2011 IEEE Symposium on Security and Privacy

Verification of Information Flow and Access Control Policies with Dependent Types

Aleksandar Nanevski Anindya Banerjee
IMDEA Software Institute
Madrid, Spain
{aleks.nanevski, anindya.banerjee}@imdea.org

Deepak Garg
Carnegie Mellon University
Pittsburgh, PA, USA
dg@cs.cmu.edu

***Abstract*—We present Relational Hoare Type Theory (RHTT), a novel language and verification system capable of expressing and verifying rich information flow and access control policies via dependent types. We show that a number of security policies which have been formalized separately in the literature can all be expressed in RHTT using only standard type-theoretic constructions such as monads, higher-order functions, abstract types, abstract predicates, and modules. Example security policies include conditional declassification, information erasure, and state-dependent information flow and access control. RHTT can reason about such policies in the presence of dynamic memory allocation, deallocation, pointer aliasing and arithmetic. The system, theorems and examples have all been formalized in Coq.**

***Keywords*-Information Flow, Access Control, Type Theory**

I. Introduction

Several challenges persist in existing work on specification and enforcement of confidentiality policies. First, many practical applications require a combination of a number of different classes of policies: authentication, authorization, conditional declassification, erasure, etc. Yet, most existing systems are tailored for enforcing specific classes of policies in isolation. Second, where policy combinations have been considered (e.g. [4, 7, 12]), policy conformance is typically formalized for simple languages without important programming features such as dynamic allocation, mutable state and pointer aliasing, or without modern modularity mechanisms that aid programming in the large. There has been little work on confidentiality policies pertaining to linked data structures (lists, trees, graphs, etc.), and even less work exists for structures that are heterogeneous; that is, structures that contain mixed secret and public data as well as mixed secret and public links. Third, despite their efficiency, enforcement mechanisms are often imprecise in their handling of implicit information flow (that arises due to program control structures such as conditionals or procedure calls) and reject perfectly secure programs.

In this paper we revisit the foundations of information flow — its specification as well as its static enforcement — and address the above challenges of policy specificity, language expressiveness and precision, simultaneously. The key insight of our work is that all the three problems can be addressed using *standard* linguistic features from dependent type theory [24]: (a) higher-order functions, abstract data types and modules, that provide for software engineering concepts such as abstraction and information hiding, and (b) a logic for higher-order assertions, including quantification over predicates, that serves as the foundation for a rich policy specification language. We additionally consider an extension of dependent types with (c) general recursion, mutable state, dynamic allocation, and pointer aliasing. We use the dependent types as a policy specification language, and typechecking (i.e., program verification) to enforce conformance of programs to policy. As is standard in type theory, we assume that programs are typechecked before they are executed.

As our first contribution, we show that a number of security policies which have been previously considered in isolation, such as declassification [14, 39], information erasure [15, 16], state-dependent access control [11, 12] and state-dependent information flow policies [7], can be combined in the same system using the mentioned type-theoretic abstractions. We explain this point further below, and illustrate it through several verified examples in the paper.

As our second contribution, we show that these policies can be enforced in the presence of dynamic allocation, deallocation, and pointer aliasing, and in particular, over programs involving linked, heterogeneous data structures. To achieve this, we employ a *semantic* definition of what constitutes confidential (high) vs. public (low) data, in contrast to most related work where variables are *syntactically* labeled with a desired security level [28, 45]. The semantic characterization allows the same variable or pointer to contain data of different security levels at different points in program execution, which gives us the needed flexibility of enforcement. The semantic characterization also facilitates precise specification of programs with implicit information flow such as procedure calls or (possibly nested) conditionals.

Our third and technically central contribution is a novel verification system, Relational Hoare Type Theory (RHTT), that integrates a programming language and a logic into a common substrate underlying all of (a)–(c) above. In more detail, RHTT provides (a) and (b) by including the type theory of the Calculus of Inductive Constructions (CiC) [25, Chapter 4], as implemented in the Coq proof assistant. To provide for (c), RHTT introduces a new type constructor STsec, which classifies side-effectful computations similar to Haskell monads [34], except that the STsec monad is indexed with a precondition and a postcondition, as in a Hoare triple. STsec types separate the imperative from the purely functional fragment of the type theory, ensuring soundness of

1081-6011/11 $26.00 © 2011 IEEE
DOI 10.1109/SP.2011.12

their combination.

RHTT's preconditions specify constraints on the environment under which it is safe to run a program, and can be used to enforce authentication and authorization policies, even when they depend on state. RHTT's postconditions are *relational* assertions; they specify the behavior of *two runs* of a program [1]. The relational formulation directly captures in the types the notion of *noninterference* [18], a prominent semantic characterization of confidentiality. Together with higher-order type theory, this provides an architecture for uniform treatment of all the policies mentioned above.

For example, we show that the fundamental linguistic abstractions required to specify and implement declassification are STsec types, modules and abstract predicates. A module can be used to delimit the scope in which data is considered public, by hiding the publicity of the data from module clients via existential type abstraction [26]. Then declassification amounts to breaking the abstraction barrier by an exported interface method that reveals this in-module publicity. This is orthogonal to revealing the data itself. The latter can always be done even without declassification, but the clients will have to use such data as if it were confidential. Declassification may be unconditional or conditional [7], where the condition might be stateful and involve, e.g., authentication.

In information erasure policies [15, 16] confidential data may be released within a delimited scope, provided there is a guarantee that such data will be erased upon exit from the scope. We show that such policies can be specified using a combination of higher-order functions with local state, modules and abstract predicates. The key facilitating component here is that STsec types may appear in argument positions in function types, which is similar to having Hoare logic where one can reason about Hoare triples hypothetically wrt. the truth of other Hoare triples. A similar combination of features can be used to grant a method access to data only if the method provably conforms with some desired confidentiality policy.

Finally, state-dependent information flow and access control policies require abstract predicates combined with mutable state. This allows expressing security policies that can change with time due to state updates [43].

Our development of RHTT overcomes a number of technical challenges. First, for relational reasoning to be applicable at all, the type system must give special status to instantiations of a program e with high values. The special status is needed so that the *same* postcondition of e can relate e's *different* instantiations. Our solution is to introduce new typing and programming primitives for abstraction and instantiation wrt. a number of variables, simultaneously (Section II). This illustrates why our type system had to be developed hand-in-hand with the associated relational verification logic, as each must possess the requisite constructs to facilitate the other. The second challenge concerns the semantic treatment of allocation and deallocation, pertaining to dynamic data structures. Existing techniques [1, 6] for modelling allocation in the relational security setting cannot cope with deallocation; hence the need for two different allocators — one for low and another for high addresses (Section III).

In a companion technical report [29] (TR in the sequel) we develop a logic for relational reasoning about RHTT programs, and a worked out verification of an example program. Inference rules of the logic have been verified sound against a semantic model, and are formally implemented as lemmas in RHTT. The soundness of our program logic, the domain theoretic implementation of our semantic model, as well as all of our examples, have been fully and formally verified in Coq. Additional technical difficulties arise in this process, but we elide them here for readability. The interested reader is invited to look at our Coq proofs, and the companion TR, which are available at http://software.imdea.org/~aleks/rhtt/.

II. RHTT BY EXAMPLES

Overview: As suggested by the introduction, this paper assumes understanding of the following aspects of type theory: (1) *Dependent function types*, used to specify how the body of a function depends on the input arguments. To illustrate, consider the type $\mathsf{vector}(n)$, of integer-storing arrays. This type is dependent on the size parameter n. A function computing the inner product of two vectors can be typed as

$$\Pi n{:}\mathsf{nat}.\,\mathsf{vector}(n) \times \mathsf{vector}(n) \to \mathsf{nat}$$

capturing the invariant that that the argument vectors must be of equal size. In RHTT, dependent function types naturally arise when specifying any kind of program behavior. (2) *Module systems (including abstract types and predicates)*, for information hiding, and as we show, declassification. (3) *Inductive types*, for specifications of programs that manipulate (possibly heterogeneous) data structures such as lists, trees, etc.

To use RHTT in practice, it is further important to be familiar with some implementation of type theory (our chosen one is Coq [25], but others exist too), as one needs to interact with the system to discharge verification conditions. Our presentation in this paper does not include such interaction aspects, and hence does not assume familiarity with Coq.

RHTT basics: types, specifications, opaque sealing: To begin with, our types must be able to express at least noninterference: that low outputs of a computation are independent of high inputs. To illustrate, assume a function $f{:}A^2{\to}A^2$, where $A^2 = A \times A$. Also, let $e.1$ and $e.2$ denote resp. the first and the second component of the ordered pair e. Then, mathematically, f's first output is independent of f's second argument iff

$$\forall x_1\ x_2\ y_1\ y_2.\, x_1 = x_2 \to f(x_1, y_1).1 = f(x_2, y_2).1$$

In other words, in two runs of f, equal x inputs, lead to equal $f(x, y).1$ outputs. This relational statement of independence can be viewed as a definition of noninterference in terms of f alone [1, 9], without recourse to outside concepts such as security lattices [8, 17]. Consequently, inputs and outputs related by equality in the two runs of f are considered low (x and $f(x, y).1$ above), and the unconstrained values (y and $f(x, y).2$) are by default considered high. So defined, the notions of low and high security are intrinsic to the considered

specification, rather than to the code itself; one is free to consider statements about f in which the inputs and outputs take other security levels.

In RHTT, program specifications are stated using a *monadic* type $\mathsf{STsec}\, A\, (p, q)$, which classifies heap-manipulating, potentially diverging computations e whose return value has type A. e's *precondition* p is a predicate over heaps, i.e., function of type $\mathsf{heap}{\rightarrow}\mathsf{prop}$. The reader can roughly think of prop as type bool which in addition to the usual logical operations supports quantifiers as well. The precondition selects a set of heaps from which e's execution will be memory-safe (e.g., there will be no dangling-pointer dereferences or run-time type errors). This automatically provides a mechanism for controlling access to heap locations, in a manner identical to that of separation logic [35]: e may only access those locations that are provably in all the heaps satisfying e's precondition, or that e allocated itself. We will illustrate access control via preconditions in subsequent examples (see, e.g., Example 4).

The *postcondition* q relates the output values, input heaps and output heaps of any *two terminating* executions of e. Thus q has the type $A^2{\rightarrow}\mathsf{heap}^2{\rightarrow}\mathsf{heap}^2{\rightarrow}\mathsf{prop}$. The postcondition does not apply if one or both of the executions of e are diverging. In that respect, our type system is *termination insensitive* [38]. While p controls access to locations x, we use q to implement information flow policies about x. This is why q is a predicate over two runs. For example, q may specify that x is low, so that e may freely propagate x's value. Or x may be high, requiring that all x-dependent outputs of e must be high too. Or x may be high but q may require all of e's final heap to be low, in which case e must deallocate or rewrite any portion of its final heap that depends on x.

RHTT is implemented via shallow-embedding into Coq, which it extends with STsec types. In the implementation of STsec types in Coq, we rely on the ability of Coq modules to perform *opaque sealing* [19, 22]; that is, hiding the implementations of various values within a module, while only exposing their types, thus forcing the clients of the module to be generic with respect to implementations of the module. Moreover, the actual implementations of opaquely-sealed functions, types and propositions cannot be recovered by clients, because RHTT does not contain constructs for pattern-matching (i.e., making observations) on the structures of such values.

We point out that our types can only describe the properties of the input and output states of the program (via pre- and postconditions), but not of intermediate states. Although this is not a significant limitation for a sequential, non-reactive language like RHTT, further work in this direction is left for future work.

Syntax, heaps, implicit flow: Consider the following program, P_1, adapted from Terauchi and Aiken [44], and presented here in a Haskell-like notation. We use side-effecting primitives such as write $x\ y$, which stores the value y into the location x; read x, which returns the contents of x; and $x \leftarrow e_1; e_2$, which sequentially composes e_1 and e_2, binding the return value of e_1 to x. In future examples, we will also use alloc x, which returns a fresh memory location initialized with x; and dealloc x, which deallocates the location x from the heap. Additionally, we use do to delimit the scope of the side-effectful computations. Our actual syntax implemented in Coq differs somewhat from the one here in the treatment of variable binding, an issue we ignore for the time being but to which we return in Section III. Further, we freely use all the constructors inherited from CiC and Coq, such as for example, functions (fun), and dependent function type constructor (Π).

$$
\begin{array}{l}
P_1 \;\widehat{=}\; \mathsf{fun}\ x\ y\ z\ lo\ hi{:}\mathsf{ptr}. \\
\quad \mathsf{do}\,(\mathsf{write}\ z\ 1; b \leftarrow \mathsf{read}\ hi; \\
\qquad \mathsf{if}\ b\ \mathsf{then}\ \mathsf{write}\ x\ 1\ \mathsf{else}\ (w \leftarrow \mathsf{read}\ z; \mathsf{write}\ x\ w); \\
\qquad u \leftarrow \mathsf{read}\ x; v \leftarrow \mathsf{read}\ y; \\
\qquad \mathsf{write}\ lo\ (u + (v\ \mathsf{mod}\ 10)))
\end{array}
$$

Pointers x, y, z, lo store integers, and hi stores a boolean. The policy is: contents of lo and y are low at program input and output, while contents of x, z, hi are high. P_1 satisfies the policy because: (1) the value of y is not modified, and (2) the value of lo is modified to store the sum of the contents of x and the contents of y modulo 10, but this sum is independent of high data: at the time of writing lo, x has been rewritten by 1 in both branches of the conditional. Thus, P_1 can be ascribed the following dependent type, U.

$$
\begin{array}{l}
U \;\widehat{=}\; \Pi x\ y\ z\ lo\ hi : \mathsf{ptr}.\ \mathsf{STsec}\ \mathsf{unit} \\
\quad (\mathsf{fun}\ i. \exists u\ v\ w\ c{:}\mathsf{nat}.\ b{:}\mathsf{bool}.\ j{:}\mathsf{heap}. \\
\qquad i = x \mapsto u \bullet y \mapsto v \bullet z \mapsto w \bullet hi \mapsto b \bullet lo \mapsto c \bullet j, \\
\quad \mathsf{fun}\ rr\ ii\ mm. \\
\qquad (ii.1\ lo = ii.2\ lo) \rightarrow (ii.1\ y = ii.2\ y) \rightarrow \\
\qquad (mm.1\ lo = mm.2\ lo \wedge mm.1\ y = mm.2\ y))
\end{array}
$$

The precondition states that P_1 must start in an initial heap i containing the five pointers x, y, z, lo, hi, with appropriately-typed contents. The heap i may be larger still; this is stated by existentially quantifying over the heap variable j. Heaps are (finite) maps from pointers to values; $x \mapsto u$ is a singleton heap containing only the location x storing value u; and $\bullet$ is *disjoint* heap union. The precondition insists that i be a disjoint union of smaller singleton heaps; hence there be no aliasing between the five pointers. The postcondition binds over three variables $rr{:}\mathsf{unit}^2$, $ii, mm{:}\mathsf{heap}^2$ which are, respectively, the pair of return values, the pair of initial heaps and the pair of ending heaps for the two runs of P_1. The postcondition states that if the contents of lo and y in the two initial heaps are equal (hence low), then they are low in the output heaps too.

Other types for P_1 are possible too. For example, we may specify that only the last digit of y is low, by replacing $ii.1\ y$ with $(ii.1\ y)\ \mathsf{mod}\ 10$ in the postcondition, and similarly with $ii.2$, $mm.1$ and $mm.2$. Or, the postcondition may state that the contents of x and z are low at the end of P_1, though not at the beginning. RHTT (like [1]) can deem arbitrary expressions as low, even though they may have high subparts. The only requirement is that the values of the expressions in two runs are the same. Because we are considering full functional verification, which STsec type a program should have is a matter of programmer's choice. The system merely

issues a proof obligation that the desired type is indeed valid, to be discharged interactively, using the logic we outline in the TR (Section 4). This proof obligation may not only be about security but also may concern full functional correctness.

Opaque sealing: The ascription of STsec types in RHTT is opaque, as mentioned earlier in this section. Even if program execution makes more values low, this knowledge cannot be utilized by clients if it is not exposed in the postcondition. For example, using P_1's type U, program

$$P_2 \;\widehat{=}\; \mathsf{fun}\ x\ y\ z\ lo\ hi.\ \mathsf{do}\ (P_1\ x\ y\ z\ lo\ hi;\ t \leftarrow \mathsf{read}\ x;\ \mathsf{return}\ t),$$

cannot be given a type in which t is low, because the postcondition in U does not expose the property that x is low at the end of P_1.

Local contexts: While the STsec type of P_1 classifies the security of the *contents* of x, y, z, lo, hi, it cannot classify the pointer addresses themselves, as the latter requires discerning the address names in the two different runs (e.g., $x.1$ and $x.2$). We therefore extend the STsec constructor with a *local context*, which is a list of types of the variables we consider local to the computation. For example, the type for P_1 in which the five pointer addresses are high, even though the contents of lo and y are low, can be written as follows, using the list $[\mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}]$ as the local context.

$$\begin{array}{l}\mathsf{STsec}\ [\mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}, \mathsf{ptr}]\ \mathsf{unit}\\ \quad(\mathsf{fun}\ x\ y\ z\ lo\ hi{:}\mathsf{ptr}.\ i{:}\mathsf{heap}.\\ \qquad \exists u\ v\ w\ c{:}\mathsf{nat}.\ b{:}\mathsf{bool}.\ j{:}\mathsf{heap}.\\ \qquad\quad i = x \mapsto u \bullet y \mapsto v \bullet z \mapsto w \bullet hi \mapsto b \bullet lo \mapsto c \bullet j,\\ \quad \mathsf{fun}\ xx\ yy\ zz\ llo\ hhi{:}\mathsf{ptr}^2.\ rr{:}\mathsf{unit}^2.\ ii\ mm{:}\mathsf{heap}^2.\\ \quad ii.1\ llo.1 = ii.2\ llo.2 \rightarrow ii.1\ yy.1 = ii.2\ yy.2 \rightarrow\\ \quad mm.1\ llo.1 = mm.2\ llo.2 \wedge mm.1\ yy.1 = mm.2\ yy.2)\end{array}$$

The type of the precondition (and similarly for postconditions) now changes to $\mathsf{ptr}^5 \rightarrow \mathsf{heap} \rightarrow \mathsf{prop}$, so that we can bind additional names for the pointers x, y, ... in the precondition, and *pairs* of pointers xx, yy, ... in the postcondition. The program syntax changes too, as the local variables now have to be bound within the scope of do. In other words, our program now looks like $P_3 \;\widehat{=}\; \mathsf{do}\ (\mathsf{fun}\ x\ y\ z\ lo\ hi.\ \mathsf{write}\ z\ 1; \ldots)$.

Remark 1. Ordinary function arguments, corresponding to the $\rightarrow$ and Π-types, can be viewed as a special kind of STsec-local arguments, where the security level is low by default. Indeed, any function $f{:}\Pi x{:}A.\ \mathsf{STsec}\ \Gamma\ (p\ x, q\ x)$ can be transformed into

$$\begin{array}{l}\mathsf{do}\ (\mathsf{fun}\ x\ \gamma_1 \ldots \gamma_n.\ f\ x\ \gamma_1 \ldots \gamma_n) :\\ \quad \mathsf{STsec}\ (A{::}\Gamma)\ B\\ \quad\ (\mathsf{fun}\ x\ \gamma_1 \ldots \gamma_n.\ p\ x\ \gamma_1 \ldots \gamma_n,\\ \quad\quad \mathsf{fun}\ xx\ \gamma\gamma_1 \ldots \gamma\gamma_n\ yy\ ii\ mm.\\ \quad\quad\quad xx.1 = xx.2 \rightarrow q\ xx\ \gamma\gamma_1 \ldots\ yy\ ii\ mm)\end{array}$$

Here, the variables $\gamma_1, \ldots, \gamma_n$ are typed with types from the local context Γ, and the postcondition explicitly declares x to be low, by inserting the hypothesis $xx.1 = xx.2$. To summarize, function arguments are always low, whereas variables in local contexts may be low, high, or subject to a more precise security specification, depending on the postcondition.

Example 1 (Nested conditionals)**.** The following program is adapted from Simonet [40]. It uses low arguments a, b, c, u, v, and a high argument x which is declared in the local context but is unrestricted by the postcondition. It nests two conditionals to compute the final result, but the result is independent of x, and hence is low. Owing to the non-trivial implicit control flow, however, most security type systems will not be able to establish this independence and typecheck the example accordingly. Simonet's type system for sum types *can* typecheck the example using types annotated with matrices containing security levels. In contrast, in RHTT the type can precisely describe the final result, y, as a function of the inputs: $y \;\widehat{=}\; (a = c)\,||\,(b = c)\,||\,u\,||\,v$. Clearly y does not depend on x and we prove that y is low by proving that $yy.1 = yy.2$ in the postcondition.

$$\begin{array}{l}P_4\ :\ \Pi a\ b\ c\ u\ v{:}\mathsf{bool}.\ \mathsf{STsec}\ [\mathsf{bool}]\ \mathsf{bool}\\ \qquad (\mathsf{fun}\ x\ i.\ \mathsf{True},\\ \qquad\ \mathsf{fun}\ xx\ yy\ ii\ mm.\ yy.1 = yy.2 \wedge mm = ii \wedge\\ \qquad\quad yy.1 = (a = c)\,||\,(b = c)\,||\,u\,||\,v) \;\widehat{=}\\ \quad \mathsf{fun}\ a\ b\ c\ u\ v.\\ \qquad \mathsf{do}(\mathsf{fun}\ x.\ t \leftarrow \mathsf{if}\ u\ \mathsf{then}\\ \qquad\qquad\qquad\qquad \mathsf{if}\ x\ \mathsf{then}\ \mathsf{return}\ a\ \mathsf{else}\ \mathsf{return}\ b\\ \qquad\qquad\qquad\quad \mathsf{else}\\ \qquad\qquad\qquad\qquad \mathsf{if}\ v\ \mathsf{then}\ \mathsf{return}\ a\ \mathsf{else}\ \mathsf{return}\ c;\\ \qquad\qquad\qquad\quad \mathsf{return}\ ((t = a)\,||\,(t = b)))\end{array}$$

Example 2 (Access control through abstraction)**.** What if we want to allow read access, but not write access to some data (or vice-versa), or that access should be made conditional upon successful authentication? To enforce this kind of access control, we employ the standard abstraction mechanisms of type theory, such as abstract types, predicates and modules. The data to be protected can be hidden behind module boundaries, so that it can be accessed only via dedicated methods that enforce access control. For example, let Alice be a module storing some integer data, say salary, whose integrity should be enforced: Alice allows the salary to be readable globally, but only Alice herself can update it, to keep the value coherent with promotions at work. Thus, she exports unconstrained functions for creating new instances and for reading the salary, but the function for writing requires a check against a password that is also stored locally. The signature, AliceSig in Figure 1, presents the specifications that Alice wants to export, and a possible module implementing AliceSig by keeping two local pointers – one for the salary, one for the password – is given in Figure 2. Referring to Figure 2, the method new takes a nat salary and a string password, and generates a new instance of Alice, initialized with this data; read_salary takes a local-context argument representing Alice, and returns her current salary; write_salary takes a new salary, a password, and an alice argument in the local context, and updates the salary only if the supplied password matches the password stored in the alice argument. Referring back to Figure 1, AliceSig specifies an abstract type alice, abstract predicates sshape and shape, a relation, srefl, between shape and sshape, and the types of the methods. Although these

```
alice : Type
sshape : alice² → nat² → string² → heap² → prop
shape ≙ fun a s p h. sshape (a, a) (s, s) (p, p) (h, h)
srefl : ∀aa ss pp ii. sshape aa ss pp ii →
    shape aa.1 ss.1 pp.1 ii.1 ∧ shape aa.2 ss.2 pp.2 ii.2
new : nat → string → STsec nil alice
  (fun i. True,
   fun aa ii mm.
     ∃ss pp hh. mm = ii •• hh ∧ sshape aa ss pp hh)
read_salary : STsec [alice] nat
  (fun a i. ∃s p j h. i = j • h ∧ shape a s p j,
   fun aa yy ii mm. ∀ss pp jj hh.
     ii = jj •• hh → sshape aa ss pp jj →
     mm = ii ∧ yy = ss)
write_salary : nat → string → STsec [alice] unit
  (fun a i. ∃s p j h. i = j • h ∧ shape a s p j,
   fun aa qq yy ii mm. ∀ss pp jj hh.
     ii = jj •• hh → sshape aa ss pp jj →
     ∃jj' ss'. mm = jj' •• hh ∧ sshape aa ss' pp jj')
```

Fig. 1. AliceSig: access control via abstract predicates.

```
type alice ≙ ptr × ptr
salary (a : alice) ≙ a.1
passwd (a : alice) ≙ a.2
sshape (aa : alice²) (ss : nat²) (pp : string²) (ii : heap²) ≙
  ii.1 = salary aa.1 ↦ ss.1 • passwd aa.1 ↦ pp.1 ∧
  ii.2 = salary aa.2 ↦ ss.2 • passwd aa.2 ↦ pp.2 ∧
    ss.1 = ss.2 ∧ pp.1 = pp.2
new s p ≙ do (x ← alloc s; y ← alloc p; return (x, y))
read_salary ≙ do (fun a. read (salary a))
write_salary s p ≙
  do (fun a. x ← read (passwd a);
        if x = p then write (salary a) s else return ())
```

Fig. 2. Implementation of AliceSig.

figures may look complicated, the reader should bear in mind that they are intended for full functional verification. Also, the definitions of the various abstract predicates and types such as sshape, shape and alice, will be hidden from the clients, and do not contribute to the complexity.

The sshape predicate is a relational invariant of the module's local state (i.e., invariant over two runs). It is parametrized over pairs of alices, nat salaries, string passwords, and heaps that are current during execution. The parametrization by all these values captures that different instances of Alice that may be allocated at run time all have different local states, which can potentially store different salaries and passwords. If we were not interested in tracking the changes to salaries and passwords, but only in restricting write access, then these can be omitted from sshape, resulting in fewer quantifiers and hence simpler STsec types for the methods.

For use in preconditions for access control, we employ the non-relational variant shape which is a diagonal of sshape, as constrained by srefl. Recall that a computation in RHTT can access locations only in those heaps that provably satisfy its precondition. Correspondingly, a method that wants to access Alice's local state, has to describe the desired parts of that state in its own precondition. This is why AliceSig keeps sshape and shape abstract. The abstraction hides the layout of Alice's local state from the clients, thus preventing them from describing the layout in their preconditions and forcing them to access Alice's local state exclusively via the exported methods. Apart from giving code for the methods, the implementation also provides a *proof* of srefl (elided here, but present in the Coq scripts).

The STsec types in Alice's methods describe several additional properties. For example, that the local state of each instance of Alice is disjoint from that of another instance. For new, this is achieved by stating that the pair of ending heaps mm extends the initial ones ii by newly allocated sections hh ($mm = ii \bullet\bullet hh$). Here $\bullet\bullet$ generalizes the disjointness operator $\bullet$ to pairs of heaps, that is, $(i_1, i_2) \bullet\bullet (h_1, h_2) = (i_1 \bullet h_1, i_2 \bullet h_2)$. For read_salary, we allow that the state in which the function executes be larger than the module's local state by allowing $ii = jj \bullet\bullet hh$ where jj names the local state and hh is the potential global part. For write_salary we require that the global part, hh, remain invariant, but the local part may be changed by storing the new salary.

Finally, the specifications expose that read_salary does not change Alice's local state ($mm = ii$ in the postcondition). On the other hand, write_salary may change the salary field, but not the password field, as the sshape predicate changes from using the salary ss to using ss', but pp persists.

Notice that the salary and password arguments in new and write_salary are ordinary function arguments, whereas alice is in the local context of STsec in read_salary and write_salary. Thus, within the scope of Alice's methods, the salary and the password are low (c.f. Remark 1) whereas the alice argument is high because it is unconstrained by the methods' pre- and postconditions. Of course, as far as clients of AliceSig are concerned, all three of these are high: the abstraction over sshape hides all relations between the stored values.

Example 3 (Declassification)**.** One consequence of making salary and password internally low is that whenever a new instance of Alice is allocated, or a salary of an existing instance is changed, the salary and password have to be computed only out of low arguments – it is not possible for Alice to store confidential data into her local fields. Additionally, the specifications of new and write_salary must hide that the stored salary and password are equal to the supplied ones. The latter are internally low, while the former are to be externally high. The hiding is achieved by existential quantification over ss and pp in the postcondition of new, and over ss' in the postcondition of write_salary.

Alice can use the internal knowledge that salary and password are low, to implement and export an additional function which declassifies her salary – that is, reveals the internal knowledge that the salary is low. This declassification can be based on arbitrary conditions – say, it is only granted if a

```
rreadable : alice² → heap² → prop
readable ≙ fun a h. rreadable (a, a) (h, h)
rrefl : ∀aa ii. rreadable aa ii →
                  readable aa.1 ii.1 ∧ readable aa.2 ii.2
grant : STsec [alice] unit
  (fun a i. ∃s p j h. i = j • h ∧ shape a s p j,
   fun aa yy ii mm. ∀ss pp jj hh.
     ii = jj •• hh → sshape aa ss pp jj →
     ∃jj'. mm = jj' •• hh ∧ sshape aa ss pp jj' ∧
           rreadable aa jj')
revoke : STsec [alice] unit
  (fun a i. ∃s p j h.
     i = j • h ∧ shape a s p j ∧ readable a j,
   fun aa yy ii mm. ∀ss pp jj hh.
     ii = jj •• hh → sshape aa ss pp jj →
     ∃jj'. mm = jj' •• hh ∧ sshape aa ss pp jj')
read_salary : STsec [alice] nat
  (fun a i. ∃s p j h.
     i = j • h ∧ shape a s p j ∧ readable a j,
   fun aa yy ii mm. ∀ss pp jj hh.
     ii = jj •• hh → sshape aa ss pp jj →
     mm = ii ∧ yy = ss)
```

Fig. 3. Extension of AliceSig with state-based read access.

correct password has been supplied.

```
declassify : Πp:string. STsec [alice] bool
  (fun a i. ∃s q j h. i = j • h ∧ shape a s q j,
   fun aa yy ii mm. ∀ss qq jj hh.
     ii = jj •• hh → sshape aa ss qq jj →
     ∃jj'. mm = jj' •• hh ∧ sshape aa ss qq jj' ∧
           yy.1 = yy.2 ∧ yy = (p = qq.1, p = qq.2) ∧
           yy.1 → ss.1 = ss.2) ≙
  fun p. do (fun a. x ← read (passwd a); return (p = x))
```

The code of declassify checks if the supplied password equals the stored one, and returns the corresponding boolean. declassify does not return the value of the salary; for that, one has to use read_salary, but the specification of declassify shows that the salary is low if declassify returned true ($yy.1 \to ss.1 = ss.2$). This is possible because the low-status of the salary has been hardwired into the implementation of sshape, and Alice can reveal it at will.

Example 4 (State-based policies)**.** Alice can implement policies that change depending on her local state. For example, she may control the granting of read access with functions grant and revoke, as specified in Figure 3. These enable and disable reading by, respectively, adding and removing a new abstract predicate – rreadable – from the knowledge exposed about Alice's local state. Typically, such functions require authentication, but for simplicity, we forgo that aspect. The postcondition of grant exposes that the newly obtained state jj' is readable, while revoke omits this property, thus revoking the read access. To associate the predicate with reading, the specification of read_salary has to require a proof of readable.

The signature keeps rreadable abstract, so that the only way readable can be derived is if rreadable has been placed into the proof context by a previous call to grant, without an intervening revoke. The signature can be implemented by extending Alice's state with an additional boolean pointer that is set and reset by grant and revoke: rreadable is in force once the boolean is set $true$. Our Coq scripts provide several different implementations of this interface.

```
bbshape : bob²→G²→heap²→prop
bshape ≙ fun b k t. bbshape (b, b) (k, k) (t, t)
brefl : ... (* similar to srefl *)

epre (a : alice) (b : bob) (j i : heap) ≙
  ∃s p k t h. i = j • t • h ∧ shape a s p j ∧ bshape b k t
epost aa bb jj yy ii mm ≙
  ∀ss pp kk tt hh. ii = jj •• tt •• hh →
      sshape aa ss pp jj → bbshape bb kk tt →
  ∃jj' tt'. mm = jj' •• tt' •• hh ∧ sshape aa ss pp jj' ∧
        bbshape bb (bcmp kk.1, bcmp kk.2) tt' ∧
        (tt.1 = tt.2 → tt'.1 = tt'.2)
```

Fig. 4. Some definitions for conditional access and erasure policies.

Example 5 (Conditional access and erasure policies)**.** Suppose Alice wants to download a program from Bob for computing tax returns. Alice is willing to let Bob access her local state and read her salary directly using read_salary, but wants to prevent Bob from stealing her secret by copying it into his own local state. Alice may insist that Bob not keep any local state, or that he deallocate all of it before termination. But this is too restrictive, for Bob may want to keep in his local state a count of how many times his program has executed. Such local state should be allowed to escape the function call as it is independent of Alice's salary. In RHTT, Alice can formulate such a permissive policy.

We start the description of Alice's specification by assuming that bob is an abstract type representing Bob's local state (usually implemented as the set of root pointers for Bob's local state). G is another abstract type representing the values that Bob keeps in his local state. For example, if Bob wants to count how many times his program has been invoked, then $G = \mathsf{nat}$. Further, we assume bbshape:$\mathsf{bob}^2 \to G^2 \to \mathsf{heap}^2 \to \mathsf{prop}$ is an abstract predicate describing Bob's local state, and bcmp:$G \to G$ describes how Bob's local state changes within one function call. In the counting example, bcmp will be the program Bob needs to run over both Alice's and Bob's local heap. The initial heaps i for his program can therefore be split in three ways: j belonging to Alice, t belonging to Bob, and the remainder h that is untouched. Predicate epre in Figure 4 describes this situation. On the other hand, epost states that Bob's local state tt' at the end stores the correct statistics (bcmp of $kk.1$ and $kk.2$), and if Bob's initial local state tt is assumed low, then tt' is low as well. In other words, Bob did not copy into tt' any of the high values that he may have read from Alice. A program that requests read access to Alice's local store, and respects

the described policy has the type

$$T \;\widehat{=}\; \mathsf{STsec}\ [\mathsf{alice},\mathsf{bob}]\ \mathsf{nat}\ (\mathsf{fun}\ a\ b\ i.\exists j.\,\mathsf{readable}\ a\ j \wedge \mathsf{epre}\ a\ b\ j\ i,\ \mathsf{fun}\ aa\ bb\ yy\ ii\ mm.\forall jj.\ \mathsf{rreadable}\ aa\ jj \to \mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm)$$

Alice now wants to ratify programs with type T by granting them read access to her salary. She can do so by exporting from her module a function ratify which removes readable from T, much like the grant program would do. After that, Bob's program can execute without needing special reading privileges. In this respect, ratify is a *higher-order* function because in its type, STsec appears in a negative (argument) position. ratify can be said to implement a conditional access policy, because it grants access only after Bob supplies a proof that his program satisfies the type T, i.e., the program does not leak Alice's salary.

$$\mathsf{ratify} : T \to \mathsf{STsec}\ [\mathsf{alice},\mathsf{bob}]\ \mathsf{nat}\ (\mathsf{fun}\ a\ b\ i.\exists j.\,\mathsf{epre}\ a\ b\ j\ i,\ \mathsf{fun}\ aa\ bb\ yy\ ii\ mm.\ \forall jj.\,\mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm) \;\widehat{=}\; \mathsf{fun}\ e : T.\,\mathsf{do}\ (\mathsf{fun}\ a\ b.\,e\ a\ b)$$

This specification can be instantiated in several ways, by choosing different values for bob, G, bbshape and bcmp. For example, if bob $\widehat{=}$ ptr, $G \widehat{=}$ nat, bcmp $\widehat{=}$ succ and bbshape $bb\ kk\ tt \;\widehat{=}\; (tt.1{=}bb.1 \mapsto kk.1 \wedge tt.2 = bb.2 \mapsto kk.2)$ then Bob's program keeps a single pointer whose content is incremented by 1 after every execution.

Bob's program which computes the tax of 24% of Alice's salary, while also keeping its invocation count, can be implemented and then immediately ratified by the following function call. Notice that by the type of ratify, the return value of Bob's program is high as there is no requirement $yy.1 = yy.2$ in epost. Hence, the fact that this value is a function of Alice's salary, is not a security leak.

$$\mathsf{ratify}\ (\mathsf{do}\ (\mathsf{fun}\ a{:}\mathsf{alice}\ b{:}\mathsf{bob}.\ x \leftarrow \mathsf{read\_salary}\ a; k \leftarrow \mathsf{read}\ b; \mathsf{write}\ b\ (k+1); \mathsf{return}\ (x * 24\%)))$$

Suppose Bob keeps the count with two nat pointers, whose contents p and q are both increased at every call, so that the overall count is the difference between the two. This is represented by taking bob $\widehat{=}$ ptr $\times$ ptr (one ptr for p and one for q), $G \widehat{=}$ nat, bcmp $\widehat{=}$ succ and bbshape $bb\ kk\ tt$ is

$$\exists pp\ qq{:}\mathsf{nat}^2.\ tt.1 = \mathsf{fst}\ (bb.1) \mapsto pp.1 \bullet \mathsf{snd}\ (bb.1) \mapsto qq.1 \wedge tt.2 = \mathsf{fst}\ (bb.2) \mapsto pp.2 \bullet \mathsf{snd}\ (bb.2) \mapsto qq.2 \wedge kk = (pp.1 - qq.1, pp.2 - qq.2))$$

Bob's program can read Alice's salary, then increment p and q by amount of the salary, and additionally, increment p by 1. In terms of required specifications for Bob's local state, the program still keeps the invocation count. However, the program is actually stealing Alice's salary, because the salary can be inferred by deducting the old value of q from the new one. Bob will fail to get such a program ratified by Alice, if he calls ratify with the argument

$$\mathsf{do}\ (\mathsf{fun}\ a{:}\mathsf{alice}\ b{:}\mathsf{bob}.\ x \leftarrow \mathsf{read\_salary}\ a; p \leftarrow \mathsf{read}\ (\mathsf{fst}\ b); q \leftarrow \mathsf{read}\ (\mathsf{snd}\ b); \mathsf{write}\ (\mathsf{fst}\ b)\ (p+x+1); \mathsf{write}\ (\mathsf{snd}\ b)\ (q+x); \mathsf{return}\ (x * 24\%))$$

ratify forces Bob to prove that his ending state is low $(tt'.1 = tt'.2)$ as defined in epost, but this is not provable if tt' stores Alice's salary x. Indeed, as x is high, Bob lacks the information that x is equal in the two runs, so he cannot prove that his pointers store equal values in two runs. For ratification, Bob will have to *erase* Alice's salary from his state, perhaps by mutating his pointers to store $p+1$ and q instead of $p+x+1$ and $q+x$. ratify may thus be said to implement an *erasure policy*, similar to those of Chong and Myers [15, 16]. Alternatively, Bob may try to declassify Alice's salary, using the function from Example 2, but then he has to provide the correct

III. Typing rules

Each command of the stateful fragment of RHTT comes with a dependent STsec type that captures the command's specification using pre- and post-conditions. We start our description with the types of the basic commands; descriptions of the other commands appear later in the section.

$$\mathsf{return} : \mathsf{STsec}\ [A]\ A\ (\mathsf{fun}\ x\ i.\,\mathsf{True},\ \mathsf{fun}\ xx\ yy\ ii\ mm.\ mm = ii \wedge yy = xx)$$

$$\mathsf{read} : \mathsf{STsec}\ [\mathsf{ptr}]\ A\ (\mathsf{fun}\ \ell\ i.\exists h{:}\mathsf{heap}\ v{:}A.\ i = \ell \mapsto v \bullet h,\ \mathsf{fun}\ \ell\ell\ yy\ ii\ mm.\ mm = ii \wedge \forall hh\ vv.\ ii = (\ell\ell.1 \mapsto vv.1, \ell\ell.2 \mapsto vv.2) \bullet\bullet hh \to yy = vv)$$

$$\mathsf{write} : \mathsf{STsec}\ [\mathsf{ptr}, A]\ \mathsf{unit}\ (\mathsf{fun}\ \ell\ v\ i.\exists h\ B{:}\mathsf{type}\ w{:}B.\ i = \ell \mapsto w \bullet h,\ \mathsf{fun}\ \ell\ell\ vv\ yy\ ii\ mm.\ \forall hh\ B_1\ B_2\ w_1{:}B_1\ w_2{:}B_2.\ ii = (\ell\ell.1 \mapsto w_1, \ell\ell.2 \mapsto w_2) \bullet\bullet hh \to mm = (\ell\ell.1 \mapsto vv.1, \ell\ell.2 \mapsto vv.2) \bullet\bullet hh)$$

$$\mathsf{dealloc} : \mathsf{STsec}\ [\mathsf{ptr}]\ \mathsf{unit}\ (\mathsf{fun}\ \ell\ i.\exists h\ B{:}\mathsf{type}\ w{:}B.\ i = \ell \mapsto w \bullet h,\ \mathsf{fun}\ \ell\ell\ yy\ ii\ mm.\ \forall hh\ B_1\ B_2\ w_1{:}B_1\ w_2{:}B_2.\ ii = (\ell\ell.1 \mapsto w_1, \ell\ell.2 \mapsto w_2) \bullet\bullet hh \to mm = hh)$$

return immediately terminates with the value that was supplied as a local argument. Its STsec constructor records the argument type in the local context, and the type of the returned value (here, both types are A). The precondition states that return can execute in any heap, as it performs no heap operations. The postcondition states that return does not change the input heaps $(mm = ii)$ and passes the input argument to the output $(yy = xx)$. The precondition of read write and dealloc all require that the initial heaps contain at

least the pointer ℓ to be read from, written to or deallocated. In the case of read, the contents of the pointer must have the expected type A. For write and dealloc, this type is irrelevant and is hence existentially quantified. The postconditions of all three commands explicitly describe the layout of the new heap and, in particular, state that parts of the input heaps that are disjoint from ℓ (hh above) remain invariant.

Allocation presents the following challenge. If under a high guard, a pointer is allocated in one branch of a conditional, but not in the other, this may constitute a leak of the high guard, if the pointer itself is of low security. Such "unmatched" allocations should therefore always produce high pointers. This is why we provide two allocation primitives: lalloc for allocating low pointer addresses, and alloc, for allocating high ones.

$$\begin{aligned}&\mathsf{lalloc} : \mathsf{STsec}\ [A]\ \mathsf{ptr}\\ &\qquad (\mathsf{fun}\ v\ i.\ \mathsf{True}, \mathsf{fun}\ vv\ yy\ ii\ mm.\\ &\qquad\quad mm = (yy.1 \mapsto vv.1, yy.2 \mapsto vv.2) \bullet\bullet\ ii \wedge\\ &\qquad\quad (ii.1 \cong ii.2 \to yy.1 = yy.2 \wedge mm.1 \cong mm.2))\end{aligned}$$

$$\begin{aligned}&\mathsf{alloc}\ : \mathsf{STsec}\ [A]\ \mathsf{ptr}\\ &\qquad (\mathsf{fun}\ v\ i.\ \mathsf{True}, \mathsf{fun}\ vv\ yy\ ii\ mm.\\ &\qquad\quad mm = (yy.1 \mapsto vv.1, yy.2 \mapsto vv.2) \bullet\bullet\ ii \wedge\\ &\qquad\quad \mathsf{even}\ yy.1 \wedge \mathsf{even}\ yy.2)\end{aligned}$$

Both commands take a local argument $v{:}A$, and return a *fresh* pointer initialized with v. The freshness is captured in the postcondition by demanding that the initial heaps ii be *disjoint* from the returned pointers yy in the equation for the ending heaps mm. However, alloc chooses the returned location non-deterministically, while lalloc is *deterministic*; that is, it returns *equal* (and hence low) pointers, when invoked under appropriate conditions. We make the two allocators operate on disjoint pools of locations: alloc always returns an *even* pointer (albeit, a randomly chosen one), while lalloc returns the next unallocated *odd* pointer. Here we rely on the property that type ptr is isomorphic to nat in our model.

Definition 1. Heaps h_1 and h_2 are *low-equivalent*, written $h_1 \cong h_2$, iff their domains contain the same odd pointers. The content of the pointers is irrelevant.

The postconditions of lalloc and alloc further capture the behavior of the two commands with respect to the $\cong$ relation. In the case of lalloc, we expose that if invoked in low-equivalent input heaps ($ii.1 \cong ii.2$), the command returns equal pointers ($yy.1 = yy.2$), and low-equivalent output heaps ($mm.1 \cong mm.2$). In the case of alloc, we expose the evenness of $yy.1$ and $yy.2$, and provide a number of lemmas, that can be used to relate evenness with $\cong$. For example, the lemma

$$\forall x{:}\mathsf{ptr}.\, \mathsf{even}\ x \to (x \mapsto v \bullet h_1 \cong h_2) \leftrightarrow (h_1 \cong h_2)$$

when iterated, can show that low equivalence of h_1 and h_2 is preserved after arbitrary number of high allocations. Other related lemmas are present in our Coq scripts.

Example 6 (The need for both allocators)**.** The following program can be given a type in which the returned pointer y is low, no matter what the boolean h is.

$$\begin{aligned}&\mathsf{do}\ (\mathsf{fun}\ h.\, \mathsf{if}\ h\ \mathsf{then}\ y \leftarrow \mathsf{lalloc}\ 2; \mathsf{return}\ y\ \mathsf{else}\\ &\qquad\qquad x \leftarrow \mathsf{alloc}\ 1; y \leftarrow \mathsf{lalloc}\ 2; \mathsf{dealloc}\ x; \mathsf{return}\ y) :\\ &\quad \mathsf{STsec}\ [\mathsf{bool}]\ \mathsf{ptr}\\ &\qquad (\mathsf{fun}\ h\ i.\, \mathsf{True},\\ &\qquad\ \mathsf{fun}\ hh\ yy\ ii\ mm.\, mm = ii \bullet\bullet (yy.1 \mapsto 2, yy.2 \mapsto 2) \wedge\\ &\qquad\qquad\qquad (ii.1 \cong ii.2 \to yy.1 = yy.2))\end{aligned}$$

The program does not typecheck if the high allocation of x is replaced by lalloc. In that case, it is possible that the two executions of the program select different branches of the conditional (depending on h). If we started with low-equivalent heaps $i_1 \cong i_2$, then at the point of allocation of y, the heaps will not be low equivalent anymore, since one of them has been extended with an odd location x, while the other has not. Thus, we cannot conclude that the returned pointer is low ($yy.1 = yy.2$).

Remark 2. Deterministic allocation forces STsec to use *large-footprint specifications*, whereby specifications describe the full heaps in which commands operate. This is in contrast to separation logic, where specifications describe only those heap parts that commands touch, and implicitly assume invariance of the remaining heap. The latter style is more succinct, but cannot support deterministic allocation [47]. With large footprints, we can specify lalloc (specifically, the antecedent $ii.1 \cong ii.2$ in the postcondition), but the invariance of untouched parts of heaps has to be stated explicitly for every program, as witnessed by the quantification over hh in the postconditions of write and dealloc. Note that *the concrete layouts of untouched parts of heaps do not need to appear in the specifications* — thus alleviating concerns of scalability of specifications. Moreover, the overhead between large and small footprint specifications is *constant*, as we discuss in Section 6 of the TR. The two styles also lead to similar proofs. What matters in proofs is the ability to effectively reason about heap disjointness, and we can do that equally well in both styles by relying on the operator $\bullet$ [32].

Another way of treating allocation in the relational setting is to model its non-determinism by means of partial bijections between pointers [1]. Then one can avoid using two different allocators, albeit at a price of increasing the complexity of reasoning. Such proposals, however, only work in the *absence of deallocation*. For example, the definition of noninterference of Amtoft et al. [1] allows that the input heaps to the computation are related by some bijection between pointers, and requires that the ending heaps are also related by a bijection. However, the ending bijection has to be an *extension* of the initial one. Obviously, such a definition cannot support deallocation, as deallocation produces smaller, not larger heaps. Alternatively, one can omit the extension requirement; but that leads to counterexamples which satisfy the weakened requirement even though they actually leak information.

We proceed to describe our constructor for sequential composition, but first we need some notational conventions. Let Γ be a list of types. We denote by $\overline{\Gamma}$ the product of all the types

in Γ, e.g., $\overline{\mathsf{nil}} = \mathsf{unit}$ and $\overline{[A,B,C]} = A{\times}B{\times}C$. We further conflate the function types $\overline{\Gamma}{\to}T$ and $\Gamma_1{\to}\Gamma_2{\to}\cdots{\to}T$, and their corresponding terms. For example, we freely interchange $\mathsf{fun}\ \gamma{:}\overline{[A,B,C]}\ldots$, or $\mathsf{fun}\ \gamma\ldots$ if the types are clear from the context, with $\mathsf{fun}\ x{:}A\ y{:}B\ z{:}C\ldots$. Similarly, we interchange $e\ (x,y,z)$ with $e\ x\ y\ z$. We hope that no confusion arises due to this abuse of notation; all of our exposition has been checked in Coq, where the notation is formally resolved.

For sequential composition $e_1; e_2$, let $e_1 : \mathsf{STsec}\,\Gamma\ A\ (p_1, q_1)$ and $e_2 : \mathsf{STsec}\,(A{::}\Gamma)\ B\ (p_2, q_2)$. Then $e_1; e_2$ first executes e_1, passing the returned value as the first local argument to e_2. Assuming $\gamma{:}\overline{\Gamma}$ and $\gamma\gamma{:}\overline{\Gamma}^2$, the STsec type for $e_1; e_2$ is

$$\begin{array}{l}\mathsf{STsec}\ \Gamma\ B\\ (\mathsf{fun}\ \gamma\ i.\, p_1\ \gamma\ i \wedge\\ \qquad \forall y\ m.\, q_1\ (\gamma,\gamma)\ (y,y)\ (i,i)(m,m) \to p_2\ (y,\gamma)\ m,\\ \mathsf{fun}\ \gamma\gamma\ yy\ ii\ mm.\\ \quad \exists vv{:}A^2.\, hh{:}heap^2.\, q_1\ \gamma\gamma\ vv\ ii\ hh \wedge\\ \qquad\qquad q_2\ ((vv.1, \gamma\gamma.1), (vv.2, \gamma\gamma.2))\ yy\ hh\ mm)\end{array}$$

In English, the precondition requires e_1 to be safe in the initial heap of the sequential composition, and that any value y and heap m obtained as output of e_1 – and which thus satisfy e_1's "squared" postcondition – make e_2 safe. The postcondition states that intermediate values vv and heaps hh exist, obtained after running e_1 but before running e_2.

As e_1's output is bound in the local context of e_2, we cannot treat this output as an ordinary functional variable, despite our suggestive notation in Section II. Indeed, as discussed previously in Section II, ordinary variables are always low, whereas the ones in the local context may be high, depending on the specification. Thus, we must rely on variable-free representation via *combinators*, as described next.

Our first combinator is for changing the local context of an STsec type. Given $\Gamma_1, \Gamma_2, f{:}\overline{\Gamma}_1{\to}\overline{\Gamma}_2$, and $e{:}\mathsf{STsec}\,\Gamma_2\ A\ (p,q)$, we can *instantiate* the local variables of e according to f, to produce a computation with context Γ_1.

$$\begin{array}{l}e\ @\ f : \mathsf{STsec}\ \Gamma_1\ B\\ \qquad (\mathsf{fun}\ \gamma.\, p\ (f\ \gamma), \mathsf{fun}\ \gamma\gamma.\, q\ (f\ \gamma\gamma.1, f\ \gamma\gamma.2))\end{array}$$

We denote by $e\ @_0\ \gamma$ the special instance of $@$, when $\Gamma_1 = \mathsf{nil}$ and hence, f is isomorphic to a tuple $\gamma{:}\overline{\Gamma}_2$. We refer to e in $e\ @\ f$ or $e\ @_0\ \gamma$ as the *head* of the instantiation, and to f as the *explicit substitution*.

Example 7. In Example 2, we implemented declassify as $\mathsf{fun}\ p.\, \mathsf{do}\ (\mathsf{fun}\ a.\, x \leftarrow \mathsf{read}\ (\mathsf{passwd}\ a); \mathsf{return}\ (p = x))$.The actual implementation using combinators is

$$\begin{array}{l}\mathsf{fun}\ p{:}\mathsf{string}.\, \mathsf{do}\ (\mathsf{read}\ @\ (\mathsf{fun}\ a.\, \mathsf{passwd}\ a);\\ \qquad\qquad \mathsf{return}\ @\ (\mathsf{fun}\ x\ a.\, (p = x)))\end{array}$$

Programs thus become lists of commands instantiated with explicit substitutions, where the domains of substitutions grow with each command to provide names for the results of previous commands. In the above example, the domain of the substitution for read includes only the variable $a{:}\mathsf{alice}$, but the substitution for return also includes $x{:}\mathsf{string}$, which names the result of the previous read (Alice's stored password). Similarly, the functions new and read_salary are reimplemented as

$$\begin{array}{rcl}\mathsf{new}\ s\ p & \widehat{=} & \mathsf{do}\ (\mathsf{alloc}\ @_0\ s; \mathsf{alloc}\ @\ (\mathsf{fun}\ x.\, p);\\ & & \qquad \mathsf{return}\ @\ (\mathsf{fun}\ y\ x.\, (x,y)))\\ \mathsf{read\_salary} & \widehat{=} & \mathsf{do}\ (\mathsf{read}\ @\ (\mathsf{fun}\ a.\, \mathsf{salary}\ a))\end{array}$$

To set the stage for discussing the combinator cond for conditionals we first consider the predicates safe and verify2. The former says that an expression $e : \mathsf{STSec}\,\mathsf{nil}\,(p,q)$ is safe to execute in any heap i satisfying precondition p, and is defined as $\mathsf{safe}\,e\,i \mathrel{\widehat{=}} p\,i$. The latter formalizes when it is that executions over *two programs*, e_1, e_2 result in heaps and values that satisfy a postcondition q. Although we have been concerned thus far with two runs of the *same* program, the above escalation becomes necessary when treating a conditional under a high guard because in two runs of the conditional different branches may be executed. Assuming $e_1 : \mathsf{STsec}\,\mathsf{nil}\ A\,(r_1,t_1)$ and $e_2 : \mathsf{STsec}\,\mathsf{nil}\ A\,(r_2,t_2)$, a pair of input heaps ii, and a predicate $q{:}A^2{\to}\mathsf{heap}^2{\to}\mathsf{prop}$, we define

$$\begin{array}{l}\mathsf{verify2}\ ii\ e_1\ e_2\ q \mathrel{\widehat{=}}\\ \quad \forall yy{:}A^2.\, mm{:}\mathsf{heap}^2.\\ \qquad (ii.1, yy.1, mm.1) \in \mathsf{runs\_of}\ e_1 \to\\ \qquad (ii.2, yy.2, mm.2) \in \mathsf{runs\_of}\ e_2 \to q\ yy\ mm\end{array}$$

Here runs_of (defined in the TR) coerces programs into relations between input heaps, output values and output heaps. It is a useful intuition to regard verify2 as a relational variant of a Hoare style specification, with programs e_1 and e_2, and postcondition q, except that the precondition has been replaced with concrete heaps $ii.1$ and $ii.2$.

Given programs $e_i{:}\mathsf{STsec}\ \Gamma\ A\ (p_i, q_i)$ for $i{=}1,2$, corresponding to branches of a conditional, and a boolean guard $b{:}\overline{\Gamma}{\to}\mathsf{bool}$ (here parametrized over a context), which type should we ascribe to the conditional? We would like to be precise, and ascribe the weakest precondition sufficient for the safety, and the strongest postcondition sound wrt. the expected semantics. Unfortunately, computing that postcondition seems impossible in the case when the boolean guard is high. Indeed, we know that q_1 (resp. q_2) relates the output heaps if both runs of the conditional chose the same branch e_1 (resp. e_2), but nothing can be said if the branches chosen in the two runs are different. Since the principal specification cannot be computed, the best we can do is ask the programmer for the desired precondition p and postcondition q, and emit proof obligations for checking that (p,q) is valid for the conditional.

$$\begin{array}{lll}\mathsf{cond} & : & \Pi b{:}\overline{\Gamma}{\to}\mathsf{bool}.\\ & & \Pi e_1{:}\mathsf{STsec}\ \Gamma\ A\ (p_1,q_1).\, \Pi e_2{:}\mathsf{STsec}\ \Gamma\ A\ (p_2,q_2).\\ & & D_1\ (b,e_1,e_2) \to D_2\ (b,e_1,e_2) \to \mathsf{STsec}\ \Gamma\ A\ (p,q).\end{array}$$

Here D_1 captures the safety of the conditional, and D_2 the

Hoare-style correctness.

$$D_1(b, e_1, e_2) \;\widehat{=}\; \forall\gamma\ i.\, p\ \gamma\ i \to \\ \quad \mathsf{safe}\ (\mathsf{if}\ b\ \gamma\ \mathsf{then}\ e_1\ @_0\gamma\ \mathsf{else}\ e_2\ @_0\gamma)\ i$$

$$D_2(b, e_1, e_2) \;\widehat{=}\; \\ \forall\gamma\gamma\ ii.\, p\ \gamma\gamma.1\ ii.1 \to p\ \gamma\gamma.2\ ii.2 \to \\ \quad \mathsf{verify2}\ ii\ (\mathsf{if}\ b\ \gamma\gamma.1\ \mathsf{then} e_1\ @_0\ \gamma\gamma.1\ \mathsf{else}\ e_2\ @_0\ \gamma\gamma.1) \\ \qquad (\mathsf{if}\ b\ \gamma\gamma.2\ \mathsf{then} e_1\ @_0\ \gamma\gamma.2\ \mathsf{else}\ e_2\ @_0\ \gamma\gamma.2) \\ \qquad (\mathsf{fun}\ yy\ mm.\, q\ \gamma\gamma\ yy\ ii\ mm)$$

The definitions of D_1 and D_2 make use of the purely-functional conditional if to define when each of the branches is taken. In this paper, we conflate cond and if and use if for both. Note that, in contrast to other relational Hoare logics [9, 46], we do not restrict the reasoning to only the situation where the same branch of the conditional is taken in both runs; nor do we need side conditions, as in Amtoft et al. [1], that prohibit updates of low variables under a high guard (which would prevent verification of P_1 in Section II).

Example 8. The function write_salary from Example 2 is implemented with combinators (omitting annotations and proofs) as follows. Notice that the guard of the conditional is a term with a local context consisting of a:alice and x:string.

$$\begin{array}{l}\mathsf{write\_salary}\ s\ p\ \widehat{=} \\ \quad \mathsf{do}\ (\mathsf{read}\ @\ (\mathsf{fun}\ a.\, \mathsf{passwd}\ a); \\ \qquad \mathsf{if}\ (\mathsf{fun}\ x\ a.\, x = p)\ \mathsf{then} \\ \qquad\quad \mathsf{write}\ @\ (\mathsf{fun}\ x\ a.\, (\mathsf{salary}\ a, s)) \\ \qquad \mathsf{else}\ \mathsf{return}\ @\ (\mathsf{fun}\ x\ a.\, ()))\end{array}$$

A development similar to the one for cond can also be carried out for the combinator do and appears in the TR. We justify the soundness of our type system by building a denotational model for STsec types. This development is fully carried out as a shallow embedding in CiC, and we have formalized it in Coq. The model, briefly described in our TR, is based on predicate transformers. We also show in our Coq scripts that STsec $\Gamma\ A\ (p, q)$ is a complete partial order, thus, supporting a combinator fix for least fixed points of continuous functions between monadic types.

IV. Linked data structures

In this section we develop a small library for linked lists to illustrate RHTT's support for stateful abstract data types (ADTs), and their interaction with information flow. Working with ADTs essentially requires a number of higher-order features. For example, to support linked lists in a reasonable way, it has to be possible to: (1) describe the layout of the lists in the heap (is the list singly-linked, doubly-linked, etc?). This requires quantification in the assertion logic, definition of predicates by recursion, and inductive definitions of types; (2) abstract the definition of the heap layout from the specification of the ADT, so that the ADT clients can freely interchange implementations with different layouts (hence the need for abstract predicates); (3) parametrize the ADT with respect to the type of list elements (hence the need for type polymorphism in both programs and the assertion logic). All of these

$$\begin{array}{l}\mathsf{linked\_list} : \mathsf{type} \\ \mathsf{shape} : \mathsf{linked\_list} \to \mathsf{list}\ T \to \mathsf{heap} \to \mathsf{prop} \\ \mathsf{sshape}\ (pp : \mathsf{linked\_list}^2)\ (vvs : (\mathsf{list}\ T)^2)\ (ii : \mathsf{heap}^2) \;\widehat{=}\; \\ \quad \mathsf{shape}\ pp.1\ vvs.1\ ii.1 \wedge \mathsf{shape}\ pp.2\ vvs.2\ ii.2 \\ \mathsf{low\_links} : \mathsf{linked\_list}^2 \to \mathsf{heap}^2 \to \mathsf{prop}\end{array}$$

$$\begin{array}{l}\mathsf{new} : \mathsf{STsec}\ \mathsf{nil}\ \mathsf{linked\_list} \\ \quad (\mathsf{fun}\ i.\, \mathsf{True}, \\ \quad\ \mathsf{fun}\ pp\ ii\ mm.\, \exists jj.\, mm = jj \bullet\!\bullet\ ii \wedge \\ \quad\quad \mathsf{sshape}\ pp\ (\mathsf{nil}, \mathsf{nil})\ jj \wedge (ii.1 \cong ii.2 \to \mathsf{low\_links}\ pp\ jj))\end{array}$$

$$\begin{array}{l}\mathsf{insert} : \mathsf{STsec}\ [\mathsf{linked\_list}, T]\ \mathsf{unit} \\ \quad (\mathsf{fun}\ p\ v\ i.\, \exists h\ j\ vs.\, i = j \bullet h \wedge \mathsf{shape}\ p\ vs\ j, \\ \quad\ \mathsf{fun}\ pp\ vv\ yy\ ii\ mm.\, \forall hh\ jj\ vvs. \\ \quad\quad ii = jj \bullet\!\bullet\ hh \to \mathsf{sshape}\ pp\ vvs\ jj \to \\ \quad \exists jj'.\, mm = jj' \bullet\!\bullet\ hh \wedge \\ \quad\quad \mathsf{sshape}\ pp\ (vv.1 :: vvs.1, vv.2 :: vvs.2)\ jj' \wedge \\ \quad\quad (\mathsf{low\_links}\ pp\ jj \to hh.1 \cong hh.2 \to \mathsf{low\_links}\ pp\ jj'))\end{array}$$

$$\begin{array}{l}\mathsf{remove} : \mathsf{STsec}\ [\mathsf{linked\_list}]\ (\mathsf{option}\ T) \\ \quad (\mathsf{fun}\ p\ i.\, \exists h\ j\ vs.\, i = j \bullet h \wedge \mathsf{shape}\ p\ vs\ j, \\ \quad\ \mathsf{fun}\ pp\ yy\ ii\ mm.\, \forall hh\ jj\ vvs. \\ \quad\quad ii = jj \bullet\!\bullet\ hh \to \mathsf{sshape}\ pp\ vvs\ jj \to \\ \quad \exists jj'.\, mm = jj' \bullet\!\bullet\ hh \wedge \\ \qquad \mathsf{sshape}\ pp\ (\mathsf{tail}\ vvs.1, \mathsf{tail}\ vvs.2)\ jj' \wedge \\ \qquad yy = (\mathsf{if}\ vvs.1\ \mathsf{is}\ v1 :: \_\ \mathsf{then\ some}\ v1\ \mathsf{else\ none}, \\ \qquad\quad \mathsf{if}\ vvs.2\ \mathsf{is}\ v2 :: \_\ \mathsf{then\ some}\ v2\ \mathsf{else\ none}) \wedge \\ \qquad (\mathsf{low\_links}\ pp\ jj \to \mathsf{low\_links}\ pp\ jj'))\end{array}$$

Fig. 5. ListSig: signature for linked lists (excerpts).

features are present in RHTT, and used in the Figures 5 and 6, which show one possible interface, ListSig, and a module, List, implementing ListSig. The interface exports methods that create a new empty list, insert an element to the head of a list, and remove the head element, should one exist.

Both ListSig and List are parametrized in the type of list elements T. The interface declares the abstract predicate shape p vs i, capturing that the heap i stores a valid *singly-linked list* whose content is the mathematical (i.e., purely-functional) sequence vs of type list T. The pointer p stores the address of the list head, so that adding new elements at the head can be done by updating p. The linkage between the elements is described by the predicate lseq x vs which recurses over the contents vs and states that each node, starting from the head x, contains a single pointer z to the next node in the linked list. The interface hides the details of shape, however, and can thus be ascribed to other implementations of shape, such as ones describing doubly-linked lists.

The interface in Figure 5 contains one more abstract predicate low_links pp ii, which we use in combination with sshape pp vvs ii, to describe that the linkage of the list stored in the heap ii is of low security, no matter the security levels of the contents vvs. The latter may be heterogeneous; that is, some elements of vvs may be of low security, while others are high. Similar to lseq, low_links recurses over the linked lists, declaring that each node is stored at a low address; that

is, an address which is equal in the two heap instances, $ii.1$ and $ii.2$. (The formal definition of low_links is elided here but appears in file llist3.v of the Coq scripts.)

The types of the methods declare how the methods modify the contents of the list as well as the linkage. For example, the shape predicate in the preconditions of insert and remove requires that the initial heaps of these methods store valid linked lists. The sshape predicate in the postconditions guarantees that valid linked lists are produced at the end. The postconditions additionally contain conjuncts describing that the methods preserve the low security level of the linkage. For example, new will allocate a fresh pointer p, and initialize it with null. If the deterministic allocator is used to obtain p, then p will be low *only if the allocator is executed in low-equivalent initial heaps*. Thus, in order to get low_links $pp\ jj$, we require an antecedent $ii.1 \cong ii.2$. Similarly, insert specifies that low_links $pp\ jj \to hh.1 \cong hh.2 \to$ low_links $pp\ jj'$. In other words, if the initial lists have low linkage, and the remainders of the global heaps are low equivalent, then we can allocate a list node with low linkage. This is so, because the initial heaps must be low equivalent under the described conditions.

The implementations of the methods are standard (Figure 6), but due to the combinator syntax, we describe them in prose. new returns a fresh pointer, initialized with null. This will be the pointer p in the shape predicate. insert takes the pointer p to the list, and a value v to insert. It reads the address of the first element (bound to variable hd), and allocates a node x whose contents field is v and next pointer field is hd. Finally, x is written to p. remove reads the address of the first element of the list p into the variable hd. If hd is null, then the list is empty, and the function terminates. Otherwise, it reads the contents of the node at hd, binding it to the variable v. p is made to point to next v, before v is deallocated.

To establish that the implementation satisfies the signature, we need a number of helper lemmas about lseq and linked_list, which are kept local to the module. For example, for lseq, we need properties that describe the behavior of lseq $x\ vs\ i$, in case x is null (then the whole list is empty), and non-null (then x points to the head). For low_links, we show that if two heaps store lists with low linkage *and equal contents*, then the heaps themselves are equal.

Example 9. The program P_5 in Figure 7 illustrates heterogeneous lists, i.e., lists that contain both high and low values. It takes a high boolean argument b, creates a new linked list, and inserts 0 (a constant, hence low) at the head. Then, depending on b, it inserts either 1 or 2, resulting in a heterogeneous list with a high first element and low second element. This is described in the postcondition by conditionals over the values of b in the two different runs ($bb.1$ and $bb.2$). Irrespective of the contents, the ending *linkage* is low, assuming we started with low-equivalent input heaps.

Example 10. The program P_6 in Figure 7 is similar to P_5, but branches on b to decide whether to remove the head element. Therefore, the length of the resulting list may differ in the

```
linked_list ≙ ptr
node : type ≙ node of (T × ptr)
elem (e : node) ≙ e.1
next (e : node) ≙ e.2

lseq (x : ptr) (vs : list T) : heap → prop ≙
  if vs is v::vt then
    fun i. ∃z:ptr j:heap.
                i = x ↦ node v z • j ∧ lseq z vt j
  else fun i. x = null ∧ i = empty_heap

shape (p : linked_list) (vs : list T) (i : heap) ≙
  ∃x:ptr. j:heap. i = p ↦ x • j ∧ lseq x vs j

new ≙ do (lalloc @0 null)
insert ≙
  do (read @ (fun p v. p);
      lalloc @ (fun hd p v. node v hd);
      write @ (fun x hd p v. (p, x)))
remove ≙
  do (read @ (fun p. p);
      if (fun hd p. hd = null) then
        return @ (fun hd p. none)
      else
        read @ (fun hd p. hd);
        write @ (fun v hd p. (p, next v));
        dealloc @ (fun _ v hd p. hd);
        return @ (fun _ _ v hd p. some (elem v)))
```

Fig. 6. Module List: implementation of singly-linked lists (excerpts).

```
P5 : STsec [bool] linked_list
     (fun b i. True,
      fun bb yy ii mm. ∃jj. mm = jj •• ii ∧
        sshape yy ([if bb.1 then 1 else 2, 0],
                   [if bb.2 then 1 else 2, 0]) jj ∧
        (ii.1 ≅ ii.2 → low_links yy jj)) ≙
   do (new @ (fun b. ());
       insert @ (fun p b. (p, 0));
       if (fun _ p b. b) then insert @ (fun _ p b. (p, 1))
       else insert @ (fun _ p b. (p, 2)) fi;
       return @ (fun _ _ p b. p))

P6 : STsec [bool] linked_list
     (fun b i. True,
      fun bb yy ii mm. ∃jj. mm = ii •• jj ∧
        sshape yy (if bb.1 then [0] else [1, 0],
                   if bb.2 then [0] else [1, 0]) jj) ≙
   do (new @ (fun b. ());
       insert @ (fun p b. (p, 0));
       insert @ (fun _ p b. (p, 1));
       if (fun _ _ p b. b) then
         remove @ (fun _ _ p b. p);
         return @ (fun _ _ _ p b. p)
       else return @ (fun _ _ p b. p))
```

Fig. 7. Programs with heterogeneous lists.

two runs, depending on b. We can specify it with the type shown in the Figure. Notice however that we cannot prove that low_links yy jj holds at the end of P_6. The length of the produced list is dependent on b, which implies that the resulting linkage may differ in two runs of P_6, and hence cannot be low itself.

Our Coq scripts implement other interfaces for linked list, where the sshape predicates are parametrized by the linkage as well. This exposes more implementation details (e.g., that the list is singly-linked), but allows more precise reasoning about linkage. For example, we may prove that executing one more conditional over b, with a call to remove in the else branch, will restore the low linkage.

We are not aware of any other system in literature that can reason statically about heterogeneous structures. In the dynamic setting, a recent example is the work of Russo et al. [37], which tracks information-flow through DOM trees, with the goal of preventing information leakage via node deletion or navigation. The system works by assigning to each node two security labels: one for the contents, and another for the existence of the node. These annotations are very specific to DOM trees, however, and it seems that the label assignment would have to be designed differently for different data structures and enforced properties. Thus, if one wants to work with a number of structures simultaneously, one must employ a very rich specification logic, just as we do.

We close with an example which combines linked lists with the Alice module from Section II.

Example 11. In Example 5, Alice ratifies Bob's tax function, which may keep local state, as long as Bob can prove that his final state does not steal Alice's salary. Here we instantiate Bob's local state to a linked list, which dynamically grows as various instances of Alice execute Bob's program, but the values stored in the linked list are always independent of any instance's salary and the list's linkage is always low. Observe from the specifications of new and insert that Bob's newly allocated nodes will be low only if he can generate them in low-equivalent heaps. To express this low equivalence the specification of epost used in ratify's specification must change as emphasized below.

$$
\begin{array}{l}
\mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm\ \widehat{=} \\
\quad \forall ss\ pp\ kk\ tt\ hh. \\
\qquad ii = jj \bullet\!\bullet\ tt \bullet\!\bullet\ hh \to \mathsf{sshape}\ aa\ ss\ pp\ jj \to \\
\qquad\quad \mathsf{bbshape}\ bb\ kk\ tt \to \\
\qquad \exists jj'\ tt'.\ mm = jj' \bullet\!\bullet\ tt' \bullet\!\bullet\ hh \wedge \mathsf{sshape}\ aa\ ss\ pp\ jj' \wedge \\
\qquad\quad \mathsf{bbshape}\ bb\ (\mathsf{bcmp}\ kk.1, \mathsf{bcmp}\ kk.2)\ tt' \wedge \\
\qquad\quad \boxed{jj.1 \bullet hh.1 \cong jj.2 \bullet hh.2} \to \\
\qquad\quad tt.1 = tt.2 \to tt'.1 = tt'.2)
\end{array}
$$

Bob can now be granted access to Alice's salary and can keep the count in a linked list. For example, the implementation below defines Bob's local state as a linked list which counts the number of times Bob's program has been called by linking in new nodes into Bob's list. The nodes are filled with 1 for simplicity, but arbitrary values would do, including dynamically computed ones, as long as they are independent of Alice's salary.

$$
\begin{array}{l}
\mathsf{bob}\ \widehat{=}\ \mathsf{linked\_list} \\
\mathsf{G}\ \widehat{=}\ \mathsf{list\ nat} \\
\mathsf{bbshape}\ (bb : \mathsf{bob}^2)\ (kk : G^2)\ (ii : \mathsf{heap}^2)\ \widehat{=} \\
\quad \mathsf{List.sshape}\ bb\ kk\ ii\ \wedge bb.1 = bb.2 \\
\mathsf{bcmp} : G \to G\ \widehat{=}\ \mathsf{fun}\ k.\, 1 :: k
\end{array}
$$

Bob's program, which reads Alice's salary, allocates a new node in his list, and then returns the computed tax for the salary, can then be created and ratified as follows.

$$
\begin{array}{l}
\mathsf{linked\_client}\ \widehat{=} \\
\quad \mathsf{ratify}\ (\mathsf{do}\ (\mathsf{read\_salary}\ @\ (\mathsf{fun}\ a\ b.\, a); \\
\qquad\qquad \mathsf{insert}\ @\ (\mathsf{fun}\ x\ a\ b.\, (b, 1)); \\
\qquad\qquad \mathsf{return}\ @\ (\mathsf{fun}\ \_\ x\ a\ b.\, x * 24\%)))
\end{array}
$$

V. Discussion

Completeness: We have informally justified the completeness of our system through several examples, covering a wide range of security relevant policies including access control, information flow, declassification, erasure, and their combinations. Unfortunately we are not aware of a clear and exhaustive formal definition of what constitutes, say, an erasure, or access-control policy, or a combination thereof. Therefore, we do not know how to state a formal completeness result.

If we focus on Cook completeness for RHTT, then, as we have argued in Section III, our specifications for all of the primitive effectful combinators compute weakest preconditions and strongest postconditions using the specifications of the components. The exception are the conditionals, for which this cannot be done when the boolean guard is high. However, RHTT is still capable of checking high conditionals against programmer-supplied postconditions. The lack of Cook completeness therefore results in an increase in code annotations that the programmer has to supply, but does not decrease the reasoning power of the logic.

Noninterference for finite security lattices: The standard notion of noninterference when locations are classified into elements of a security lattice is compatible with RHTT and can be expressed in postconditions of RHTT programs. If each variable of a program is classified at some level of a finite lattice L, then the program is noninterfering if for each $\ell \in L$, the following holds: $(xx_1.1 = xx_1.2 \wedge \ldots \wedge xx_n.1 = xx_n.2) \to (yy_1.1 = yy_1.2 \wedge \ldots \wedge yy_m.1 = yy_m.2)$ where $xx_1, \ldots, xx_n$ are pairs of values of variables at or below security level ℓ in the two initial heaps and $yy_1, \ldots, yy_n$ are pairs of values of variables at security level ℓ in the two final heaps. If the lattice L is finite, then the noninterference property can be represented in the postcondition of a program as the *conjunction* of such requirements for each level $\ell \in L$.

This representation can be combinatorially explosive, if one uses one conjunct for each variable. But, in our higher-order assertion logic we can introduce predicates that abstract over a number of such conjuncts at once, and hence avoid the

explosion. For example, if the variables in question all stand for the contents of some linked list, we can define a predicate that conjoins equations of the above form for each value reachable from the head of the list. Note that reachability can be expressed in higher-order but not in first-order logic. We leave a more detailed exploration of the above representation to future work.

On proof sizes: We have found that the size of interactive proofs is not too overwhelming in general. However, the amount of interaction varies with programs. Programs with complex loop invariants usually require large proofs, whereas simpler programs can be verified in just a number of lines proportional to the size of the program.

Programs that branch on high boolean guards invariably have larger proofs than programs that branch only on low: the latter always choose the same branches of conditionals in two runs, so the verification of the two runs proceeds in lockstep. High-branching programs can choose branches asymmetrically, thus doubling the number of proof obligations. In addition, when branches are chosen asymmetrically, the proofs usually require some mathematical insight from the programmer (for example, algebraic simplification of expressions) in order to argue that the high secret has not been leaked. The latter, however, seems unavoidable, and inherent to the nature of programs branching on high guards.

To substantiate, consider the programs from Examples 2 and 3, our first examples that do not branch on high. We have the following statistics given as the pair (*code+spec size, proof size*). For new, we have (7, 5); for read_salary (7, 4); for write_salary (18, 15) and for declassify (11, 5). The above proofs share common definitions and lemmas which are altogether 10 lines long.

The program P_2 in Example 1, which contains nested conditionals and branching on high, is implemented using 36 lines of code, most of which are inlined user-supplied annotations. The corresponding proof is 44 lines long.

We have also implemented examples that iterate over linked data structures (not presented in the paper, but available in the accompanying Coq scripts). In a program for in-place list reversal, in which the linkage of the list is high, the code and annotations together take 43 lines. The proof is 94 lines long, because there is a high conditional branching on a null-pointer check.

VI. Related work

Banerjee et al. [7] specify expressive declassification policies using Hoare style specifications (termed flowspecs); preconditions thereof are conjunctions of ordinary state conditions based on first-order logic (for specifying conditions *when* declassification can happen) as well as relational predicates (that specify *what* is being declassified) [39]. We extend the ideas in [7] and consider a higher-order imperative language and also a policy specification language based on higher-order logic, where Hoare-style specifications may appear in negative (i.e., argument) positions, which is required for conditional access and erasure policies.

A recent line of work [23, 36] uses type-theoretic technology, namely Haskell, to specify and enforce information-flow properties in a non-dependently-typed setting. While Haskell already provides the important higher-order constructs for abstraction and modularity, non-dependent types by definition cannot specify behaviors that are dependent on some condition such as authorization, conformance to a policy, or local state. Thus, we do not think they can be used directly to enforce involved security policies such as the ones considered in this paper.

Some other recent languages, with somewhat similar high-level goals to ours, and which use some form of dependent types are Fine [41], Fable [42], FX [10], Aglet [27], F7 [11] and Aura [20]. They all support some, but not all features that we provide in RHTT.

In Swamy et al.'s purely functional programming language Fine [41], access and information flow policies can mention attributes like high and low, that statically label data. The type system enforces these policies by tracking flows of attributes. Unlike RHTT, Fine's type system does not track changes to the state (heap), so the effect of state in policies must be simulated through ghost variables, whose (static) updates are governed by specifications of primitive functions. A token passing mechanism based on affine kinds ensures that at most one static state is valid at each program point, but it makes programming in Fine inconvenient. Fine includes a simple module system which allows a programmer to hide type definitions, but does not allow abstraction over predicates as RHTT does. In an earlier language, Fable [42], data can be statically labelled with attributes that can be used to enforce both access control and information flow policies. However, Fable's type system lacks the affine kinds of Fine as well as Fine's logic-based sublanguage for policies and, therefore, cannot be used to reason about state-dependent policies.

The language FX [10] succeeds Fine with the purpose of verifying stateful programs that permit object allocation, mutation and deallocation. The type system of FX admits computation (Hoare) types and caters to the verification of safety properties of FX programs by translating into Fine programs and typechecking the latter. The translation is a simulation under strong bisimilarity, rather than the stronger property that well-typed FX programs are translated into well-typed Fine programs. The verification of security policies, particularly, of non-safety properties such as noninterference, is not the overarching goal of FX's type system, although a lattice of labels can be encoded and used to prove, e.g., an integrity property that untrusted data does not get consumed at trusted sinks. A proof of noninterference is not supplied; as in most label-based security type systems, such a proof cannot be carried out in FX's (or Fine's) type system directly (in contrast to our work) but rather must be established as a metatheorem of the type system by reasoning about two runs of programs. As regards reasoning about stateful higher-order programs, the formalization is left for future work and we expect that it will elucidate how the type system reasons about (security properties of) unbounded dynamic data structures e.g., linked

lists, trees with back pointers etc., that contain significant use of aliased mutable objects. In particular, because FX proposes to reason about aliasing using a library of permissions the above formalization might be delicate.

Morgenstern and Licata have recently proposed a type system called Aglet [27], for enforcement of state-dependent access control policies. Aglet is an extension of Agda [33] with a computation monad similar to our STsec types. However, Aglet's computation monad lacks semantics and, consequently, the soundness of its inference rules has to be taken on faith (in contrast, the RHTT model is formalized in Coq). Moreover, the pre- and post-conditions of Aglet's computation monad can only mention a restricted form of state, namely, a mutable list of authorization-relevant credentials, which can be used to discharge authorization obligations at various program points. Due to this restriction, Aglet cannot be used to reason about data structures written in Agda. Also, Aglet's postconditions do not consider simultaneous runs of programs. As a result of these limitations, Aglet cannot be used to represent many of our examples. On the other hand, we believe that examples from the paper on Aglet can be expressed in RHTT easily.

Borgström et al. [11] reason about access control behavior of programs in an extension of F7 that has a state monad with pre- and post-conditions. Although the state monads in their work and ours are technically similar, that work differs from ours in two significant ways. First, the goals are different: whereas we consider enforcement of information flow properties and declassification in addition to access control properties, Borgström et al. consider access control and show how the state monad can be used to enforce different flavors of it, *viz.* role-based, stack-based, and history-based. Second, in common with other work based in F7, a priori evidence for discharging verification conditions in Borgström et al's work is programmer specified assumptions that are not necessarily semantically grounded, and verification is correct only to the extent that these assumptions are correct. In contrast to their axiomatic approach, we verify the soundness of our type theory on a semantic model. Nonetheless, due to the common state-monad based approach, and RHTT's more general type system, we believe that Borgström et al's work can be encoded in RHTT without much change. As a first step in this direction, our Coq scripts contain an example that shows how RHTT supports reasoning about principals and roles.

The languages Aura [20], PCML_5 [5], and PCAL [13], based on the proof-as-authorization paradigm [2], enforce logic-represented access policies by statically ensuring that each call to a protected interface is accompanied by proper authorization. Although work in the context of Aura shows that noninterference can be encoded [21], Aura currently does not handle state in the form that we consider in this paper. However, it is conceivable that mutable state can be added to Aura along the lines of the STsec monad.

The Paralocks language [12] also allows logic-based access control policies that are enforced statically in the type system. Information flow policies can be encoded as a specific mode of access control as, for instance, is demonstrated through an encoding of Myers' and Liskov's Decentralized Label Model. Like Fine, Paralocks includes two kinds of state, of which, one, called *locks*, is tracked through the type system, while the other is not. Locks are boolean variables that can be used to encode a wide range of policies. The semantics of Paralocks is trace-based and, like gradual release [3], uses a knowledge-based definition of information leaks. A meta-theorem guarantees that access policies of a well-typed program are respected at all program points during the program's execution.

Finally, RHTT extends the work on Hoare Type Theory (HTT) [31] and Ynot [30] with the ability to reason relationally about security. HTT and Ynot implement via dependent types a higher-order variant of Hoare logic for single program runs. Thus, they cannot enforce relational properties such as the various information flow policies, which are specified via two program runs. A more technical discussion of the differences between these systems and RHTT appears in the TR.

VII. Conclusion

We have presented RHTT, a system implemented in Coq that is targeted for full interactive verification of state-based access control and information flow policies via dependent types. Examples of such security policies include declassification, information erasure and state-based access control and information flow. We have presented typing rules for the stateful fragment of RHTT and implemented a semantic model that provides a denotation to every well-typed RHTT program. We have also developed a logic for discharging verification conditions that arise in the verification process.

Currently, RHTT does not support reasoning about trace-based, temporal properties. For example, while it is intuitively clear that our specification of functions grant, revoke, read_salary (Example 4) indeed encodes a temporal discipline on the usage of read_salary (e.g., "no reads occur unless a grant has occurred and no revoke has occurred after the grant") this cannot be formally proved in our logic itself. We note that very little is known on how enforcement of trace-based properties, in security or other areas such as concurrency, interacts with type theoretic constructions such as higher-order functions, abstract types or modules. We intend to investigate this in the future, in the context of reactive, non-deterministic and concurrent higher-order languages.

Acknowledgements: Thanks to Gilles Barthe, Alexey Gotsman, Boris Köpf, Jamie Morgenstern, Greg Morrisett and David Naumann for their comments on earlier drafts. We thank Trent Jaeger for his advice on improving the presentation of the paper.

This research was partially supported by Madrid Regional Government Project S2009TIC-1465 Prometidos; MICINN Projects TIN2009-14599-C03-02 Desafios and TIN2010-20639 Paran10; EU Projects GA-231620 Hats and NoE-256980 Nessos; Ramon y Cajal grant RYC-2010-07433; AMAROUT grant PCOFUND-GA-2008-229599; U.S. NSF Trustworthy Computing grant 1018061 "Compositional End-to-End Security for Systems", and the U.S. AFOSR MURI "Collaborative Policies and Assured Information Sharing".

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *POPL*, 2006.

[2] A. W. Appel and E. W. Felten, "Proof-carrying authentication," in *ACM CCS*, 1999.

[3] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE Symp. Security and Privacy*, 2007.

[4] A. Askarov and A. Myers, "A semantic framework for declassification and endorsement," in *ESOP*, 2010.

[5] K. Avijit, A. Datta, and R. Harper, "Distributed programming with distributed authorization," in *TLDI*, 2010.

[6] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *JFP*, vol. 15, no. 2, pp. 131–177, 2005.

[7] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and their modular static enforcement," in *IEEE Symp. Security and Privacy*, 2008.

[8] D. Bell and L. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Tech. Rep. MTR-2547, 1973.

[9] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *POPL*, 2004.

[10] J. Borgstrom, J. Chen, and N. Swamy, "Verifying stateful programs with substructural state and Hoare types," in *PLPV*, 2011.

[11] J. Borgström, A. D. Gordon, and R. Pucella, "Roles, stacks, histories: A triple for Hoare," *JFP*, forthcoming.

[12] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *POPL*, 2010.

[13] A. Chaudhuri and D. Garg, "PCAL: Language support for proof-carrying authorization systems," in *ESORICS*, 2009.

[14] S. Chong and A. C. Myers, "Security policies for downgrading," in *ACM CCS*, 2004.

[15] ——, "Language-based information erasure," in *CSFW*, 2005.

[16] ——, "End-to-end enforcement of erasure and declassification," in *CSF*, 2008.

[17] D. Denning, "A lattice model of secure information flow," *CACM*, vol. 19, no. 5, pp. 236–242, 1976.

[18] J. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. Security and Privacy*, 1982, pp. 11–20.

[19] R. Harper and M. Lillibridge, "A type-theoretic approach to higher-order modules with sharing," in *POPL*, 1994.

[20] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic, "AURA: A programming language for authorization and audit," in *ICFP*, 2008.

[21] L. Jia and S. Zdancewic, "Encoding information flow in Aura," in *PLAS*, 2009.

[22] X. Leroy, "Manifest types, modules, and separate compilation," in *POPL*, 1994.

[23] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Comput. Sci.*, vol. 411, no. 19, pp. 1974–1994, 2010.

[24] P. Martin-Löf, *Intuitionistic Type Theory*. Bibliopolis, 1984.

[25] The Coq development team, *The Coq proof assistant reference manual*, LogiCal project, INRIA, 2009, version 8.2. [Online]. Available: http://coq.inria.fr

[26] J. C. Mitchell and G. D. Plotkin, "Abstract types have existential type," *ACM Trans. Prog. Lang. Syst.*, vol. 10, no. 3, pp. 470–502, 1988.

[27] J. Morgenstern and D. Licata, "Security-typed programming within dependently-typed programming," in *ICFP*, 2010.

[28] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL*, 1999.

[29] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies via dependent types," IMDEA Software Iinstitute, Tech. Rep. IMDEA-SW-TR-2010.0, 2010.

[30] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal, "Ynot: Dependent types for imperative programs," in *ICFP*, 2008.

[31] A. Nanevski, J. G. Morrisett, and L. Birkedal, "Hoare type theory, polymorphism and separation," *JFP*, vol. 18, no. 5-6, pp. 865–911, 2008.

[32] A. Nanevski, V. Vafeiadis, and J. Berdine, "Structuring the verification of heap-manipulating programs," in *POPL*, 2010.

[33] U. Norell, "Towards a practical programming language based on dependent type theory," Ph.D. dissertation, Chalmers University of Technology, 2007.

[34] S. L. Peyton Jones and P. Wadler, "Imperative functional programming," in *POPL*, 1993.

[35] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," in *LICS*, 2002.

[36] A. Russo, K. Claessen, and J. Hughes, "A library for lightweight information-flow security in Haskell," in *Haskell Symposium*, 2008.

[37] A. Russo, A. Sabelfeld, and A. Chudnov, "Tracking information flow in dynamic tree structures," in *ESORICS*, 2009.

[38] A. Sabelfeld and D. Sands, "A PER model of secure information flow in sequential programs," in *ESOP*, 1999.

[39] ——, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[40] V. Simonet, "Fine-grained information flow analysis for a λ-calculus with sum types," in *CSFW*, 2002.

[41] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in Fine," in *ESOP*, 2010.

[42] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *IEEE Symp. Security and Privacy*, 2008, full version: Technical report CS-TR-4895, Univ. Maryland.

[43] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *CSFW*, 2006.

[44] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *SAS*, 2005.

[45] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[46] H. Yang, "Relational separation logic," *Theoretical Comput. Sci.*, vol. 375, pp. 308–334, 2007.

[47] H. Yang and P. W. O'Hearn, "A semantic basis for local reasoning," in *FoSSaCS*, 2002.