Dependent Type Theory for Verification of Information Flow and Access Control Policies

ALEKSANDAR NANEVSKI, IMDEA Software Institute
and ANINDYA BANERJEE, IMDEA Software Institute
and DEEPAK GARG, Max Planck Institute for Software Systems

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.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory — Semantics; D.4.6 [**Security and Protection**]: Access Controls, Information Flow Controls, Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions.

General Terms: Security, Verification, Languages

Additional Key Words and Phrases: Information Flow, Access Control, Type Theory

1. 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. [Askarov and Myers 2010; Banerjee et al. 2008; Broberg and Sands 2010]), 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, data 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

Expanded and revised version of a paper originally appearing in *IEEE Symposium on Security and Privacy*, 2011.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

three problems can be addressed using *standard* linguistic features from dependent type theory [Martin-Löf 1984]: (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 [Chong and Myers 2004; Sabelfeld and Sands 2009], information erasure [Chong and Myers 2005, 2008], state-dependent access control [Borgström et al. 2011; Broberg and Sands 2010] and state-dependent information flow policies [Banerjee et al. 2008], 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 [Myers 1999; Volpano et al. 1996]. 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) [The Coq development team 2009, Chapter 4], as implemented in the Coq proof assistant. To provide for (c), RHTT introduces a novel type constructor STsec, which classifies side-effectful computations similar to Haskell monads [Peyton Jones and Wadler 1993], 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 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 [Amtoft et al. 2006]. The relational formulation directly captures in the types the notion of *non-interference* [Goguen and Meseguer 1982], 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. Following standard literature on information flow [Goguen and Meseguer 1982], we define a datum to be ***public*** or ***low*** if *its value is known to be denotationally equal in two runs of a program*, and ***confidential*** or ***high*** otherwise. 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 [Mitchell and Plotkin 1988]. 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 [Banerjee et al. 2008], where the condition might be stateful and involve, e.g., authentication.

In information erasure policies [Chong and Myers 2005, 2008] 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 [Swamy et al. 2006].

Our fourth contribution is the development of a logic for relational reasoning about RHTT programs. Inference rules of the logic have been verified sound against a semantic model (Sections 3 and 4), and are formally implemented as lemmas in RHTT. The distinguishing characteristic of our work is that we can reason relationally about structurally dissimilar programs. This is in contrast to previous relational logics [Benton 2004; Yang 2007], which support reasoning only about programs with similar control flow, and provide inference rules for conditionals and loops with only low boolean guards. The latter is known to be limiting in practice and prior work proposes workarounds through dynamic checks [Austin and Flanagan 2010] and testing [Birgisson et al. 2012], but not statically.

Our development of RHTT overcomes a number of technical challenges.

— The first challenge is that for relational reasoning to be applicable at all, the type system must give special status to instantiations, in two runs, of a program e with high values. The special status is needed so that the *same* postcondition of e can relate e's instantiations with *different* high values in the two runs. Our solution is to introduce new typing and programming primitives for abstraction and instantiation wrt. a number of variables, simultaneously (Section 2): technically, this is done by defining a new function space for functions that depend on high arguments.
— The second challenge concerns the semantic treatment of allocation and deallocation, pertaining to dynamic data structures. Existing techniques [Amtoft et al. 2006; Banerjee and Naumann 2005] 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 3).

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.[1] In the process we also addressed the following technical difficulties for the purposes of scalability of reasoning: (a) Separation of reasoning about memory safety from the conformance to the security policy. (b) Development of a library for reasoning about complete partial orders in Coq, for treating general recursive functions. The interested reader is invited to look at our Coq proofs which are available at `http://software.imdea.org/~aleks/rhtt/`. An online appendix containing all proof rules is available at `http://software.imdea.org/~aleks/rhtt/proof_rules.pdf`

[1]The semantic model (Section 3.3) and the logic for discharging verification conditions (Section 4) are additional material not present in the conference version of the paper [Nanevski et al. 2011]. A worked out example (Section 4.4) shows how the logic is used.

Attack model. As demonstrated by our examples, many different attack models can be encoded using RHTT preconditions and postconditions. In particular, we can establish standard noninterference for a program, by proving in RHTT's logic that pointers stipulated as public contain denotationally equal values in the output heap of the program if they contain denotationally equal values in the input heap. We can also represent, and prove absence of attacks in, stronger attack models. For example, we can represent an attack model where the adversary can observe the shape of the heap by relying on the fact that pointers are simply Coq's natural numbers in our denotational model.

2. 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 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 [The Coq development team 2009], 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.

2.1. RHTT basics: types, specifications, opaque sealing

To begin with, our types must be able to express at least non-interference: 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 non-interference in terms of f alone [Amtoft et al. 2006; Benton 2004], without recourse to outside concepts such as security lattices [Bell and LaPadula 1973; Denning 1976]. 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., a function of type $\mathsf{heap} \to \mathsf{prop}$. 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 [Reynolds 2002]: 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 2.5).

The *postcondition* q relates the output values, input heaps and output heaps of any *two terminating* executions of e. Thus q has the type $A^2 \to \mathsf{heap}^2 \to \mathsf{heap}^2 \to \mathsf{prop}$. 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* [Sabelfeld and Sands 1999]. 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* [Harper and Lillibridge 1994; Leroy 1994]; 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 stress 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.

NOTATION: Examples in the sequel will often use the following notation. Variables of squared types, e.g, A^2, will often be written, e.g., aa. We also write a_1 and a_2 for the two projections of the pair aa.

2.2. Syntax, heaps, implicit flow

Consider the following program, P_1, adapted from Terauchi and Aiken [2005], and presented here in a Haskell-like notation. We use side-effecting primitives such as $\mathsf{write}\ x\ y$, which stores the value y into the location x; $\mathsf{read}\ x$, 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 $\mathsf{alloc}\ x$, which returns a fresh memory location initialized with x; and $\mathsf{dealloc}\ x$, 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 3. Further, we freely use all the constructors inherited from CiC and Coq, such as for example, functions (fun), and dependent function type constructor (Π).

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

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.

$$\begin{array}{l} U \;\widehat{=}\; \Pi x\ y\ z\ lo\ hi : \mathsf{ptr}.\,\mathsf{STsec}\ \mathsf{unit} \\ \quad (\mathsf{fun}\ i.\ \exists u\ v\ w\ c : \mathsf{nat}, b : \mathsf{bool}, j : \mathsf{heap}. \\ \qquad i = x \mapsto u \bullet y \mapsto v \bullet z \mapsto w \bullet hi \mapsto b \bullet lo \mapsto c \bullet j, \\ \quad \mathsf{fun}\ rr\ ii\ mm. \\ \qquad (i_1\ lo = i_2\ lo) \to (i_1\ y = i_2\ y) \to \\ \qquad (m_1\ lo = m_2\ lo\ \wedge\ m_1\ y = m_2\ y)) \end{array}$$

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. (Notice our use of the notation mentioned earlier: rr, ii, mm are variables of squared types. Also i_1, i_2 refer to the first and second components of ii and m_1, m_2 refer to the first and second components of mm.)

Other types for P_1 are possible too. For example, we may specify that only the last digit of y is low, by replacing $i_1\ y$ with $(i_1\ y)\ \mathsf{mod}\ 10$ in the postcondition, and similarly with i_2, m_1 and m_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 [Amtoft et al. 2006]) 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 be 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 Section 4. This proof obligation not only may be about security but also may concern full functional correctness.

2.3. Opaque sealing

The ascription of STsec types in RHTT is opaque, as mentioned earlier in this section. Clients of a program can use knowledge about low values in the output of the program only if this knowledge is exposed in the program's postcondition, even if the program's execution actually makes more values low. 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, even though it actually is.

2.4. 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.

$$\begin{array}{l}\mathsf{STsec}\ [\mathsf{ptr},\mathsf{ptr},\mathsf{ptr},\mathsf{ptr},\mathsf{ptr}]\ \mathsf{unit}\\ \quad(\mathsf{fun}\ x\ y\ z\ lo\ hi : \mathsf{ptr}, i : \mathsf{heap}.\\ \qquad \exists u\ v\ w\ c : \mathsf{nat}, b : \mathsf{bool}, j : \mathsf{heap}.\\ \qquad\quad i = x \mapsto u \bullet y \mapsto v \bullet z \mapsto w \bullet hi \mapsto b \bullet lo \mapsto c \bullet j,\\ \quad \mathsf{fun}\ xx\ yy\ zz\ llo\ hh : \mathsf{ptr}^2, rr : \mathsf{unit}^2, ii\ mm : \mathsf{heap}^2.\\ \quad (i_1\ llo_1 = i_2\ llo_2) \to (i_1\ y_1 = i_2\ y_2) \to\\ \quad (m_1\ llo_1 = m_2\ llo_2) \wedge (m_1\ y_1 = m_2\ y_2))\end{array}$$

The type of the precondition (and similarly for postconditions) now changes to $\mathsf{ptr}^5 \to \mathsf{heap} \to \mathsf{prop}$, 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* 2.1** Ordinary function arguments, corresponding to the $\to$ 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 $f : \Pi x : A.\, \mathsf{STsec}\ \Gamma\ B\ (p\ x, q\ x)$ can be transformed into

$$\begin{array}{l}\mathsf{do}\ (\mathsf{fun}\ x\ \gamma_1 \ldots \gamma_n.\, f\ x\ \gamma_1 \ldots \gamma_n) :\\ \quad \mathsf{STsec}\ (A{::}\Gamma)\ B\\ \qquad (\mathsf{fun}\ x\ \gamma_1 \ldots \gamma_n\ i.\ p\ x\ \gamma_1 \ldots \gamma_n\ i,\\ \qquad\ \mathsf{fun}\ xx\ \gamma\gamma_1 \ldots \gamma\gamma_n\ yy\ ii\ mm.\\ \qquad\qquad x_1 = x_2 \to q\ xx\ \gamma\gamma_1 \ldots\ yy\ ii\ mm)\end{array}$$

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 $x_1 = x_2$. Thus we are asserting that the output depends on the low argument x. Note that the transformation says that the Π type is equivalent to the STsec type —the proof appears in file secmod.v of the Coq scripts.

In summary, 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 2.2 (Nested conditionals) The following program is adapted from Simonet [2002]. 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: $y \ \widehat{=}\ (a = c)\,||\,(b = c)\,||\,u\,||\,v$. (Here $||$ is boolean disjunction.) Clearly y does not

α : Type
sshape : $\alpha^2 \to \mathsf{nat}^2 \to \mathsf{string}^2 \to \mathsf{heap}^2 \to \mathsf{prop}$
shape $\widehat{=}$ fun $a\ s\ p\ h.$ sshape $(a,a)\ (s,s)\ (p,p)\ (h,h)$
srefl : $\forall aa\ ss\ pp\ ii\ .$ sshape $aa\ ss\ pp\ ii \to$
shape $a_1\ s_1\ p_1\ i_1 \wedge$ shape $a_2\ s_2\ p_2\ i_2$
new : nat $\to$ string $\to$ STsec nil α
(fun i. True,
fun $aa\ ii\ mm.\ \exists ss\ pp\ hh\ .\ mm = ii \bullet\bullet hh \wedge$ sshape $aa\ ss\ pp\ hh$)
read_salary : STsec $[\alpha]$ nat
(fun $a\ i.\exists s\ p\ j\ h.\ i = j \bullet h \wedge$ shape $a\ s\ p\ j$,
fun $aa\ yy\ ii\ mm.\forall ss\ pp\ jj\ hh.$
$ii = jj \bullet\bullet hh \to$ sshape $aa\ ss\ pp\ jj \to$
$mm = ii \wedge yy = ss$)
write_salary : nat $\to$ string $\to$ STsec $[\alpha]$ unit
(fun $a\ i.\exists s\ p\ j\ h.\ i = j \bullet h \wedge$ shape $a\ s\ p\ j$,
fun $aa\ qq\ yy\ ii\ mm.\forall ss\ pp\ jj\ hh.$
$ii = jj \bullet\bullet hh \to$ sshape $aa\ ss\ pp\ jj \to$
$\exists jj'\ ss'.\ mm = jj' \bullet\bullet hh \wedge$ sshape $aa\ ss'\ pp\ jj'$)

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

depend on x and we prove that y is low by proving that $y_1 = y_2$ in the postcondition.

P_4 : $\Pi a\ b\ c\ u\ v$: bool. STsec [bool] bool
(fun $x\ i$. True,
fun $xx\ yy\ ii\ mm.\ y_1 = y_2 \wedge mm = ii \wedge$
$y_1 = (a = c)\ ||\ (b = c)\ ||\ u\ ||\ v)\ \widehat{=}$
fun $a\ b\ c\ u\ v$.
do (fun $x.\ t \leftarrow$if u then
if x then return a else return b
else
if v then return a else return c;
return $((t = a)\ ||\ (t = b))$)

Example 2.3 (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 A be a module storing some integer data, say salary, whose integrity should be enforced: A allows the salary to be readable globally, but only the module itself can update it. Thus A 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 ASig in Figure 1 presents the specifications exported by A. Figure 2 shows a possible ASig implementation that uses two local pointers – one for the salary and one for the password.

Referring to Figure 2, the method new takes a nat salary and a string password, and generates a new A instance initialized with this data; read_salary takes a local-context argument a and returns the current salary; write_salary takes a new salary, a password, a local-context argument a, and updates the salary only if the supplied password matches the password stored in a. Referring back to Figure 1, ASig specifies an abstract type α, abstract predicates

$$
\begin{array}{l}
\mathsf{type}\ \alpha\ \widehat{=}\ \mathsf{ptr} \times \mathsf{ptr} \\
\mathsf{salary}\ (a : \alpha)\ \widehat{=}\ a.1 \\
\mathsf{passwd}\ (a : \alpha)\ \widehat{=}\ a.2 \\
\mathsf{sshape}\ (aa : \alpha^2)\ (ss : \mathsf{nat}^2)\ (pp : \mathsf{string}^2)\ (ii : \mathsf{heap}^2)\ \widehat{=} \\
\quad i_1 = \mathsf{salary}\ a_1 \mapsto s_1 \bullet \mathsf{passwd}\ a_1 \mapsto p_1\ \wedge \\
\quad i_2 = \mathsf{salary}\ a_2 \mapsto s_2 \bullet \mathsf{passwd}\ a_2 \mapsto p_2\ \wedge \\
\qquad s_1 = s_2\ \wedge\ p_1 = p_2 \\
\mathsf{new}\ s\ p\ \widehat{=}\ \mathsf{do}\ (x \leftarrow \mathsf{alloc}\ s; y \leftarrow \mathsf{alloc}\ p; \mathsf{return}\ (x, y)) \\
\mathsf{read\_salary}\ \widehat{=}\ \mathsf{do}\ (\mathsf{fun}\ a.\, \mathsf{read}\ (\mathsf{salary}\ a)) \\
\mathsf{write\_salary}\ s\ p\ \widehat{=} \\
\quad \mathsf{do}\ (\mathsf{fun}\ a.\, x \leftarrow \mathsf{read}\ (\mathsf{passwd}\ a); \\
\qquad\qquad \mathsf{if}\ x = p\ \mathsf{then}\ \mathsf{write}\ (\mathsf{salary}\ a)\ s\ \mathsf{else}\ \mathsf{return}\ ())
\end{array}
$$

Fig. 2. Implementation of ASig.

sshape and shape, a relation srefl between shape and sshape, and the types of the methods. The local-context arguments of read_salary and write_salary have type α. Although these figures may look complicated, the reader should bear in mind that they are intended for full functional verification. Also, the implementations of the various abstract predicates and types such as sshape, shape and α, 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, that is, sshape is invariant over two runs. It is parametrized over pairs of α's, nat salaries, string passwords, and heaps that are current during execution. The parametrization by all these values captures that different instances of A 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 the local state of A, has to describe the desired parts of that state in its own precondition. This is why ASig keeps sshape and shape abstract. The abstraction hides the layout of A's local state from the clients, thus preventing them from describing the layout in their preconditions and forcing them to access A'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 proof essentially exhibits that shape is indeed a diagonal of sshape, i.e., the two predicates are not related arbitrarily. srefl is used in the verification of the implementations of new, read_salary and write_salary.

The STsec types used in the methods of A describe several additional properties. One such property is that the local state of each instance of A 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 $\bullet\bullet$ is defined as the generalization of 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 the state in which the function executes to be larger than the module's local state by allowing $ii = jj \bullet\bullet hh$ where jj names A's local state and hh is the

potential global part. For write_salary we require that the global part, hh, remains invariant, but the local part may be changed by storing the new salary: this is reflected by the heap jj' in the postcondition.

The specifications expose that read_salary does not change 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.

The salary and password arguments in new and write_salary are ordinary function arguments, whereas α is in the local context of STsec in read_salary and write_salary. Thus, within the scope of the methods of A, the salary and the password are low (c.f. Remark 2.1) whereas the α argument is high because it is unconstrained by the methods' pre- and post-conditions. Of course, as far as clients of ASig are concerned, all three of these are high: the abstraction over sshape hides all relations between the stored values.

One consequence of making salary and password internally low is that whenever a new instance of A 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 A to store confidential data into its 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 hiding is achieved by existential quantification over ss and pp in the postcondition of new, and over ss' in the postcondition of write_salary.

Thus, while the integer s and the string p are themselves low, it is the act of writing them into A's local state that makes them private to A. A well-typed client that gains access to s and p will not be able to establish that these are equal to what is stored into A's local state, as that fact does not follow from the specs of new and write_salary.

Example 2.4 (Declassification) Module A can use the internal knowledge that salary and password are low, to implement and export an additional function which declassifies salary – that is, reveals the internal knowledge that salary is low. This declassification can be based on arbitrary conditions – say, it is only granted if a correct password has been supplied.

$$
\begin{array}{l}
\mathsf{declassify} : \Pi p : \mathsf{string}.\, \mathsf{STsec}\ [\alpha]\ \mathsf{bool} \\
\quad (\mathsf{fun}\ a\ i.\ \exists s\ q\ j\ h\ .\ i = j \bullet h\ \wedge\ \mathsf{shape}\ a\ s\ q\ j, \\
\quad\ \mathsf{fun}\ aa\ yy\ ii\ mm.\ \forall ss\ qq\ jj\ hh\ . \\
\qquad ii = jj \bullet\!\bullet\ hh \rightarrow \mathsf{sshape}\ aa\ ss\ qq\ jj \rightarrow \\
\qquad \exists jj'\ .\ mm = jj' \bullet\!\bullet\ hh\ \wedge\ \mathsf{sshape}\ aa\ ss\ qq\ jj'\ \wedge \\
\qquad\qquad y_1 = y_2\ \wedge\ yy = (p = q_1, p = q_2)\ \wedge \\
\qquad\qquad y_1 \rightarrow s_1 = s_2)\ \widehat{=} \\
\quad \mathsf{fun}\ p.\ \mathsf{do}\ (\mathsf{fun}\ a.\ x \leftarrow \mathsf{read}\ (\mathsf{passwd}\ a); \mathsf{return}\ (p = x))
\end{array}
$$

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 postcondition of declassify shows that the salary is low if declassify returned true ($y_1 \rightarrow s_1 = s_2$). This is possible because the low-status of the salary has been hardwired into the implementation of sshape, and therefore can be revealed at will.

Example 2.5 (State-based policies) Module A can support policies that change depending on local state. For example, it 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 A'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.

```
rreadable : α² → heap² → prop
readable ≜ fun a h. rreadable (a, a) (h, h)
rrefl : ∀aa ii . rreadable aa ii →
                    readable a₁ i₁ ∧ readable a₂ i₂
grant : STsec [α] 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 [α] 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 [α] 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 ASig with state-based read access.

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 A'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.

Example 2.6 (Conditional access and erasure policies) Consider the scenario where A is prepared to share the (confidential) salary information in its local state with a client module B in order for it to compute A's taxes using method B.compute_tax. While A permits such sharing, it wants to prevent compute_tax from stealing the salary by copying it into B's local state. A may insist that B not have any local state, or that it deallocate all of the local state before compute_tax terminates. This requirement is too restrictive, for it precludes B from maintaining persistent, but innocuous, local state such as a count of how many times compute_tax has executed. In general any client local state not derived from the confidential data, should be allowed to escape the function call. In RHTT, one can formulate such a permissive policy using higher-order STsec types.

Our goal then is to export at A's interface a function ratify which grants a client access to salary provided the client furnishes a proof that it *erases* salary and all information derived from salary when it terminates. Such a function will be parametric over the client's local state —denoted by the abstract type β — and G, defining the values stored in the client's local state, as well as function bcmp and relational predicate bbshape. (Signature ASig must be augmented with these four items.) bcmp defines how the values given by G evolve after each call to ratify. The relational predicate bbshape : $\beta^2 \to G^2 \to \mathsf{heap}^2 \to \mathsf{prop}$ defines the heap representation over two runs of the client's local state β^2 and local values G^2. The initial heap i on which the client method will operate can be split in three ways: j belonging

$$
\begin{array}{l}
\mathsf{bbshape} : \beta^2 \to G^2 \to \mathsf{heap}^2 \to \mathsf{prop} \\
\mathsf{bshape} \;\widehat{=}\; \mathsf{fun}\ b\ k\ t.\, \mathsf{bbshape}\ (b,b)\ (k,k)\ (t,t) \quad (*\ \text{Note that } k : \mathsf{G}\ *) \\
\mathsf{brefl} : \ldots\ (*\ \text{similar to } \mathsf{srefl}\ *) \\
\\
\mathsf{epre}\ (a:\alpha)\ (b:\beta)\ (j\ i : \mathsf{heap}) \;\widehat{=}\; \\
\quad \exists s\ p\ k\ t\ h.\, i = j \bullet t \bullet h \wedge \mathsf{shape}\ a\ s\ p\ j \wedge \mathsf{bshape}\ b\ k\ t \\
\mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm \;\widehat{=}\; \\
\quad \forall ss\ pp\ kk\ tt\ hh.\, ii = jj \bullet\!\bullet tt \bullet\!\bullet hh \to \\
\qquad \mathsf{sshape}\ aa\ ss\ pp\ jj \to \mathsf{bbshape}\ bb\ kk\ tt \to \\
\quad \exists jj'\ tt'\ .\ mm = jj' \bullet\!\bullet tt' \bullet\!\bullet hh \wedge \mathsf{sshape}\ aa\ ss\ pp\ jj' \wedge \\
\qquad \mathsf{bbshape}\ bb\ (\mathsf{bcmp}\ k_1, \mathsf{bcmp}\ k_2)\ tt' \wedge \\
\qquad (t_1 = t_2 \to t_1' = t_2')
\end{array}
$$

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

to A, t belonging to B, and the remainder h that is untouched. Predicate epre in Figure 4 describes this situation.

On the other hand, epost states that B's local state tt' at the end stores the correct statistics (bcmp of k_1 and k_2). Moreover if B's initial local state tt is assumed low, then tt' is low as well. In other words, the client method did not copy into tt' any of the high values that it may have read from A's local state. A program that requests read access to A's local state, and respects the described policy has the type

$$
\begin{array}{l}
T \;\widehat{=}\; \mathsf{STsec}\ [\alpha,\beta]\ \mathsf{nat} \\
\qquad (\mathsf{fun}\ a\ b\ i.\, \exists j.\, \mathsf{readable}\ a\ j \wedge \mathsf{epre}\ a\ b\ j\ i, \\
\qquad\ \mathsf{fun}\ aa\ bb\ yy\ ii\ mm.\, \forall jj. \\
\qquad\quad \mathsf{rreadable}\ aa\ jj \to \mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm)
\end{array}
$$

Module A can now ratify a program such as compute_tax. Provided this program has type T, A can grant it read access to salary. The function ratify removes readable from T, much like the grant program would. After that, a client method such as compute_tax 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 a proof that compute_tax satisfies type T, i.e., it does not leak salary.

$$
\begin{array}{l}
\mathsf{ratify} : T \to \\
\quad \mathsf{STsec}\ [\alpha,\beta]\ \mathsf{nat} \\
\qquad (\mathsf{fun}\ a\ b\ i.\, \exists j.\, \mathsf{epre}\ a\ b\ j\ i, \\
\qquad\ \mathsf{fun}\ aa\ bb\ yy\ ii\ mm. \\
\qquad\quad \forall jj.\, \mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm) \;\widehat{=}\; \\
\quad \mathsf{fun}\ e : T.\, \mathsf{do}\ (\mathsf{fun}\ a\ b.\, e\ a\ b)
\end{array}
$$

This specification of ratify can be instantiated in several ways, by choosing different values for β, G, bbshape and bcmp. For example, if $\beta \;\widehat{=}\; \mathsf{unit}$, $G \;\widehat{=}\; \mathsf{unit}$, and bbshape states that B's local heap is empty, then compute_tax does not keep any local state, and A, correctly, ratifies it. Alternatively, if

$$
\begin{array}{l}
\beta \;\widehat{=}\; \mathsf{ptr} \qquad G \;\widehat{=}\; \mathsf{nat} \qquad \mathsf{bcmp} \;\widehat{=}\; \mathsf{succ} \\
\mathsf{bbshape}\ bb\ kk\ tt \;\widehat{=}\; (t_1 = b_1 \mapsto k_1 \wedge t_2 = b_2 \mapsto k_2)
\end{array}
$$

then compute_tax keeps a single pointer whose content is incremented by 1 after every execution.

Suppose compute_tax computes the tax of 24% of the salary, while also keeping its invocation count. It can be implemented and then immediately ratified by the following function call. Notice that by the type of ratify, the return value of compute_tax is high as there is no requirement $y_1 = y_2$ in epost. Hence, the fact that this value is a function of salary, is not a security leak.

$$\begin{array}{l}\mathsf{ratify}\ (\mathsf{do}\ (\mathsf{fun}\ a:\alpha\ b:\beta.\\ \qquad\qquad x \leftarrow \mathsf{read\_salary}\ a; k \leftarrow \mathsf{read}\ b;\\ \qquad\qquad \mathsf{write}\ b\ (k+1); \mathsf{return}\ (x * 24\%)))\end{array}$$

Suppose compute_tax 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

$$\begin{array}{l}\beta\ \widehat{=}\ \mathsf{ptr} \times \mathsf{ptr}\ (\text{*one ptr for } p \text{ and one for } q\text{*})\\ G\ \widehat{=}\ \mathsf{nat} \qquad \mathsf{bcmp}\ \widehat{=}\ \mathsf{succ}\\ \mathsf{bbshape}\ bb\ kk\ tt\ \widehat{=}\\ \qquad \exists pp\ qq : \mathsf{nat}^2\ .\ t_1 = \mathsf{fst}\ (b_1) \mapsto p_1 \bullet \mathsf{snd}\ (b_1) \mapsto q_1\ \wedge\\ \qquad\qquad\qquad\qquad t_2 = \mathsf{fst}\ (b_2) \mapsto p_2 \bullet \mathsf{snd}\ (b_2) \mapsto q_2\ \wedge\\ \qquad\qquad\qquad\qquad kk = (p_1 - q_1, p_2 - q_2)\end{array}$$

Now compute_tax can read salary, then increment p and q by amount of the salary, and additionally, increment p by 1. In terms of required specifications for B's local state, the program still keeps the invocation count. However, the program is actually stealing salary, because the salary can be inferred by deducting the old value of q from the new one. Such a program will fail to be ratified. If ratify is called with the argument

$$\begin{array}{l}\mathsf{do}\ (\mathsf{fun}\ a:\alpha\ b:\beta.\\ \qquad x \leftarrow \mathsf{read\_salary}\ a; p \leftarrow \mathsf{read}\ (\mathsf{fst}\ b); q \leftarrow \mathsf{read}\ (\mathsf{snd}\ b);\\ \qquad \mathsf{write}\ (\mathsf{fst}\ b)\ (p+x+1); \mathsf{write}\ (\mathsf{snd}\ b)\ (q+x);\\ \qquad \mathsf{return}\ (x * 24\%))\end{array}$$

ratify forces compute_tax to prove that its ending state is low ($t_1' = t_2'$) as defined in epost, but this is not provable if tt' stores the salary x. Indeed, as x is high, compute_tax lacks the information that x is equal in the two runs, so it cannot prove that his pointers store equal values in two runs. For ratification, compute_tax must erase salary, perhaps by mutating 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 2005, 2008].

3. TYPING RULES AND SEMANTIC MODEL

3.1. 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 primitive commands; descriptions of the other commands appear later in the section.

Typing rules for primitive commands.

$$
\begin{array}{l}
\mathsf{return} \ : \mathsf{STsec}\ [A]\ A \\
\qquad (\mathsf{fun}\ x\ i.\ \mathsf{True}, \\
\qquad\ \mathsf{fun}\ xx\ yy\ ii\ mm.\ mm = ii\ \wedge\ yy = xx) \\
\\
\mathsf{read} \ : \mathsf{STsec}\ [\mathsf{ptr}]\ A \\
\qquad (\mathsf{fun}\ \ell\ i.\ \exists h : \mathsf{heap}\ v : A\ .\ \ i = \ell \mapsto v \bullet h, \\
\qquad\ \mathsf{fun}\ \ell\ell\ yy\ ii\ mm.\ mm = ii\ \wedge\ \forall hh\ vv\ . \\
\qquad\quad ii = (\ell_1 \mapsto v_1, \ell_2 \mapsto v_2) \bullet\bullet hh \rightarrow yy = vv) \\
\\
\mathsf{write} \ : \mathsf{STsec}\ [\mathsf{ptr}, A]\ \mathsf{unit} \\
\qquad (\mathsf{fun}\ \ell\ v\ i.\ \exists h\ B : \mathsf{type}\ w : B\ .\ \ i = \ell \mapsto w \bullet h, \\
\qquad\ \mathsf{fun}\ \ell\ell\ vv\ yy\ ii\ mm.\ \forall hh\ B_1\ B_2\ w_1 : B_1\ w_2 : B_2\ . \\
\qquad\quad ii = (\ell_1 \mapsto w_1, \ell_2 \mapsto w_2) \bullet\bullet hh \rightarrow \\
\qquad\quad mm = (\ell_1 \mapsto v_1, \ell_2 \mapsto v_2) \bullet\bullet hh) \\
\\
\mathsf{dealloc} \ : \mathsf{STsec}\ [\mathsf{ptr}]\ \mathsf{unit} \\
\qquad (\mathsf{fun}\ \ell\ i.\ \exists h\ B : \mathsf{type}\ w : B\ .\ \ i = \ell \mapsto w \bullet h, \\
\qquad\ \mathsf{fun}\ \ell\ell\ yy\ ii\ mm.\ \forall hh\ B_1\ B_2\ w_1 : B_1\ w_2 : B_2\ . \\
\qquad\quad ii = (\ell_1 \mapsto w_1, \ell_2 \mapsto w_2) \bullet\bullet hh \rightarrow mm = hh)
\end{array}
$$

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 return 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.

Typing rules for allocation. 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.

$$
\begin{array}{l}
\mathsf{lalloc} \ : \mathsf{STsec}\ [A]\ \mathsf{ptr} \\
\qquad (\mathsf{fun}\ v\ i.\ \mathsf{True}, \\
\qquad\ \mathsf{fun}\ vv\ yy\ ii\ mm. \\
\qquad\quad mm = (y_1 \mapsto v_1, y_2 \mapsto v_2) \bullet\bullet ii\ \wedge \\
\qquad\quad (i_1 \cong i_2 \rightarrow y_1 = y_2\ \wedge\ m_1 \cong m_2)) \\
\\
\mathsf{alloc} \ : \mathsf{STsec}\ [A]\ \mathsf{ptr} \\
\qquad (\mathsf{fun}\ v\ i.\ \mathsf{True}, \\
\qquad\ \mathsf{fun}\ vv\ yy\ ii\ mm. \\
\qquad\quad mm = (y_1 \mapsto v_1, y_2 \mapsto v_2) \bullet\bullet ii\ \wedge \\
\qquad\quad \mathsf{even}\ y_1\ \wedge\ \mathsf{even}\ y_2)
\end{array}
$$

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, lalloc returns *equal* (and hence low) pointers, when invoked under appropriate conditions. Denotationally, the two allocators operate on disjoint pools of locations: alloc always returns a *randomly* chosen unallocated *even* pointer, while lalloc returns the *next* unallocated *odd* pointer (although we did not reflect that the low pointers are odd in the specification of lalloc, as we explain below). Here we rely on the property that type ptr is isomorphic to nat in our model.

Definition 3.1 (Low-equivalent heaps) 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 ($i_1 \cong i_2$), the command returns equal pointers ($y_1 = y_2$), and low-equivalent output heaps ($m_1 \cong m_2$). In the case of alloc, we expose the evenness of y_1 and y_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. This lemma, as well as a few other lemmas in our library, uses even-ness to argue that adding a pointer to a heap preserves low equivalence. This is why the type of alloc exposes that the returned pointers are even. On the other hand, we do not need similar lemmas about odd pointers and low equivalence, which is why we decided not to expose that returned pointers are odd in the type of lalloc.

Example 3.2 (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.

$$\begin{array}{l}\mathsf{do}\ (\mathsf{fun}\ h.\ \mathsf{if}\ h\ \mathsf{then}\ y \leftarrow \mathsf{lalloc}\ 2; \mathsf{return}\ y\ \mathsf{else}\\ \qquad\qquad x \leftarrow \mathsf{alloc}\ 1; y \leftarrow \mathsf{lalloc}\ 2; \mathsf{dealloc}\ x; \mathsf{return}\ y) :\\ \quad \mathsf{STsec}\ [\mathsf{bool}]\ \mathsf{ptr}\\ \quad\ (\mathsf{fun}\ h\ i.\ \mathsf{True},\\ \quad\ \mathsf{fun}\ hh\ yy\ ii\ mm.\ mm = ii \bullet\!\!\bullet (y_1 \mapsto 2, y_2 \mapsto 2)\ \wedge\\ \qquad\qquad\qquad (i_1 \cong i_2 \to y_1 = y_2))\end{array}$$

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 only one of them would have been extended with an odd location x. Thus, we cannot conclude $y_1 = y_2$, that is, the returned pointer is low.

***Remark* 3.3** 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 [Yang and O'Hearn 2002]. With large footprints, we can specify lalloc (specifically, the antecedent $i_1 \cong i_2$ 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. 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 • [Nanevski et al. 2010].

Another way of treating allocation in the relational setting is to model its non-determinism by means of partial bijections between pointers [Amtoft et al. 2006; Banerjee and Naumann 2005]. 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 non-interference of Amtoft et al. [2006] allows the input heaps to the computation to be related by some bijection between pointers, and requires the ending heaps also to be 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. For example, consider a 2-node, acyclic, singly-linked list, with the usual implementation: Each node has a *data* field and a *nxt* pointer that points to the next node in the list. Let p be the list header, and let $p.data = 0$, $p' = p.nxt$ and $p'.data = 0$. Suppose also that all *data* are low and all *nxt* pointers are low. Now consider the conditional

$$\mathsf{if}\ \text{high}\ \mathsf{then}\ \textit{flip}\ \mathsf{else}\ \mathsf{skip}$$

where *flip* reverses the list, so that now the list header is p'. Suppose the first run of the program takes the then branch ¡¡¡¡¡¡¡ .mine and the second run takes the else branch. In this case there exists a bijection $\{(p', p), (p'.nxt, p.nxt), (p'.data, p.data)\}$. Yet by observing that p and p' are different addresses, an attacker gains information about the high guard. ======= and the second run takes the else branch. In this case there exists a bijection $\{(p', p), (p'.nxt, p.nxt), (p'.data, p.data)\}$. Yet, by observing that p and p' are different addresses, an attacker can recover information about the high guard. ¿¿¿¿¿¿¿ .r1732

Typing rule for sequential composition. 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

$$e_1 : \mathsf{STsec}\,\Gamma\,A\,(p_1, q_1) \text{ and } e_2 : \mathsf{STsec}\,(A::\Gamma)\,B\,(p_2, q_2)$$

Then $e_1; e_2$ first executes e_1, passing the returned value as the first local argument to e_2. Assuming

$$\begin{aligned}
&\gamma : \overline{\Gamma} \text{ and } \gamma\gamma : \overline{\Gamma}^2\\
&p_1 : \overline{\Gamma} \to \mathsf{heap} \to \mathsf{prop}\\
&p_2 : \mathsf{A} \times \overline{\Gamma} \to \mathsf{heap} \to \mathsf{prop}\\
&q_1 : \overline{\Gamma}^2 \to \mathsf{A}^2 \to \mathsf{heap}^2 \to \mathsf{heap}^2 \to \mathsf{prop}\\
&q_2 : (\mathsf{A} \times \overline{\Gamma})^2 \to \mathsf{B}^2 \to \mathsf{heap}^2 \to \mathsf{heap}^2 \to \mathsf{prop}
\end{aligned}$$

the STsec type for $e_1; e_2$ is

$$\begin{array}{l}\mathsf{STsec}\ \Gamma\ B\\ (\mathsf{fun}\ \gamma\ i.\ p_1\ \gamma\ i\ \wedge\\ \qquad \forall y\ m\ .\ q_1\ (\gamma,\gamma)\ (y,y)\ (i,i)\ (m,m) \to p_2\ (y,\gamma)\ m,\\ \ \mathsf{fun}\ \gamma\gamma\ yy\ ii\ mm.\\ \quad \exists vv : \mathsf{A}^2, hh : \mathsf{heap}^2\ .\ q_1\ \gamma\gamma\ vv\ ii\ hh\ \wedge\\ \qquad\qquad q_2\ ((v_1,\gamma_1),(v_2,\gamma_2))\ yy\ hh\ mm)\end{array}$$

In English: (a) the precondition requires e_1 to be safe in the initial heap i of the sequential composition —thus the $p_1\ \gamma\ i$; and (b) 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, i.e., $p_2\ (y,\gamma)\ m$ holds. The satisfaction of e_1's squared postcondition is reflected by $q_1\ (\gamma,\gamma)\ (y,y)\ (i,i)\ (m,m)$ above.

The postcondition states that intermediate values vv and heaps hh exist, obtained after running e_1 but before running e_2.

3.2. Combinators

Because e_1's output is bound in the local context of e_2, it could be low or high; so we cannot treat this output as an ordinary functional variable, despite our suggestive notation in Section 2. Indeed, as discussed previously in Section 2, 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.

The combinator @. 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. In the following, with $\gamma : \overline{\Gamma}_1$ and $\gamma\gamma : \overline{\Gamma}_1^2$, we have

$$\begin{array}{l}e\ @\ f : \mathsf{STsec}\ \Gamma_1\ A\\ \qquad (\mathsf{fun}\ \gamma\ i.\ p\ (f\ \gamma)\ i, \mathsf{fun}\ \gamma\gamma\ yy\ ii\ mm.\ q\ (f\ \gamma_1, f\ \gamma_2)\ yy\ ii\ mm)\end{array}$$

When $\Gamma_1 = \mathsf{nil}$, then $\overline{\Gamma}_1 = \mathsf{unit}$ and f is isomorphic to a tuple $\widehat{\gamma} : \overline{\Gamma}_2$. In these situations we will write $e\ @_0\ \widehat{\gamma}$ instead of the longer $e\ @\ (\mathsf{fun}\ \_ : \mathsf{unit}.\widehat{\gamma})$. Note that $e\ @_0\ \widehat{\gamma}$ has empty local context as its type

$\mathsf{STsec}\ \mathsf{nil}\ A\ (\mathsf{fun}\ i.\ p\ \widehat{\gamma}\ i, <<<<<<<\ .mine \mathsf{fun}\ ii.\ q\ (\widehat{\gamma},\widehat{\gamma})\ ii) =======\ \mathsf{fun}\ yy\ ii\ mm.\ q\ (\widehat{\gamma},\widehat{\gamma})\ yy\ ii\ mm) >>>>>>>$

¡¡¡¡¡¡¡ .mine indicates. We refer to e in $e\ @\ f$ as the *head* of the instantiation, and to f as the *explicit substitution*. ======= indicates. We refer to e in $e\ @\ f$ as the *head* of the instantiation, and to f as the *explicit substitution*. ¿¿¿¿¿¿¿ .r1732

Example 3.4 In Example 2.3, we implemented new as

$$\mathsf{fun}\ s\ p.\mathsf{do}\ (x \leftarrow \mathsf{alloc}\ s; y \leftarrow \mathsf{alloc}\ p; \mathsf{return}\ (x,y))$$

The actual implementation using combinators is

$$\begin{array}{l}\mathsf{fun}\ s\ p.\mathsf{do}\ (\mathsf{alloc}\ @_0\ s;\\ \qquad\qquad \mathsf{alloc}\ @\ (\mathsf{fun}\ x.\ p);\\ \qquad\qquad \mathsf{return}\ @\ (\mathsf{fun}\ y\ x.(x,y)))\end{array}$$

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 the first alloc is empty, and we only instantiate alloc with the salary variable s, to allocate a new location storing A's salary. For the second alloc, the domain of the substitution includes the

variable x, which names the result of the previous alloc. In principle, x could be used in computing the value passed to alloc, though we do not do that here, and just allocate the second location and store into it A's password p. The domain of the substitution for return includes an additional y, which names the output of the preceding alloc, and the program returns the pair (x, y).

Similarly, the function read_salary is reimplemented as

$$\mathsf{read\_salary} \;\widehat{=}\; \mathsf{do}\ (\mathsf{read}\ @\ (\mathsf{fun}\ a.\, \mathsf{salary}\ a))$$

The combinator do. The combinator do mediates between the semantics of types, and the logic of assertions. The role of do is to change the type of $e : \mathsf{STsec}\ \Gamma\ A\ (p_1, q_1)$ into $\mathsf{STsec}\ \Gamma\ A\ (p_2, q_2)$, if a proof can be constructed, possibly interactively, that e can be ascribed the precondition p_2 and postcondition q_2. This ascription can be made if two properties are proved.

First, e should be safe to execute in any heap satisfying p_2; that is, it should be possible to strengthen the precondition and prove $\forall\gamma : \overline{\Gamma}.\, i : \mathsf{heap}.\, p_2\ \gamma\ i \to p_1\ \gamma\ i$. We will use a somewhat different proposition, $C_1(e)$ below, which exposes that safety can be proved by syntax-directed reasoning on the structure of the involved program, a property we exploit in our verification rules (Section 4).

$$C_1(e) \;\widehat{=}\; \forall\gamma : \overline{\Gamma}.\, i : \mathsf{heap}.\, p_2\ \gamma\ i \to \mathsf{safe}\ (e\ @_0\ \gamma)\ i$$

where safe e is defined as the precondition given by e's type. That is, if $e' : \mathsf{STsec}\ \mathsf{nil}\ (p, q)$ with empty local context, then:

$$\mathsf{safe}\,(e') \;\widehat{=}\; p$$

Second, it should be possible to change the postcondition q_1 of e into q_2. Semantically we prove that two executions of e result in heaps and values satisfying q_2. We capture this property via the following predicate, which formalizes the reasoning in a relational Hoare logic over *two programs* [Benton 2004; Yang 2007]. 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

$$\begin{array}{l}\mathsf{verify2}\ ii\ e_1\ e_2\ q\ \widehat{=}\\ \quad \forall yy : A^2, mm : \mathsf{heap}^2\ .\\ \qquad (i_1, y_1, m_1) \in \mathsf{runs\_of}\ e_1 \to\\ \qquad (i_2, y_2, m_2) \in \mathsf{runs\_of}\ e_2 \to q\ yy\ mm\end{array}$$

¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r1732 Here, runs_of coerces programs into relations between input heaps, output values and output heaps. We will define it further below, for our particular model; we will find from that definition that e.g., i_1 is a heap satisfying r_1. (*AB: Check.*) verify2 is almost like a Hoare quadruple from relational Hoare logic except that it lacks the preconditions. We add the preconditions by the following definition, noting from above that $e\ @_0\ \gamma_1$ and $e\ @_0\ \gamma_2$ have empty local contexts.

$$\begin{array}{l}C_2(e) \;\widehat{=}\; \forall\gamma\gamma : \overline{\Gamma}^2, ii : \mathsf{heap}^2\ .\\ \qquad p_2\ \gamma_1\ i_1 \to p_2\ \gamma_2\ i_2 \to\\ \qquad \mathsf{verify2}\ ii\ (e\ @_0\ \gamma_1)\ (e\ @_0\ \gamma_2)\\ \qquad\qquad (\mathsf{fun}\ yy\ mm.\ q_2\ \gamma\gamma\ yy\ ii\ mm)\end{array}$$

In other words, if the input heaps are assumed to satisfy the new precondition p_2, then the two instantiations of e satisfy the new postcondition q_2. Notice how the two instantiations of e are actually different programs, which is why verify2 had to take two program arguments.

We now allow changing the type of e only if proofs of both $C_1(e)$ and $C_2(e)$ are provided as arguments to the constructor do, corresponding to the typing rule:

$$\begin{array}{l}\mathsf{do} : \Pi e : \mathsf{STsec}\ \Gamma\ A\ (p_1, q_1).\\ \qquad C_1(e) \to C_2(e) \to \mathsf{STsec}\ \Gamma\ A\ (p_2, q_2).\end{array}$$

***Remark* 3.5** In practice, our implementation in Coq allows that the proofs of $C_1(e)$ and $C_2(e)$ can be left out when using do. In such cases, the system emits the appropriate proof obligations, to be discharged at some later point, possibly interactively. For this reason, we consider do to have only one explicit argument e; the arguments standing for proofs of $C_1(e)$ and $C_2(e)$ can be ignored as they merely serve to guide the generation of verification conditions for e.

The combinator cond*.* Finally, a development similar to the one for do has to be carried out for conditionals as well. 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 choose 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.

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

Here D_1 captures the safety of the conditional, and D_2 the Hoare-style correctness.

$$\begin{array}{l}D_1(b, e_1, e_2)\ \widehat{=}\ \forall\gamma\ i\ .\ p\ \gamma\ i \to\\ \qquad\qquad \mathsf{safe}\ (\mathsf{if}\ b\ \gamma\ \mathsf{then}\ e_1\ @_0\ \gamma\ \mathsf{else}\ e_2\ @_0\ \gamma)\ i\\ D_2(b, e_1, e_2)\ \widehat{=}\\ \quad \forall\gamma\gamma\ ii.p\ \gamma_1\ i_1 \to p\ \gamma_2\ i_2 \to\\ \qquad \mathsf{verify2}\ ii\,(\mathsf{if}\ b\ \gamma_1\ \mathsf{then}\ e_1\ @_0\ \gamma_1\ \mathsf{else}\ e_2\ @_0\ \gamma_1)\\ \qquad\qquad (\mathsf{if}\ b\ \gamma_2\ \mathsf{then}\ e_1\ @_0\ \gamma_2\ \mathsf{else}\ e_2\ @_0\ \gamma_2)\\ \qquad\qquad (\mathsf{fun}\ yy\ mm.\ q\ \gamma\gamma\ yy\ ii\ mm)\end{array}$$

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 [Benton 2004; Yang 2007], 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 needed by Amtoft et al. [2006], that prohibit updates of low variables under a high guard (which would prevent verification of P_1 in Section 2).

Example 3.6 The function write_salary from Example 2.3 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 $a : \alpha$ and $x : \mathsf{string}$.

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

3.3. Semantic model

We justify the soundness of our type system by building a denotational model for STsec types. This development can be fully carried out as a shallow embedding in CiC that we have formalized in Coq. The model is based on predicate transformers [Dijkstra 1975]. In the first step, we temporarily ignore the local context Γ and the postcondition q, and build the type $\mathsf{prog}\ A\ p$ of stateful programs that return values of type A and are safe to execute in heaps satisfying p.

We make each element of $\mathsf{prog}\ A\ p$ be a function taking a *set of heaps* satisfying p (not just a single heap), and producing a set of possible output values and output heaps. By manipulating sets of heaps, as opposed to singleton heaps, we immediately obtain the right denotational structure needed for modeling fixed points. In particular, $\mathsf{prog}\ A\ p$ will be a complete partial order, under the subset ordering (pointwise lifted to functions on sets), with the bottom of the ordering being the function that always returns the empty set. However, we must ensure that such functions are *coherent*, in the sense that when applied to sets of heaps, they behave as if they were applied to singleton heaps. To set the stage formally,

$$\mathsf{prog}\ A\ p\ \widehat{=}\ \{f : \mathsf{ideal}\ p{\to}A{\to}\mathsf{heap}{\to}\mathsf{prop} \mid \mathsf{coherent}\ f\}$$

where

$$\begin{array}{rcl}\mathsf{ideal}\ p & \widehat{=} & \{r : \mathsf{heap}{\to}\mathsf{prop} \mid r \sqsubseteq p\}\\ r \sqsubseteq p & \widehat{=} & \forall x : \mathsf{heap}.\, x \in r \to x \in p\end{array}$$

and where, e.g., $x \in p$ is shorthand for the application $p\ x$. Now, $f : \mathsf{ideal}\ p{\to}A{\to}\mathsf{heap}{\to}\mathsf{prop}$ is ***coherent*** iff f's action on a set of heaps is a union of f's actions on the elements of the set:

$$\forall r : \mathsf{ideal}\ p. \forall x : A. \forall m : \mathsf{heap}.\, m \in f\ r\ x \leftrightarrow m \in \bigcup_{i:\mathsf{heap}, i\in r} f\ \{i\}\ x$$

By making $f : \mathsf{prog}\ A\ p$ apply only to (sets of) heaps satisfying p, as opposed to all heaps, we avoid the need to model programs that go wrong during their execution, perhaps because of memory errors (reading a dangling pointer) or type errors (reading a pointer storing a boolean, when an integer is expected). The RHTT typechecker will reject such applications as ill-typed expressions which do not need to be given semantics.

We now define the relation $\mathsf{runs\_of}\ f : \mathsf{heap}{\times}A{\times}\mathsf{heap}{\to}\mathsf{prop}$, mentioned previously in this section, which relates input heaps, output values and output heaps produced by executions of f where $f : \mathsf{prog}\ A\ p$.

$$\mathsf{runs\_of}\ f\ \ \widehat{=}\ \ \{(i, y, m) \mid i \in p\ \wedge\ m \in f\ \{i\}\ y\}$$

Next, we restore the type list Γ. Given parametrized precondition $p : \overline{\Gamma} \to \mathsf{heap} \to \mathsf{prop}$, parametrized programs are defined inductively on the structure of Γ as shown below.

$$\mathsf{pprog}\ \Gamma\ A\ p\ \ \widehat{=}\ \ \begin{cases}\mathsf{prog}\ A\ p & \text{if } \Gamma = \mathsf{nil}\\ \Pi x : B.\,\mathsf{pprog}\ \Gamma'\ A\ (p\ x) & \text{if } \Gamma = B{::}\Gamma'\end{cases}$$

Now, $\mathsf{STsec}\ \Gamma\ A\ (p, q)$ is defined as a subset of $\mathsf{pprog}\ \Gamma\ A\ p$, consisting of those programs whose two executions satisfy the postcondition q.

$$\begin{array}{l}\mathsf{STsec}\ \Gamma\ A\ (p, q) \ \widehat{=} \\ \quad \{c : \mathsf{pprog}\ \Gamma\ A\ p \mid \forall \gamma\gamma\ ii\ yy\ mm. \\ \qquad (i_1, y_1, m_1) \in \mathsf{runs\_of}\ (c\ \gamma_1) \rightarrow \\ \qquad (i_2, y_2, m_2) \in \mathsf{runs\_of}\ (c\ \gamma_2) \rightarrow q\ \gamma\gamma\ yy\ ii\ mm\}\end{array}$$

Because satisfaction of q is part of the definition of the STsec type, any program that can be ascribed an STsec type must automatically satisfy q. In particular, if q asserts noninterference, then it is immediate the program is noninterferent; there is no need for a separate proof of this fact as in a type system such as DCC [Abadi et al. 1999, Theorem 4.2], Fine [Swamy et al. 2010, Theorem 2], or Fable [Swamy et al. 2008, Theorem 25 in the full version].

We further prove in our Coq scripts the following:

Lemma 3.7 $\mathsf{STsec}\ \Gamma\ A\ (p, q)$, with the ordering inherited from $\mathsf{prog}\ A\ p$ and $\mathsf{pprog}\ \Gamma\ A\ p$, is a complete partial order.

iiiiiii .mine ======= ¿¿¿¿¿¿¿ .r1732 Thus, our model supports a combinator for least fixed points of continuous functions between STsec types:

$$\begin{array}{c}\mathsf{fix} : \Pi f : \mathsf{STsec}\ \Gamma\ A\ (p, q) \rightarrow \mathsf{STsec}\ \Gamma\ A\ (p, q). \\ \mathsf{continuous}\ f \rightarrow \mathsf{STsec}\ \Gamma\ A\ (p, q)\end{array}$$

In the Coq script secmod.v, we have implemented denotations for all the combinators presented in this section. Here we discuss only the denotations for sequential composition and do, and omit the denotations of other constructors as they are straightforward.

Denotation of sequential composition. Sequential composition $e_1; e_2$ is modeled essentially by the relational composition of the runs_of relations induced by e_1 and e_2. In other words, assume we are given a context value γ, ideal r denoting the set of possible input heaps i for the sequential composition, and the ending value $y : A$ and ending heap m. Then the denotation $[\![e_1; e_2]\!]$ relates the above values iff there exists an initial heap $i \in r$, and an intermediate heap h and value x obtained after executing e_1 but before executing e_2, such that the denotation of e_1 relates (i, x, h) and denotation of e_2 relates (h, y, m). More formally:

$$\begin{array}{l}m \in [\![e_1; e_2]\!]\ r\ y \leftrightarrow \exists i\ h\ x. \\ \quad i \in r \wedge (i, x, h) \in \mathsf{runs\_of}\ ([\![e_1]\!]\ \gamma) \wedge (h, y, m) \in \mathsf{runs\_of}\ ([\![e_2]\!]\ (x, \gamma))\end{array}$$

Notice how we pass the context value γ to the denotation of e_1, but the denotation of e_2 receives γ cons'd with the intermediate result x computed by e_1, thus capturing that the output value of e_1 is bound to the first local variable of e_2.

Denotation of do. The do command strengthens the precondition and weakens the postcondition of e. It does not change the behavior of e, but merely restricts that e can only be applied to input heaps that satisfy the strengthened precondition. Thus do e is modeled as a restriction of the denotation $[\![e]\!]$ to input heaps that satisfy the strengthened precondition. If p is a strengthened precondition, and $r : \mathsf{ideal}\ p$, then:

$$m \in [\![\mathsf{do}\ e]\!]\ r\ y \leftrightarrow \exists i.\, i \in r \wedge (i, y, m) \in \mathsf{runs\_of}\ ([\![e]\!]\ \gamma)$$

We have also shown that all our program constructors are modeled by continuous functions. Such continuity lemmas are useful for discharging the continuity obligations required by fix as we show in the next section.

4. LOGIC FOR DISCHARGING VERIFICATION CONDITIONS

As illustrated in Section 3, our program semantics relies on two main predicates: $\mathsf{safe}\, e\, i$ establishing that e is safe in the heap i, and $\mathsf{verify2}\, ii\, e_1\, e_2\, r$ establishing that r holds after the execution of e_1 and e_2 in heaps i_1 and i_2, respectively. The proof obligations that RHTT generates are mostly of this form. For example, in the case of a program with conditionals, the system will issue proof obligations in the form of appropriate D_1 and D_2 predicates, to be discharged in interaction with the system. When proving programs involving recursion, the system also issues obligations about domain-theoretic continuity. In this section, we illustrate how we reason about obligations with safe (Section 4.1), verify2 (Section 4.2) and continuous (Section 4.3). Proving obligations is largely directed by the syntax of the programs that appear as arguments to these three predicates. The inference rules of RHTT used in discharging the proof obligations related to safe and verify are proved sound in the Coq script seclog.v. The inference rules related to continuity are in secmod.v. The online appendix gives the full set of proof rules for safety.

4.1. Establishing safety

The first case to consider is when the programs in the proof obligation are instantiations of the form $e\ @_0\ \gamma$. The type system only issues instantiation containing $@_0$ at the top level, rather than the more general $@$, because the program e must be fully instantiated before it can be considered for execution against a heap.

Our strategy in such situations is to push $@_0$ further into the program, to eventually reveal some primitive command at the top-level of e. In the case of safe we have the following proof rules for dealing with $@_0$. Application of any of the proof rules replaces the goal in the conclusion with a premise in which $@_0$ has been pushed inside.

$$\textsc{sval\_inst}\ \frac{\mathsf{safe}\,(e\ @_0\ (f\ \gamma))\,i}{\mathsf{safe}\,((e\ @\ f)\ @_0\ \gamma)\,i} \qquad \textsc{sbnd\_push}\ \frac{\mathsf{safe}\,((e_1\ @_0\ \gamma);(e_2\ @\ (\mathsf{fun}\ y.\ (y,\gamma))))\,i}{\mathsf{safe}\,((e_1;e_2)\ @_0\ \gamma)\,i}$$

$$\textsc{sbnd\_inst}\ \frac{\mathsf{safe}\,((e_1\ @_0\ (f\ \gamma));e_2)\,i}{\mathsf{safe}\,((e_1\ @\ f\ @_0\ \gamma);e_2)\,i}$$

Note that in the Coq scripts, the inference rules are implemented as (proved) lemmas in the CiC meta logic. For example, the above rules SVAL_INST and SBND_PUSH are implemented as lemmas sval_inst and sbnd_push below.

$$\begin{aligned}&\mathsf{sval\_inst} \quad : \mathsf{safe}\ (e\ @_0\ (f\ \gamma))\ i \to \mathsf{safe}\ ((e\ @\ f)\ @_0\ \gamma)\ i\\ &\mathsf{sbnd\_push} : \mathsf{safe}\ ((e_1\ @_0\ \gamma);(e_2\ @\ (\mathsf{fun}\ y.(y,\gamma))))\ i \to \mathsf{safe}\ ((e_1;e_2)\ @_0\ \gamma)\ i\end{aligned}$$

The second case to consider in the proofs of safety is when the verification of e cannot be reduced to verification of simpler subprograms, but e is not a primitive command. Such cases arise, for example, if e is a call to a previously verified program, if e is a conditional, or if e is a variable standing for an unknown program. If the type of e is $\mathsf{STsec}\ \Gamma\ A\ (p,q)$, we reason using p and q.

$$\textsc{sval\_do}\ \frac{e : \mathsf{STsec}\ \Gamma\ A\ (p,q) \qquad p\ \gamma\ i}{\mathsf{safe}\,(e\ @_0\ \gamma)\,i}$$

$$\textsc{sbnd\_do}\ \frac{\begin{array}{c}e : \mathsf{STsec}\ \Gamma\ A\ (p,q)\\ (p\ \gamma\ i) \Rightarrow (\forall\ y\ m.\ q\ (\gamma,\gamma)\ (y,y)\ (i,i)\ (m,m) \Rightarrow \mathsf{safe}\,(e_2\ @_0\ y)\,m)\end{array}}{\mathsf{safe}\,((e\ @_0\ \gamma);\ e_2)\,i}$$

To prove the safety of $e\ @_0\ \gamma$ in the heap i, SVAL_DO merely issues a proof obligation that the precondition of p holds for γ and i. Similarly, SBND_DO issues a proof obligation that

the safety of the continuation e_2 can be infered out of the precondition p and postcondition q of e.

The third case to consider in the proofs of safety is when e is one of the primitive commands. Then SVAL_DO and SBND_DO can be specialized to utilize the knowledge of p and q of the particular commands. For example, we show below the specialization of SVAL_DO to return and to write, and of SBND_DO to alloc.

$$\textsc{sval\_ret}\ \frac{}{\mathsf{safe}\,(\mathsf{return}\ @_0\ x)\,i} \qquad \textsc{sval\_write}\ \frac{}{\mathsf{safe}\ (\mathsf{write}\ @_0\ (x,v))\ (x \mapsto w \bullet i)}$$

$$\textsc{sbnd\_alloc}\ \frac{\forall \ell : \mathsf{ptr}\ .\ \mathsf{safe}\,(e\ @_0\ \ell)\,(\ell \mapsto v \bullet i)}{\mathsf{safe}\,((\mathsf{alloc}\ @_0\ v); e)\,i}$$

In SVAL_RET, x is the value that is being immediately returned (see the type of return in Section 3). As the precondition of return is always True, return is always trivially safe. In the case of SVAL_WRITE, writing a value v into the location x is safe in heaps that do contain x, possibly initialized with some w. In SBND_ALLOC, allocating a fresh high pointer ℓ initialized with v is always safe. Thus, if e is a continuation of alloc, the whole composition is safe if e can be proved safe in the heap in which the existing heap i has been extended with $\ell \mapsto v$. A similar rule SBND_LALLOC exists for allocating low pointers, but is omitted here.

Sometimes, even though the top primitive command is revealed, the corresponding proof rule does not apply because the heap expression is not in the expected form. For example, sval_write applies to the heap $x \mapsto w \bullet i$, but not to $i \bullet x \mapsto w$. In those situations, we have to first rearrange the heap expressions using the commutativity and associativity of $\bullet$.

$$\begin{array}{l}\mathsf{unC} : h_1 \bullet h_2 = h_2 \bullet h_1\\ \mathsf{unA} : h_1 \bullet (h_2 \bullet h_3) = (h_1 \bullet h_2) \bullet h_3\end{array}$$

4.2. Establishing postconditions

A similar strategy as in establishing safety applies when reasoning about $\mathsf{verify2}\ ii\ e_1\ e_2\ r$. However now we have two kinds of proof rules: those that apply to both programs simultaneously, and those that apply to only one of them. The first kind is used when reasoning relationally. Typically, relational verification starts with the same command, say $e : \mathsf{STsec}\ \Gamma\ A\ (p, q)$, which is instantiated with two different explicit substitutions to obtain e_1 and e_2. Since q relates two runs of e in different heaps, we can advance the verification in both i_1 and i_2 simultaneously, and strip e from the residual goal. In case e is a return, or alloc, the appropriate lemmas are as follows.

$$\textsc{val\_ret}\ \frac{r(v_1, v_2)(i_1, i_2)}{\mathsf{verify2}\ i_1\ i_2\ (\mathsf{return}\ @_0\ v_1)\ (\mathsf{return}\ @_0\ v_2)\ r}$$

$$\textsc{bnd\_alloc}\ \frac{\begin{array}{c}\forall \ell_1, \ell_2 : \mathsf{ptr}\ .\ \mathsf{verify2}\,(\ell_1 \mapsto v_1 \bullet i_1)\,(\ell_2 \mapsto v_2 \bullet i_2)\,(e_1\ @_0\ \ell_1)\,(e_2\ @_0\ \ell_2)\,r'\\ r' = \mathsf{fun}\ yy\ mm.\ (\mathsf{even}\ \ell_1 \wedge \mathsf{even}\ \ell_2) \Rightarrow r\ yy\ mm\end{array}}{\mathsf{verify2}\ i_1\ i_2\ ((\mathsf{alloc}\ @_0\ v_1); e_1)\ ((\mathsf{alloc}\ @_0\ v_2); e_2)\ r}$$

In VAL_RET, for example, when both programs are returns of (possibly different) values v_1 and v_2, a proof of verify2 proceeds by proving that the postcondition r holds of (v_1, v_2). In BND_ALLOC, we change the goal about high allocation to a goal about continuation, where the considered heaps are extended appropriately with fresh, even, locations.

We can continue to perform relational verification until we reach a stage when the two programs in verify2 do *not* start with the same command, for example, when the two programs are obtained from different branches of a high conditional. In this situation we need the second kind of lemmas. We advance the verification of the programs separately, until

some common structure is revealed (when we can again perform relational verification) or until the verification is over. This is the purpose of proof rules VRFS1 and VRFS2. In these rules verify is a unary, i.e., non-relational version of verify2 that operates on single programs rather than pairs of programs. VRFS1 advances the verification by starting from the left while VRFS2 advances the verification by starting from the right. Note that these rules are generalized versions of self-composition [Barthe et al. 2004; de Roever and Engelhardt 1998; Gries 1993; Reynolds 1981]. Self-composition converts relational reasoning about two runs of one and the same program $e_1 = e_2 = e$, to the twice-iterated ordinary verification of that program. Here, however, we allow for two different programs e_1 and e_2. The rules are bidirectional, which we emphasize with the doubled rule line. One can apply the rules to transform a verify2 goal into a verify goal, advance the verification, and once a common structure of the programs has been uncovered, move back to relational reasoning.

$$\textsc{vrfs1}\ \frac{\mathsf{verify}\ i_1\ e_1\ (\mathsf{fun}\ y_1\ m_1.\ \mathsf{verify}\ i_2\ e_2\ (\mathsf{fun}\ y_2\ m_2.\ r\ y_1\ y_2\ m_1\ m_2))}{\mathsf{verify2}\ i_1\ i_2\ e_1\ e_2\ r}$$

$$\textsc{vrfs2}\ \frac{\mathsf{verify}\ i_2\ e_2\ (\mathsf{fun}\ y_2\ m_2.\ \mathsf{verify}\ i_1\ e_1\ (\mathsf{fun}\ y_1\ m_1.\ r\ y_1\ y_2\ m_1\ m_2))}{\mathsf{verify2}\ i_1\ i_2\ e_1\ e_2\ r}$$

To point out the similarity between proof rules for verify and verify2 we present the following proof rules that are analogues of VAL_RET and BND_ALLOC above.

$$\textsc{val\_ret}\mathrm{V}\ \frac{r\ v\ i}{\mathsf{verify}\ i\ (\mathsf{return}\ @_0\ v)\ r}$$

$$\textsc{bnd\_alloc}\mathrm{V}\ \frac{\forall \ell : \mathsf{ptr}\ .\ \mathsf{verify}\ (\ell \mapsto v \bullet i)\ (e\ @_0\ \ell)\ (\mathsf{fun}\ y\ m.\ \mathsf{even}\ \ell \Rightarrow r\ y\ m)}{\mathsf{verify}\ i\ ((\mathsf{alloc}\ @_0\ v); e)\ r}$$

Apart from the above rules we have structural rules such as the rules of conjunction.

$$\textsc{vrf2\_conj}\ \frac{\mathsf{verify2}\ i_1\ i_2\ e_1\ e_2\ r_1 \qquad \mathsf{verify2}\ i_1\ i_2\ e_1\ e_2\ r_2}{\mathsf{verify2}\ i_1\ i_2\ e_1\ e_2\ (\mathsf{fun}\ y\ m.\ r_1\ y\ m\ \wedge\ r_2\ y\ m)}$$

$$\textsc{vrf\_conj}\ \frac{\mathsf{verify}\ i\ e\ r_1 \qquad \mathsf{verify}\ i\ e\ r_2}{\mathsf{verify}\ i\ e\ (\mathsf{fun}\ y\ m.\ r_1\ y\ m\ \wedge\ r_2\ y\ m)}$$

The next two structural rules implement relational and unary versions of the usual Hoare-style rule for sequential composition, whereby the postcondition q of e is used as the precondition for the verification of the continuation e_1 in the first run and e_2 in the second run. They can be seen as rules for symbolic evaluation of a sequential composition of e, where e is unknown, except for ¡¡¡¡¡¡¡ .mine its specification. ======= its specification. ¿¿¿¿¿¿¿ .r1732

$$\textsc{bnd\_do}\ \frac{\begin{array}{c} e : \mathsf{STsec}\ \Gamma\ A\ (p, q) \qquad p\ \gamma\ i_1 \qquad p\ \gamma\ i_2 \qquad e_j : \mathsf{STsec}\ [A]\ B\ (p_j, q_j),\ j \in \{1, 2\} \\ \forall y_1\ y_2\ m_1\ m_2\ .\ q\ (\gamma, \gamma)\ (y_1, y_2)\ (i_1, i_2)\ (m_1, m_2) \Rightarrow \mathsf{verify2}\ m_1\ m_2\ (e_1\ @_0\ y_1)\ (e_2\ @_0\ y_2)\ r \end{array}}{\mathsf{verify2}\ i_1\ i_2\ (x \leftarrow (e\ @_0\ \gamma);\ e_1)\ (y \leftarrow (e\ @_0\ \gamma);\ e_2)\ r}$$

$$\textsc{bnd\_do}\mathrm{V}\ \frac{\begin{array}{c} e : \mathsf{STsec}\ \Gamma\ A\ (p, q) \qquad p\ \gamma\ i \qquad e_2 : \mathsf{STsec}\ [A]\ B\ (p_2, q_2) \\ \forall y\ m\ .\ q\ (\gamma, \gamma)\ (y, y)\ (i, i)\ (m, m) \Rightarrow \mathsf{verify}\ m\ (e_2\ @_0\ y)\ r \end{array}}{\mathsf{verify}\ i\ (x \leftarrow (e\ @_0\ \gamma);\ e_2)\ \ r}$$

The next two structural rules implement relational and unary versions of the Hoare-style rule of consequence which asserts that postcondition q of e can be weakened to r provided

q implies r.

$$\textsc{val\_do}\ \frac{e : \mathsf{STsec}\ \Gamma\ A\ (p,q) \qquad p\ \gamma\ i_1 \qquad p\ \gamma\ i_2 \qquad \forall y_1\ y_2\ m_1\ m_2\ .\ q\ (\gamma,\gamma)\ (y_1,y_2)\ (i_1,i_2)\ (m_1,m_2) \Rightarrow r\ (y_1,y_2)\ (m_1,m_2)}{\mathsf{verify2}\ i_1\ i_2\ (e\ @_0\ \gamma)\ (e\ @_0\ \gamma)\ r}$$

$$\textsc{val\_do}\mathrm{V}\ \frac{e : \mathsf{STsec}\ \Gamma\ A\ (p,q) \qquad p\ \gamma\ i \qquad \forall y\ m\ .\ q\ (\gamma,\gamma)\ (y,y)\ (i,i)\ (m,m) \Rightarrow r\ y\ m}{\mathsf{verify}\ i\ (e\ @_0\ \gamma)\ r}$$

4.3. Establishing continuity

When typechecking a recursive program, in addition to the proof obligations for safe and verify, RHTT will emit a proof obligation for establishing continuity of the function that the program represents. Such proof obligations arise out of the type of the program combinator fix shown in Section 3.3, which is explicitly parametrized by the obligation continuous f, where f is the function that is being recursed over.

Typically, the function over which we are recursing will be in an expanded form; that is, it will look like fun $e.\, e'$, where e' is the body of recursive function, and e is the name for the recursive call. Both e and e' have the type ¡¡¡¡¡¡¡ .mine STsec Γ A (p,q). ======= STsec Γ A (p,q). ¿¿¿¿¿¿¿ .r1732 Discharging proof obligations about continuity for such functions follows the structure of e', and is always straightforward. We merely apply the rules stating that the program constructors such as instantiation and sequential composition preserve continuity (the rules INST_STCONT and BND_STCONT shown below). This strips off the program constructors. Once a recursive call to e is reached the rule ID_STCONT applies and concludes the proof. If some primitive command such as read or write is reached, then the rule CONST_STCONT applies and concludes the proof. No other cases can appear in programs written using our combinators, which make the rules below sufficient for proving continuity. ¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r1732

$$\textsc{inst\_stcont}\ \frac{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e')}{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e'\ @\ \gamma)}$$

$$\textsc{bnd\_stcont}\ \frac{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e_1) \qquad \mathsf{continuous}\ (\mathsf{fun}\ e.\, e_2)}{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e_1; e_2)}$$

$$\textsc{id\_stcont}\ \frac{}{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e)} \qquad\qquad \textsc{const\_stcont}\ \frac{e \notin \mathsf{FV}(e')}{\mathsf{continuous}\ (\mathsf{fun}\ e.\, e')}$$

4.4. Using the proof rules

In this section we illustrate the working of the RHTT logic by sketching how to verify that the implementation of new in Figure 2 matches its specification from Figure 1, that is:

$$\begin{array}{l}\mathsf{new} : \mathsf{nat} \to \mathsf{string} \to \\ \quad \mathsf{STsec}\ \mathsf{nil}\ \mathsf{alice}\ (\mathsf{fun}\ i.\, \mathsf{True}, \\ \qquad\qquad \mathsf{fun}\ aa\ ii\ mm.\, \exists ss\ pp\ hh.\, mm = ii \bullet\bullet hh \wedge \mathsf{sshape}\ aa\ ss\ pp\ hh)\end{array}$$

where the implementation of new, using combinators is as follows:

$$\begin{array}{l}\mathsf{new}\ s\ p = \mathsf{do}\ (\mathsf{alloc}\ @_0\ s; \\ \qquad\qquad\qquad\quad \mathsf{alloc}\ @\ (\mathsf{fun}\ x.\, p); \\ \qquad\qquad\qquad\quad \mathsf{return}\ @\ (\mathsf{fun}\ y\ x.\, (x,y)))\end{array}$$

The code of new says: allocate new locations, x, y containing s, p respectively. The allocations are done by the high allocator, so x, y are high pointers. Return the pair (x, y).

When the type system encounters the do command in the above code, it issues two proof obligations, corresponding to the predicates C_1 and C_2 from the typing rule of do in Section 3.2. The first obligation requires showing that the program within do is safe, and thereby has the form:

$$\begin{array}{l}\forall i : \mathsf{heap}.\, \mathsf{safe}((\mathsf{alloc}\,@_0\, s;\\ \qquad\qquad\qquad\qquad \mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p);\\ \qquad\qquad\qquad\qquad \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@_0\,())\ i\end{array}$$

The second obligation requires showing that two runs of the program within do satisfy the provided relational postcondition. Thus:

$$\begin{array}{l}\forall i_1\ i_2 : \mathsf{heap}.\\ \quad \mathsf{verify2}\ i_1\ i_2\\ \qquad ((\mathsf{alloc}\,@_0\, s; \mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p); \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@_0\,())\\ \qquad ((\mathsf{alloc}\,@_0\, s; \mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p); \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@_0\,())\\ \qquad (\mathsf{fun}\,(yy : \mathsf{alice}^2)\ (m : \mathsf{heap}^2).\\ \qquad\quad \exists ss : \mathsf{nat}^2.\, \exists pp : \mathsf{nat}^2.\, \exists hh : \mathsf{heap}^2.\, m = (i_1, i_2) \bullet\bullet\, hh \wedge \mathsf{sshape}\ yy\ ss\ pp\ hh)\end{array}$$

Both obligations require instantiating the do-enclosed program with a unit element (), which signifies an empty explicit substitution, corresponding to the empty local context of the STsec type of new.

We now focus on discharging the safety obligation. We first apply the SBND_PUSH lemma, to push the instantiation with the empty explicit substitution inside the program. This transforms the original safety obligation into the following.

$$\begin{array}{l}\mathsf{safe}\ ((\mathsf{alloc}\,@_0\, s)\,@_0\,();\\ \qquad (\mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p);\\ \qquad\ \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@\,(\mathsf{fun}\ z.\,(z,())))\ i\end{array}$$

Now SBND_INST applies to reassociate the instantiations in the first alloc command, causing the first command to be transformed into $\mathsf{alloc}\,@_0\,(s\ ())$. In our notational convention about explicit substitutions from Section 3.2, s is isomorphic to $\mathsf{fun}\, x : \mathsf{unit}.\, s$. Hence, the application $s\ ()$ simply transforms into s, and the subgoal becomes

$$\begin{array}{l}\mathsf{safe}\ (\mathsf{alloc}\,@_0\, s;\\ \qquad (\mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p);\\ \qquad\ \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@\,(\mathsf{fun}\ z.\,(z,())))\ i\end{array}$$

Next, we apply SBND_ALLOC, to symbolically execute the first command, and reduce the goal to

$$\begin{array}{l}\forall l : \mathsf{ptr}.\, \mathsf{safe}\ (((\mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p);\\ \qquad\qquad \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@\,(\mathsf{fun}\ z.\,(z,()))\,@_0\ l)\ (\ell \mapsto s \bullet i)\end{array}$$

Picking a fresh l for the quantified variable, we can apply SVAL_INST to push in the instantiation with l. This changes the term $\mathsf{fun}\ z.\,(z,())$ into $(\mathsf{fun}\ z.\,(z,()))\ l$, which itself beta reduces to $(l,())$, transforming our goal into

$$\begin{array}{l}\mathsf{safe}((\mathsf{alloc}\,@\,(\mathsf{fun}\ x.\, p);\\ \qquad \mathsf{return}\,@\,(\mathsf{fun}\ y\ x.\,(x,y)))\,@_0\,(l,()))\ (\ell \mapsto s \bullet i)\end{array}$$

After this we apply SBND_PUSH to push the instantiation with $(l,())$ into the sequential composition, and so on. We can continue applying such lemmas till the proof obligation for

safety is discharged. The actual explicit proof carried out in Coq looks like

$$\begin{array}{l}\mathsf{move} \Rightarrow i.\\ \mathsf{apply:}\ \mathsf{sbnd\_push};\ \mathsf{apply:}\ \mathsf{sbnd\_inst};\ \mathsf{apply:}\ \mathsf{sbnd\_alloc} \Rightarrow l.\\ \mathsf{apply:}\ \mathsf{sval\_inst};\ \mathsf{apply:}\ \mathsf{sbnd\_push};\ \mathsf{apply:}\ \mathsf{sbnd\_inst};\ \mathsf{apply:}\ \mathsf{sbnd\_alloc} \Rightarrow l'.\\ \mathsf{apply:}\ \mathsf{sval\_inst};\ \mathsf{apply:}\ \mathsf{sval\_inst};\ \mathsf{apply:}\ \mathsf{sval\_ret}.\end{array}$$

At each point in the proof, there is only one lemma that can apply. Hence the proof can be constructed fully automatically. We have implemented an automated procedure sstep, to do just that, simplifying the above proof to:

$$\mathsf{move} \Rightarrow i;\ \mathsf{do}\ ![\mathsf{sstep} \Rightarrow ?].$$

To discharge the subgoal corresponding to the verify2 obligation, we must show that with a pair of input heaps i_1, i_2, and two copies of the code (inside do), the postcondition is satisfied. First we massage the subgoal using applications of several lemmas for pushing explicit substitutions into the programs. (We have another automation routine, called vstep, which applies the lemmas as appropriate.) At the end of the execution of vstep, we will have variables l_1 and l_2 for the pointers storing the salaries in the two runs, and l_1' and l_2' for the pointers storing the passwords in the two runs. The subgoal looks as follows.

$$\begin{array}{l}\exists ss : \mathsf{nat}^2.\ \exists pp : \mathsf{nat}^2.\ \exists hh : \mathsf{heap}^2.\\ \quad (l_1' \mapsto p \bullet (l_1 \mapsto s \bullet i_1),\ l_2' \mapsto p \bullet (l_2 \mapsto s \bullet i_2)) = (i_1, i_2)\ \bullet\bullet\ hh \wedge\\ \quad \mathsf{sshape}\ ((l_1, l_1'), (l_2, l_2'))\ ss\ pp\ hh\end{array}$$

Now we instantiate the existentially bound variables ss, pp with (s, s) and (p, p) respectively, so that the proof obligation becomes

$$\begin{array}{l}\exists hh : \mathsf{heap}^2.\\ \quad (l_1' \mapsto p \bullet (l_1 \mapsto s \bullet i_1),\ l_2' \mapsto p \bullet (l_2 \mapsto s \bullet i_2)) = (i_1, i_2)\ \bullet\bullet\ hh \wedge\\ \quad \mathsf{sshape}\ ((l_1, l_1'), (l_2, l_2'))\ (s, s)\ (p, p)\ hh\end{array}$$

Now we use associative and commutative laws to rewrite the left-hand side of the equality above so that the proof obligation becomes

$$\begin{array}{l}\exists hh : \mathsf{heap}^2.\\ \quad (i_1 \bullet (l_1' \mapsto p \bullet l_1 \mapsto s),\ i_2 \bullet (l_2' \mapsto p \bullet l_2 \mapsto s)) = (i_1, i_2)\ \bullet\bullet\ hh \wedge\\ \quad \mathsf{sshape}\ ((l_1, l_1'), (l_2, l_2'))\ (s, s)\ (p, p)\ hh\end{array}$$

Next, we let the system pick logical variables to instantiate the existential with; let us call these variables α and β – they will be instantiated with concrete values further below. Now there are two subgoals that need to be discharged corresponding to the two conjuncts. They are:

$$\begin{array}{l}(i_1 \bullet (l_1' \mapsto p \bullet l_1 \mapsto s),\ i_2 \bullet (l_2' \mapsto p \bullet l_2 \mapsto s)) = (i_1, i_2)\ \bullet\bullet\ (\alpha, \beta) \text{ and}\\ \mathsf{sshape}\ ((l_1, l_1'), (l_2, l_2'))\ (s, s)\ (p, p)\ (\alpha, \beta)\end{array}$$

The first conjunct is discharged by noting that

$$(i_1, i_2)\ \bullet\bullet\ (\alpha, \beta) = i_1 \bullet \alpha, i_2 \bullet \beta$$

so that there are unique solutions for α and β:

$$\alpha\ =\ l_1' \mapsto p\ \bullet\ l_1 \mapsto s \text{ and } \beta\ =\ l_2' \mapsto p\ \bullet\ l_2 \mapsto s.$$

The second subgoal is easy to discharge by appealing to the definition of sshape (see Figure 2). Our actual proof in Coq is the following.

$$\begin{array}{l}\mathsf{move} \Rightarrow i_1\ i_2;\ \mathsf{vstep} \Rightarrow l_1\ l_2;\ \mathsf{vstep} \Rightarrow l_1'\ l_2';\ \mathsf{vstep}.\\ \mathsf{rewrite}\ -(\mathsf{unC}\ i_1)\ -(\mathsf{unCA}\ i_1)\ -(\mathsf{unC}\ i_2)\ -(\mathsf{unCA}\ i_2)\ /\mathsf{plus2}.\\ \mathsf{eexists}\ (s, s), (p, p), (\_, \_);\ \mathsf{do}\ !\mathsf{split} \Rightarrow /=; \mathsf{rewrite}\ \mathsf{unC}.\end{array}$$

```
linked_list : type
shape : linked_list → list T → heap → prop
sshape (pp : linked_list²) (ww : (list T)²) (ii : heap²) ≙
  shape p₁ w₁ i₁ ∧ shape p₂ w₂ i₂
low_links : linked_list² → heap² → prop

new : STsec nil linked_list
  (fun i. True,
   fun pp ii mm. ∃jj. mm = jj •• ii ∧
    sshape pp (nil, nil) jj ∧ (i₁ ≅ i₂ → low_links pp jj))

insert : STsec [linked_list, T] unit
  (fun p v i. ∃h j w. i = j • h ∧ shape p w j,
   fun pp vv yy ii mm. ∀hh jj ww.
     ii = jj •• hh → sshape pp ww jj →
   ∃jj'. mm = jj' •• hh ∧
     sshape pp (v₁ :: w₁, v₂ :: w₂) jj' ∧
     (low_links pp jj → h₁ ≅ h₂ → low_links pp jj'))

remove : STsec [linked_list] (option T)
  (fun p i. ∃h j w. i = j • h ∧ shape p w j,
   fun pp yy ii mm. ∀hh jj ww.
     ii = jj •• hh → sshape pp ww jj →
   ∃jj'. mm = jj' •• hh ∧
       sshape pp (tail w₁, tail w₂) jj' ∧
       yy = (if w₁ is v₁::_ then some v₁ else none,
             if w₂ is v₂::_ then some v₂ else none) ∧
       (low_links pp jj → low_links pp jj'))
```

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

5. 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 features are present in RHTT, and used in 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 $p\ w\ i$, capturing that the heap i stores a valid *singly-linked list* whose content is the mathematical (i.e., purely-functional) sequence w 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 w which recurses over the contents w 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 ww 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 ww. The latter may be heterogeneous; that is, some elements of ww 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, i_1 and i_2. (The formal definition of low_links is elided here but appears in file ¡¡¡¡¡¡¡ .mine llist3.v of the Coq scripts.) ======= llist3.v of the Coq scripts.) ¿¿¿¿¿¿¿ .r1732

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 j of these methods ¡¡¡¡¡¡¡ .mine store valid linked lists. ======= store valid linked lists. ¿¿¿¿¿¿¿ .r1732 The sshape predicate in the post-conditions 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 $i_1 \cong i_2$. Similarly, insert specifies that low_links pp $jj \to h_1 \cong h_2 \to$ 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 e. p is made to point to next e, before hd is deallocated. ¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r1732

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 w i, in case x is null (then the whole list is empty: note that empty denotes the empty heap), and non-null (then x points to the head).

$$\begin{aligned}&\mathsf{lseq\_null} : \forall w\ i.\, \mathsf{lseq}\ \mathsf{null}\ w\ i \to w = \mathsf{nil} \wedge i = \mathsf{empty}\\ &\mathsf{lseq\_pos} : \forall w\ x\ i.\, x \neq \mathsf{null} \to \mathsf{lseq}\ x\ w\ i \to\\ &\qquad \exists\ z\ j.\, i = x \mapsto \mathsf{node}\ (\mathsf{head}\ w, z) \bullet j \wedge \mathsf{lseq}\ z\ (\mathsf{tail}\ w)\ j\end{aligned}$$

For low_links, we show that if two heaps store lists with low linkage *and equal contents*, then the heaps themselves are equal.

$$\begin{aligned}\mathsf{low\_linkR} : \forall w\ pp\ ii.\mathsf{sshape}\ pp\ (w, w)\ ii \to\\ \mathsf{low\_links}\ pp\ ii \to i_1 = i_2\end{aligned}$$

Example 5.1 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.

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

lseq (x : ptr) (w : list T) : heap → prop ≙
  if w 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

shape (p : linked_list) (w : list T) (i : heap) ≙
  ∃x:ptr. j:heap. i = p ↦ x • j ∧ lseq x w 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 e hd p. (p, next e));
        dealloc @ (fun _ e hd p. hd);
        return @ (fun _ _ e hd p. some (elem e)))
```

Fig. 6. Module List: implementation of singly-linked lists (excerpts). empty denotes the empty heap.

This is described in the postcondition by conditionals over the values of b in the two different runs (b_1 and b_2). Irrespective of the contents, the ending *linkage* is low, assuming we started with low-equivalent input heaps.

Example 5.2 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 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 except Beringer [2010] which considers relational reasoning about information flow for bytecode. In the dynamic setting, a recent example is the work of Russo et al. [2009], 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.

```
P5 : STsec [bool] linked_list
       (fun b i. True,
        fun bb yy ii mm. ∃jj. mm = jj •• ii ∧
          sshape yy ([if b1 then 1 else 2, 0],
                     [if b2 then 1 else 2, 0]) jj ∧
          (i1 ≅ i2 → 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 b1 then [0] else [1, 0],
                     if b2 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.

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 module A from Section 2.

Example 5.3 In Example 2.6, A ratifies the client program (such as B.compute_tax) which may keep local state, as long as its final state does not steal A's salary. Now we instantiate B's local state to a linked list, which dynamically grows as various instances of A executes the client 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 newly allocated nodes during client execution will be low only if they can be generated in low-equivalent heaps. To express this low equivalence the specification of epost used in ratify's specification must change as emphasized below.

$$
\begin{array}{l}
\mathsf{epost}\ aa\ bb\ jj\ yy\ ii\ mm\ \widehat{=} \\
\quad \forall ss\ pp\ kk\ tt\ hh. \\
\qquad ii = jj \bullet\bullet tt \bullet\bullet hh \to \mathsf{sshape}\ aa\ ss\ pp\ jj \to \\
\qquad\quad \mathsf{bbshape}\ bb\ kk\ tt \to \\
\quad \exists jj'\ tt'. mm = jj' \bullet\bullet tt' \bullet\bullet hh \wedge \mathsf{sshape}\ aa\ ss\ pp\ jj' \wedge \\
\qquad\quad \mathsf{bbshape}\ bb\ (\mathsf{bcmp}\ k_1, \mathsf{bcmp}\ k_2)\ tt' \wedge \\
\qquad\quad \boxed{j_1 \bullet h_1 \cong j_2 \bullet h_2} \to \\
\qquad\quad t_1 = t_2 \to t_1' = t_2'
\end{array}
$$

The client can now be granted access to A's salary and can keep the count in a linked list. For example, the implementation below defines B's local state as a linked list which counts

the number of times the client program has been called by linking in new nodes into its 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 salary.

$$
\begin{array}{l}
\beta \;\widehat{=}\; \mathsf{linked\_list} \\
\mathsf{G} \;\widehat{=}\; \mathsf{list\ nat} \\
\mathsf{bbshape}\ (bb : \beta^2)\ (kk : G^2)\ (ii : \mathsf{heap}^2) \;\widehat{=}\; \\
\quad \mathsf{List.sshape}\ bb\ kk\ ii\ \wedge b_1 = b_2 \\
\mathsf{bcmp} : G \to G \;\widehat{=}\; \mathsf{fun}\ k.\, 1 :: k
\end{array}
$$

The client program, which reads salary, allocates a new node in its list, and then returns the computed tax for the salary, can then be created and ratified as follows.

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

6. DISCUSSION

Small vs. large footprint specifications. In one of the previous iterations of the system, we did have small footprint specifications, before we realized that deterministic allocation is important. In that iteration of the system, the method new from Figure 1, had the type

$$\mathsf{nat} \to \mathsf{string} \to \mathsf{STsec\ nil\ alice}\ (\mathsf{fun}\, i.\, i = \mathsf{empty}, \mathsf{fun}\, aa\ ii\ mm.\, \exists ss\ pp.\ \ \mathsf{sshape}\ aa\ ss\ pp\ mm)$$

When we switched to large footprint specification, this became:

$$
\begin{array}{l}
\mathsf{nat} \to \mathsf{string} \to \mathsf{STsec\ nil\ alice} \\
\qquad\qquad (\mathsf{fun}\, i.\, \mathsf{True}, \mathsf{fun}\, aa\ ii\ mm.\, \exists ss\ pp\ hh.\, mm = ii \bullet\!\!\bullet hh \wedge \mathsf{sshape}\ aa\ ss\ pp\ hh)
\end{array}
$$

¡¡¡¡¡¡¡ .mine In other words, one more existentially quantified variable (hh) and one more assertion expressing disjointness between the initial and allocated heap ($mm = ii \bullet\!\!\bullet hh$). The $\mathsf{fun}\, i.\, i = \mathsf{empty}$ in the precondition gets converted to $\mathsf{fun}\, i.\, \exists h.\, i = \mathsf{empty} \bullet h$ in the large footprint specification, which then yields $\mathsf{fun}\, i.\, \mathsf{True}$. (*AB: Please check.*) ✓

In other examples, where the preconditions were not so simple as here, we needed one more variable and one more assertion for the precondition too. As parts of the precondition sometimes have to appear in the postcondition, for the purposes of correct scoping, this may add one more variable and one more assertion in the postcondition as well. Altogether, at most three new variables, and three more disjointness assertions between those variables. This pattern applied throughout the development and caused minimal refactoring on our already written proofs. ======= In other words, the postcondition is extended with one more existentially quantified variable (hh) and one more assertion expressing disjointness between the initial and allocated heap ($mm = ii \bullet\!\!\bullet hh$). In other examples, where the preconditions were not simply True as here, we needed one more variable and one more equation for the precondition too. As parts of the precondition sometimes have to appear in the postcondition, for the purposes of correct scoping, this may add one more variable and one more equation in the postcondition as well. Altogether, at most three new variables, and three more disjointness equations between those variables. This pattern applied throughout the development and caused minimal refactoring on our already written proofs. ¿¿¿¿¿¿¿ .r1732

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 3, 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. In particular, the programmer-supplied annotations are also in ¡¡¡¡¡¡¡ .mine higher-order logic, and therefore can be as expressive as needed. ======= higher-order logic, and therefore can be as expressive as needed. ¿¿¿¿¿¿¿ .r1732

Non-Interference for finite security lattices. Consider variables $x : H$, $y : M$ and $z : L$ where H represents high, M represents medium and L represents low in a security lattice with $L \leq M \leq H$. The security policy is that (a) z is independent of both x and y and (b) y is independent of x. We can encode this policy in our setting, with e.g., zz representing the pair of input values of z in the two runs and zz' representing the pair of output values, as the conjunction of

$$z_1 = z_2 \rightarrow z_1' = z_2' \quad \text{(a)}$$
$$z_1 = z_2 \rightarrow y_1 = y_2 \rightarrow y_1' = y_2' \quad \text{(b)}$$

Instead of a conjunct for each level L, M, of the security lattice, we can nest the above to obtain the equivalent

$$z_1 = z_2 \rightarrow$$
$$(z_1' = z_2' \wedge (y_1 = y_2 \rightarrow y_1' = y_2'))$$

This encoding can be generalized to express non-interference like properties on contents of variables drawn from an arbitrary finite security lattice. However, RHTT cannot be used to reason about non-interference properties of pointer addresses if the security lattice contains more than two elements (i.e., low and high) *and* dynamic allocation is used. This is because reasoning about non-interference of pointer addresses requires separate allocators for each level of the lattice, with the property that the pointers returned by an allocator call at a level l be influenced only by prior allocations at levels lower than l. In its current form, RHTT has only two allocators, so it can be used to reason about two levels of pointer addresses only. Given any finite security lattice, it seems straightforward to design a different version of RHTT with enough allocators to reason about non-interference of pointer addresses in that lattice, but we do not know of a method to combine more than one lattice within a single version of RHTT.

Ultimately the requirement of two allocators is not fully satisfactory because it goes against our desire to prevent security levels from ¡¡¡¡¡¡¡ .mine permeating specifications. The crux of the issue is the non-determinism of dynamic allocation. This non-determinism is not expressible in the currently implemented model of RHTT and in the specifications. Thus an output of a fully non-deterministic allocator cannot be considered low. A proper account of non-determinism is necessary, however, for enforcing lack of leakage in even slightly non-trivial dynamic data structures such as binary search trees. In such structures, an innocuous move such as rebalancing the tree upon a certain condition may leak information about some data, unless randomization is introduced to mask the condition so that the structure is balanced at random times. We leave such exploration to future work. (*AB: Aleks please take care of this para.*) ======= permeating specifications. The crux of the ✓ issue is the randomness of dynamic allocation. This randomness cannot be captured in the current RHTT specifications, where we only describe the properties of final values in two runs, rather than final sets of values, or final *probability distributions*. In a setting where

specifications can describe probability distributions, pointers returned by an allocator in two runs may be considered low if they are chosen at random from two equal distributions, even if the pointers themselves may be different. ¿¿¿¿¿¿¿ .r1732

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 2.3 and 2.4, 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 2.2, 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.

7. RELATED WORK

¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r1732 Banerjee et al. [2008] 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) [Sabelfeld and Sands 2009]. We extend the ideas of Banerjee et al. [2008] 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 [Li and Zdancewic 2010; Russo et al. 2008] 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 [Swamy et al. 2010], Fable [Swamy et al. 2008], FX [Borgström et al. 2011], Aglet [Morgenstern and Licata 2010], F7 [Bengtson et al. 2011] and Aura [Jia et al. 2008]. They all support some, but not all features that we provide in RHTT.

In Swamy et al.'s purely functional programming language Fine [Swamy et al. 2010], 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 [Swamy et al. 2008], 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 F* [Swamy et al. 2011] combines Fine's affine types with F7's path-sensitive assertions and automatic discharge of verification conditions using the SMT solver Z3. Verification of information flow properties in F* is similar to that in Fine and differs from our work in ways described above. Like F7, the use of path-sensitive assertions allows verification of some trace properties, which are beyond the scope of RHTT. Use of an SMT solver allows for automatic discharge of verification conditions and the authors report success with approximately 20,000 lines of code, but it is unclear how much additional manual work is needed to guide the SMT solver.

The language FX [Borgström et al. 2011] 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 non-interference, 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 non-interference 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 [Morgenstern and Licata 2010], for enforcement of state-dependent access control policies. Aglet is an extension of Agda [Norell 2007] 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. [2011] 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 [Jia et al. 2008], PCML_5 [Avijit et al. 2010], and PCAL [Chaudhuri and Garg 2009], based on the proof-as-authorization paradigm [Appel and Felten 1999], 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 non-interference can be encoded [Jia and Zdancewic 2009], 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 [Broberg and Sands 2010] 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 [Askarov and Sabelfeld 2007], 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 and Ynot [Nanevski et al. 2008] with the ability to reason relationally about security. The addition caused significant changes in the semantics and the usage of the language. For example, because of the reasoning about two-runs, the modeling of STsec deals with CPO's, whereas for HTT complete lattices sufficed. The logic for discharging verification conditions in RHTT has to reason separately about safety and correctness, whereas in HTT, safety and correctness could be captured in one judgment, corresponding to the separation logic triple. In RHTT we use large footprint specification and combinator syntax, whereas HTT used small footprints for programming monadically. On the other hand, we were able to reuse from HTT the library for reasoning about heap disjointness, to keep RHTT proofs relatively short.

8. 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.

Beyond what has been achieved in this paper much remains to be done. While we can specify expressive program properties, and verify that programs comply with them, it is currently impossible to reason about specifications themselves. For example, we cannot reason that indirect inference (say using aggregation operators such as average) do not lead to unwanted leaks. For that, one might need to reason about quantitative information flow and apply knowledge-based reasoning [Fagin et al. 1995].

Currently, RHTT does not support reasoning about trace-based, ¡¡¡¡¡¡¡ .mine temporal properties. (Thus RHTT cannot express arbitrary hyperproperties [Clarkson and Schneider 2010] over a pair of traces. RHTT's model only relates states at the beginning and end of a trace.) ======= temporal properties. (Thus RHTT cannot express arbitrary hyperproperties [Clarkson and Schneider 2010] over a pair of traces. RHTT's model only relates states at the beginning and end of a trace.) ¿¿¿¿¿¿¿ .r1732 For example, while it is intuitively clear that our specification of functions grant, revoke, read_salary (Example 2.5) 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. The type systems of F7 and its successors have taken steps towards proving trace properties like injective and non-injective correspondence assertions in the context of message passing concurrency. We intend to extend RHTT for trace-based properties in the context of reactive, non-deterministic and concurrent ¡¡¡¡¡¡¡ .mine higher-order languages. ======= higher-order languages. ¿¿¿¿¿¿¿ .r1732

A related property is parametricity of the type system of Coq – that opaque sealing does not divulge the actual implementation of the sealed data. This property has recently been proved for a class of pure type systems (including the calculus of constructions) by Bernardy et al. [2012]. The closest related proofs in the imperative world of which we are aware, are the recent ones for ML with references [Ahmed et al. 2009] and for separation logic [Birkedal and Yang 2008; Thamsborg et al. 2012]. The parametricity property can even be internalized into a pure type system [Bernardy and Moulin 2012], enabling a form of reasoning similar to the logic of Plotkin and Abadi [1993]. We intend to investigate in the future if and how this proof can be extended to mutable state and our STsec types.

We also intend to investigate the use of relations other than equality, such as distance metrics and continuity (small changes to the distance of inputs cause small changes to the distance of outputs), in the postconditions of STsec types. Such definitions may be particularly useful in the context of differential privacy guarantees [Dwork et al. 2006; Reed and Pierce 2010].

ACKNOWLEDGMENTS

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. We thank the anonymous referees for their detailed comments on 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 Project 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

Abadi, M., Banerjee, A., Heintze, N., and Riecke, J. G. 1999. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages (POPL).*

147–160.

Ahmed, A., Dreyer, D., and Rossberg, A. 2009. State-dependent representation independence. In *ACM Symposium on Principles of Programming Languages (POPL)*. 340–353.

Amtoft, T., Bandhakavi, S., and Banerjee, A. 2006. A logic for information flow in object-oriented programs. In *ACM Symposium on Principles of Programming Languages (POPL)*. 91–102.

Appel, A. W. and Felten, E. W. 1999. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security (CCS)*. 52–62.

Askarov, A. and Myers, A. 2010. A semantic framework for declassification and endorsement. In *European Symposium on Programming (ESOP)*. 64–84.

Askarov, A. and Sabelfeld, A. 2007. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy (S&P)*. 207–221.

Austin, T. H. and Flanagan, C. 2010. Permissive dynamic information flow analysis. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 3:1–3:12.

Avijit, K., Datta, A., and Harper, R. 2010. Distributed programming with distributed authorization. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. 27–38.

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

Banerjee, A., Naumann, D. A., and Rosenberg, S. 2008. Expressive declassification policies and their modular static enforcement. In *IEEE Symposium on Security and Privacy (S&P)*. 339–353.

Barthe, G., D'Argenio, P. R., and Rezk, T. 2004. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop (CSFW)*. 100–114.

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

Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., and Maffeis, S. 2011. Refinement types for secure implementations. *TOPLAS 33,* 2, 8:1–8:45.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symposium on Principles of Programming Languages (POPL)*. 14–25.

Beringer, L. 2010. Relational bytecode correlations. *J. Log. Algebr. Program. 79,* 7, 483–514.

Bernardy, J.-P., Jansson, P., and Paterson, R. 2012. Proofs for free — parametricity for dependent types. *JFP 22,* 2, 107–152.

Bernardy, J.-P. and Moulin, G. 2012. A computational interpretation of parametricity. In *IEEE Symp. on Logic in Computer Science (LICS)*. 135–144.

Birgisson, A., Hedin, D., and Sabelfeld, A. 2012. Boosting the permissiveness of dynamic information-flow tracking by testing. In *European Symposium on Research in Computer Security (ESORICS)*. 55–72.

Birkedal, L. and Yang, H. 2008. Relational parametricity and separation logic. *Logical Methods in Computer Science 4,* 2:6, 1–27.

Borgström, J., Chen, J., and Swamy, N. 2011. Verifying stateful programs with substructural state and Hoare types. In *ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*. 15–26.

Borgström, J., Gordon, A. D., and Pucella, R. 2011. Roles, stacks, histories: A triple for Hoare. *Journal of Functional Programming 21,* 2, 159–207.

Broberg, N. and Sands, D. 2010. Paralocks: role-based information flow control and beyond. In *ACM Symposium on Principles of Programming Languages (POPL)*. 431–444.

Chaudhuri, A. and Garg, D. 2009. PCAL: Language support for proof-carrying autho-

rization systems. In *European Symposium on Research in Computer Security (ESORICS)*. 184–199.

Chong, S. and Myers, A. C. 2004. Security policies for downgrading. In *ACM Conference on Computer and Communications Security (CCS)*. 198–209.

Chong, S. and Myers, A. C. 2005. Language-based information erasure. In *IEEE Computer Security Foundations Workshop (CSFW)*. 241–254.

Chong, S. and Myers, A. C. 2008. End-to-end enforcement of erasure and declassification. In *IEEE Computer Security Foundations Symposium (CSF)*. 98–111.

Clarkson, M. R. and Schneider, F. B. 2010. Hyperproperties. *Journal of Computer Security 18,* 6, 1157–1210.

de Roever, W.-P. and Engelhardt, K. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press.

Denning, D. 1976. A lattice model of secure information flow. *Commun. ACM 19,* 5, 236–242.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of program. *Commun. ACM 18,* 8, 453–457.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)*. 265–284.

Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. Y. 1995. *Reasoning About Knowledge*. MIT Press.

Goguen, J. and Meseguer, J. 1982. Security policies and security models. In *IEEE Symposium on Security and Privacy (S&P)*. 11–20.

Gries, D. 1993. Data refinement and the transform. In *Program Design Calculi*, M. Broy, Ed. Springer. International Summer School at Marktoberdorf.

Harper, R. and Lillibridge, M. 1994. A type-theoretic approach to higher-order modules with sharing. In *ACM Symposium on Principles of Programming Languages (POPL)*. 123–137.

Jia, L., Vaughan, J. A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., and Zdancewic, S. 2008. AURA: A programming language for authorization and audit. In *International Conference on Functional Programming (ICFP)*. 27–38.

Jia, L. and Zdancewic, S. 2009. Encoding information flow in Aura. In *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*. 17–29.

Leroy, X. 1994. Manifest types, modules, and separate compilation. In *ACM Symposium on Principles of Programming Languages (POPL)*. 109–122.

Li, P. and Zdancewic, S. 2010. Arrows for secure information flow. *Theor. Comput. Sci. 411,* 19, 1974–1994.

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

The Coq development team. 2009. *The Coq proof assistant reference manual*. LogiCal project, INRIA. Version 8.2.

Mitchell, J. C. and Plotkin, G. D. 1988. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10,* 3, 470–502.

Morgenstern, J. and Licata, D. 2010. Security-typed programming within dependently-typed programming. In *International Conference on Functional Programming (ICFP)*. 169–180.

Myers, A. C. 1999. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*. 228–241.

Nanevski, A., Banerjee, A., and Garg, D. 2011. Verification of information flow and access control policies via dependent types. In *IEEE Symposium on Security and Privacy (S&P)*. 165–179.

Nanevski, A., Morrisett, J. G., and Birkedal, L. 2008. Hoare type theory, polymorphism and separation. *Journal of Functional Programming 18,* 5-6, 865–911.

Nanevski, A., Vafeiadis, V., and Berdine, J. 2010. Structuring the verification of heap-manipulating programs. In *ACM Symposium on Principles of Programming Languages (POPL)*. 261–274.

Norell, U. 2007. Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology.

Peyton Jones, S. L. and Wadler, P. 1993. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*. 71–84.

Plotkin, G. D. and Abadi, M. 1993. A logic for parametric polymorphism. In *Typed Lambda Calculus and Applications (TLCA)*. 361–375.

Reed, J. and Pierce, B. C. 2010. Distance makes the types grow stronger. In *International Conference on Functional Programming (ICFP)*. 157–168.

Reynolds, J. C. 1981. *The Craft of Programming*. Prentice-Hall.

Reynolds, J. C. 2002. Separation logic: a logic for shared mutable data structures. In *IEEE Symp. on Logic in Computer Science (LICS)*. 55–74.

Russo, A., Claessen, K., and Hughes, J. 2008. A library for light-weight information-flow security in Haskell. In *Haskell Symposium*. 13–24.

Russo, A., Sabelfeld, A., and Chudnov, A. 2009. Tracking information flow in dynamic tree structures. In *European Symposium on Research in Computer Security (ESORICS)*. 86–103.

Sabelfeld, A. and Sands, D. 1999. A PER model of secure information flow in sequential programs. In *European Symposium on Programming (ESOP)*. 40–58.

Sabelfeld, A. and Sands, D. 2009. Declassification: Dimensions and principles. *Journal of Computer Security 17,* 5, 517–548.

Simonet, V. 2002. Fine-grained information flow analysis for a λ-calculus with sum types. In *IEEE Computer Security Foundations Workshop (CSFW)*. 223–237.

Swamy, N., Chen, J., and Chugh, R. 2010. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming (ESOP)*. 529–549.

Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., and Yang, J. 2011. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming (ICFP)*. 266–278.

Swamy, N., Corcoran, B. J., and Hicks, M. 2008. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy (S&P)*. 369–383. Full version: Technical report CS-TR-4895, Univ. Maryland.

Swamy, N., Hicks, M., Tse, S., and Zdancewic, S. 2006. Managing policy updates in security-typed languages. In *IEEE Computer Security Foundations Workshop (CSFW)*. 202–216.

Terauchi, T. and Aiken, A. 2005. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*. 352–367.

Thamsborg, J., Birkedal, L., and Yang, H. 2012. Two for the price of one: Lifting separation logic assertions. *Logical Methods in Computer Science 8,* 3.

Volpano, D. M., Irvine, C. E., and Smith, G. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4,* 2/3, 167–188.

Yang, H. 2007. Relational separation logic. *Theor. Comput. Sci. 375*, 308–334.

Yang, H. and O'Hearn, P. W. 2002. A semantic basis for local reasoning. In *Foundations of Software Science and Computational Structures (FoSSaCS)*. 402–416.

Received Month Year; revised Month Year; accepted Month Year