\!] \vdash x : \mathcal{T}[\![T]\!]$. Since $\mathcal{V}[\![x]\!] = x$, $[\![E]\!] \vdash \mathcal{V}[\![x]\!] : \mathcal{T}[\![T]\!]$.

(EXP ASSUME) derives $E \vdash \mathbf{assume}\ (s)C : \{(s)\mathsf{True}\}\ \_{:}\mathsf{unit}\ \{(s_1)((s = s_1) \wedge C)\}$ from $E, s, s_1 \vdash \diamond$ and $\mathrm{fv}(C) \subseteq \mathrm{dom}(E, s)$. Let $M \triangleq \mathcal{E}[\![\mathbf{assume}\ (s)C]\!] = \mathbf{fun}\, s \to \mathbf{let}\ \_ = \mathbf{assume}\ [\![C]\!]\ \mathbf{in}\ ((), s)$ (by definition) and

$$\mathcal{T}[\![\{(s)\mathsf{True}\}\ \_{:}\mathsf{unit}\ \{(s_1)((s = s_1) \wedge C)\}]\!] = \Pi s : \{s : \mathsf{state} \mid \mathsf{True}\}.\ \Sigma\_ : \mathsf{unit}.\ \{s_1 : \mathsf{state} \mid s = s_1 \wedge [\![C]\!]\}$$

which we can write as $\Pi s : T.\ \Sigma\_ : \mathsf{unit}.\ U$. Then $[\![E]\!] \vdash M : \Pi s : T.\ \Sigma\_ : \mathsf{unit}.\ U$ can be derived by the following tree: (R VAL FUN) ((R EXP LET) $[T = \{[\![C]\!]\}]$ ((R EXP ASSUME), (R VAL PAIR) ((R VAL UNIT), (R VAL REFINE) ((R VAL VAR), (R DERIVE))))).

(EXP ASSERT) derives $E \vdash \mathbf{assert}\ (s)C : \{(s)C\}\ \_{:}\mathsf{unit}\ \{(s_1)s = s_1\}$ from $E, s, s_1 \vdash \diamond$ and $\mathrm{fv}(C) \subseteq \mathrm{dom}(E, s)$. Let $M \triangleq \mathcal{E}[\![\mathbf{assert}\ (s)C]\!]$ so that $M = \mathbf{fun}\, s \to \mathbf{let}\ \_ = \mathbf{assert}\ [\![C]\!]\ \mathbf{in}\ ((), s)$ (by definition) and

$$\mathcal{T}[\![\{(s)C\}\ \_{:}\mathsf{unit}\ \{(s_1)s = s_1\}]\!] = \Pi s : \{s : \mathsf{state} \mid [\![C]\!]\}.\ \Sigma\_ : \mathsf{unit}.\ \{s_1 : \mathsf{state} \mid s = s_1\}$$

which we can write as $\Pi s : T.\ \Sigma\_ : \mathsf{unit}.\ U$. Then $[\![E]\!] \vdash M : \Pi s : T.\ \Sigma\_ : \mathsf{unit}.\ U$ can be derived by the following tree: (R VAL FUN) ((R EXP LET) $[T = \mathsf{unit}]$ ((R EXP ASSERT) ((R DERIVE)), (R VAL PAIR) ((R VAL UNIT), (R VAL REFINE) ((R VAL VAR), (R DERIVE))))).

(VAL UNIT) derives $E \vdash () : \mathsf{unit}$ from $E \vdash \diamond$. By 1 $[\![E]\!] \vdash \diamond$, so $[\![E]\!] \vdash () : \mathsf{unit}$. By definition $\mathcal{V}[\![()]\!] = ()$ and $\mathcal{T}[\![\mathsf{unit}]\!] = \mathsf{unit}$, so $[\![E]\!] \vdash \mathcal{V}[\![()]\!] : \mathcal{T}[\![\mathsf{unit}]\!]$.

Induction cases:

(EXP SUBSUM) derives $E \vdash A : F'$ from $E \vdash A : F$ and $E \vdash F <: F'$, where $F = \{(s_0)C_0\}\, x{:}T\, \{(s_1)C_1\}$, and the same for F'.
By induction $[\![E]\!] \vdash \mathcal{E}[\![A]\!] : \mathcal{T}[\![F]\!]$ using (R VAL FUN) as the top-level rule. Then $[\![E]\!], s_0 : T \vdash \mathcal{E}_c[\![A]\!]s_0 : U$.
The only rule that can derive $E \vdash F <: F'$ is (SUB COMP). By its preconditions $\mathrm{fv}(C_0, C_0') \subseteq \mathrm{dom}(E, s_0)$, $\mathrm{fv}(C_1, C_1') \subseteq \mathrm{dom}(E, s_0, x, s_1)$, $\mathbf{vld}(C_0' \Rightarrow C_0)$, $\mathbf{vld}((C_0' \wedge C_1) \Rightarrow C_1')$ and $E, s_0 \vdash T <: T'$.
By induction $[\![E]\!], s_0 : \mathsf{state} \vdash \mathcal{T}[\![T]\!] <: \mathcal{T}[\![T']\!]$ (*). By Lemma 4 and Lemma 5 $\mathrm{fv}([\![C_0]\!], [\![C_0']\!]) \subseteq \mathrm{dom}([\![E, s_0]\!])$ and $\mathrm{fv}([\![C_1]\!], [\![C_1']\!]) \subseteq \mathrm{dom}([\![E, s_0, x, s_1]\!])$. By Corollary 1 $\mathbf{vld}([\![C_0']\!] \Rightarrow [\![C_0]\!])$ (**) and $\mathbf{vld}(([\![C_0']\!] \wedge [\![C_1]\!]) \Rightarrow [\![C_1']\!])$ (***).
We derive $[\![E]\!] \vdash \mathcal{E}[\![A]\!] : \mathcal{T}[\![F']\!]$ by (R VAL FUN) ((R EXP SUBSUM) ((R BOUND WEAKENING) ((R SUB REFINE) ((R SUB REFL), (R DERIVE) (**)), (R SUB PAIR) ((R BOUND WEAKENING) ((R SUB REFINE LEFT REFL) (*), (R SUB REFINE) ((R SUB REFL), (R DERIVE) (***))))).

(EXP RETURN) derives $E \vdash M : \{(s_0)\mathsf{True}\}\, \_{:}T\, \{(s_1)s_0 = s_1\}$ from $E, s_0 \vdash M : T$. By induction $[\![E, s_0]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$. By definition $[\![E, s_0]\!] = [\![E]\!], s_0{:}\mathsf{state}$ and $\mathcal{E}[\![M]\!] = \mathbf{fun}\, s_0 \to (\mathcal{V}[\![M]\!], s_0)$ (for some $(s_0 \notin \mathrm{fv}(M))$). Moreover, let

$$U \triangleq \mathcal{T}[\![\{(s_0)\mathsf{True}\}\, \_{:}T\, \{(s_1)s_0 = s_1\}]\!] = \\ \Pi s_0 : \{s_0 : \mathsf{state} \mid \mathsf{True}\}.\ \Sigma\_ : \mathcal{T}[\![T]\!].\ \{s_1 : \mathsf{state} \mid s_0 = s_1\}$$

We derive $[\![E]\!] \vdash \mathcal{E}[\![M]\!] : U$ by (R VAL FUN) ((R VAL PAIR) (Induction Hypothesis, (R VAL REFINE) ((R VAL VAR), (R DERIVE)))).

(EXP EQ) derives $E \vdash M = N : \{(s_0)\mathsf{True}\}\, x{:}\mathsf{bool}\, \{(s_1)s_1 = s_0 \wedge C\}$ where $C = (x = \mathbf{true} \Leftrightarrow M = N)$ from $E \vdash M : T$, $E \vdash N : U$, $x \notin \mathrm{fv}(M, N)$ and $E, s_0, s_1 \vdash \diamond$.
By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$ and $[\![E]\!] \vdash \mathcal{V}[\![N]\!] : \mathcal{T}[\![U]\!]$. By Lemma 4 $x \notin \mathrm{fv}(\mathcal{V}[\![M]\!], \mathcal{V}[\![N]\!])$. Thus $[\![E]\!] \vdash M' : T'$ (*), where $M' \triangleq (\mathcal{V}[\![M]\!] = \mathcal{V}[\![N]\!])$ and $T' \triangleq \{x{:}\mathsf{bool} \mid [\![C]\!]\}$.
By definition, $\mathcal{E}[\![M = N]\!] = \mathbf{fun}\, s \to \mathbf{let}\, b = (\mathcal{V}[\![M]\!] = \mathcal{V}[\![N]\!])\ \mathbf{in}\ (b, s) = \mathbf{fun}\, s \to N'$, where $N' \triangleq \mathbf{let}\, b = M'\ \mathbf{in}\ (b, s)$, and

$$\mathcal{T}[\![\{(s)\mathsf{True}\}\, x{:}\mathsf{bool}\, \{(s_1)s = s_1 \wedge C\}]\!] = \\ \Pi s : \{s : \mathsf{state} \mid \mathsf{True}\}.\ \Sigma\_ : \mathsf{unit}.\ \{s_1 : \mathsf{state} \mid s = s_1 \wedge [\![C]\!]\}$$

which we can write as $\Pi s : U'.\ \Sigma x : \mathsf{bool}.\ U''$. We derive $[\![E]\!] \vdash \mathbf{fun}\, s \to N' : \Pi s : T.\ \Sigma x : \mathsf{bool}.\ U$ by (R VAL FUN) ((R EXP LET)$[T']$ (*, (R VAL PAIR) ((R VAL VAR), (R VAL REFINE) ((R VAL VAR), (R DERIVE))))).

(STATEFUL EXP LET) derives the judgment

$$E \vdash \mathbf{let}\, x_1 = A\ \mathbf{in}\ B : \{(s_0)C_0\}\, x_2{:}T_2\, \{(s_2)C_2\}$$

from $E \vdash A : \{(s_0)C_0\}\, x_1{:}T_1\, \{(s_1)C_1\}$ and

$$E, s_0, x_1 : T_1 \vdash B : \{(s_1)C_1\}\, x_2{:}T_2\, \{(s_2)C_2\}$$

where $\{s_1, x_1\} \cap \mathrm{fv}(T_2, C_2) = \varnothing$. By induction, we have

$$[\![E]\!] \vdash \mathcal{E}[\![A]\!] : \mathcal{T}[\![\{(s_0)C_0\}\, x_1{:}T_1\, \{(s_1)C_1\}]\!]$$

and $[\![E, s_0, x_1 : T_1]\!] \vdash \mathcal{E}[\![B]\!] : \mathcal{T}[\![\{(s_1)C_1\}\, x_2{:}T_2\, \{(s_2)C_2\}]\!]$ using (R VAL FUN) as the top-level rule.
By Lemma 4 $\{s_1, x_1\} \cap \mathrm{fv}(\mathcal{T}[\![T_2]\!], [\![C_2]\!]) = \varnothing$. By definition
$[\![E, s_0, x_1 : T_1]\!] = [\![E]\!]$, $s_0 : \mathsf{state}$, $x_1 : \mathcal{T}[\![T_1]\!]$,
$\mathcal{E}[\![\mathbf{let}\ x_1 = A\ \mathbf{in}\ B]\!] = \mathbf{fun}\, s_0 \to \mathbf{let}\ (x_1, s_1) = \mathcal{E}_c[\![A]\!] s_0\ \mathbf{in}\ \mathcal{E}_c[\![B]\!] s_1$ and

$$\mathcal{T}[\![\{(s_i)C_i\}\, x_j{:}T_j\, \{(s_j)C_j\}]\!] = \\ \Pi s_i : \{s_i : \mathsf{state} \mid [\![C_i]\!]\}.\ \Sigma x_j : \mathcal{T}[\![T_j]\!].\ \{s_j : \mathsf{state} \mid [\![C_j]\!]\}$$

which we can write as $\Pi s_i : U_i.\ U_{ij}$.
By (R VAL FUN) we get $[\![E]\!], s_0 : U_0 \vdash \mathcal{E}_c[\![A]\!] s_0 : U_{01}$ (*) and
$[\![E, s_0, x_1 : T_1]\!], s_1 : U_1 \vdash \mathcal{E}_c[\![B]\!] s_1 : U_{12}$ **. Then $[\![E]\!] \vdash \mathcal{E}[\![\mathbf{let}\ x_1 = A\ \mathbf{in}\ B]\!] : \Pi s_0 : U_0.\ U_{01}$ by (R VAL FUN) ((R EXP LET)$[U_{01}]$ (*, (R EXP SPLIT) ((R VAL VAR),**))).

(VAL PAIR) derives $E \vdash (M, N) : \Sigma x : T.\ U$ from $E \vdash M : T$ and $E \vdash N : U\{^M/_x\}$. By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$ and $[\![E]\!] \vdash \mathcal{V}[\![N]\!] : \mathcal{T}[\![U\{^M/_x\}]\!]$. By Lemma 6 $\mathcal{T}[\![U\{^M/_x\}]\!] = \mathcal{T}[\![U]\!]\{^{\mathcal{V}[\![M]\!]}/_x\}$, so $[\![E]\!] \vdash (\mathcal{V}[\![M]\!], \mathcal{V}[\![N]\!]) : \Sigma x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![U]\!]$. By definition $\mathcal{V}[\![(M, N)]\!] = (\mathcal{V}[\![M]\!], \mathcal{V}[\![N]\!])$ and $\Sigma x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![U]\!] = \mathcal{T}[\![\Sigma x : T.\ U]\!]$, so $[\![E]\!] \vdash \mathcal{V}[\![(M, N)]\!] : \mathcal{T}[\![\Sigma x : T.\ U]\!]$.

(STATEFUL EXP SPLIT) derives $E \vdash \mathbf{let}\ (x, y) = M\ \mathbf{in}\ A : F$ from $E \vdash M : (\Sigma x : T.\ U)$, $E, x : T, y : U \vdash A : ((x, y) = M) \rightsquigarrow F$ and $\{x, y\} \cap \mathrm{fv}(F) = \varnothing$.
Assume that $F = \{(s)C_1\}\, z{:}T'\, \{(s_2)C_2\}$; then $((x, y) = M) \rightsquigarrow F = \{(s)((x, y) = M) \wedge C_1\}\, z{:}T'\, \{(s_2)C_2\}$.
By Lemma 4 $\{x, y\} \cap \mathrm{fn\,fv}(\mathcal{T}[\![F]\!]) = \varnothing$. By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : (\Sigma x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![U]\!])$ (*) and $[\![E]\!], x : \mathcal{T}[\![T]\!], y : \mathcal{T}[\![U]\!] \vdash \mathcal{E}[\![A]\!] : U'$ where

$$U' \triangleq \Pi s : \{s : \mathsf{state} \mid ((x, y) = \mathcal{V}[\![M]\!]) \wedge [\![C_1]\!]\}.\ \Sigma z : \mathcal{T}[\![T']\!].\ \{s_2 : \mathsf{state} \mid [\![C_2]\!]\}$$

(which we can write as $\Pi s : U_1'.\ U_2'$) using (R VAL FUN) as the top-level rule.
Then $[\![E]\!], x : \mathcal{T}[\![T]\!], y : \mathcal{T}[\![U]\!], s : U_1' \vdash \mathcal{E}[\![A]\!] s : U_2'$(**).
Let $N \triangleq \mathcal{E}[\![\mathbf{let}\ (x, y) = M\ \mathbf{in}\ A]\!] = \mathbf{fun}\, s \to \mathbf{let}\ (x, y) = \mathcal{V}[\![M]\!]\ \mathbf{in}\ \mathcal{E}_c[\![A]\!] s$ (by definition). We derive $[\![E]\!] \vdash N : \mathcal{T}[\![F]\!]$ by (R VAL FUN) ((R EXP SPLIT) (*, **)).

(STATEFUL VAL FUN) derives $E \vdash \mathbf{fun}\, x \to A : \Pi x : T.\ F$ from $E, x : T \vdash A : F$. By induction $[\![E, x : T]\!] \vdash \mathcal{E}[\![A]\!] : \mathcal{T}[\![F]\!]$. By definition $[\![E, x : T]\!] = [\![E]\!], x : \mathcal{T}[\![T]\!]$, so $[\![E]\!] \vdash \mathbf{fun}\, x \to \mathcal{E}[\![A]\!] : \Pi x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![F]\!]$. By definition $\mathcal{V}[\![\mathbf{fun}\, x \to$

$A\rrbracket = \mathbf{fun}\, x \to \mathcal{E}[\![A]\!]$ and $\mathcal{T}[\![\Pi x : T.\ F]\!] = \Pi x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![F]\!]$, so $[\![E]\!] \vdash \mathcal{V}[\![\mathbf{fun}\, x \to A]\!] : \mathcal{T}[\![\Pi x : T.\ F]\!]$.

(STATEFUL EXP APPL) derives $E \vdash M\ N : F\{^N/_x\}$ from $E \vdash M : (\Pi x : T.\ F)$ and $E \vdash N : T$. By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : (\Pi x : \mathcal{T}[\![T]\!].\ \mathcal{T}[\![F]\!])$ (*) and $[\![E]\!] \vdash \mathcal{V}[\![N]\!] : \mathcal{T}[\![T]\!]$(**).
By definition $\mathcal{E}[\![M\ N]\!] = \mathbf{fun}\, s \to \mathbf{let}\ f = \mathcal{V}[\![M]\!]\ \mathcal{V}[\![N]\!]\ \mathbf{in}\ f\ s$. Then, $[\![E]\!] \vdash \mathcal{E}[\![M\ N]\!] : \mathcal{T}[\![F]\!]\{^{\mathcal{V}[\![N]\!]}/_x\}$ by (R VAL FUN) ((R EXP LET) $[\mathcal{T}[\![F]\!]\{^{\mathcal{V}[\![N]\!]}/_x\}]$ ((R EXP APPL) (*, **), (R EXP APPL) ((R VAL VAR), (R VAL VAR)))).
Finally, by Lemma 6 $\mathcal{T}[\![F\{^N/_x\}]\!] = \mathcal{T}[\![F]\!]\{^{\mathcal{V}[\![N]\!]}/_x\}$.

(VAL INL INR FOLD) derives $E \vdash h\ M : U$ from $h : (T, U)$, $E \vdash U$ and $E \vdash M : T$. By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$. Using Lemma 6 $h : (\mathcal{T}[\![T]\!], \mathcal{T}[\![U]\!])$. By (2) $[\![E]\!] \vdash \mathcal{T}[\![T]\!]$, so $[\![E]\!] \vdash h\ \mathcal{V}[\![M]\!] : \mathcal{T}[\![U]\!]$. By definition $h\ \mathcal{V}[\![M]\!] = \mathcal{V}[\![h\ M]\!]$, so $[\![E]\!] \vdash \mathcal{V}[\![h\ M]\!] : \mathcal{T}[\![U]\!]$.

(STATEFUL EXP MATCH INL INR FOLD) derives $E \vdash \mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B : F$ from $E \vdash M : T, h : (U, T), x \notin \mathrm{fv}(F), E, x : U \vdash A : (h\, x = M) \rightsquigarrow F$ and $E \vdash B : (\forall x.h\ x \neq M) \rightsquigarrow F$.
Assume that $F = \{(s)C_1\}\, z{:}T'\, \{(s_2)C_2\}$; then $C \rightsquigarrow F = \{(s)C \wedge C_1\}\, z{:}T'\, \{(s_2)C_2\}$.
By Lemma 4 $x \notin \mathrm{fn}\,\mathrm{fv}(\mathcal{T}[\![F]\!])$. Using Lemma 6 $h : (\mathcal{T}[\![T]\!], \mathcal{T}[\![U]\!])$.
By induction $[\![E]\!] \vdash \mathcal{V}[\![M]\!] : \mathcal{T}[\![T]\!]$ (*), $[\![E]\!], x : \mathcal{T}[\![U]\!] \vdash \mathcal{E}[\![A]\!] : U^{(h\ x = \mathcal{V}[\![M]\!])}$ and $[\![E]\!] \vdash \mathcal{E}[\![B]\!] : U^{(\forall x.h\ x \neq \mathcal{V}[\![M]\!])}$ where

$$U^C \triangleq \Pi s : \{s : \mathsf{state} \mid [\![C \wedge C_1]\!]\}.\ \ \Sigma z : \mathcal{T}[\![T']\!].\ \ \{s_2 : \mathsf{state} \mid [\![C_2]\!]\}$$

(which we can write as $\Pi s : U_1^C.\ U_2$) using (R VAL FUN) as the top-level rule.
Then $[\![E]\!], x : \mathcal{T}[\![U]\!], s : U_1^C \vdash \mathcal{E}[\![A]\!]s : U_2$(**) and $[\![E]\!], s : U_1^C \vdash \mathcal{E}[\![B]\!]s : U_2$(***)
Let $N \triangleq \mathcal{E}[\![\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B]\!] = \mathbf{fun}\, s \to \mathbf{match}\ \mathcal{V}[\![M]\!]\ \mathbf{with}\ h\ x \to \mathcal{E}_c[\![A]\!]s\ \mathbf{else}\ \mathcal{E}_c[\![B]\!]s$ (by definition). We derive $[\![E]\!] \vdash N : \mathcal{T}[\![F]\!]$ by (R VAL FUN) ((R EXP MATCH INL INR FOLD) (*, (R EXCHANGE)$[E' = \varnothing]$ ((R BOUND WEAKENING) $[\{s : \mathsf{state} \mid [\![C_1]\!]\} <: \{s : \mathsf{state} \mid (\forall x.h\ x \neq \mathcal{V}[\![M]\!]) \wedge [\![C_1]\!]\}]$ ((R SUB REFINE) ((R SUB REFL), (R DERIVE)), (R WEAKENING) $[\mu = \_ : \{(\forall x.h\, x \neq \mathcal{V}[\![M]\!])\}]$ (**))), (R EXCHANGE)$[E' = \_ : \{h\, x = \mathcal{V}[\![M]\!]\}]$ ((R EXCHANGE)$[E' = \varnothing]$ ((R BOUND WEAKENING) $[\{s : \mathsf{state} \mid [\![C_1]\!]\} <: \{s : \mathsf{state} \mid (h\ x = \mathcal{V}[\![M]\!]) \wedge [\![C_1]\!]\}]$ ((R SUB REFINE) ((R SUB REFL), (R DERIVE)), (R WEAKENING) $[\mu = \_ : \{(h\ x = \mathcal{V}[\![M]\!])\}]$ (***))))))). □

1.8.1 Safety

We define $[\![\mathcal{R}]\!]$ by $[\![\mathbf{let}\ x = \mathcal{R}\ \mathbf{in}\ B]\!] \triangleq \mathbf{let}\ y = [\![\mathcal{R}]\!]\ \mathbf{in\ let}\ (x, s') = y\ \mathbf{in}\ \mathcal{E}_c[\![B]\!]s'$ 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 7. *If* $[\![\mathbf{assume}\ S]\!] \Rsh \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_s\}$ *is statically safe, then* (A, M, S) *has not failed.*

Proof. The configuration (A,M,S) has failed iff $A = \mathcal{R}[\textbf{assert}\ (s)C]$ and we cannot derive $S \vdash C\{^M/_s\}$. If $A = \mathcal{R}[\textbf{assert}\ (s)C]$ then $B \triangleq \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_s\} = [\![\mathcal{R}]\!][\textbf{let}\ \_ = \textbf{assert}\ C\{^{\mathcal{V}[\![M]\!]}/_s\}\ \textbf{in}\ ((),s)]$. If $[\![\textbf{assume}\ S]\!] \Rsh B$ is statically safe, then $[\![S]\!] \vdash C\{^{\mathcal{V}[\![M]\!]}/_s\}$. Thus $[\![S]\!]$ and $C\{^{\mathcal{V}[\![M]\!]}/_s\}$ are first-order, so S and $C\{^M/_s\}$ are first-order. By Corollary 1, $[\![S]\!] \vdash C\{^{\mathcal{V}[\![M]\!]}/_s\}$. □

Lemma 8. *If* $(A_0,N_0,S_0) \to (A_1,N_1,S_1)$ *then* $[\![\textbf{assume}\ S_0]\!] \Rsh \mathcal{E}_c[\![A_0]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_s\} \to^* [\![\textbf{assume}\ S_1]\!] \Rsh \mathcal{E}_c[\![A_1]\!]s\{^{\mathcal{V}[\![N_1]\!]}/_s\}$

Proof. The proof is by induction on the derivation of the transition. Let $M_0^r \triangleq \mathcal{E}_c[\![A_0]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_s\}$ and $M_1^r \triangleq \mathcal{E}_c[\![A_1]\!]s\{^{\mathcal{V}[\![N_1]\!]}/_s\}$. When $S_0 = S_1$ and $M_0^r \to^* M_1^r$ then the statement of the lemma follows by the use of (R RED FORK 2) at every reduction. We group the transitions into three groups.

For the reductions of the core calculus, we have $S_0 = S_1$, so we need only show that $M_0^r \to^* M_1^r$. We also have $N_0 = N_1$, and we let $N \triangleq \mathcal{V}[\![N_0]\!] (= \mathcal{V}[\![N_1]\!])$.

(RED LET) Here $A_0 = \textbf{let}\ x = M\ \textbf{in}\ A$ and $A_1 = A\{^M/_x\}$. Then $M_0^r = \textbf{let}\ y = (\mathcal{V}[\![M]\!],N)\ \textbf{in let}\ (x,s) = y\ \textbf{in}\ \mathcal{E}_c[\![A]\!]s$.
By (R RED LET VAL), $M_0^r \to M'$ with $M' \triangleq \textbf{let}\ (x,s) = (\mathcal{V}[\![M]\!],N)\ \textbf{in}\ \mathcal{E}_c[\![A]\!]s$. By (R RED SPLIT), $M' \to M''$ with $M'' \triangleq \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_x\}\{^N/_s\}$. By Lemma 6, $\mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_x\} = \mathcal{E}_c[\![A\{^M/_x\}]\!]s$, so $M'' = M_1^r$.

(RED FUN) Here $A_0 = (\textbf{fun}\,x \to A)\ M$ and $A_1 = A\{^M/_x\}$. Then $M_0^r = \textbf{let}\ f = (\textbf{fun}\,x \to \textbf{fun}\,s \to \mathcal{E}_c[\![A]\!]s)\ \mathcal{V}[\![M]\!]\ \textbf{in}\ f\ N$.
By (R RED LET)((R RED FUN)), we have $M_0^r \to M'$ with $M' \triangleq \textbf{let}\ f = \textbf{fun}\,s \to \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_x\}\ \textbf{in}\ f\ N$. By Lemma 6, $\mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![M]\!]}/_x\} = \mathcal{E}_c[\![A\{^M/_x\}]\!]s$. By (R RED LET VAL), $M' \to M''$ with $M'' \triangleq \textbf{fun}\,s \to \mathcal{E}_c[\![A\{^M/_x\}]\!]s\ N$. By (R RED FUN), $M'' \to \mathcal{E}_c[\![A\{^M/_x\}]\!]s\{^N/_s\} = M_1^r$.

(RED EQ) Here $A_0 = (M_1 = M_2)$, $A_1 = \textbf{true}$, and $M_1 = M_2 = M$ for some M. Then $M_0^r = \textbf{let}\ b = (\mathcal{V}[\![M]\!] = \mathcal{V}[\![M]\!])\ \textbf{in}\ (b,N)$.
By (R RED LET)((R RED EQ)), $M_0^r \to M'$ with $M' \triangleq \textbf{let}\ b = \textbf{true}\ \textbf{in}\ (b,N)$. By (R RED LET VAL), $M' \to (\textbf{true},N) = M_1^r$.

(RED NEQ) Here $A_0 = (M_1 = M_2)$, $A_1 = \textbf{true}$, and $M_1 \neq M_2$. Then $M_0^r = \textbf{let}\ b = (\mathcal{V}[\![M_1]\!] = \mathcal{V}[\![M_2]\!])\ \textbf{in}\ (b,N)$.
By injectivity of $\mathcal{V}[\![]\!]$, $\mathcal{V}[\![M_1]\!] \neq \mathcal{V}[\![M_2]\!]$. Thus, by (R RED LET)((R RED EQ)), $M_0^r \to M'$ with $M' \triangleq \textbf{let}\ b = \textbf{false}\ \textbf{in}\ (b,N)$. By (R RED LET VAL), $M' \to (\textbf{false},N) = M_1^r$.

(RED SPLIT) Here $A_0 = \textbf{let}\ (x,y) = (M_1,M_2)\ \textbf{in}\ A$ and $A_1 = A\{^{M_1}/_x\}\{^{M_2}/_y\}$. Then, $M_0^r = \textbf{let}\ (x,y) = (\mathcal{V}[\![M_1]\!],\mathcal{V}[\![M_2]\!])\ \textbf{in}\ \mathcal{E}_c[\![A]\!]s\{^N/_s\}$.
By (R RED SPLIT) $M_0^r \to M'$ with $M' \triangleq \mathcal{E}_c[\![A]\!]s\{^N/_s\}\{^{\mathcal{V}[\![M_1]\!]}/_x\}\{^{\mathcal{V}[\![M_2]\!]}/_y\}$. Since s is fresh and N is closed, we can commute the substitutions and use Lemma 6 to get $M' = M_1^r$.

(RED MATCH) Here $A_0 = \textbf{match}\ (h\ M)\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B$ and $A_1 = A\{^M/_x\}$. Then, $M_0^r = \textbf{match}\ (h\ \mathcal{V}[\![M]\!])\ \textbf{with}\ h\ x \to \mathcal{E}_c[\![A]\!]s\{^N/_s\}\ \textbf{else}\ \mathcal{E}_c[\![B]\!]s\{^N/_s\}$.
By (R RED MATCH) $M_0^r \to M'$ with $M' \triangleq \mathcal{E}_c[\![A]\!]s\{^N/_s\}\{^{\mathcal{V}[\![M]\!]}/_x\}$. Since s is fresh and N is closed, we can commute the substitutions and use Lemma 6 to get $M' = M_1^r$.

(RED MISMATCH) Here $A_0 = \textbf{match}\ (h'\ M)\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B$, $h \neq h'$ and $A_1 = B$. Then, $M_0^r = \textbf{match}\ (h'\ \mathcal{V}[\![M]\!])\ \textbf{with}\ h\ x \to\ \mathcal{E}_c[\![A]\!]s\{^N/_s\}\ \textbf{else}\ \mathcal{E}_c[\![B]\!]s\{^N/_s\}$. By (R RED MATCH) $M_0^r \to \mathcal{E}_c[\![B]\!]s\{^N/_s\} = M_1^r$.

The second group is the reductions involving the state.

(RED ASSUME) Here $A_0 = \textbf{assume}\ (s)C$, $S_1 = S_0 \cup \{\!|C\{^{N_0}/_s\}|\!\}$, $N_0 = N_1$ and $A_1 = ()$. We let $N \triangleq \mathcal{V}[\![N_0]\!]$. Then $M_0^r = \textbf{let}\ \_ = \textbf{assume}\ [\![C]\!]\{^N/_s\}\ \textbf{in}\ ((),N)$. By Lemma 6, $[\![C]\!]\{^N/_s\} = [\![C\{^{N_0}/_s\}]\!]$.
By (HEAT ASSUME ()), (HEAT FORK LET) and (HEAT TRANS), we have $M_0^r \Rrightarrow \textbf{assume}\ [\![C\{^{N_0}/_s\}]\!] \Cap \textbf{let}\ \_ = ()\ \textbf{in}\ ((),N)$.
By (R RED LET VAL), $\textbf{let}\ \_ = ()\ \textbf{in}\ ((),N) \to ((),N) = M_1^r$. By (R RED FORK 2), $M_0^r \Rrightarrow\!\to M'$ with $M' \triangleq \textbf{assume}\ [\![C\{^{N_0}/_s\}]\!] \Cap M_1^r$.
By repeated use of (HEAT TRANS), (HEAT FORK COMM), (HEAT FORK ASSOC), (HEAT FORK 1) and (HEAT FORK 2), $[\![\textbf{assume}\ S_0]\!] \Cap M' \Rrightarrow [\![\textbf{assume}\ S_1]\!] \Cap M_1^r$.
Finally, $[\![\textbf{assume}\ S_0]\!] \Cap M_0^r \to [\![\textbf{assume}\ S_1]\!] \Cap M_1^r$ by (R RED FORK 2) and (R RED HEAT).

(RED ASSERT) Here $A_0 = \textbf{assert}\ (s)C$, $S_1 = S_0$, $N_0 = N_1$ and $A_1 = ()$. We let $N \triangleq \mathcal{V}[\![N_0]\!]$ and $S \triangleq S_0$. Then $M_0^r = \textbf{let}\ \_ = \textbf{assert}\ [\![C]\!]\{^N/_s\}\ \textbf{in}\ ((),N)$.
By (R RED LET) ((R RED ASSERT)), $M_0^r \to M'$ with $M' \triangleq \textbf{let}\ \_ = ()\ \textbf{in}\ ((),N)$.
By (R RED LET VAL), $M' \to ((),N) = M_1^r$.

(RED GET) Here $A_0 = \textbf{get}()$, $N_0 = N_1$, $S_0 = S_1$ and $A_1 = N_1$, and we have $M_0^r = (\mathcal{V}[\![N_0]\!], \mathcal{V}[\![N_0]\!]) = M_1^r$.

(RED SET) Here $A_0 = \textbf{set}(N_1)$, $S_0 = S_1$ and $A_1 = ()$. Here $M_0^r = ((), \mathcal{V}[\![N_1]\!]) = M_1^r$.

The third group only consists of the contextual rule.

(RED CTX) By rewriting the derivation of the reduction, we need only consider the case of $\mathcal{R} = \textbf{let}\ x = [\,]\ \textbf{in}\ B$. Assume that $(A,N_0,S_0) \longrightarrow (A',N_1,S_1)$. By induction there is k such that $[\![\textbf{assume}\ S_0]\!] \Cap \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_s\} \to^k [\![\textbf{assume}\ S_1]\!] \Cap \mathcal{E}_c[\![A']\!]s\{^{\mathcal{V}[\![N_1]\!]}/_s\}$.
If $k = 0$, the leaf rule of the reduction was (RED GET) or (RED SET), so $S_0 = S_1$ and $M_0^r = M_1^r$, and we are done.
Otherwise,

$$
\begin{aligned}
&[\![\textbf{assume}\ S_0]\!] \Cap M_0^r = \\
&\quad [\![\textbf{assume}\ S_0]\!] \Cap \textbf{let}\ y = \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_s\}\ \textbf{in let}\ (x,s') = y\ \textbf{in}\ \mathcal{E}_c[\![B]\!]s' \Rrightarrow \\
&\quad \textbf{let}\ y = [\![\textbf{assume}\ S_0]\!] \Cap \mathcal{E}_c[\![A]\!]s\{^{\mathcal{V}[\![N_0]\!]}/_s\}\ \textbf{in let}\ (x,s') = y\ \textbf{in}\ \mathcal{E}_c[\![B]\!]s' \to^k \\
&\quad \textbf{let}\ y = [\![\textbf{assume}\ S_1]\!] \Cap \mathcal{E}_c[\![A']\!]s\{^{\mathcal{V}[\![N_1]\!]}/_s\}\ \textbf{in let}\ (x,s') = y\ \textbf{in}\ \mathcal{E}_c[\![B]\!]s' \Rrightarrow \\
&\qquad\qquad [\![\textbf{assume}\ S_1]\!] \Cap M_1^r.
\end{aligned}
$$

Using (R RED HEAT), we then get $[\![\textbf{assume}\ S_0]\!] \Cap M_0^r \to^k [\![\textbf{assume}\ S_1]\!] \Cap M_1^r$. □

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

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

Proof. By Theorem 2 and Corollary 2, it suffices to prove $\varnothing \vdash \mathcal{E}[\![A]\!]\ \mathcal{V}[\![M]\!] : U$ for some U. By Proposition 4, 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 $U \triangleq \Sigma\_ : T.\ \{s' : \mathcal{T}[\![\mathsf{state}]\!] \mid \mathsf{True}\}$. □

1.8.2 Generalized Instantiation

The translation from RIF to RCF does not depend on the particular state type or API functions. In order to execute a RIF program, we need to instantiate both. In the paper, we have only used two state-manipulating functions: **get**() and **set**$(\cdot)$, where the state type is a type in RIF. We generalize this notion as follows: An instance is given by

- An RCF type T, writing $\mathsf{state} \triangleq T$; and
- API function names (i.e., variables) $g_1, \ldots, g_n$ with RIF types $\Pi x_i : T_i.\ F_i$; and
- an RCF implementation N such that
  $\varnothing \vdash N : \mathcal{T}[\![\Pi x_1 : T_1.\ F_1]\!] * \ldots * \mathcal{T}[\![\Pi x_n : T_n.\ F_n]\!]$; and
- a RIF (initial) environment
  $E^0 \triangleq g_1 : \Pi x_i : T_i.\ F_i, \ldots, g_n : \Pi x_n : T_n.\ F_n$; and
- predicate symbols $p_1, \ldots, p_m$.

As an example, given state type $\mathcal{T}[\![T]\!]$ we can implement **get**() $\triangleq$ **fun** s $\rightarrow$(s,s) and **set** $\triangleq$ **fun** m $\rightarrow$**fun** s $\rightarrow$((),m) at the appropriate types.

Given a generalized instance of RIF as above, we define the configuration (A, M, S) to be *safe* if and only if the RCF expression **assume** S;**let** $(g_1, \ldots, g_n) = N$ **in** $\mathcal{E}[\![A]\!]\ M$ is safe.

Theorem 4 (Safety for RIF).
If $E^0 \vdash A : \{(s)C\}\_ : T\ \{(s')\mathsf{True}\}$ *(in RIF) and*
$\varnothing \vdash$ **let** $(g_1, \ldots, g_n) = N$ **in** $M : \{s : \mathsf{state} \mid [\![C]\!]\}$ *(in RCF) then*
the configuration (A, M, True) *is safe.*

Proof. By Theorem 2, it suffices to prove $\varnothing \vdash$ **assume** True;**let** $(g_1, \ldots, g_n) = N$ **in** $\mathcal{E}[\![A]\!]\ M : U$ for some U.

By assumption $E^0 \vdash A : \{(s)C\}\_ : T\ \{(s')\mathsf{True}\}$. By Proposition 4(8) $[\![E^0]\!] \vdash \mathcal{E}[\![A]\!] : \Pi s : \{s : \mathsf{state} \mid [\![C]\!]\}.\ \Sigma\_ : T.\ \{s' : \mathsf{state} \mid \mathsf{True}\}$ (*).

By assumption $\varnothing \vdash M : \{s : \mathsf{state} \mid [\![C]\!]\}$ (**), so by (R EXP APPL) (*, (R WEAKENING) (**)) $[\![E^0]\!] \vdash \mathcal{E}[\![A]\!]\ M : U$ with $U \triangleq \Sigma\_ : T.\ \{s' : \mathsf{state} \mid \mathsf{True}\}$.

By definition $[\![E^0]\!] = x_1 : \mathcal{T}[\![\Pi x_1 : T_1.\ F_1]\!], \ldots, x_n : \mathcal{T}[\![\Pi x_n : T_n.\ F_n]\!]$. By assumption $\varnothing \vdash N : \mathcal{T}[\![\Pi x_1 : T_1.\ F_1]\!] * \mathcal{T}[\![\Pi x_n : T_n.\ F_n]\!]$. This yields $\varnothing \vdash$ **let** $(g_1, \ldots, g_n) = N$ **in** $\mathcal{E}[\![A]\!]\ M : U$. By (R EXP LET), (R EXP ASSUME) and (R WEAKENING), we get $\varnothing \vdash$ **assume** True;**let** $(g_1, \ldots, g_n) = N$ **in** $\mathcal{E}[\![A]\!]\ M : U$. □

References

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

M. Abadi and C. Fournet. Access control based on execution history. In *Network and Distributed System Security Symposium (NDSS'03)*, pages 107–121. The Internet Society, 2003.

M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1–2):273–309, 2001.

R. Atkey. Parameterized notions of computation. *Journal of Functional Programming*, 19:355–376, 2009.

A. Banerjee and D. Naumann. History-based access control and secure information flow. In *Construction And Analysis of Safe, Secure, And Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *LNCS*, pages 27–48. Springer, 2005a.

A. Banerjee and D. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005b.

M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. In *European Symposium On Research In Computer Security (ESORICS'07)*, volume 4734 of *LNCS*, pages 203–218. Springer, 2007.

M. Y. Becker and P. Sewell. Cassandra: flexible trust management, applied to electronic health records. In *17th IEEE Computer Security Foundations Workshop (CSFW'04)*, pages 139–154, June 2004.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. Technical Report MSR–TR–2008–118, Microsoft Research, 2008. A preliminary, abridged version appears in the proceedings of CSF'08.

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

L. Cardelli. Typechecking dependent types and subtypes. In *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 45–57. Springer, 1986.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, et al. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

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

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

B. Dutertre and L. de Moura. The YICES SMT solver. Available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

D. F. Ferraiolo and D. R. Kuhn. Role based access control. In *Proc. National Computer Security Conference*, pages 554–563, 1992.

J. Filliâtre and C. Marché. Multi-prover Verification of C Programs. In *International Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.

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

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

C. Flanagan and M. Abadi. Types for safe locking. In *European Symposium on Programming (ESOP'99)*, volume 1576 of *LNCS*, pages 91–108. Springer, 1999.

C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, 2003.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 141–156. Springer, 2005.

C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies in distributed systems. In *20th IEEE Computer Security Foundation Symposium (CSF'07)*, pages 31–45, 2007.

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

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

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

A. D. Gordon and C. Fournet. Principles and applications of refinement types. Technical Report MSR–TR–2009–147, Microsoft Research, 2009.

A. D. Gordon and A. S. A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 11(4):451–521, 2003.

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

C. Gunter. *Semantics of programming languages*. MIT Press, 1992.

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

L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: Preliminary technical results. Technical Report MS-CIS-08-10, University of Pennsylvania, 2008.

K. W. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP*, volume 4421 of *LNCS*, pages 505–519. Springer, 2007.

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

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

E. Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

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

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

B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's type theory*. Clarendon Press Oxford, 1990.

B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

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

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

G. D. Plotkin. Denotational semantics with partial functions. Unpublished lecture notes, CSLI, Stanford University, July 1985.

F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, 2005.

S. Ranise and C. Tinelli. *The SMT-LIB Standard: Version 1.2*, 2006.

Y. Régis-Gianas and F. Pottier. A Hoare logic for call-by-value functional programs. In *Mathematics of Program Construction (MPC'08)*, volume 5133 of *LNCS*, pages 305–335. Springer, 2008.

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

J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

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

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

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12: 157–171, 1986.

P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.

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