A Monadic Framework for Relational Verification

Applied to Information Security, Program Equivalence, and Optimizations

Niklas Grimm[1] Kenji Maillard[2,3] Cédric Fournet[4] Cătălin Hriţcu[2] Matteo Maffei[1] Jonathan Protzenko[4]
Tahina Ramananandro[4] Aseem Rastogi[4] Nikhil Swamy[4] Santiago Zanella-Béguelin[4]

[1]Vienna University of Technology [2]Inria Paris [3]ENS Paris [4]Microsoft Research

Abstract

Relational properties describe multiple runs of one or more programs. They characterize many useful notions of security, program refinement, and equivalence for programs with diverse computational effects, and they have received much attention in the recent literature. Rather than developing separate tools for special classes of effects and relational properties, we advocate using a general purpose proof assistant as a unifying framework for the relational verification of effectful programs. The essence of our approach is to model effectful computations using monads and to prove relational properties on their monadic representations, making the most of existing support for reasoning about pure programs.

We apply this method in F⋆ and evaluate it by encoding a variety of relational program analyses, including information flow control, semantic declassification, program equivalence and refinement at higher order, correctness of program optimizations. By relying on SMT-based automation, unary weakest preconditions, user-defined effects, and monadic reification, we show that, compared to unary properties, verifying relational properties requires little additional effort from the F⋆ programmer.

1 Introduction

Generalizing unary properties (which describe single runs of programs), *relational* properties describe multiple runs of one or more programs. Relational properties are useful when reasoning about program refinement, approximation, equivalence, provenance, as well as many notions of security. A great many relational program analyses have been proposed in the recent literature, including works by Antonopoulos et al. (2017); Asada et al. (2016); Banerjee et al. (2016); Barthe et al. (2012, 2013b, 2014, 2015); Benton et al. (2009); Ştefan Ciobâcă et al. (2016); Godlin and Strichman (2010); Hedin and Sabelfeld (2012); Kundu et al. (2009); Küsters et al. (2015); Yang (2007); Zaks and Pnueli (2008); Murray et al. (2013); Fehrenbach and Cheney (2016); Bauereiß et al. (2016, 2017); and Çiçek et al. (2017). While some systems have been designed for the efficient verification of specialized relational properties of programs (notably information-flow type systems, e.g., Sabelfeld and Myers (2003a)), others support larger classes of properties. These include tools based on product program constructions for automatically proving relations between first-order imperative programs (e.g., SymDiff (Lahiri et al. 2012) and Descartes (Sousa and Dillig 2016)), as well as relational program logics (Benton 2004) that support interactive verification of relational properties within proof assistants (e.g., EasyCrypt (Barthe et al. 2012) and RHTT (Nanevski et al. 2013)).

We provide a framework in which relational logics and other special-purpose tools can be recast on top of a general method for relational reasoning. The method is simple: we use monads to model and program effectful computations; and we reveal the pure monadic representation of an effect in support of specification and proof. Hence, we reduce the problem of relating effectful computations to relating their pure representations, and then apply the advanced tools available for reasoning about pure programs.

While this method should be usable for a variety of proof assistants, we choose to work in F⋆ (Swamy et al. 2016), a dependently typed programming language and proof assistant. By relying on its support for SMT-based automation, unary weakest preconditions, and user-defined effects (Ahman et al. 2017), we demonstrate, through a diverse set of examples, that our approach enables the effective verification of relational properties with an effort comparable to proofs of unary properties in F⋆ and to proofs in relational logics with SMT-based automation.

Being based on an expressive semantic foundation, our approach can be directly used to verify relational properties of programs. Additionally, we can still benefit from more specialized automated proof procedures, such as syntax-directed relational type systems, by encoding them within our framework. Hence, our approach facilitates comparing and composing special-purpose relational analyses with more general-purpose semi-interactive proofs; and it encourages prototyping and experimenting with special-purpose analyses with a path towards their certified implementations.

1.1 A first example

We sketch the main ideas on a proof of equivalence for the two stateful, recursive functions below, a task not easily accomplished using specialized relational program logics:

```
let rec sum_up r lo hi = if lo≠hi then (r := !r+lo; sum_up r (lo+1) hi)
let rec sum_dn r lo hi = if lo≠hi then (r := !r+hi−1; sum_dn r lo (hi−1))
```

Both functions sum all numbers between lo and hi into some accumulator reference r, the former function by counting up and the latter function by counting down.

Unary reasoning about monadic computations As a first step, we embed these computations within a dependently typed language. There are many proposals for how to do this—one straightforward approach is to encapsulate effectful computations within a parameterized monad (Atkey 2009). In F*, as in the original Hoare Type Theory (Nanevski et al. 2008), these monads are indexed by a computation's pre- and postconditions and proofs are conducted using a unary program logic (i.e., not relational), adapted for use with higher-order, dependently typed programs. Beyond state, F* supports reasoning about unary properties of a wide class of user-defined monadic effects, where the monad can be chosen to best suit the intended style of unary proof.

Relating reified effectful terms Our goal is to conveniently state and prove properties that relate effectful terms, e.g., prove sum_up and sum_dn equivalent. We do so by revealing the monadic representation of these two computations as pure state-passing functions. However, since doing this naïvely would preclude the efficient implementation of primitive effects, such as state in terms of a primitive heap, our general method relies on an explicit *monadic reification* coercion for exposing the pure monadic representation of an effectful computation in support of relational reasoning. Thus, in order to relate effectful terms, one simply reasons about their pure reifications. Turning to our example, we prove the following lemma, stating that running sum_up and sum_dn in the same initial states produces equivalent final states. (A proof is given in §2.4.)

```
r:ref int → lo:int → hi:int{hi ≥ lo} → h:heap{r ∈ h} →
  reify (sum_up r lo hi) h ∼ reify (sum_dn r lo hi) h
```

Flexible specification and proving style with SMT-backed automation Although seemingly simple, proving sum_up and sum_dn equivalent is cumbersome, if at all possible, in most prior relational program logics. Prior relational logics rely on common syntactic structure and control flow between multiple programs to facilitate the analysis. To reason about transformations such as loop reversal, rules that exploit syntactic similarity are not very useful and instead a typical proof in prior systems may involve several indirections, e.g., first proving the full functional correctness of each loop with respect to a purely functional specification and then showing that the two specifications are equivalent. Through monadic reification, effectful terms are *self-specifying*, removing the need to rewrite the same code in purely-functional style just to enable specification and reasoning.

Further, whereas many prior systems are specialized to proving binary relations, it can be convenient to structure proofs using relations of a higher arity, a style naturally supported by our method. For example, a key lemma in the proof of the equivalence above is an inductive proof of a ternary relation, which states that sum_up is related to sum_up on a prefix combined with sum_dn on a suffix of the interval [lo, hi).

Last but not least, using the combination of typechecking, weakest precondition calculation, and SMT solving provided by F*, many relational proofs go through with a degree of automation comparable to existing proofs of unary properties, as highlighted by the examples in this paper.

1.2 Contributions and outline

We propose a methodology for relational verification (§2), covering both broadly applicable ingredients such as representing effects using monads and exposing their representation using monadic reification, as well as our use of specific F* features that enable proof flexibility and automation. All these ingredients are generic, i.e., none of them is specific to the verification of relational properties.

The rest of the paper is structured as a series of case studies illustrating our methodology at work. Through these examples we aim to show that our methodology enables comparing and composing various styles of relational program verification in the same system, thus taking a step towards unifying many prior strands of research. Also these examples cover a wide range of applications that, when taken together, exceed the ability of all previous tools for relational verification of which we are aware. Our examples are divided into three sections that can be read in any order, each being an independent case study:

Transformations of effectful programs (§3) We develop an extensional, semantic characterization of a stateful program's read and write effects, based on the relational approach of Benton et al. (2006). Based on these semantic read and write effects, we derive lemmas that we use to prove the correctness of common program transformations, such as swapping the order of two commands and eliminating redundant writes. Going further, we encode Benton's (2004) relational Hoare logic in our system, providing a syntax-directed proof system for relational properties as a special-purpose complement to directly reasoning about a program's effects.

Information-flow control (§4) We encode several styles of static information-flow control analyses, while accounting for declassification. Highlighting the ability to compose various proof styles in a single framework, we combine automated, type-based security analysis with SMT-backed, semantic proofs of noninterference.

Proofs of algorithmic optimizations (§5) With a few exceptions, prior relational program logics apply to first-order programs and provide incomplete proof rules that exploit syntactic similarities between the related programs. Not being bound by syntax, we prove relations of higher arities (e.g., 4-ary and 6-ary relations) between higher-order,

While this coercion is inspired by Filinski's (1994) reify operator, we only use it to reveal the pure representation of an effectful computation in support of specification and proof, whereas Filinski's main use of reification was to uniformly implement monads using continuations.

effectful programs with differing control flow by reasoning directly about their reifications. We present two larger examples: First, we show how to memoize a recursive function using McBride's (2015) partiality monad and we prove it equivalent to the original non-memoized version. Second, we implement an imperative union-find data structure, adding the classic union-by-rank and path compression optimizations in several steps and proving stepwise refinement.

From these case studies, we conclude that our method for relational reasoning about reified monadic computations is both effective and versatile. We are encouraged to continue research in this direction, aiming to place proofs of relational properties of effectful programs on an equal footing with proofs of pure programs in F⋆ as well as other proof assistants and verification tools.

The code for the examples in this paper is available at
https://github.com/FStarLang/FStar/tree/master/examples/rel
Compared to this code, the listings in the paper are edited for clarity and sometimes omit uninteresting details. The appendices describe additional case studies that we omit because of space, including cryptographic security proofs and dynamic information flow control.

2 Methodology for relational verification

In this section we review in more detail the key F⋆ features we use and how each of them contributes to our verification method for relational properties. Two of these features are general and broadly applicable: (§2.1) modeling effects using monads and keeping the effect representation abstract to support efficient implementation of primitive effects and (§2.3) using monadic reification to expose the effect representation. The remaining features are more specific to F⋆ and enable proof flexibility and automation: (§2.2) using a unary weakest precondition calculus to produce verification conditions in an expressive dependently typed logic; (§2.4) using dependent types together with pre- and postconditions to express arbitrary relational properties of reified computations; (§2.4) embedding the dependently typed logic into SMT logic to enable the SMT solver to reason by computation.

None of these generic ingredients is tailored to the verification of relational properties, and while F⋆ is currently the only verification system to provide all these ingredients in a unified package, each of them also appears in other systems. This makes us hopeful that this relational verification method can also be applied with other proof assistants (e.g., Coq, Lean, Agda, Idris, etc.), for which the automation would likely come in quite different styles.

2.1 Modeling effects using monads

At the core of F⋆ is a language of dependently typed, total functions. Function types are written x:t → Tot t' where the co-domain t' may depend on the argument x:t. Since it is the default in F⋆, we often drop the Tot annotation (except where needed for emphasis) and also the name of the formal argument when it is unnecessary, e.g., we write int → bool for _:int → Tot bool. We also write #x:t → t' to indicate that the argument x is implicitly instantiated.

Our first step is to describe effects using monads built from total functions (Moggi 1989). For instance, here is the standard monadic representation of state in F⋆ syntax.

```
type st (mem:Type) (a:Type) = mem → Tot (a ∗ mem)
```

This defines a type st indexed by types for the memory (mem) and the result (a). We use st as the representation type of a new STATE_m effect we add to F⋆, with the total qualifier enabling the termination checker for STATE_m computations.

```
total new_effect {
 STATE_m (mem:Type) : a:Type → Effect
 with repr = st mem;
    return = λ(a:Type) (x:a) (m:mem) → x, m;
    bind = λ(a b:Type) (f:st mem a) (g:a → st mem b) (m:mem) →
      let z, m' = f m in g z m';
    get = λ() (m:mem) → m, m; put = λ(m:mem) _ → (), m }
```

This defines the return and bind of this monad, and two actions: get for obtaining the current memory, and put for updating it. The new effect STATE_m is still parameterized by the type of memories, which allows us to choose a memory model best suited to the programming and verification task at hand. We often instantiate mem to heap (a map from references to their values, as in ML), obtaining the STATE effect shown below—we use other memory types in §4 and §5.

```
total new_effect STATE = STATE_m heap
```

While such monad definitions could in principle be used to directly extend the implementation of any functional language with the state effect, a practical language needs to allow keeping the representation of some effects abstract so that they are efficiently implemented primitively (Peyton Jones 2010). F⋆ uses its simple module system to keep the monadic representation of the STATE effect abstract and implements it under the hood using the ML heap, rather than state passing (and similarly for other primitive ML effects such as exceptions). Whether implemented primitively or not, the monadic definition of each effect is always the *model* used by F⋆ to reason about effectful code, both intrinsically using a (non-relational) weakest precondition calculus (§2.2) and extrinsically using monadic reification (§2.3).

For the purpose of verification, monads provide great flexibility in the modeling of effects, which enables us to express relational properties and to conduct proofs at the right level of abstraction. For instance, in §4.3 we extend a state monad with extra ghost state to track declassification, and in §5.1 we define a partiality monad for memoizing recursive functions. Moreover, since the difficulty of reasoning about effectful code is proportional to the complexity of the effect, we do not use a single full-featured monad for all code; instead we define custom monads for sub-effects and relate them using monadic lifts. For instance, we define a READER monad for computations that only read the store, lifting READER to STATE only where necessary (§4.1 provides a detailed example). While F⋆ code is always written in an ML-like direct style, the F⋆ typechecker automatically inserts binds, returns and lifts under the hood (Swamy et al. 2011).

2.2 Unary weakest preconditions for user-defined effects and intrinsic proof

For each user-defined effect, F⋆ derives a weakest precondition calculus for specifying unary properties and computing verification conditions for programs using that effect (Ahman et al. 2017). Each effect definition induces a computation type indexed by a predicate transformer describing that computation's effectful semantics.

For state, we obtain a computation type 'STATE a wp' indexed by a result type a and by wp, a predicate transformer of type (a ⟶ heap ⟶ Type) ⟶ heap ⟶ Type, mapping postconditions (relating the result and final state of the computation) to preconditions (predicates on the initial state). The types of the get and put actions of STATE are specified as:

```
val get : unit ⟶ STATE heap (λ post (h:heap) ⟶ post h h)
val put : h':heap ⟶ STATE unit (λ post (h:heap) ⟶ post () h')
```

The type of get states that, in order to prove any postcondition post of 'get ()' evaluated in state h, it suffices to prove post h h, whereas for put h' it suffices to prove post () h'. F⋆ users find it more convenient to index computations with pre- and postconditions as in HTT (Nanevski et al. 2008), or sometimes not at all, using the following abbreviations:

```
ST a (requires p) (ensures q) = STATE a (λ post h₀ ⟶
                    p h₀ ∧ (∀ (x:a) (h₁:heap). q h₀ x h₁ ⟹ post x h₁))
St a = ST a (requires (λ _ ⟶ ⊤)) (ensures (λ _ _ _ ⟶ ⊤))
```

F⋆ computes weakest preconditions generically for any effect. Intuitively, this works by putting the code into an explicit monadic form and then translating the binds, returns, actions, and lifts from the expression level to the weakest precondition level. This enables a convenient form of *intrinsic* proof in F⋆, i.e., one annotates a term with a type capturing properties of interest; F⋆ computes a weakest precondition for the term and compares it to the annotated type using a built-in subsumption rule, checked by an SMT solver.

For example, the sum_up function from §1.1 can be given the following type:

```
r:ref int ⟶ lo:nat ⟶ hi:nat{hi ≥ lo} ⟶
          ST unit (requires λh ⟶ r ∈ h) (ensures λ_ _ h ⟶ r ∈ h)
```

This is a dependent function type, for a function with three arguments r, lo, and hi returning a terminating, stateful computation. The *refinement* type hi:nat{hi ≥ lo} restricts hi to only those natural numbers greater than or equal to lo. The computation type of 'sum_up r lo hi' simply requires and ensures that its reference argument r is present in the memory. F⋆ computes a weakest precondition from the implementation of sum_up (using the types of (!) and (:=) provided by the heap memory model used by STATE) and proves that its inferred specification is subsumed by the user-provided annotation. The same type can also be given to sum_dn.

2.3 Exposing effect definitions via reification

Intrinsic proofs of effectful programs in F⋆ are inherently restricted to unary properties. Notably, pre- and postconditions are required to be pure terms, making it impossible for specifications to refer directly to effectful code, e.g., sum_up cannot directly use itself or sum_dn in its specification. To overcome this restriction, we need a way to coerce a terminating effectful computation to its underlying monadic representation which is a pure term—Filinski's (1994) monadic reification provides just that facility.

Each new effect in F⋆ induces a reify operator that exposes the representation of an effectful computation in terms of its underlying monadic representation (Ahman et al. 2017). For the STATE effect, F⋆ provides the following (derived) rule for reify, to coerce a stateful computation to a total, explicitly state-passing function of type heap ⟶ t ∗ heap. The argument and result types of reify e are refined to capture the pre- and postconditions intrinsically proved for e.

$$\frac{S;\Gamma \vdash e : \mathsf{ST}\ \mathsf{t}\ (\mathsf{requires}\ \mathsf{pre})\ (\mathsf{ensures}\ \mathsf{post})}{S;\Gamma \vdash \mathsf{reify}\ e : \mathsf{h{:}heap\{pre\ h\}} \longrightarrow \mathsf{Tot}\ (\mathsf{r{:}(t{*}heap))\{post\ h\ (fst\ r)\ (snd\ r)\}}}$$

The semantics of reify is to traverse the term and to gradually expose the underlying monadic representation. We illustrate this below for STATE, where the constructs on the right-hand side of the rules are the pure implementations of return, bind, put, and get as defined on page 3, but with type arguments left implicit:

$$\begin{aligned}
\mathsf{reify}\ (\mathsf{return}\ e) &\rightsquigarrow \mathsf{STATE.return}\ e\\
\mathsf{reify}\ (\mathsf{bind}\ \mathsf{x} \longleftarrow e_1\ \mathsf{in}\ e_2) &\rightsquigarrow \mathsf{STATE.bind}\ (\mathsf{reify}\ e_1)\ (\lambda \mathsf{x} \!\longrightarrow \mathsf{reify}\ e_2)\\
\mathsf{reify}\ (\mathsf{get}\ e) &\rightsquigarrow \mathsf{STATE.get}\ e\\
\mathsf{reify}\ (\mathsf{put}\ e) &\rightsquigarrow \mathsf{STATE.put}\ e
\end{aligned}$$

Armed with reify, we can write an *extrinsic* proof of a lemma relating sum_up and sum_dn (discussed in detail in §2.4), i.e., an "after the fact" proof that is separate from the definition of sum_up and sum_dn and that relates their reified executions. We further remark that in F⋆ the standard operational semantics of effectful computations is modeled in terms of reification, so proving a property about a reified computation is really the same as proving the property about the evaluation of the computation itself.

The reify operator clearly breaks the abstraction of the underlying monad and needs to be used with care. Ahman et al. (2017) show that programs that do not use reify (or its converse, reflect) can be compiled efficiently. Specifically, if the computationally relevant part of a program is free of reify then the STATE computations can be compiled using primitive state with destructive updates.

Less frequently, we use `reify`'s dual, `reflect`, which packages a pure function as an effectful computation.

To retain these benefits of abstraction, we rely on F★'s module system to control how the abstraction-breaking reify coercion can be used in client code. In particular, when abstraction violations cannot be tolerated, we use F★'s Ghost effect (explained in §2.4) to mark reify as being usable only in computationally irrelevant code, limiting the use of monadic reification to specifications and proofs. This allows one to use reification even though effects like state and exceptions are implemented primitively in F★.

2.4 Extrinsic specification and proof, eased by SMT-based automation

We now look at the proof relating sum_up and sum_dn in detail, explaining along the way several F★-specific idioms that we find essential to making our method work well.

Computational irrelevance (Ghost effect) The Ghost effect is used to track a form of computational irrelevance. Ghost t (requires pre) (ensures post) is the type of a pure computation returning a value of type t satisfying post, provided pre is valid. However, this computation must be erased before running the program, so it can only be used in specifications and proofs.

Adding proof irrelevance (Lemma) F★ provides two closely related forms of proof irrelevance. First, a pure term e:t can be given the refinement type x:t{ϕ} when it validates the formula ϕ[e/x], although no proof of ϕ is materialized. For example, borrowing the terminology of Nogin (2002), the value () is a *squashed* proof of u:unit{$0 \leq 1$}. Combining proof and computation irrelevance, e : Ghost unit pre (λ() ⟶ post) is a squashed proof of pre ⟶ post. This latter form is so common that we write it as Lemma (requires pre) (ensures post), further abbreviated as Lemma post when pre is ⊤.

Proof relating sum_up and sum_dn Spelling out the main lemma of §1.1, our goal is a value of the following type:

```
val equiv_sum_up_dn (r:ref int) (lo:int) (hi:int{hi ≥ lo}) (h:heap{r ∈ h})
: Lemma (v r (reify (sum_up r lo hi) h) == v r (reify (sum_dn r lo hi) h))
```

where v r (_, h) = h.[r] and h.[r] selects the contents of the reference r from the heap h.

An attempt to give a trivial definition for equiv_sum_up_dn that simply returns a unit value () fails, because the SMT solver cannot automatically prove the strong postcondition above. Instead our proof involves calling an auxiliary lemma sum_up_dn_aux, proving a ternary relation:

```
val sum_up_dn_aux (r:ref int) (lo:int) (mid:int{mid ≥ lo})
                            (hi:int{hi ≥ mid}) (h:heap{r ∈ h})
: Lemma (v r (reify (sum_up r lo hi) h)
              == v r (reify (sum_dn r lo mid) h)
                + v r (reify (sum_up r mid hi) h) − h.[r])
   (decreases (mid − lo))
let equiv_sum_up_dn r lo hi h = sum_up_dn_aux r lo hi hi h
```

While the statement of equiv_sum_up_dn is different from the statement of sum_up_dn_aux, the SMT-based automation fills in the gaps and accepts the proof sketch. In particular, the SMT solver figures out that sum_up r hi hi is a no-op by looking at its reified definition. In other cases, the user has to provide more interesting proof sketches that include not only calls to lemmas that the SMT solver cannot automatically apply but also the cases of the proof and the recursive structure. This is illustrated by the following proof:

```
let rec sum_up_dn_aux r lo mid hi h =
  if lo ≠ mid then (sum_up_dn_aux r lo (mid − 1) hi h;
                    sum_up_commute r mid hi (mid − 1) h;
                    sum_dn_commute r lo (mid − 1) (mid − 1) h)
```

This proof is by induction on the difference between mid and lo (as illustrated by the decreases clause of the lemma, this is needed because we are working with potentially-negative integers). If this difference is zero, then the property is trivial since the SMT solver can figure out that sum_dn r lo lo is a no-op. Otherwise, we call sum_up_dn_aux recursively for mid − 1 as well as two further commutation lemmas (not shown) about sum_up and sum_dn and the SMT automation can take care of the rest.

Encoding computations to SMT So how did F★ figure out automatically that sum_up r hi hi and sum_dn r lo lo are no-ops? For a start the F★ normalizer applied the semantics of reify sketched in §2.3 to partially evaluate the term and reveal the monadic representation of the STATE effect by traversing the term and unfolding the monadic definitions of return, bind, actions and lifts. In the case of reify (sum_up r hi hi) h, for instance, reduction intuitively proceeds as follows:

```
reify (sum_up r hi hi) h
  ⇝ reify (if hi ≠ hi then (r := !r + lo; sum_up r (lo + 1) hi)) h
  ⇝* if hi ≠ hi then (STATE.bind (reify (Ref.read r) h) (λ x ⟶
                  STATE.bind (reify (Ref.upd r (x + lo))) (λ _ ⟶
                  reified_sum_up r (hi + 1) hi))) h
      else STATE.return () h
  ⇝* if hi ≠ hi then let x, h' = reify (Ref.read r) h in
                  let _, h'' = reify (Ref.upd r (x + lo)) h' in
                  reified_sum_up r (hi + 1) hi h''
      else ((), h)
```

What is left is pure monadic code that F★ then encodes to the SMT solver in a way that allows it to reason by computation (Aguirre et al. 2016). In the case of reify (sum_up r hi hi) h the SMT solver can trivially show that hi ≠ hi is false and thus the computation returns the pair ((), h).

While our work did not require any extension to F★'s theory (Ahman et al. 2017), we significantly improved F★'s logical encoding to perform normalization of open terms based on the semantics of reify (a kind of symbolic execution) before calling the SMT solver. This allowed us to scale and validate the theory of Ahman et al. (2017) from a single 2-line example to the ≈5,300 lines of relationally verified code presented in this paper.

Subject	Section	1st run (ms)	Replay (ms)	Loc
Loops	1.1	79757	2085	143
Reorderings	3.1	3857	2169	193
Benton (2004)	3.2	98072	62264	1504
Static IFC	4.1	44832	7672	917
Hybrid IFC	4.2	37826	1315	55
Declassification	4.3	15064	3463	257
Memoization	5.1	8583	7210	719
Union-find	5.2	46725	12367	332
Cryptography	A	8234	7396	627
IFC Monitor	D	22118	6726	611
Total		365068	112667	5358

Table 1. Code size (lines of code without comments) and proof-checking time (ms) for our examples.

2.5 Empirical evaluation of our methodology

For this first example, we reasoned directly about the semantics of two effectful terms to prove their equivalence. However, we often prefer more structured reasoning principles to prove or enforce relational properties, e.g., by using program logics, syntax-directed type systems, or even dynamic analyses. In the rest of this paper, we show through several case studies, that these approaches can be accommodated, and even composed, within our framework.

Table 1 summarizes the empirical evaluation from these case studies. Each row describes a specific case study, its size in lines of source code, and the verification time using F⋆ and the Z3-4.5.1 SMT solver. The verification times were collected on an Intel Xeon E5-1650 at 3.5 GHz and 16GB of RAM. The "1st run" column indicates the time it takes F⋆ and Z3 to find a proof. This proof is then used to generate hints (unsat cores) that can be used as a starting point to verify subsequent versions of the program. The "replay" column indicates the time it takes to verify the program given the hints recorded in the first run. Proof replay is usually significantly faster, indicating that although finding a proof may initially be quite expensive, revising a proof with hints is fast, which greatly aids interactive proof development.

3 Correctness of program transformations

Several researchers have devised custom program logics for verifying transformations of imperative programs (Barthe et al. 2009; Benton 2004; Carbin et al. 2012). We show how to derive similar rules justifying the correctness of generic program transformations within our monadic framework. We focus on stateful programs with a fixed-domain, finite memory. We leave proving transformations of commands that dynamically allocate memory to future work.

3.1 Generic transformations based on read- and write-footprints

Here and in the next subsection, we represent a command c as a function of type unit ⟶ St unit that may read or write arbitrary references in memory.

```
type command = unit ⟶ St unit
```

In trying to validate transformations of commands, it is traditional to employ an effect system to delimit the parts of memory that a command may read or write. Most effect systems are unary, syntactic analyses. For example, consider the classic frame rule from separation logic:

$$\{P\}c\{Q\} \Rightarrow \{P * R\}c\{Q * R\}$$

The command c requires ownership of a subset of the heap P in order to execute, then returns ownership of Q to its caller. Any distinct heap fragment R remains unaffected by the function. Reading this rule as an effect analysis, one may conclude that c may read or write the P-fragment of memory—however, this is just an approximation of c's extensional behavior. Benton et al. (2006) observe that a more precise, semantic characterization of effects arises from a relational perspective. Adopting this perspective, one can define the footprint of a command extensionally, using two unary properties and one binary property.

Capturing a command's write effect is easy with a unary property, 'writes c ws' stating that the initial and final heaps agree on the contents of their references, except those in ws.

```
type addrs = S.set addr
let writes (c:command) (ws:addrs) = ∀(h:heap).
 let h' = snd (reify (c ()) h) in
 (∀ r. r ∈ h ⟺ r ∈ h') ∧ (* no allocation *)
 (∀ r. addr_of r ∉ ws ⟹ h.[r] == h'.[r]) (* only refs in ws changed *)
```

Stating that a command only reads references rs is similar in spirit to noninterference (§4.1). Interestingly, it is impossible to describe the set of locations that a command may read without also speaking about the locations it may write. The relation 'reads c rs ws' states that if c writes at most the references in ws, then executing c in heaps that agree on the references in rs produces heaps that agree on ws, i.e., c does not depend on references outside rs.

```
let equiv_on (rs:addr_set) (h0:heap) (h1:heap) =
 ∀a (r:ref a). addr_of r ∈ rs ∧ r ∈ h0 ∧ r ∈ h1 ⟹ h0.[r] == h1.[r]
let reads (c:command) (rs ws:addrs) = ∀(h0 h1: heap).
 let h'0, h'1 = snd (reify (c ()) h0), snd (reify (c ()) h1) in
 (equiv_on rs h0 h1 ∧ writes c ws) ⟹ equiv_on ws h'0 h'1
```

Putting the pieces together, we define a read- and write-footprint-indexed type for commands:

```
type cmd (rs ws:addrs) = c:command{writes c ws ∧ reads c rs ws}
```

One can also define combinators to manipulate footprint-indexed commands. For example, here is a '>>' combinator for sequential composition. Its type proves that read and write-footprints compose by a pointwise union, a higher-order relational property; the proof requires an (omitted) auxiliary lemma seq_lem (recall that variables preceded by a # are implicit arguments):

```
let seq (#r1 #w1 #r2 #w2 : addrs) (c1:cmd r1 w1) (c2:cmd r2 w2) :
  command = c1(); c2()
```

```
let (>>) #r1 #w1 #r2 #w2 (c1:cmd r1 w1) (c2:cmd r2 w2) :
  cmd (r1 ∪ r2) (w1 ∪ w2) = seq_lem c1 c2; seq c1 c2
```

Making use of relational footprints, we can prove other relations between commands, e.g., equivalences that justify program transformations. Command equivalence $c_0 \sim c_1$ states that running c_0 and c_1 in identical initial heaps produces (extensionally) equal final heaps.

```
let (~) (c0:command) (c1:command) = ∀h.
  let h0, h1 = snd (reify (c0 ()) h), snd (reify (c1 ()) h) in
  ∀(r:ref α). (r ∈ h0 ⟺ r ∈ h1) ∧ (r ∈ h0 ⟹ h0.[r] == h1.[r])
```

For instance, we can prove that two commands can be swapped if they write to disjoint sets, and if the read footprint of one does not overlap with the write footprint of the other—this lemma is identical to a rule for swapping commands in a logic presented by Barthe et al. (2009).

```
let swap #rs1 #rs2 #ws1 #ws2 (c1:cmd rs1 ws1) (c2:cmd rs2 ws2)
  :Lemma (requires (disjoint ws1 ws2 ∧ disjoint rs1 ws2 ∧
                                    disjoint rs2 ws1))
              (ensures ((c1 >> c2) ~ (c2 >> c1)))
  = ∀_intro (λ h → let _ = reify (c1 ()) h, reify (c2 ()) h in
     () <: Lemma (equiv_on_h (c1 >> c2) (c2 >> c1) h))
```

In Appendix B we verify two other common command-transformations based on similar equivalences flavor: idempotence of commands and elimination of redundant writes.

3.2 Relational Hoare Logic

Beyond generic footprint-based transformations, one may also prove program-specific equivalences. Several logics have been devised for this, including, e.g., Benton's (2004) Relational Hoare logic (RHL). We show how to derive RHL within our framework by proving the soundness of each of its rules as lemmas about a program's reification.

Model To support potentially diverging computations, we instrument shallowly-embedded effectful computations with a *fuel* argument, where the value of the fuel is irrelevant for the behavior of a terminating computation.

```
type computation = f: (fuel:nat → St bool)
   { ∀h fuel fuel' . fst (reify (f fuel) h) == true ∧ fuel' > fuel
      ⟹ reify (f fuel') h == reify (f fuel) h }
let terminates_on c h = ∃fuel . fst (reify (c fuel) h) == true
```

We model effectful expressions whose evaluation always terminates and does not change the memory state, and assignments, conditionals, sequences of computations, and potentially diverging while loops.

Deriving RHL An RHL judgement 'related c_1 c_2 pre post' (where c_1, c_2 are effectful computations, and pre, post are relations over memory states) means that the executions of c_1, c_2 starting in memories h_1, h_2 related by pre, both diverge or both terminate with memories h_1', h_2' related by post.

```
let related (c1 c2 : computation) (pre post: (heap → heap → prop)) =
 (* if precondition holds on initial memory states, then *)
 ∀h1 h2 . pre h1 h2 ⟹
 (* c1 and c2 both terminate or both diverge, and *)
 ((c1 `terminates_on` h1 ⟺ c2 `terminates_on` h2) ∧
  (∀ fuel h1' h2' . (reify (c1 fuel) h1 == (true, h1') ∧
     reify (c2 fuel) h2 == (true, h2')) ⟹ (* if both terminate, *)
   post h1' h2')) (* then postcondition holds on final memory states *)
```

From these reification-based definitions, we prove every rule of RHL. Of the 20 rules and equations of RHL presented by Benton (2004), 16 need at most 5 lines of proof annotation each, among which 10 need none and are proven automatically. Rules related to while loops often require some manual induction on the fuel.

With RHL in hand, we can prove program equivalences applying syntax-directed rules, focusing the intellectual effort on finding and proving inductive invariants to relate loop bodies. When RHL is not powerful enough, we can escape back to the reification of commands to complete a direct proof in terms of the operational semantics. In Appendix C we provide a detailed sketch of a program-specific equivalence built using our embedding of RHL in F★.

4 Information-flow control

In this section, we present a case study examining various styles of information-flow control (IFC), a security paradigm based on *noninterference* (Goguen and Meseguer 1982), a property that compares two runs of a program differing only in the program's secret inputs and requires the non-secret outputs to be equal. Many special-purpose systems, including syntax-directed type systems, have been devised to enforce noninterference-like security properties (see, e.g., Hedin and Sabelfeld 2012; Sabelfeld and Myers 2006).

We start our IFC case study by encoding a classic IFC type system (Volpano et al. 1996) for a small deeply-embedded imperative language and proving its correctness (§4.1). In order to augment the permissiveness of our analysis we then show how to compose our IFC type system with precise semantic proofs (§4.2). As IFC is often too strong for practical use, the final step in our IFC case study is a semantic treatment of declassification based on delimited release (Sabelfeld and Myers 2003b) (§4.3). An additional case study on a runtime monitor for IFC is presented in Appendix D. We conclude that our method for relational verification is flexible enough to accommodate various IFC disciplines, allowing comparisons and compositions within the same framework.

4.1 Deriving an IFC type system

Consider the following small *while* language consisting of expressions, which may only read from the heap, but not modify it, and commands, which may write to the heap and branch, depending on its contents. The definition of the language should be unsurprising, the only subtlety worth noting is the decr expression in the while command, a metric

$$\textsc{CSub}\ \frac{\Gamma, \mathrm{pc} : l_1 \vdash c \qquad l_2 \leq l_1}{\Gamma, \mathrm{pc} : l_2 \vdash c} \qquad \textsc{CAssign}\ \frac{\Gamma \vdash e : \Gamma(r)}{\Gamma, \mathrm{pc} : \Gamma(r) \vdash r := e}$$

$$\textsc{CCond}\ \frac{\Gamma \vdash e : l \qquad \Gamma, \mathrm{pc} : l \vdash c_1 \qquad \Gamma, \mathrm{pc} : l \vdash c_2}{\Gamma, \mathrm{pc} : l \vdash \text{if } e = 0 \text{ then } c_1 \text{ else } c_2}$$

Figure 1. A classic IFC type system (selected rules)

used to ensure loop termination.

$$\begin{array}{lll} e & ::= & i \mid r \mid e_1 \oplus e_2 \\ c & ::= & \text{skip} \mid r := e \mid c_1; c_2 \mid \text{if } e = 0 \text{ then } c_1 \text{ else } c_2 \\ & & \mid \text{while } e \neq 0 \text{ do } c \,(\text{decr } e') \end{array}$$

A classic IFC type system Volpano et al. (1996) devise an IFC type system to check that programs executing over a memory containing both secrets (stored in memory locations labeled High) and non-secrets (in locations labeled Low) never leak secrets into non-secret locations. The type system includes two judgments $\Gamma \vdash e : l$, which states that the expression e (with free variables in Γ) depends only on locations labeled l or lower; and $\Gamma, \mathrm{pc} : l \vdash c$, which states that a command c in a context that is *control-dependent* on the contents of memory locations labeled l, does not leak secrets. Some selected rules of their system, as adapted to our example language, are shown in Figure 1.

Multiple effects to structure the* while *interpreter We deeply embed the syntax of *while* in F⋆ using data types exp and com, for expressions and commands, respectively. The expression interpreter interp_exp only requires reading the value of the variables from the store, whereas the command interpreter, interp_com, also requires writes to the store, where store is an integer store mapping a fixed set of integer references 'ref int' to int. Additionally, interp_com may also raise an Out_of_fuel exception when it detects that a loop may not terminate (e.g., because the claimed metric is not actually decreasing). We could define both interpreters using a single effect, but this would require us to prove that interp_exp does not change the store and does not raise exceptions. Avoiding the needless proof overhead, we use a Reader monad for interp_exp and StExn, a combined state and exceptions monad, for interp_com. By defining Reader as a sub_effect of StExn, expression interpretation is transparently lifted by F⋆ to the larger effect when interpreting commands. Using these effects, interp_exp and interp_com form a standard, recursive, definitional interpreter for *while*, with the following trivial signatures.

```
val interp_exp: exp → Reader int
val interp_com: com → StExn unit
```

Deriving IFC typing for expressions For starters, we use a store_labeling = ref int → label, where label ∈ {High, Low}, to partition the store between secrets (High) and non-secrets (Low). An expression is noninterferent at level l when its interpretation does not depend on locations labeled greater than l in the store. To formalize this, we define a notion of *low-equivalence* on stores, relating stores that agree on the contents of all Low-labeled references, and noninterferent expressions (at level Low, i.e., ni_exp env e Low) as those whose interpretation is identical in low-equivalent stores.

```
type low_equiv (env:store_labeling) (s0 s1:store) =
    ∀(r:ref int). env x=Low ⟹ s0.[r] == s1.[r]
let ni_exp (env:store_labeling) (e:exp) (l:label) =
    ∀(s0 s1:store). (low_equiv env s0 s1 ∧ l == Low) ⟹
        reify (interp_exp e) s0 == reify (interp_exp e) s1
```

With this definition of noninterference for expressions we capture the semantic interpretation of the typing judgment $\Gamma \vdash e : l$: if the expression e can be assigned the label Low, then the computation of e is only influenced by Low values.

Deriving IFC typing for commands As explained previously, the judgment $\Gamma, \mathrm{pc} : l \vdash c$ deems c noninterferent when run in context control-dependent only on locations whose label is at most l. More explicitly, the judgment establishes the following two properties: (1) locations labeled below l are not modified by c—this is captured by no_write_down, a unary property; (2) the command c does not leak the contents of a High location to Low location—this is captured by ni_com', a binary property.

```
let run c s = match reify (interp_com c) s with
  | Inr Out_of_fuel, _ → Loops | _, s' → Returns s'
let no_write_down env c l s = match run c s with
  | Loops → ⊤| Returns s' → ∀(i:id). env i < l ⟹ s'.[i] == s.[i]
let ni_com' env c l s0 s1 = match run c s0, run c s1 with
  | Returns s0', Returns s1' → low_equiv env s0 s1 ⟹
        low_equiv env s0' s1'
  | Loops, _ | _, Loops → ⊤
```

The type system is termination-insensitive, meaning that a program may diverge depending on the value of a secret. Consider, for instance, two runs of the program `while hi <> 0 do {skip}; lo := 0`, one with `hi = 0` and another with `hi = 1`. The first run terminates and writes to `lo`; the second run loops forever. As such, we do not expect to prove noninterference in case the program loops. Putting the pieces together, we define $\Gamma, \mathrm{pc} : l \vdash c$ to be ni_com Γ c l.

```
let ni_com (env:store_labeling) (c:com) (l:label) =
  (∀ s0 s1. ni_com' env c l s0 s1) ∧ (∀ s. no_write_down env c l s)
```

As in the case of expression typing, we derive each rule of the command-typing judgment as a lemma about ni_com. For example, here is the statement for the CCond rule:

```
val cond_com (env:store_labeling) (e:exp) (ct:com) (cf:com) (l:label)
: Lemma (requires (ni_exp env e l ∧ ni_com env ct l ∧ ni_com env cf l))
        (ensures (ni_com env (If e ct cf) l))
```

The proofs of many of these rules are partially automated by SMT—they take about 250 lines of specification and proof in F⋆. Once proven, we use these rules to build a certified, syntax-directed typechecker for *while* programs that repeatedly applies these lemmas to prove that a program satisfies ni_com. This typechecker has the following type:

```
val tc_com : env:store_labeling → c:com →
  Exn label (requires ⊤) (ensures λInl l → ni_com env c l | _ → ⊤)
```

4.2 Combining syntactic IFC analysis with semantic noninterference proofs

Building on §4.1, we show how programs that fall outside the syntactic information-flow typing discipline can be proven secure using a combination of typechecking and semantic proofs of noninterference. This example is evocative (though at a smaller scale) of the work of Küsters et al. (2015), who combine automated information-flow analysis in the Joana analyzer (Hammer and Snelting 2009) with semantic proofs in the KeY verifier for Java programs (Darvas et al. 2005; Scheben and Schmitt 2011). In contrast, we sketch a combination of syntactic and semantic proofs of relational properties in *a single* framework. Consider the following *while* program, where the label of c and lo is Low and the label of hi is High.

```
  while c ≠0 do hi := lo + 1; lo := hi + 1; c := c − 1 (decr c)
```

The assignment lo := hi + 1 is ill-typed in the type system of §4.1, since it directly assigns a High expression to a Low location. However, the previous command overwrites hi so that hi does not contain a High value anymore at that point. As such, even though the IFC type system cannot prove it, the program is actually noninterferent. To prove it, one could directly attempt to prove ni_com for the entire program, which would require a strong enough (relational) invariant for the loop. A simpler approach is to prove just the sub-program hi := lo + 1; lo := hi + 1 (c_s) noninterferent, while relying on the type system for the rest of the program. The sub-program can be automatically proven secure:

```
let c_s_ni () : Lemma (ni_com env c_s Low) = ()
```

This lemma has exactly the form of the other standard, typing rules proven previously, except it is specialized to the command in question. As such, c_s_ni can just be used in place of the standard sequence-typing rule (CSeq) when proving the while loop noninterferent.

We can even modify our automatic typechecker from §4.1 to take as input a list of commands that are already proved noninterferent (by whichever means), and simply look up the command it tries to typecheck in the list before trying to typecheck it syntactically. The type (and omitted implementation) of this typechecker is very similar to that of tc_com, the only difference is the extra list argument:

```
val tc_com_hybrid : env:store_labeling → c:com →
  list (cl:(com∗label){ni_com env (fst cl) (snd cl)}) →
  Exn label (ensures λol → Inl? ol ⟹ ni_com env c (Inl?.v ol))
```

We can complete the noninterference proof automatically by passing the (c_s, Low) pair proved in ni_com by lemma c_s_ni (or directly by SMT) to this hybrid IFC typechecker:

```
let c_loop_ni () : Lemma (ensures ni_com env c_loop Low) =
  c_s_ni(); ignore (reify (tc_com_hybrid env c_loop [c_s, Low]) ())
```

Checking this in F* works by simply evaluating the invocation of tc_com_hybrid; this reduces fully to Inl Low and the intrinsic type of tc_com_hybrid ensures the postcondition.

4.3 Semantic declassification

Beyond noninterference, reasoning directly about relational properties allows us to characterize various forms of *declassification* where programs intentionally reveal some information about secrets. For example, Sabelfeld and Myers (2003b) propose *delimited release*, a discipline in which programs are allowed to reveal the value of only certain pure expressions.

In a simple example by Sabelfeld and Myers some amount of money (k) is transferred from one account (hi) to another (lo). Simply by observing whether or not the funds are received, the owner of the lo account gains some information about the other account, namely whether or not hi contained at least k units of currency—this is, however, by design.

```
let transfer (k:int) (hi:ref int) (lo:ref int) =
  if k < !hi then (hi := !hi − k; lo := !lo + k)
```

To characterize this kind of intentional release of information, delimited release describes two runs of a program in initial states where the secrets, instead of being arbitrary, are related in some manner, e.g., the initial states agree on the value of the term being explicitly declassified. This is easily captured in our setting. For example, we can prove the following lemma for transfer, which shows that lo gains no more information than intended.

```
let transfer_ok (k:int) (hi lo:ref int{addr_of lo ≠addr_of hi})
  (s0 s1:heap{lo ∈ s0 ∧ hi ∈ s0 ∧ lo ∈ s1 ∧ hi ∈ s1}) : Lemma
     (∗ initial memories agree on lo and on the declassified term ∗)
     (requires (s0.[lo] == s1.[lo] ∧ (k < s0.[hi] ⟺ k < s1.[hi])))
     (ensures ((snd (reify (transfer k hi lo) s0)).[lo] ==
                  (snd (reify (transfer k hi lo) s1)).[lo])) = ()
```

Delimited release was about the *what* dimension of declassification (Sabelfeld and Sands 2009). We also built a very simple model that is targeted at the *when* dimension, illustrating a customization of the monadic model to the target relational property. For instance, to track when information is declassified, we augment the state with a bit recording whether the secret component of the state was declassified and is thus allowed to be leaked.

```
type ifc_state = { secret:int; public:int; release:bool }
new_effect STATE_IFC = STATE_h ifc_state
```

In this case the noninterference property depends on the extra instrumentation bit we added to the state.

```
let ni (f:unit → St unit) =
 ∀s0 s1. let (_, s0'), (_, s1') = reify (f ()) s0, reify (f ()) s1 in
  s0'.release ∨ s1'.release ∨ (low_equiv s0 s1 ⟹ low_equiv s0' s1')
```

5 Program optimizations and refinement

This section presents two complete examples to prove a few, classic algorithmic optimizations correct. These properties are very specific to their application domains and a special-purpose relational logic would probably not be suitable. Instead, we make use of the generality of our approach to prove application-specific relational properties (including 4- and 6-ary relations) of higher-order programs with local state. In contrast, most prior relational logics are specialized to proving binary relations, or, at best, properties of *n* runs of a single first-order program (Sousa and Dillig 2016).

5.1 Effect for memoizing recursive functions

First, we look at memoizing total functions, including memoizing a function's recursive calls based on a partiality representation technique due to McBride (2015). We prove that a memoized function is extensionally equal to the original.

We define a custom effect Memo for this task. Memo is a state monad where the state consists of a (partial, finite) mapping from the domain type of the functions (dom) to their codomain type (codom). This effect has two actions:

- get : dom ⟶ Memo (option codom), which returns a memoized value if it exists; and
- put : dom ⟶ codom ⟶ Memo unit, which adds a new memoization pair to the state.

Take 1: Memoizing total functions Our goal is to turn a total function g into a memoized function f computing the same values as g. This relation between f's reification and g is captured by the computes predicate below, depending on an invariant of the memoization state, valid_memo. A memoization state h is valid for memoizing some total function g : (dom ⟶ codom) when h is a subset of the graph of g:

```
let valid_memo (h:memo_st) (g:dom ⟶ codom) =
  for_all_prop (λ (x,y) ⟶ y == g x) h
let computes (f: dom ⟶ Memo codom) (g:dom ⟶ codom) =
  ∀h0. valid_memo h0 g ⟹ (∀ x. (let y, h1 = reify (f x) h0 in
                                    y == g x ∧ valid_memo h1 g))
```

We have f `computes`g when given any state h0 containing a subgraph of g, f x returns g x and maintains the invariant that the result state h1 is a subgraph of g. It is easy to program and verify a simple memoizing function:

```
let memoize (g : dom ⟶ codom) (x:dom) =
  match get x with Some y ⟶ y | None ⟶ let y = g x in put x y; y
let memoize_computes g :Lemma ((memoize g) `computes` g) = ...
```

The proof of this lemma is straightforward: we only need to show that the value y we get back from the heap in the first branch is indeed g x which is enforced by the valid_memo in the precondition of computes.

Take 2: Memoizing recursive calls Now, what if we want to memoize a recursive function, for example, a function computing the Fibonacci sequence? We also want to memoize the intermediate recursive calls, and in order to achieve it, we need an explicit representation of the recursive structure of the function. Following McBride (2015), we represent this by a function x:dom ⟶ partial_result x, where a partial result is either a finished computation of type codom or a request for a recursive call together with a continuation.

```
type partial_result (x0:dom) =
  | Done : codom ⟶ partial_result x0
  | Need : x:dom{x ≺ x0} ⟶ cont:(codom ⟶ partial_result x0) ⟶
         partial_result x0
```

As we define the fixed point using Need x f, we crucially require x ≺ x0, meaning that the value of the function is requested at a point x where function's definition already exists. For example encoding Fibonacci amounts to the following code where the 2 recursive calls in the second branch have been replaced by applications of the Need constructor:

```
let fib_skel (x:dom) : partial_result x =
  if x ≤ 1 then Done 1 else
    Need (x − 1) (λ y1 ⟶ Need (x − 2) (λ y2 ⟶ Done (y1 + y2)))
```

We define the fixpoint of such a function representation f:

```
let rec fixp (f: x:dom ⟶ partial_result x) (x0:dom) : codom =
  let rec complete_fixp x = function
    | Done y ⟶ y
    | Need x' cont ⟶ let y = fixp f x' in complete_fixp x (cont y)
  in complete_fixp x0 (f x0)
```

To obtain a memoized fixpoint, we need to memoize functions defined only on part of the domain, x:dom{p x}.

```
let partial_memoize (p:dom ⟶ Type)
  (f : x:dom{p x} ⟶ Memo codom) (x:dom{p x}) =
  match get x with Some y ⟶ y | None ⟶ let y = g x in put x y; y
let rec memoize_rec (f: x:dom ⟶ partial_result x) (x0:dom) =
  let rec complete_memo_rec x :Memo codom = function
    | Done y ⟶ y
    | Need x' cont ⟶
      let y = partial_memoize (λ y ⟶ y ≺ x) (memoize_rec f) x' in
      complete_memo_rec (cont y)
  in complete_memo_rec x0 (f x0)
```

It is relatively easy to prove by structural induction on the code of memoize_rec that, for any skeleton of a recursive function f, we have that (memoize_rec f) `computes`(fixp f). The harder part is proving that fixp fib_skel is extensionally equal to fibonacci, the natural recursive definition of the Fibonacci sequence, since these two functions are not syntactically similar—but at least this proof involves reasoning only about pure functions. The good news is that having already proven that memoize_rec fib_skel computes fixp fib_skel, we gain a proof of the equivalence of memoize_rec fib_skel to fibonacci by transitivity.

This abstract model could be implemented efficiently, for instance by an imperative hash-table with a specific memory-management policy.

5.2 Stepwise refinement and n-ary relations: Union-find with two optimizations

In this section, we prove several classic optimizations of a union-find data structure introduced in several stages, each a refinement. For each refinement step, we employ relational verification to prove that the refinement preserves the canonical structure of union-find. We specify correctness using, in some cases, 4- and 6-ary relations, which are easily manipulated in our monadic framework.

Basic union-find implementation A union-find data structure maintains disjoint partitions of a set, such that each element belongs to exactly one of the partitions. The data structure supports two operations: find, that identifies to which partition an element belongs, and union, that takes as input two elements and combines their partitions.

An efficient way to implement the union-find data structure is as a forest of disjoint trees, one tree for each partition, where each node maintains its parent and the root of each tree is the designated representative of its partition. The find operation returns the root of a given element's partition (by traversing the parent links), and the union operation simply points one of the roots to the other.

We represent a union-find of set $[0, n-1]$ as the type 'uf_forest n' (below), a sequence of ref cells, where the i^{th} element in the sequence is the i^{th} set element, containing its parent and the list of all the nodes in the subtree rooted at that node. The list is computationally irrelevant (i.e., *erased*)—we only use it to express the disjointness invariant and the termination metric for recursive functions (e.g. find).

```
type elt (n:ℕ) = i:ℕ{i < n} × erased (list ℕ)
type uf_forest (n:ℕ) = s:seq (ref (elt n)){length s = n}
```

The basic find and union operations are shown below, where set and get are stateful functions that read and write the i^{th} index in the uf sequence. Reasoning about mutable pointer structures requires maintaining invariants regarding the liveness and separation of the memory referenced by the pointers. While important, these are orthogonal to the relational refinement proofs—so we elide them here, but still prove them intrinsically in our code.

```
let rec find #n uf i = let p, _ = get uf i in if p = i then i else find uf p
let union #n uf i1 i2 = let r1, r2 = find uf i1, find uf i2 in
    let _, s1 = get uf r1 in let _, s2 = get uf r2 in
    if r1 ≠r2 then (set uf r1 (r2, s1); set uf r2 (r2, union s1 s2))
```

Union by rank The first optimization we consider is union_by_rank, which decides whether to merge r_1 into r_2, or vice versa, depending on the heights of each tree, aiming to keep the trees shallow. We prove this optimization in two steps, first refining the representation of elements by adding a rank field to elt n and then proving that union_by_rank maintains the same set partitioning as union.

```
type elt (n:ℕ) = i:ℕ{i < n} × ℕ × erased (list nat) (* added a rank *)
```

We formally reason about the refinement by proving that the outputs of the find and union functions do not depend on the newly added rank field. The rank_independence lemma (a 4-ary relation) states that find and union when run on two heaps that differ only on the rank field, output equal results and the resulting heaps also differ only on the rank field.

```
let equal_but_rank uf h1 h2 = ∀ i. parent uf i h1 = parent uf i h2
                                   ∧ subtree uf i h1 = subtree uf i h2
let rank_independence #n uf i i1 i2 h1 h2 : Lemma
(requires (equal_but_rank uf h1 h2))
(ensures (let (r1,f1), (r2,f2) = reify (find uf i) h1,reify (find uf i) h2 in
 let (_,u1), (_,u2) = reify (union uf i1 i2) h1,reify (union uf i1 i2) h2 in
 r1 == r2 ∧ equal_but_rank uf f1 f2 ∧ equal_but_rank uf u1 u2))
```

Next, we want to prove the union_by_rank refinement sound. Suppose we run union on a heap h producing h_1; and suppose we run union_by_rank in h producing h_2. Clearly, we cannot prove that find for a node j returns the same result in h_1 and h_2. But we prove that the canonical structure of the forest is the same in h_1 and h_2, by showing that two nodes are in the same partition in h_1 if and only if they are in the same partition in h_2:

```
val union_by_rank_refinement #n uf i1 i2 h j1 j2 : Lemma
  (let (_, h1), (_, h2) =
    reify (union uf i1 i2) h, reify (union_by_rank uf i1 i2) h in
   fst (reify (find uf j1) h1) == fst (reify (find uf j2) h1) ⟺
     fst (reify (find uf j1) h2) == fst (reify (find uf j2) h2))
```

This property is 6-ary relation, relating 1 run of union and 1 run of union_by_rank to 4 runs of find—its proof is a relatively straightforward case analysis.

Path compression Finally, we consider find_compress, which, in addition to returning the root for an element, sets the root as the new parent of the element to accelerate subsequent find queries. To prove the refinement of find to find_compress sound, we prove a 4-ary relation showing that if running find on a heap h results in the heap h_1, and running find_compress on h results in the heap h_2, then the partition of a node j is same in h_1 and h_2. This also implies that find_compress retains the canonical structure of the union-find forest.

```
val find_compress_refinement #n uf i h j
  : Lemma (let (r1, h1), (r2, h2) =
    reify (find uf i) h, reify (find_compress uf i) h in
    r1 == r2 ∧ fst (reify (find uf j) h1) == fst (reify (find uf j) h2))
```

6 Related work

Much of the prior related work focused on checking specific relational properties of programs, or general relational properties using special-purpose logics. In contrast, we argue that proof assistants that support reasoning about pure and effectful programs can, using our methodology, model and verify relational properties in a generic way. The specific incarnation of our methodology in F* exploits its efficient implementation of effects enabled by abstraction and controlled reification; a unary weakest precondition calculus as a base for relational proofs; SMT-based automation; and the convenience of writing effectful code in direct style with returns, binds, and lifts automatically inserted.

Static IFC tools Sabelfeld and Myers (2003a) survey a number of IFC type systems and static analyses for showing noninterference, trading completeness for automation. More recent verification techniques for IFC aim for better completeness (Amtoft and Banerjee 2004; Amtoft et al. 2012; Banerjee et al. 2016; Barthe et al. 2014; Beringer and Hofmann 2007; Nanevski et al. 2013; Rabe 2016; Scheben and Schmitt 2011), while compromising automation. The two approaches can be combined, as discussed in in §4.2.

Relational program logics and type systems A variety of program logics for reasoning about general relational properties have been proposed previously (Aguirre et al. 2017; Barthe et al. 2009; Benton 2004; Yang 2007), while others apply general relational logics to specific domains, including access control (Nanevski et al. 2013), cryptography (Barthe et al. 2009, 2012, 2013a; Petcher and Morrisett 2015), differential privacy (Barthe et al. 2013b; Zhang and Kifer 2017), mechanism design (Barthe et al. 2015), cost analysis (Çiçek et al. 2017), program approximations (Carbin et al. 2012).

RF*, is worth pointing out for its connection to F*. Barthe et al.'s (2014) extend a prior, value-dependent version of F* (Swamy et al. 2013) with a probabilistic semantics and a type system that combines pRHL with refinement types. Like many other relational Hoare logics, RF* provided an incomplete set of rules aimed at capturing many relational properties by intrinsic typing only.

In this paper we instead provide a versatile generic method for relational verification based on modeling effectful computations using monads and proving relational properties on their monadic representations, making the most of the support for full dependent types and SMT-based automation in the latest version of F*. This generic method can both be used directly to verify programs or as a base for encoding specialized relational program logics.

Product program constructions Product program constructions and self-composition are techniques aimed at reducing the verification of k-safety properties (Clarkson and Schneider 2010) to the verification of traditional (unary) safety proprieties of a product program that emulates the behavior of multiple input programs. Multiple such constructions have been proposed (Barthe et al. 2016) targeted for instance at secure IFC (Barthe et al. 2011; Naumann 2006; Terauchi and Aiken 2005; Yasuoka and Terauchi 2014), program equivalence for compiler validation (Zaks and Pnueli 2008), equivalence checking and computing semantic differences (Lahiri et al. 2012), program approximation (He et al. 2016). Sousa and Dillig's (2016) recent Descartes tool for k-safety properties also creates k copies of the program, but uses lockstep reasoning to improve performance by more tightly coupling the key invariants across the program copies. Recently Antonopoulos et al. (2017) propose a tool that obtains better scalability by using a new decomposition of programs instead of using self-composition for k-safety problems.

Other program equivalence techniques Beyond the ones already mentioned above, many other techniques targeted at program equivalence have been proposed; we briefly review several recent works: Benton et al. (2009) do manual proofs of correctness of compiler optimizations using partial equivalence relations. Kundu et al. (2009) do automatic translation validation of compiler optimizations by checking equivalence of partially specified programs that can represent multiple concrete programs. Godlin and Strichman (2010) propose proof rules for proving the equivalence of recursive procedures. Lucanu and Rusu (2015) and Ştefan Ciobâcă et al. (2016) generalize this to a set of co-inductive equivalence proof rules that are language-independent. Automatically checking the equivalence of processes in a process calculus is an important building block for security protocol analysis (Blanchet et al. 2008; Chadha et al. 2016).

Semantic techniques Many semantic techniques have been proposed for reasoning about relational properties such as observational equivalence, including techniques based on binary logical relations (Ahmed et al. 2009; Benton et al. 2009, 2013, 2014; Dreyer et al. 2010, 2011, 2012; Mitchell 1986), bisimulations (Koutavas and Wand 2006; Sangiorgi et al. 2011; Sumii 2009) and combinations thereof (Hur et al. 2012, 2014). While these very powerful techniques are often not directly automated, they can be used to provide semantic correctness proofs for relational program logics (Dreyer et al. 2010, 2011) and other verification tools (Benton et al. 2016).

7 Conclusion

This paper advocates verifying relational properties of effectful programs using generic tools that are not specific to relational reasoning: monadic effects, reification, dependent types, non-relational weakest preconditions, and SMT-based automation. Our experiments in F* verifying relational properties about a variety of examples show the wide applicability of this approach. One of the strong points is the great flexibility in modelling effects and expressing relational properties about code using these effects. The other strong point is the good balance between interactive control, SMT-based automation, and the ability to encode even more automated specialized tools where needed. Thanks to this, the effort required from the F* programmer for relational verification seems on par with non-relational reasoning in F* and with specialized relational program logics.

References

A. Aguirre, C. Hriţcu, C. Keller, and N. Swamy. From F* to SMT (extended abstract). Talk at 1st International Workshop on Hammers for Type Theories (HaTT), 2016.

A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P. Strub. A relational logic for higher-order programs. *CoRR*, abs/1703.05042, 2017.

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In Shao and Pierce (2009).

T. Amtoft and A. Banerjee. Information flow analysis in logical form. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. 2004.

T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In P. Degano and J. D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. 2012.

T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for k-safety. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017), to appear., 2017.

K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. *Science of Computer Programming*, 2016.

R. Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19:335–376, 2009.

A. Banerjee, D. A. Naumann, and M. Nikouei. Relational logic with framing and hypotheses. In A. Lal, S. Akshay, S. Saurabh, and S. Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*. 2016.

G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In Shao and Pierce (2009).

G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *11th International Conference on Mathematics of Program Construction*. 2012.

G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. Easycrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013a.

G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9:1–9:49, 2013b.

G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. *POPL*. 2014.

G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. *POPL*. 2015.

G. Barthe, J. M. Crespo, and C. Kunz. Product programs and relational program logics. *J. Log. Algebr. Meth. Program.*, 85(5):847–859, 2016.

T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. 2016.

T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 2017.

M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *Advances in Cryptology – EUROCRYPT 2006*, 2006.

N. Benton. Simple relational correctness proofs for static analyses and program transformations. *POPL*. 2004.

N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In N. Kobayashi, editor, *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*. 2006.

N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*. 2009.

N. Benton, M. Hofmann, and V. Nigam. Proof-relevant logical relations for name generation. *TLCA*. 2013.

N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. *POPL*. 2014.

N. Benton, A. Kennedy, M. Hofmann, and V. Nigam. Counting successes: Effects and transformations for non-deterministic programs. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 2016.

L. Beringer and M. Hofmann. Secure information flow and program logics. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*. 2007.

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 2012.

G. Castagna and A. D. Gordon, editors. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, 2017. ACM.

R. Chadha, V. Cheval, Ştefan Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.

E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. In Castagna and Gordon (2017).

M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18 (6):1157–1210, 2010.

Ştefan Ciobâcă, D. Lucanu, V. Rusu, and G. Rosu. A language-independent proof system for full program equivalence. *Formal Asp. Comput.*, 28(3): 469–497, 2016.

Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. 2005.

D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful adts. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010.

D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.

D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5): 477–528, 2012.

S. Fehrenbach and J. Cheney. Language-integrated provenance. In J. Cheney and G. Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United*

Kingdom, September 5-7, 2016. 2016.

A. Filinski. Representing monads. *POPL*. 1994.

B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. In Z. Manna and D. A. Peled, editors, *Time for Verification, Essays in Memory of Amir Pnueli*. 2010.

J. A. Goguen and J. Meseguer. Security policies and security models. *1982 IEEE Symposium on Security and Privacy*, 00:11, 1982.

C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

S. He, S. K. Lahiri, and Z. Rakamaric. Verifying relative safety, accuracy, and termination for program approximations. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*. 2016.

D. Hedin and A. Sabelfeld. A perspective on information-flow control. In T. Nipkow, O. Grumberg, and B. Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. In J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 2012.

C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, 2014.

V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In Morrisett and Jones (2006).

S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 2009.

R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr. A hybrid approach for proving noninterference of Java programs. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. 2015.

S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 2012.

D. Lucanu and V. Rusu. Program equivalence by circular reasoning. *Formal Asp. Comput.*, 27(4):701–726, 2015.

C. McBride. Turing-completeness totally free. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. 2015.

J. C. Mitchell. Representation independence and data abstraction. In *POPL '86*. 1986.

E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.

J. G. Morrisett and S. L. P. Jones, editors. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 2006. ACM.

L. Moura and N. Bjørner. Efficient e-matching for smt solvers. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*. 2007.

T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 2013.

A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM TOPLAS*, 35 (2):6, 2013.

D. A. Naumann. From coupling relations to mated invariants for checking information flow. In D. Gollmann, J. Meier, and A. Sabelfeld, editors, *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings*. 2006.

A. Nogin. Quotient types: A modular approach. *TPHOLs*. 2002.

A. Petcher and G. Morrisett. The foundational cryptography framework. In R. Focardi and A. C. Myers, editors, *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. 2015.

S. Peyton Jones. *Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, 2010.

M. N. Rabe. *A temporal logic approach to Information-flow control*. PhD thesis, Saarland University, 2016.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003a.

A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, 2003b.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, 2006.

A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*. 2009.

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

D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.

C. Scheben and P. H. Schmitt. Verification of information flow properties of java programs without approximations. In B. Beckert, F. Damiani, and D. Gurov, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*. 2011.

Z. Shao and B. C. Pierce, editors. *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, 2009. ACM.

M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 2016.

E. Sumii. A complete characterization of observational equivalence in polymorphic *lambda*-calculus with general references. In E. Grädel and R. Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*. 2009.

N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. *ICFP*, 2011.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. 2005.

D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.

H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.

H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theor. Comput. Sci.*, 538:167–182, 2014.

A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*. 2008.

D. Zhang and D. Kifer. LightDP: towards automating differential privacy proofs. *POPL*. 2017.

The appendices present additional case studies.

Cryptographic security proofs (Appendix A) We show how to model basic game steps of code-based cryptographic proofs of security (Bellare and Rogaway 2006) by proving equivalences between probabilistic programs. We prove perfect secrecy of one-time pad encryption, and a crucial lemma in the proof of semantic security of ElGamal encryption, an elementary use of Barthe et al.'s (2009) probabilistic relational Hoare logic.

Two additional command transformations (Appendix B) Expanding on §3.1, we prove the correctness of two more command transformations using semantic footprints: idempotence of commands and elimination of redundant writes.

RHL example (Appendix C) Making use of the RHL embedding from §3.2, we prove an example hoisting an assignment out of a loop.

Soundness of an IFC monitor (Appendix D) Using reification we prove the noninterference of a dynamic IFC monitor implemented as checks in an effectful interpreter.

A Cryptographic security proofs

We show how to construct a simple model for reasoning about probabilistic programs that sample values from discrete distributions. In this model, we prove the soundness of rules of probabilistic Relational Hoare Logic (pRHL) (Barthe et al. 2009) allowing one to derive (in-)equalities on probability quantities from pRHL judgments. We illustrate our approach by formalizing two simple cryptographic proofs: the perfect secrecy of one-time pad encryption and a crucial lemma used by Barthe et al. (2009) in the proof of semantic security of ElGamal encryption.

The simplicity of our examples pales in comparison with complex proofs formalized in specialized tools based on pRHL like EasyCrypt (Barthe et al. 2012) or FCF (Petcher and Morrisett 2015), yet our examples hint at a way to prototype and explore proofs in pRHL with a low entry cost.

A.1 A monad for random sampling

We begin by defining a monad for sampling from the uniform distribution over bitvectors of a fixed length q. We implement the monad as the composition of the state and exception monads where the state is a finite tape of bitvector values together with a pointer to a position in the tape. The RAND effect provides a single action, sample, which reads from the tape the value at the current position and advances the pointer to the next position, or raises an exception if the pointer is past the end of the tape.

```
type value = bv q
type tape = seq value
type id = i:ℕ{i < size}
type store = id ∗ tape
type rand a = store → M (option a ∗ id)
total new_effect {
  RAND: a:Type → Effect
  with repr = rand a;
       bind = λ(a b:Type) (c:rand a) (f:a → rand b) s →
               let r, next = c s in
               match r with
               | None → None, next
               | Some x → f x (next, snd s);
       return = λ(a:Type) (x:a) (next,_) → (Some x, next);
       sample = λ() s → let next, t = s in
                  if next + 1 < size then (Some (t n), n + 1)
                  else (None, n) }
effect Rand a = RAND a (λ initial_tape post → ∀x. post x)
```

Assuming a uniform distribution over initial tapes, we define the unnormalized measure of a function p:a → ℕ with respect to the denotation of a reified computation in f:Rand a as

```
let mass f p = sum (λ t → let r,_ = f (0, t) in p r)
```

where sum: (tape → ℕ) → ℕ is the summation operator over finite tapes. When p only takes values in $\{0, 1\}$, it can be regarded as an *event* whose probability with respect to the distribution generated by f is

$$\Pr[f : p] = \frac{1}{|\text{tape}|} \times \sum_{t\ \in\ \text{tape}} p\ (\text{fst}\ (f\ t)) = \frac{\text{mass}\ f\ p}{|\text{tape}|}$$

We use the shorthand $\Pr[f = v] = |\text{tape}|^{-1} \times \text{mass}\ f\ (\text{point}\ v)$ for the probability of a successful computation returning a value v, where let point x = λy → if y = Some x then 1 else 0.

A.2 Perfect secrecy of one-time pad encryption

The following effectful program uses a one-time key k sampled uniformly at random to encrypt a bitvector m:

```
let otp (m:value) : Rand value = let k = sample () in m ⊕ k
```

We show that this construction, known as *one-time pad*, provides *perfect secrecy*. That is, a ciphertext does not give away any information about the encrypted plaintext, provided the encryption key is used just once. Or equivalently, the distribution of the one-time pad encryption of a message is independent of the message itself, $\forall m_0,\ m_1,\ c.\ \Pr[\text{otp}\ m_0 = c] = \Pr[\text{otp}\ m_1 = c]$. We prove this by applying two rules of pRHL, namely [R-Rand] and [PrLe]. The former allows us to relate the results of two probabilistic programs by showing a bijection over initial random tapes that would make the relation hold (intuitively, permuting equally probable initial tapes does not change the resulting distribution over final tapes). The latter allows us to infer a probability inequality from a proven relation between probabilistic programs. Together, the two rules allow us to prove the following lemma:

```
val mass_leq: #a:Type → #b:Type →
  c1:(store → M (a ∗ id)) → c2:(store → M (b ∗ id)) →
  p1:(a → nat) → p2:(b → nat) → bij:bijection → Lemma
  (requires (∀ t. let r1,_ = c1 (to_id 0,t) in
                  let r2,_ = c2 (to_id 0,bij.f t) in p1 r1 ≤ p2 r2))
  (ensures (mass c1 p1 ≤ mass c2 p2))
```

The proof is elementary from rearranging terms in summations according to the given bijection. The following secrecy proof of one-time pad is immediate from this lemma using as bijection on initial tapes λt ⟶ upd t 0 (t 0 ⊕ m0 ⊕ m1):

```
val otp_secure: m0:value ⟶ m1:value ⟶ c:value ⟶ Lemma
  (let f0, f1 = reify (otp m0), reify (otp m1) in
   mass f0 (point c) == mass f1 (point c))
```

A.3 A step in the proof of semantic security of ElGamal encryption

Another example following a similar principle is a probabilistic equivalence used in the proof of semantic security of ElGamal encryption by Barthe et al.'s (2009). This equivalence, named `mult_pad` in that paper, proves the independence of the adversary's view from the hidden bit b that the adversary has to guess in the semantic security indistinguishability game, and thus shows that the adversary cannot do better than a random guess.

ElGamal encryption is parametric on a cyclic group of order q, and a generator g. Roughly stated, the equivalence says that if one applies the group operation to a uniformly distributed element of the group and some other element, the result is uniformly distributed, that is $z \overset{\$}{\leftarrow} \mathbb{Z}_q; \zeta \leftarrow g^z \times m_b$ and $z \overset{\$}{\leftarrow} \mathbb{Z}_q; \zeta \leftarrow g^z$ induce the same distribution on ζ (which is thus independent of b). To prove this, we modify the RAND effect to use random tapes of elements of $\mathbb{Z}_q$ rather than bitvectors, an define

```
let elgamal0 (m:group) : Rand group = let z = sample () in g^z
let elgamal1 (m:group) : Rand group = let z = sample () in (g^z) * m
```

and prove, again using mass_leq, the following lemma

```
val elgamal_equiv: m:group ⟶ c:group ⟶ Lemma
  (let f1, f2 = reify (elgamal0 m), reify (elgamal1 m) in
  mass f1 (point c) == mass f2 (point c))
```

B Additional command transformations

Our first equivalence, listed below, shows that if a command's read and write footprints are disjoint, then it is idempotent. The proofs of idem and the other lemmas below are perhaps peculiar to SMT-based proofs. In all cases, the proofs involve simply mentioning the terms reify (c ()) h, which suffice to direct the SMT solver's quantifier instantiation engine towards finding a proof. While more explicit proofs are certainly possible, with experience, concise SMT-based proofs can be easier to write.

```
let idem #rs #ws (c:cmd rs ws):
  Lemma (requires (disjoint rs ws)) (ensures ((c >> c) ~ c))
  = ∀_intro (λ h ⟶ let (), h1 = reify (c ()) h in
      let _ = reify (c ()) h1 in ()
      <: Lemma (equiv_on_h (c >> c) c h))
```

Next, we show elimination of redundant writes by proving that c1 >> c2 is equivalent to c2 if c1's write footprint is (a) a subset of c2's write footprint, and (b) disjoint from c2's read footprint.

```
let redundant_writes #rs1 #rs2 #ws1 #ws2
  (c1:cmd rs1 ws1) (c2:cmd rs2 ws2)
  : Lemma (requires (disjoint ws1 rs2 ∧ ws1 ⊆ ws2))
          (ensures ((c1 >> c2) ~ c2))
  = ∀_intro (λ h ⟶ let _ = reify (c1 ()) h, reify (c2 ()) h in
      () <: Lemma (equiv_on_h (c1 >> c2) c2 h))
```

C RHL Example

Following Benton (2004), we prove an example hoisting an assignment out of a loop:

$$\vdash \underbrace{\boxed{\begin{array}{l}\text{while } (I < N)\\ \quad X := Y + 1;\\ \quad I := I + X\end{array}}}_{L} \rightsquigarrow \underbrace{\boxed{\begin{array}{l}X := Y + 1;\\ \text{while } (I < N)\\ \quad I := I + X\end{array}}}_{R} :$$

$$\underbrace{\boxed{\begin{array}{l}I_{\text{left}} = I_{\text{right}} \wedge\\ N_{\text{left}} = N_{\text{right}} \wedge\\ Y_{\text{left}} = Y_{\text{right}}\end{array}}}_{\Phi} \Rrightarrow \begin{array}{l}I_{\text{left}} = I_{\text{right}} \wedge\\ N_{\text{left}} = N_{\text{right}} \wedge\\ Y_{\text{left}} = Y_{\text{right}}\end{array}$$

In other words, the judgement above preserves the invariant Φ stating that the two programs L and R compute the same values for I, N, Y, with X being neglected (which is already useful enough if X is known to be dead in the code following the while loops).

```
let proof () : Lemma (ensures (related L R Φ Φ)) =
  (* intermediate invariants for the loop bodies *)
  let Φ1 = Φ ∧ (X_right = Y_right + 1) in
  let Φ2 = Φ1 ∧ (X_left = X_right) in
  assert (related skip (assign X (Y + 1)) Φ Φ1); (* dead assign *)
  assert (related (assign X (Y + 1)) skip Φ1 Φ2); (* dead assign *)
  assert (related (assign I (I + X)) (assign I (I + X)) Φ2 Φ2); (* assign *)
  assert (related (seq (assign X (Y + 1)) (assign I (I + X)))
                          (assign i (I + X)) Φ1 Φ2); (* seq, elim. skip *)
  r_while (I < N) (I < N) (seq (assign X (Y + 1)) (assign I (I + X)))
          (assign I (I + X)) Φ1;
  (* seq, elim. skip *)
  assert (related L (while (I < N) (assign I (Y + 1))) Φ1 Φ)
```

r_while $B\ B'\ C\ C'\ \Phi$:

$$\frac{\vdash C \rightsquigarrow C' : \Phi \wedge B_{\text{left}} \wedge B'_{\text{right}} \Rrightarrow \Phi \wedge (B_{\text{left}} = B'_{\text{right}})}{\begin{array}{c}\vdash \text{while } B \text{ do } C \rightsquigarrow \text{while } B' \text{ do } C' : \Phi \wedge (B_{\text{left}} = B'_{\text{right}}) \Rrightarrow\\ \Phi \wedge \neg(B_{\text{left}} \vee B'_{\text{right}})\end{array}}$$

The proof shows that applications of RHL rules (including dead assignment rules) are actually syntax-directed, so that the only nontrivial effort needed is to provide the intermediate verification condition relating the bodies of the loops.

In more detail, for a given proposition ϕ, assert ϕ tries to prove ϕ and, if successful, adds ϕ to the proof context as a

EVar

$$S,\Gamma \vdash r \longrightarrow \langle S\,(r)\,,\Gamma\,(r)\rangle$$

EInt

$$\frac{i : \mathrm{int}}{S,\Gamma \vdash i \longrightarrow \langle i, \mathrm{L}\rangle}$$

EBinOp

$$\frac{S \vdash e_1 \longrightarrow \langle v_1, l_1\rangle \qquad S,\Gamma \vdash e_2 \longrightarrow \langle v_2, l_2\rangle}{S,\Gamma \vdash e_1 \oplus e_2 \longrightarrow \langle v_1 \oplus v_2, l_1 \sqcup l_2\rangle}$$

CAssign

$$\frac{\begin{array}{c} S,\Gamma \vdash e \longrightarrow \langle v_e, l_e\rangle \qquad \Gamma\,(r) = l_r \\ l_e \sqcup \mathrm{pc} \leq l_r \end{array}}{S,\Gamma,\mathrm{pc} \vdash r := e \longrightarrow S\,[r \mapsto v_e]}$$

CCondTrue

$$\frac{S,\Gamma \vdash e \longrightarrow \langle v_e, l_e\rangle \qquad v_e = 0 \qquad S,\Gamma,(\mathrm{pc} \sqcup l_e) \vdash c_1 \longrightarrow S_1}{S,\Gamma,\mathrm{pc} \vdash \text{if } e = 0 \text{ then } c_1 \text{ else } c_2 \longrightarrow S_1}$$

Figure 2. Semantics of the IFC monitor

fact that can be automatically reused by the later parts of the proof. To prove ϕ, proof search relies not only on the current proof context, but also on those lemmas in the global context that are associated with *triggering patterns*: if the shape of ϕ matches the triggering pattern of some lemma f in the global context, then f is applied (*triggered*) and the proof search recursively goes on with the preconditions of f. This proof search is actually performed by the Z3 SMT solver through *e-matching* (Moura and Bjørner 2007).

In our example , assert (related skip (assign X $(Y + 1)$) Φ Φ_1) tries to prove that an assignment can be erased; based on the syntax of both commands of the relation, e-matching successfully selects the corresponding dead assignment rule of RHL. In fact, this assert also allows specifying the intermediate condition Φ_1 that is to be used to verify the rest of the bodies of L and R, which cannot always be guessed by proof search. Alternatively, the user can also explicitly apply an RHL rule by directly calling the corresponding lemma, which is illustrated by the call to r_while to prove that the two while loops are related. In that case, the postcondition of the lemma is added to the proof context for the remainder of the proof. This way, the user can avoid explicitly spelling out the fact proven by the lemma; moreover, since the lemma to apply is explicitly given, the SMT solver only has to prove the preconditions of the lemma, if any.

This example is 33 lines of F* code and takes 25 seconds to check. This time could be improved substantially. However, perhaps more interesting, this experiment suggests developing tactics to automatically use Benton's RHL whenever possible, while still keeping the possibility to escape back to semantic approaches wherever RHL is not powerful enough. We leave this as future work.

D Soundness of an IFC monitor

Another popular technique for the enforcement of IFC are runtime monitors: the idea is to dynamically track the security labels of expressions and to check them at runtime in order to detect IFC violations, which cause the execution to halt. Here we implement an interpreter for the while language presented in §4.1 extended with the security monitor proposed by Sabelfeld and Russo (2009): a selection of the semantic rules is reported in Figure 2. The store S maps references to integers, while the store labeling Γ maps references to security labels, which are then used to derive labels for expressions. Assignments are subject to the expected security checks at run-time.

We embed the monitor in F*, obtaining a machine-checked proof of soundness for it. The interpretation functions for expressions and commands have the following signatures:

```
val interp_exp_monitor: store_labeling → exp → Reader (int ∗ label)
val interp_com_monitor: store_labeling → label → com → StExn unit
```

We prove termination-insensitive non-interference for interpretation with the monitor and capture this with the following lemma:

```
val dyn_ifc (s0:store) (s1:store) (env:store_labeling) (c:com) (pc:label) :
    Lemma (requires (low_equiv env s0 s1))
      (ensures (match (reify (interp_com_monitor env pc c)) s0,
                      (reify (interp_com_monitor env pc c)) s1 with
               | (Inl _, s0'), (Inl _, s1')→ low_equiv env s0' s1'
               | _ → ⊤))
```

Intuitively, we show that for any two low-equivalent initial stores, the two resulting stores are also low equivalent, if the interpretation with the monitor terminates without a runtime exception.

While the result looks similar to the one shown for the type system, there is a subtle difference in the enforced security property. Consider the following example where the label of hi is High and the label of lo is Low:

```
if (hi=0) skip else lo := 0
```

The assignment to a low reference on the else branch is leaking information about the value of the high reference in the conditional expression. Nevertheless, if the then-branch of the conditional is taken, the monitor will not report a violation, as it does not inspect the else-branch. This example does however not break our theorem, since our theorem only relates pairs of programs that terminate normally, while for all stores in which the else branch is taken, the execution of the interpreter halts with an error. The monitor is collapsing the implicit-flow channel into an erroneous termination channel, thereby enforcing error-insensitive non-interference. For comparison, notice that the (termination-insensitive) type system from §4.1 accepts a variant of the program above, in which the low assignment is replaced by a non-terminating loop.