ALEXANDRU IOAN CUZA UNIVERSITY OF IAŞI

FACULTY OF COMPUTER SCIENCE

MASTER'S THESIS

Secure F*-ML interoperability for IO programs

proposed by

Cezar-Constantin Andrici

Session: July, 2021

Advisors

Associate Professor Ştefan Ciobâcă, PhD

Tenured faculty Cătălin Hriţcu, PhD

ALEXANDRU IOAN CUZA UNIVERSITY OF IAŞI

FACULTY OF COMPUTER SCIENCE

Secure F*-ML interoperability for IO programs

Cezar-Constantin Andrici

Session: July, 2021

Advisors

Associate Professor Ştefan Ciobâcă, PhD

Tenured faculty Cătălin Hriţcu, PhD

Acknowledgements

I am not sure if it is common to have an acknowledgement section for a Master's thesis, but the work here was done over the last two years with the help and support of a lot of people, therefore I have to thank them.

I want to thank Ştefan for introducing me to research during my Bachelor's. I had his support during these years, which motivated me to participate further. With his support, I managed to participate in three top conferences and even got a travel grant to participate in VMCAI Winter School, New Orleans in 2020.

I want to thank Cătălin for the two great opportunities to work with him and his team during two fun and challenging internships at INRIA, Paris and MPI-SP, Bochum. Moreover, for accepting me as PhD student at MPI-SP, Bochum.

For this thesis, I collaborated with my two advisors and with Guido Martínez, Exequeil Rivas and Éric Tanter. I want to thank all of them for their advice and guidance during this thesis. We met to work together a few times each month in the last two years, and in this time, we tried a lot of topics and ideas until we decided on what to focus. For sure it was an interesting experience to work at the intersection of so many topics.

Thank to the people from the Faculty of Computer Science and from the University. I am really glad that I had the chance to focus on research during my Master's, and this will have not been possible without the opportunities offered by them.

Finally, I want to thank to my colleagues from C414, Andreea, Alex, Paul, Radu and Carmine, Jeremy, Thèo, Rob to make the last two years a fun experience, and for the many discussions about research and academia, and for the exchange of feedback and advice.

Contents

1 Problem and Motivation

Web servers are complex applications that handle sensitive information. Their security is essential because they usually are public, therefore exposed to a wide range of attacks. Even if there are different ways to verify and test if they satisfy the desired security guarantees, something as simple as installing a new plugin could require the entire process of certification to be restarted.

Most web servers have a plugin system through which third-party software components can be installed to extend the base functionality. Having a plugin system is a common practice, because the installation of a plugin is usually simple and adds complex functionality with little effort, but this is problematic as well because it comes with security risks. The installed plugin can have unintended behavior, steal secrets or install malicious code.

One way to verify if a program satisfies a specification is by using a verification-enabled language such as Coq, Dafny or F*. In these languages, a programmer can code the web server and give a specification for the desired security guarantees. Then, the language checks if the program satisfies the specification, and if not, it warns the programmer. An example of a specification may be: "the program never opens the file */etc/passwd*". The problem with these languages is that they need the entire source code of the application to check if the specification is satisfied, therefore it is not possible to verify the web server without also verifying the plugins. This is inconvenient, since we may not have access to the plugin's source code, or if we have, it implies to try to

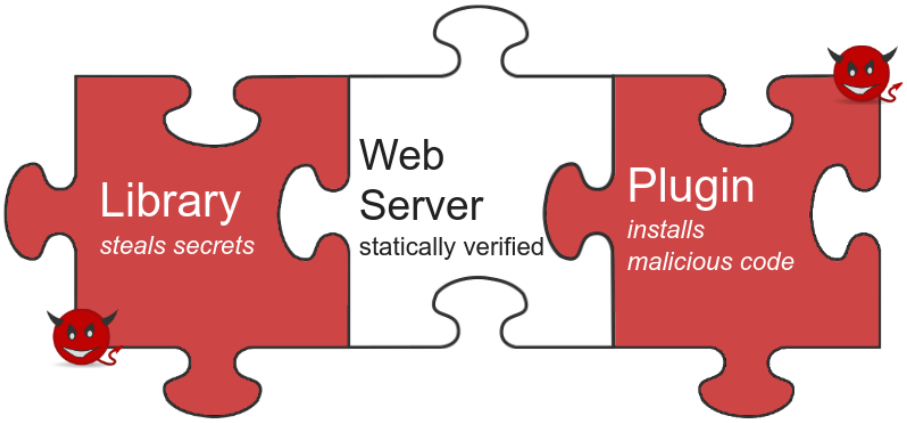

Figure 1: The statically verified web server is linked with a third-party plugin and library that are malicious and try to steal secrets, therefore breaking all the proved guaranties.

verify the code written by third parties which would take away the simplicity of using them.

This implies that after we tried and verified the web server, the specification is lost once the web server is linked with plugins since they may be adversarial. This is bad since the specification usually contains important security properties and correctness guarantees.

The same problem also makes it difficult to adopt static verification for large applications. Static verification is hard to do and takes a lot of effort. Even if the critical components would make sense to verify, it is not worth it, because the specification is lost once the critical component interacts with other unverified components.

Therefore, we propose to study the interoperability between verified and unverified code and how we are able to prove that the resulting program satisfies a specification. Our goal is to create a mechanism through which a programmer is able to integrate a statically verified component with an unverified piece of code, and still be able to prove safety properties about the whole program.

We study this problem in the context of the verification-enabled programming language F* [Swamy et al., 2016] developed at Microsoft Research and Inria. F* is used to verify big projects such as the whole HTTPS stack, including TLS 1.2 and TLS 1.3, and also the cryptographic primitives. After verification, F* extracts to OCaml, C, WASM or ASM code.

Contributions:

- We present a setup that enables writing top level specifications about a whole program composed by statically or dynamically verified components;
- We extend the IO Dijkstra monad in a novel way that enables mixing dynamic and static verification;
- We provide a proof embeded in F* to show that our model is correct;
- We present a case study about a stateless terminating web server, and we show it respects a top-level trace property.

2 Background

F* is a functional programming language with a complex type system aimed at program verification. It differs from other verification-enabled languages because in F* an expression has a type and a computational effect.

The following piece of code shows the type of the operation read.

```
val read : (fd:file_descr) → IO string
 (requires (fun (h:trace) → is_open fd h))
 (ensures (fun _ msg lt → lt = [Read fd msg]))
```

The operation read is in the effect IO (input-output) and returns a string. To be able to call this operation, the pre-condition must be met and in return it guarantees the post-condition. The specification of read requires the file descriptor given as an argument to have been open before, by using the operation openfile, and not closed in the meantime. This property is checked by looking back in the trace of events to see if an open event was registered for that file descriptor. The operation read guarantees that during its execution only a read is done from the file descriptor given as argument.

A way to explain IO traces is with a simple example. Let the program webserver1, be:

```
let webserver1 () :
    IO unit (ensures (fun h _ lt → (Openfile "/etc/passwd") not in lt)) =
  let fd = openfile "data.csv" in
  let r = read fd in
  close fd
```

If the program webserver1 runs successfully it produces the following IO trace: [Openfile "data.csv"fd; Read fd "some-data"; Close fd], where fd is returned by the openfile operation. The trace is a list of events corresponding to the order in which input-output operations were called. To write trace properties, we use two variables in the specification: h and lt. Variable h represents the history of events until the function was called. Variable lt represents the local trace, meaning the trace of events that occurred during the execution of the function.

Using static verification in F*, we can check if the program webserver1 satisfies the trace property that is in the post-condition: "the program does not open the file */etc/passwd*"; in this case, the program satisfies the trace property, because it opens *data.csv* and not */etc/passwd*. The advantage of using static verification to enforce trace

properties is that the trace and the enforcement of the trace properties exist only at the specification level, therefore there are no trace and no checks at runtime.

Lets take another program, webserver2, with the same specification, where plugin is an unknown function:

```
let webserver2 plugin :
    IO unit
      (ensures (fun _ _ lt → (Openfile "/etc/passwd") not in lt)) =
  let fd = openfile "data.csv" in
  let r = plugin fd in
  close fd
```

For the program webserver2, we can not say how a successful trace looks like. We can say it starts with an Openfile, but we do not know what events is the plugin producing. Therefore, there is no way to use static verification to check that the program webserver2 satisfies the property "the program does not open the file */etc/passwd*".

Runtime verification [Bartocci et al., 2018, Lamport and Schneider, 1984, Schneider, 2000] can help us to verify the program webserver2. The way runtime verification works is by adding a monitor that observes each input-output operation that is happening during the execution. Before each operation, the monitor checks if the specification is satisfied; if so, it allows it; otherwise, it halts the execution. This is called *instrumentation* and we say *the program webserver2 is instrumented*. A big drawback is that these checks happen at runtime and they can add quite a penalty to performance. Moreover, the trace must be allocated at runtime, meaning it also consumes memory. Actually, for runtime verification automatas are preferred instead of traces for efficiency reasons. We prefer trace properties because they are much easier to work with and think about, but we are careful to not make our solution to only work with traces.

Since runtime verification is a field on its own, we try to create a mechanism to make possible the static verification of a program such that later can be linked with another one instrumented. We assume that tools that can instrument programs in the way we want exist and can be used.

3 Approach and uniqueness

We propose the following setup that combines static verification with dynamic verification in what we call hybrid enforcement of trace properties. Be a whole program W that is composed of the partial program P and the context C, where P is statically verified and C is arbitrary and instrumented, we would like to show about W that it satisfies the IO trace property π.

The benefits of this setup are:

- The partial program P can be statically verified to satisfy π;
- P can be linked to arbitrary contexts created by third-party developers if they are instrumentable;
- Fewer runtime checks are needed. P is already statically verified, therefore only C needs instrumentation to satisfy π. Therefore, fewer runtime checks are added, which should increase performance compared to instrumenting the entire W (see Figure 2);
- It is possible to show that W satisfies the trace property π.

What is monitorable? Havelund et al. [2018] define monitorability purpose as detecting violations of properties, therefore they consider safety properties over infinite

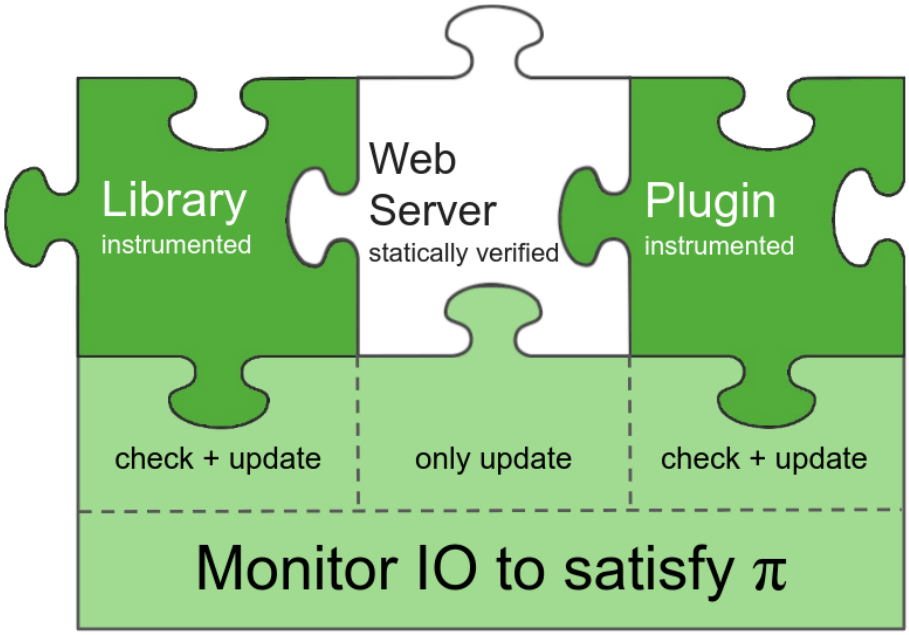

Figure 2: The statically verified web server is linked with a third-party plugin and library instrumented and monitored.

execution. Safety properties are sets of infinite traces. Any violation of a safety property can be detected with a finite sequence, therefore the detection of a bad prefix is computable. For us, monitoring has the purpose to detect and prevent violations of properties. Since our monitoring is done by using instrumentation, before any event is produced, the monitor tests if the following operation violates the safety property, therefore the execution never produces a bad trace. This also implies that the safety property must allow for the instrumentation to halt the execution.

F* does not have a primitive effect to enable verification of IO programs, therefore we define our own. We define operations for file management and for socket communication, but this is one possible instantiation. In our implementation of the input-output effect, the output type of the *i.o.* operations is either an error or another type, therefore all i.o. operations by default can return errors.

The pre- and post-condition are defined over finite traces that are "computed" by observing the computation; each time an i.o. operation is called, an event is appended at the beginning of the trace; an event is a tuple of the name of the operation, the arguments and the result. The properties restrict the behavior of the program to only a subset of all possible traces. The traces exist only at the specification level and are not actually computed at runtime.

The pre-condition allows writing properties over the entire history of events, while the post-condition is written only over the result of the computation and the events produced by it. The specification of read, write, close requires the file descriptor given as an argument to have been open before, by using the operation openfile, and not closed in the meantime.

Up to this point, we presented an effect that allows verifying statically IO programs, but does not support dynamic checks of trace properties because the trace exists only at the specification level.

To enable dynamic verification, we extend our IO effect with a new silent operation called get_trace obtaining a new effect we call IIO, for "instrumented IO". The operation is silent because it does not produce events. The get_trace operation guarantees that it returns the trace computed until now, without producing an event. This operation allows us to mix static and dynamic checking easily, creating a seamless interoperability between the two for enforcing IO trace properties. As a future work, we think the effect IIO can be used to enable gradual enforcement of trace properties.

The IIO monad enables us to convert pre-conditions to runtime checks by using

a technique called wrapping. We call this transformation wrap. It consists of wrapping an initial function in a new function with trivial pre- and post-conditions. The new function adds the pre-condition as a runtime check before calling the initial function. The original post-condition is changed to accept the possibility of failure. In the following piece of code we present the exported version of the operation read.

```
let read' (fd:file_descr) : IIO (maybe string)
 (requires (fun h → True))
 (ensures (fun h r lt →
   (is_open fd h ⟹ Inl? r ∧ lt = [Read fd (Inl?.v r)]) ∧
   (˜(is_open fd h) ⟹ lt = [] ∧ r == (Inr Contract_failure)))) =
 if is_open fd (get_trace ()) then (Inl (read fd))
 else (Inr Contract_failure)
```

4 Case study: The Web Server

We report on a nontrivial case study that uses these ideas to extend a web server previously verified in F⋆ with a safe ML-plugin mechanism. Our goal is that the mechanism described above be usable to integrate a verified web server in F⋆ with plugins written in OCaml, and show that global properties are enforced.

One use case we target is to be able to define a global property π that restricts the behaviour of the entire application. A second use case is to be able to reason about the values returned by the plugin to the web server.

We showcase in Figure 3 a stateless terminating web server and we present how our mechanism works. The web server accepts as an argument a plugin. In our case, the web server is the verified partial program P, and the plugin is the arbitrary instrumented C. The web server requires, before being called, that the entire history is empty. Then the web server opens a socket, accepts the first connection, and passes it to the plugin. We want to show about the whole program (web server + plugin) that it satisfies the safety property π: "the program never opens the file */etc/passwd*".

The plugin's specification has two parts: 1) the safety property π was enforced during the entire execution of the plugin, and 2) if the execution of the plugin was successful, the message returned has a length smaller than 500.

The safety property π is enforced statically and dynamically, therefore it is implemented using the runtime checks pi that accepts the trace until now and the next

operation to be executed. It returns true if the execution should continue, or false if it should be halted. We show in Figure 4 the runtime check pi that is used together with hybridly_enforced to specify the property π.

4.1 Instrumentation of the plugin

We give an example of an adversarial plugin in Figure 5. It is a simple example that allows us to illustrate how the instrumentation works. First, since the plugin is written by a third party, we should give it a effect that models a safe subset of ML. The instrumentation should bring the unverified code into the effect IIO, but this is not always possible because it may contain other effects such as state, exceptions or non-termination, which are not supported by the IIO effect. Therefore, for now we define a synonym effect for IO that does not have pre- and post-conditions. We call this effect MIO, it stands for "ML-ish IO". Since the plugin has this effect, we know it only contains i.o. operations, but we do not know anything about how they are used, therefore we say it has "no specification".

```
type plugin_type = fd:file_descr →
  IIO (maybe string) (requires (fun h → is_open fd h))
            (ensures (fun h r lt → hybridly_enforced pi h lt ∧
                                   match r with
                                   | Inl msg → length msg < 500
                                   | Inr err → True)

let webserver (plugin:plugin_type) :
  IIO unit (requires (fun h → h = []))
           (ensures (fun _ _ lt → hybridly_enforced pi [] lt)
  let s = socket () in
  setsockopt s SO_REUSEADDR true;
  bind s "0.0.0.0" 3000;
  listen s 5;
  ...
  let client = accept s in
  plugin client;
  ...
```

Figure 3: The webserver used in the case study and the expected type for the plugin.

```
let pi (h:trace) action : bool =
  match action with
  | Openfile file_name → file_name != "/etc/passwd"
  | _ → true

let rec hybridly_enforced pi h lt : bool =
  match lt with
  | [] → true
  | hd :: t →
    let action = convert_event_to_action hd in
    if pi h action then hybridly_enforced pi (hd::h) t
    else false
```

Figure 4: The pi is a runtime check that can be used to enforce the IO trace property π: "the program never opens the file */etc/passwd*". The method hybridly_enforced is used to convert the pi into a trace property.

We define a new transformation function, instrument, that accepts a safety property π and handles each i.o. operation of a function by first checking if the property π is satisfied. If so, then it executes the corresponding i.o. operation; otherwise, it returns a contract failure. The resulting plugin after instrumentation is conceptually equivalent to the plugin' in Figure 6.

It is possible to write such a transformation function that instruments another function because F* has introspections capabilities. Since IO is an effect, F* can reveal its computational representation and refine it to satisfy a safety property π and then create a computation of the effect IIO.

4.2 Extraction of the web server

Since the web server is written in F*, it must be extracted to a different target language to actually compile and run it. F* supports extraction to a few languages. We focused on extraction to OCaml. During extraction, an actual OCaml implementation of the i.o. operations and get_trace must be provided.

Because of our approach, there is great flexibility on how the monitoring can be done at runtime. We can take advantage of existing work that automatically extracts a monitor from the specification that must be enforced. Depending on how the monitoring and the instrumentation are done, the implementation may differ. The monitor

```
let plugin (fd:file_descr) : MIO string =
  let fd' = openfile "/etc/passwd" in
  "message"
```

Figure 5: Example of adversarial plugin.

```
val plugin' : plugin_type
let plugin' fd =
  let r = (
    let fd' = if pi (get_trace ()) (Openfile "/etc/passwd") then
                openfile "/etc/passwd"
              else throw Contract_failure in
    "message"
  ) in
  if length r < 500 then r
  else throw Contract_failure
```

Figure 6: The adversarial plugin from Figure 5 instrumented.

has the liberty to observe only some operations and to store the trace where and how it finds efficient and secure [Pothier and Tanter, 2011].

The simplest implementation is for each i.o. operation to append an event to the trace and the get_trace operation to return the entire trace. This is done by wrapping each i.o. operation in a new function that updates the trace before calling the operation itself.

$$\begin{array}{ccccc} \mathrm{P}^S_\pi : \mathrm{IIO\ a}\ \pi & \bowtie^S_\pi & \mathrm{C}^S_\pi : \mathrm{IIO\ a}\ \pi & = & \mathrm{W}^S_\pi : \mathrm{IIO\ a}\ \pi \\ \Big\downarrow \mathrm{extract} & & \Big\uparrow \mathrm{instrument} & & \\ \mathrm{P}^T_\pi : \mathrm{MIIO\ a} & \bowtie^T_\pi & \mathrm{C}^T : \mathrm{MIO\ a} & = & \mathrm{W}^T : \mathrm{MIIO\ a} \end{array}$$

Figure 7: The model consists in the source language (top) and the target language (bottom). To model the source language we used the monadic effect IIO; to model the target language we used MIIO for the partial program and MIO for the context.

4.3 Top-level security guarantee

We model our mechanism in F* using a shallow embedding. We do this to show global security guarantees are enforced if our mechanism is used. The proof is mechanized in F*. Until now, we used P and C to denote the verified partial program and the unverified context, but for the proof we need to distinguish between the partial program in the source and in the target. Therefore, we use P_π^S and P_π^T. The P_π^S denotes the verified web server in F* and P_π^T the extracted web server. Equivalently, C^T is the plugin before instrumentation and C_π^S is the instrumented plugin. We show in Figure 7 how the partial program and the context are linked in the source and in the target. The $\bowtie_\pi^T$ denotes the linking in the target which returns a whole program. The linker in the target language does the instrumentation of the C^T; this is why it is indexed with π.

The property we show about our mechanism is the following, where the down arrow is the compilation from source to target, and Beh returns the set of traces possible by the whole program.

$$\forall \pi\ P_\pi^S\ C^T.\ \mathsf{Beh}(C^T \bowtie_\pi^T (P_\pi^S \downarrow)) \subseteq \pi$$

The proof is easy if we think about it in terms of web server and plugin. The linker instruments the plugin such that it would match the type-and-effect expected by the extracted web server. However, we already showed statically that the web server satisfies the trace property π if such a plugin is provided. □

We present this proof in greater detail in section 7, but we use the intuition from here.

5 Defining effects in F*

In theory, computational effects can be: *pure*, *state*, *exceptions*, *input-output*, *divergence*, *probabilities*, *concurrency*, or a combination of these.

In F*, there is one base "effect" called Pure. For example, an expression whose inferred computational effect is Pure is an expression that always terminates and does not have any other side effects except *pure*. But, the effect Pure is much more complex; Pure is indexed by the returned type and a pre- and a post-condition, therefore the inferred type of an expression would look like Pure a pre post with the semantics: the expression is pure, can be evaluated if pre holds, it returns a result of type a, and it guarantees post.

In theory, all effects in F* should be defined over the Pure effect, and they should form a lattice where the effects are connected by lifts. In practice, there are other primitive effects such as Dv (for possibly divergent code) and ML (for arbitrary code).

Dijkstra [1975] proposed a weakest pre-condition sematics that relates results and final states to pre-conditions on input states. It was found out that using monads it was a natural way to implement the calculus to compute the weakest pre-condition of computations, therefore it became natural to use the Dijkstra monad for program verification [Swamy et al., 2013, Jacobs, 2015]. Ahman et al. [2017] showed that Dijkstra monads can be obtained for free if the underlying computational monad is in the continuation-passing style. Maillard et al. [2019] have shown that "any monad morphism between a computational monad and a specification monad gives rise to a Dijkstra monad" that makes it easy to verify IO programs. They implemented a framework for defining primitve Dijkstra monads in F*. We use the mechanism introduced by Rastogi et al. [2020], that enables defining Dijkstra monads on top of the Pure effect, to define our monadic effect, IO, for use by the trusted code.

5.1 The IO Effect

In F* an easy way to define new effects is by using Dijkstra monads which are a bundle between a computational monad, a specification monad and an effect observation between the two.

For our IO monadic effect, we chose a computational monad that is parametric in the underlying primitive operations by using a free monad [Bauer and Pretnar, 2015] that can accept any interface [Letan et al., 2021]. In addition, the free monad can be

used to implement other effects such as exceptions, state and non-determinism by extending the interface [Bauer and Pretnar, 2015] and we plan to integrate them in future work.

```
type op_sig (op:Type) = { args:(op → Type); res:(op → Type); }

type free (op:Type) (s:op_sig op) (a:Type) : Type =
| Call : l:op → s.args l → (s.res l → free op s a) → free op s a
| Return : a → free op s a

let free_return (op:Type) (s:op_sig op) (a:Type) (x:a) : free op s a = Return x

let rec free_bind (op:Type) (s:op_sig op) (a b:Type) (l:free op s a) (k:a→ free op s b) : free op s b = (* omitted *)
```

Computational monad - io. For simplicity, we instantiate free with the following operations: Openfile, Read and Close. Our example does not contain the usual write primitive, but it can be defined as other operations. These account for one possible instantiation of the monad. The approach is general and it is easy to extend.

The output type of the i.o. operations is either an error or another type. We use the type maybe a that is equal to either a exn, where exn are exceptions. This behavior is similar to the one in C, where the i.o. operations return a value greater than 0 if the operation was successful, or less than 0 otherwise. We do not include the exceptions effect, that could remove the need for returning maybes, because it would hide some of the implications of the design choices we did.

```
type maybe a = either a exn

type io_cmds = | Openfile | Read | Close
let io_args (cmd:io_cmds) : Type =
  match cmd with | Openfile → string | Read → file_descr | Close → file_descr
let io_res (cmd:io_cmds) : Type =
  match cmd with | Openfile → file_descr | Read → string | Close → unit
let io_resm (cmd:io_cmds) = maybe (io_res cmd)
let io_sig : op_sig io_cmds = { args = io_args; res = io_resm }

type io a = free io_cmds io_sig a
let io_return (a:Type) (x:a) : io a = free_return _ _ _ x
```

Specification monad - hist. The role of a specification monad is to specify properties of our i.o. operations. We chose a specification monad that allows writing a

pre-condition over the entire history of events, while the post-condition is written over the result of the computation and the events produced by it (we refer to it as local trace).

```
let hist_post a = a → lt:trace → Type0
let hist a = h:trace → hist_post a → Type0
let hist_return (a:Type) (x:a) : hist a = fun _ p → p x []

let hist_bind (a b:Type) (w : hist a) (kw : a → hist b) : hist b =
  fun h p →
    w h (fun r lt →
      kw r ((List.rev lt) @ h) (fun r' lt' → p r' (lt @ lt')))
```

Effect observations. An effect observation relates the computational monad with the specifications, providing insight to the potential effects of the computation. We associate to each operation of the free monad an event represented by a dependent pair of type: cmd:op & io_args cmd & io_resm cmd. Each time an operation is called, an event that contains the arguments and the result of the operation is appended to the local trace. The local trace is just a list of events.

```
let rec io_interpretation #a (m : io a) (p : hist_post a) : Type0 =
  match m with
  | Return x → p x []
  | Call cmd args fnc → ∀res. (io_interpretation (fnc res) (fun x lt → post x ((| cmd, argz, rez |) :: lt)))
```

TODO: explain better how layered effects are used to make a Dijkstra monad

We bundle the 3 elements into a Dijkstra monad we call IO using the Layered Indexed Effects mechanism of F* Rastogi et al. [2020], and now we can write primitives as read. We also define a synonym effect MIO (stands for "MLish IO"), which treats the special case when the pre- and post-conditions are trivial. MIO is for IO, what Tot is for Pure.

```
effect MIO (a:Type) = IO a (fun _ → True) (fun _ _ _ → True)
```

5.2 The Instrumented IO Effect

As one can expect, the event history from the specification monad cannot be accessed by the computational monad to test dynamically if some conditions hold. We could extend the computational monad with state and keep track of another history

at the computational level, but that would not be very modular. Instead, we chose to extend our free monad with one more operation, called GetTrace, that returns the trace when is called. Therefore, we assume that when running the code, the operation returns the actual trace at that moment.

To accommodate this change, we define the type cmds and redefine the type io_cmds as a refinement of cmds. We also define a new signature called inst_sig that contains only the operation GetTrace. Our new computational monad would be called IIO from "instrumented io" and its signature is the sum between io_sig and inst_sig.

```
type cmds = | Openfile | Read | Close | GetTrace
type io_cmds = x:cmds{x ≠GetTrace}
type inst_cmds = x:cmds{x = GetTrace}
let inst_args (cmd:inst_cmds) : Type = match cmd with | GetTrace → unit
let inst_res (cmd:inst_cmds) : Type = match cmd with | GetTrace → trace
let inst_sig : op_sig inst_cmds = { args = inst_args; res = inst_res }

let iio_sig = add_sig cmds io_sig inst_sig
type iio a = free cmds inst_sig a
```

To obtain a Dijkstra monad from iio, we use the same specification monad, hist, and we slightly change the effect observations in two ways. The first change is that it now computes the history along the way, and second is that it treats GetTrace specially. We treat it separately because the operation does not produce a visible event and because we know the value returned by it, which is the computed history until now (our assumption).

```
let rec iio_interpretation #a (m : iio a) (h : trace) (p : hist_post a) : Type0 =
  match m with
  | Return x → p x []
  | Call GetTrace args fnc → iio_interpretation (fnc h) h p (** inst_cmds **)
  (** io_cmds **)
  | Call cmd args fnc → ∀res. (let e = (| cmd, argz, rez |) in
      iio_interpretation (fnc res) (e::h) (fun x lt → post x (e :: lt))
```

We also use the Indexed Layered Effects mechanism to create the effect IIO. Obviously, it is very easy to define a lift from IO to IIO since one is a subset of the other.

5.3 From pre-conditions to runtime checks

With this new effect, we can access the trace in the computational monad, therefore, now it is possible to convert an IO pre-condition into a runtime check.

Propositions in F* live in the type Type0, but this type can not be converted into a boolean expression, since not all propositions are decidable. Instead of decidable, we define the type class checkable for predicates p : t → Type0 for which there is a function check : t → bool s.t. $\forall x.\text{check}\ x \implies p\ x$. For soundness, it does not matter which check function is provided if check is a sub approximation of p. A simple instance is that any bool function is a checkable predicate.

```
class checkable (#t:Type) (p : t → Type0) = { check : (x:t → b:bool{b ⟹ p x}) }
instance general_is_checkable t (c : t → bool) : checkable (fun x → c x) = { check = fun x → c x }
```

We further define the function wrap_iio that accepts a IIO a pre post computation and exports it to IIO a True True. The wrap_iio wraps the original function in a new function where first it tests dynamically if the pre-condition holds. Because of that, the output of the function must be changed to accomodate either a result or a contract failure. Because we use quite often IIO a True True, we define a synonym effect called MIIO a which stands for "MLish instrumented IO".

```
let wrap_iio (#t1:Type) (#t2:Type) (pre:t1→ trace→ Type0) {| d:checkable2 pre |}
  (post:t1→ trace→ maybe t2→ trace→ Type0)
  (f:(x:t1) → IIO t2 (pre x) (post x))
  (x:t1) : MIIO (maybe t2) =
  if d.check2 x (get_trace ()) then (Inl (f x)) else (Inr Contract_failure)
```

MIIO is a synonym effect defined on top of IIO. We say it has "no specification". By no-specification we mean that the pre- and post-conditions are True. But, even if the MIIO effect has no pre-conditions on its own, if it wants to call an IIO that has a pre-condition, it still needs to prove that the pre-condition is respected. Also, since MIIO has "no specification", it means any computation in MIIO can be safely extracted by F*, therefore, we define an instance of the type class ml.

```
effect MIIO (a:Type) = IIO a (fun _ → True) (fun _ _ _ → True)
```

If you get lost with so many effects, I hope Figure 8 will make things easier. Also, as a recap, the four effects stand for:

- IO - input-output operations **with** pre- and post-conditions

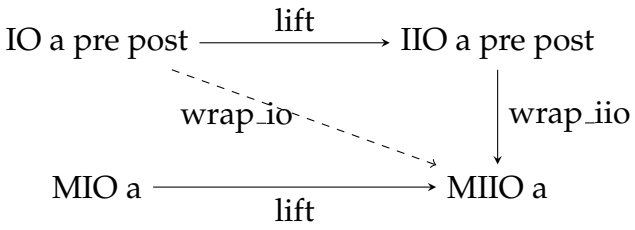

Figure 8: Secure transformation functions between effects. The lift from IO to IIO automatically works from MIO to MIIO since they are synonyms. The dashed line can be obtained by composing the lift with the wrap function.

- MIO - input-output operations **without** pre- and post-conditions
- IIO - instrumented IO operations **with** pre- and post-conditions
- MIIO - IIO operations **without** pre- and post-conditions

6 Secure extraction mechanism for F*

We mentioned that pre-condition must be converted into runtime checks during extraction for preserving security properties, and we showed that the top-level specification hold. However, the type system in F* is much more powerful and supports rich types that do not have an equivalent in the target language (e.g. OCaml). Therefore, in this section we introduce an extraction mechanism that is more general.

6.1 Extracting rich types

We start here by explaining how security guarantees are lost by extraction with a more basic example. We choose as a basic example the recursive pure factorial function with a refinement on the argument. In Figure 9, on the left we present the implementation of the factorial function in F* and on the right the resulting code after the extraction to OCaml. We can see that the refinement is erased during extraction, meaning that the pure (total) factorial function in F* is now divergent in OCaml for inputs smaller than 0 (e.g., factorial −5). This is a problem because the function now has new behavior we do not want.

```
val factorial: n:int{x≥ 0} → Tot int
let rec factorial n =
  if n = 0 then 1 else n ∗ (factorial (n − 1))
```

```
let rec factorial n =
  if n = 0 then 1 else n ∗ (factorial (n − 1))
```

Figure 9: Example of F* code (on the left) extracted to OCaml (on the right).

Inspired by previous work in gradual and hybrid types, we implement an automatic mechanism that wraps strongly typed functions inside a weakly typed version.

We implement the mechanism by defining the following collection of type classes: ml, exportable and importable. The ml class is used to label weak types. Weak types are types from F* that have a corresponding type in OCaml. These types are extracted by F* as they are, therefore we consider they are extracted safely. For brevity, we give example of only a few instances: for the type int and file_descr, for the type option a and for the arrow in the effect Tot which has constraints on the argument's and return's type. Therefore, an arrow in Tot can be safely extracted if its input and output types are weak types labeled with ml.

```
class ml (t:Type) = { }
```

```
instance ml_int : ml int = { }
instance ml_file_descr : ml file_descr = {}
instance ml_option a {| ml a |} : ml (option a) = { }
```

The classes exportable and importable are used to define transformation functions between strong types and weak types. A type a is exportable to type b if b is an ml type and there is a function export from a to b. Similar, a type a is importable from a type b if b is a ml type and there is an import function from b to option a (an import can fail, in which case we return None). Two obvious instances are that any ml type is exportable and importable with itself using the identity function. Another instance is for the type option a which is exportable if a is exportable.

```
class exportable (t : Type) = { etype : Type; export : t → etype; ml_etype : ml etype }
class importable (t : Type) = { itype : Type; import : itype → option t; ml_itype : ml itype }

let mk_exportable (#t1 t2 : Type) {| ml t2 |} (exp : t1 → t2) : exportable t1 =
  { etype = t2; export = exp; ml_etype = solve }
let mk_importable (t1 #t2 : Type) {| ml t1 |} (imp : t1 → option t2) : importable t2 =
  { itype = t1; import = imp; ml_itype = solve }

instance exportable_ml t {| ml t |} : exportable t = mk_exportable t (fun x → x)
instance importable_ml t {| ml t |} : importable t = mk_importable t (fun x → Some x)

instance exportable_option t {| exportable t |} : exportable (option t) =
  mk_exportable t (fun x → match x with | Some x' → Some (export x') | None → None)
```

Using these four type classes we add an instance to export refined types. A refined type is importable if the predicate used for refining is checkable. The import function then checks if the predicate holds and returns an option.

```
instance importable_refinement t {| d:importable t |} (rp : t → Type0) {| checkable rp |} :
  Tot (importable (x:t{rp x})) =
  mk_importable (d.itype) (fun (x:d.itype) →
      match import x with | Some x' → if check #t #rp x' then Some x' else None | None → None)
```

We extend the type classes for the other ml types above and also with instances that show how pairs and dependent pairs are exportable and importable.

To finish our factorial example, we define a extract_tot function for Tot arrows. The function says that a pure total arrow (a → Tot b) is *extractable* if the input type is importable and the output type is exportable. Since the input type is importable, this

means the import of the input may fail. The Tot effect does not support exceptions, therefore, because the import may fail, the extracted function must have the output type option b. Thinking about our factorial example, the input type (x:int{x > 0}) is a refinement type with a checkable predicate and the output type (int) is ml.

```
let extract_tot #a #b {| d1:importable a |} {| d2:exportable b |}
  (f:(a → Tot b)) : d1.itype → Tot (option d2.etype) =
  fun (x:d1.itype) →
    match import x with | Some x' → Some (export (f x')) | None → None
```

To safely extract our factorial function, we use the extract_tot function to wrap it with the contract and then extract the wrapped version to OCaml.

Right now, the extracted factorial looks like this in OCaml. Prims.int is the type of mathematical integers. Obj.magic is an unsafe primitive from OCaml which is widely used in F* extraction mechanism. Obj.magic is changing the actual type of the argument to the expected type with no checking.

```
let (extracted_factorial : Prims.int → Prims.int FStar_Pervasives_Native.option) =
  fun uu___ →
    (Obj.magic
       (export
          (exportable_arrow
             (importable_refinement (importable_ml ml_file_descr) ()
                (Obj.magic (general_is_checkable (fun x → x ≥ Prims.int_zero))))
             (exportable_ml ml_file_descr))
          factorial))
      uu___
```

The code above is equivalent to the following simplified version:

```
let extracted_factorial : Prims.int → Prims.int FStar_Pervasives_Native.option) =
  fun x →
    if x > 0 then Some (factorial x)
    else None
```

6.2 Extracting arrows

A first question may be why we did not define a ml instance for Tot arrows, since F* knows how to extract them by default. There are several reasons.

First, arrows are not importable. A Tot arrow is not importable, because its semantics implies the function is pure (we understand by pure that the function does not

have any side-effects, and we treat divergence as a side-effect, therefore a pure function is also a total function). To be able to verify that an arbitrary function respects these properties, we would need powerful introspection capabilities which languages like ML do not have. Therefore, it is possible to import the input or output type of the arrow, but not the "effect" itself. We conclude this paragraph with the following statement that we use further to explain our design decisions:

Statement 1 *Arrows are not importable at runtime.*

Second, since Statement 1 applies, the extraction of higher-order types gets complicated. See section 6.4 related to higher-order.

Third, complex effects such as IO use ghost state at the specification level, but to enforce properties at runtime, the state has to be materialized. This problem is better described in section 5.

6.3 Extracting arrows with specs

There are other effects in F* that allow defining computations with pre- and post-conditions, such as Pure. Pure is just the effect Tot + pre- and post-conditions, therefore they can be mixed. We can rewrite our previous factorial function using this effect (figure 10).

```
val factorial': n:int → Pure int (requires (n ≥ 0)) (ensures (fun r → True))
let rec factorial' n = if n = 0 then 1 else n * (factorial (n – 1))
```

Figure 10: Factorial implemented in the *Pure* effect in F*.

For such arrows with pre- and post-conditions, we propose a two-step exporting: first we export the specs of the function and second, we export the input and output types.

The first step involves wrapping the initial function in a new function with trivial pre- and post-conditions. The new function adds the pre-condition as a runtime check before calling the initial function. The original post-condition is not checked by the new function since it does not affect the correctness.

For the Pure effect, an arrow with the type a → Pure b pre post becomes after the first step a → Tot (option b). As example, we show here the wrap_pure, which does the wrapping of the specs for Pure.

```
let wrap_pure (a b:Type) (pre:a → Type0) {| d:checkable pre |} (post:a → b → Type0)
  (f: (x:a → Pure b (pre x) (post x))) (x:a) : Tot (option b) = if d.check x then Some (f x) else None
```

The second step involves weakening the input and output types, meaning that a → Tot b becomes a′ → Tot (option b′), where a′ and b′ are weak types corresponding to a and b.

Putting the two steps together, we write our extract function for Pure arrows. A Pure arrow is extractable if the input type is importable, the output type is exportable and the pre-condition is checkable.

```
let extract_pure #t1 #t2 {| d1:importable t1 |} {| d2:exportable t2 |}
  (pre : t1 → Type0) {| d3:checkable pre |}
  (post : t1 → t2 → Type0)
  (f:(x:t1 → Pure t2 (pre x) (post x))) : d1.itype → Tot (option d2.etype) =
    fun (x:d1.itype) →
      match import x with
      | Some x′ → export (wrap_pure pre post f x′)
      | None → None
```

Figure 11 shows graphically what was explained until now. On the left, export and import are defined only on types that are not arrows. On the right side, it is shown the two-step exporting process for an arrow. Once an arrow has ml types and has no pre- and post-condition, F* extracts it safely to OCaml.

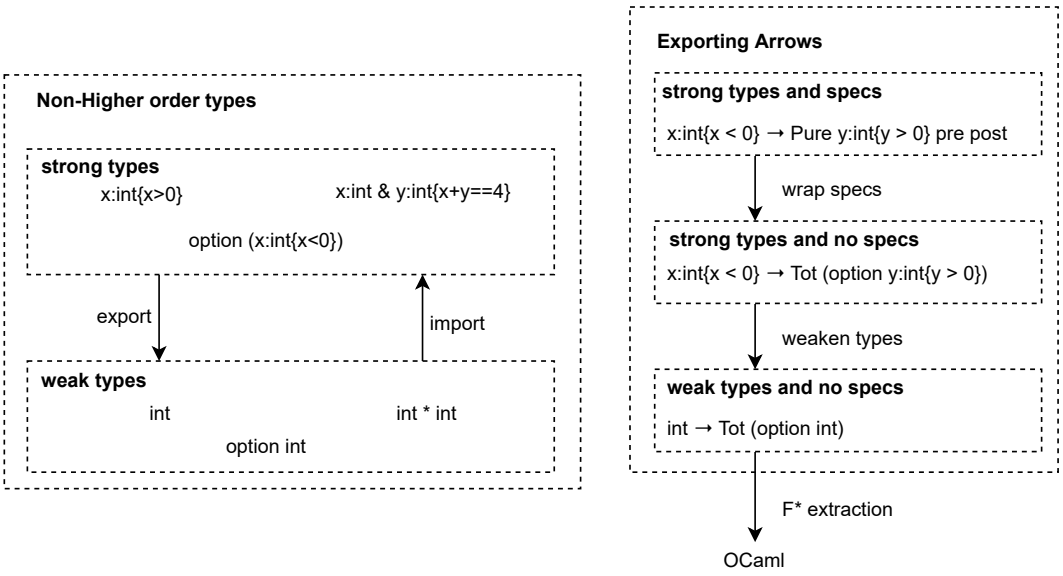

Figure 11: Diagram showing all methods together

This section may seem simple since pre- and post-conditions for the Pure effect

may be encoded as refinements, but later, we present how to extract the IO effect which has specs over a ghost state.

6.4 Extracting higher-order arrows

Working with higher-order functions is more complicated. For example, it is possible to define the following function f : g:(unit → Tot unit) → Tot unit. The semantics of the type of f is: f is a pure function that accepts a pure function g as argument.

Extracting f to a language as OCaml, means erasing the effects, therefore extracted f would accept any function of type unit → unit (e.g. of attack, let rec g () = g () where g is divergent). To preserve the semantics, it will imply that f should reject any g that is divergent, but OCaml does not support this kind of introspection. Therefore, f can not protect itself from adversarial code to preserve its semantics. This problems extends to all effects.

Statement 2 *Since arrows are not importable at runtime (Statement 1), arrows that expect an arrow as an argument can not be safely extracted.*

To be able to extract to OCaml, we assume the linker has the power to introspect and instrument g such that it becomes a pure function. Therefore, if the introspection says that g respects the specification or that g is instrumentable, f is linked with g or with the instrumented version of g. One problem raises from function g being instrumented. Since the instrumentation may stop the execution, the instrumented version of g has two possible outcomes: a result or a failure raised by the instrumentation. This means that the instrumented g needs a different type: unit → Tot (option unit). Therefore, function f in effect Tot can not accept an instrumented g because the types do not match.

With the previous assumption, that the linker can instrument g, it is possible to write a safely extractable f′ that expects an instrumented g as argument.

Statement 3 *Arrows that expect instrumented arrows as arguments, can be safely exported by relying on the linker to do a proper instrumentation.*

Statement 4 *Arrows that expect instrumented arrows as arguments, must accept failure of the instrumented arrow as an outcome. (e.g., as an* option *or as an exception)*

Right now, we can not provide an example of a linker that introspects functions and decides if a function is in Tot or instruments them to be in Tot, because we do not have a reifiable effect for non-termination.

What about a function p: unit → Tot q:(unit → Tot unit)? Function p is pure and returns a pure function q. Since p is verified, p is guaranteed to return a pure q. But for p to be extractable, the output type of q has to be exportable (in this case, it is).

7 Model of secure interoperability

In this section, we provide a model and a proof of secure interoperability, an extension to the proof from chapter 4 with the extraction mechanism from chapter 6.

We note a verified partial program with $Prog^S_{I,\pi}$ that is linked with an arbitrary context noted with $Ctx^S_{I,\pi}$. The linking results into a whole program noted with $Whole^S_{I,\pi}$. The π in the notation means that the program respects the monitorable property π. For *Prog* and *Ctx*, the π represents that its respects π if linked with any arbitrary counterpart that respects π. The I in the notation represents the interface which the parts share and we'll define its type later. The S in the notation stands for source which in our case is the language F*, and to represent the target language (OCaml) we'll use the letter T and we have $Prog^T_{I,\pi}$, Ctx^T_I and $Whole^T_I$.

Shared interface. In the model, the partial programs share between them an interface. The interface contains weak and rich types for the input and output of the context, a weak type for the output of the whole program, and a post-condition enforced dynamically after the context's execution. F* language has strong types (e.g. dependent types), which OCaml does not have, therefore our interface has to include for each strong type in the source, a weak type in the target and a function that converts one into the other. The notation: $\uparrow^a_b$ with type $b \to a$ to be read as "import weak type b to strong type a", and arrow $\downarrow^a_b$ with type $a \to b$ to be read as "export strong type a to weak type b".

$$
\begin{aligned}
&\text{type interface} = \{ \\
&\quad \text{ctx\_arg}^S : \text{Type};\ \text{ctx\_arg}^T : \text{Type};\ \downarrow^{\text{ctx\_arg}^S}_{\text{ctx\_arg}^T} : \text{ctx\_arg}^S \to \text{ctx\_arg}^T; \\
&\quad \text{ctx\_ret}^S : \text{Type};\ \text{ctx\_ret}^T : \text{Type};\ \downarrow^{\text{ctx\_ret}^S}_{\text{ctx\_ret}^T} : \text{ctx\_ret}^S \to \text{ctx\_ret}^T; \\
&\quad \uparrow^{\text{ctx\_ret}^S}_{\text{ctx\_ret}^T} : \text{ctx\_ret}^T \to \text{ctx\_ret}^S; \\
&\quad \text{ctx\_post} : \text{ctx\_arg}^S \to \text{trace} \to \text{maybe ctx\_ret}^S \to \text{trace} \to \text{bool}; \\
&\quad \text{whole\_ret} : \text{Type}; \\
&\}
\end{aligned}
\tag{1}
$$

Therefore, we define the following types for the source parts of the program:

$$\text{Ctx}^S_{I,\pi} = (x : I.\text{ctx\_arg}^S) \to \text{IIO}\ (\text{maybe}\ I.\text{ctx\_ret}^S)\ \pi\ (I.\text{ctx\_post}\ x)$$

$$\text{Prog}^S_{I,\pi} = \text{Ctx}^S_{I,\pi} \to \text{IIO}\ I.\text{whole\_ret}\ \pi$$

and the following types for the target:

$$\text{Whole}_I^T = \text{unit} \to \text{MIIO}\ (\text{maybe}\ I.\text{ctx\_ret}^T)$$

$$\text{Ctx}_I^T = I.\text{ctx\_arg}^T \to \text{MIO}\ I.\text{ctx\_ret}^T$$

$$\text{ICtx}_{I,\pi}^T = I.\text{ctx\_arg}^T \to \text{IIO}\ (\text{maybe}\ I.\text{ctx\_ret}^T)\ \pi$$

$$\text{Prog}_{I,\pi}^T = \text{ICtx}_{I,\pi}^T \to \text{MIIO}\ (\text{maybe}\ I.\text{ctx\_ret}^T)$$

$$\bowtie_{I,\pi}^T : \text{Ctx}_I^T \to \text{Prog}_{I,\pi}^T \to \text{Whole}_I^T$$

$$\bowtie_{I,\pi}^T = \lambda C.\ \lambda P.\ \lambda().\ \text{P (instrument\_MIO C)} \quad \textit{(presented in section 7.2)}$$

The context should be in the MIO effect, because it has no specification and it should not know about the instrumentation, but the compiled partial program expects an already instrumented context (because arrows are not importable). Therefore, we use an intermediate type $\text{ICtx}_{I,\pi}^T$, which allows us to define a partial program, and we assume the linker does the instrumentation of the context by using instrument_MIO.

$$\downarrow_{\text{Prog}_I^T}^{\text{Prog}_{I,\pi}^S} : I : \text{interface} \to \text{Prog}_{I,\pi}^S \to \text{Prog}_I^T$$

$$\downarrow_{\text{Prog}_I^T}^{\text{Prog}_{I,\pi}^S} = \text{extract\_HO\_iio} \quad \textit{(presented in section 7.1)}$$

7.1 Extraction of a higher-order IIO arrow

We define a extract function for higher-order IIO arrows. The extract_HO_iio wraps the higher-order arrow f into a new function with trivial specs in the MIIO effect. The new function expects a g with weak types and no post-condtion, but in the IIO effect. Thereore, the first thing extract_HO_iio does is to strengthen g' before passing it to f using strengthen_iio, and then wrap the call into a runtime check that guarantees that the default pre-condition holds.

```
let extract_HO_iio
  (a b c:Types) {| d1:exportable a |} {| d2:importable b |} {| d3:exportable c |}
  (pi:monitorable_prop)
  (post: a → trace → (m:maybe b) → trace → (r:Type0{Inr? m ⟹ r})) {| d3:checkable4 post |}
  (f: g:(x:a → IIO (maybe b) pi (fun _ → True) (post x)) → IIO c pi (fun _ → True) (fun _ _ _ → True)))
  (g': d1.etype → IIO (maybe d2.itype) pi (fun _ → True) (fun _ _ _ → True)) :
  MIIO (maybe c.etype) =
  if enforced_globally pi (get_trace ()) then
```

```
    (Inl (export (f (strenghten_iio pi g' post))))
  else (Inr Contract_failure)

let strengthen_iio
  (a b:Type) {| d1:exportable a |} {| d2:importable b |}
  (pi:monitorable_prop)
  (f: d1.etype → IIO d2.itype (fun _ → True) (fun _ _ _ → True))
  (post: a → trace → maybe b → trace → Type0)) {| d3:checkable4 post |}
  (x:a) : IIO (maybe b) (fun _ → True) (post x) =
  let h = get_trace () in
  match import (f (export x) with
  | Some x' →
    let lt = extract_local_trace h in
    if d3.check4 x h (Inl x') lt then (Inl x')
    else (Inr Contract_failure)
  | None → (Inr Contract_failure)
```

7.2 Instrumentation

F* supports the kind of powerful introspection we need to instrument a plugin therefore we give an example of a linker: instrument_MIO. The introspection is done by using reification which reveals the monad tree of the computation. Then, the tree is passed to an interpreter that executes the tree in the IIO effect, which adds before each node a runtime check that enforces π if needed. In the end, the instrumentation determines the local trace produced by the computation and enforces the post-condition, but it is not guaranteed. The instrumentation guarantees the post-condition holds or a Contract_failure is returned. The instrumentation does not have to check dynamically if the pre-condition is respected.

```
let rec interpreter (#a:Type) (tree:iio a) (pi:monitorable_prop) :
  IIO (maybe a) pi (fun _ → True) (fun _ _ _ → True) =
  match tree with
  | Return r → (Inl r)
  | Call GetTrace argz fnc → interpreter (fnc (Inl (get_trace ()))) pi
  | Call cmd argz fnc →
    if pi (get_trace ()) (| cmd, argz |) then
      let rez = run_cmd cmd pi argz in
      interpreter (fnc (Inl rez)) pi
    else (Inr Contract_failure)
```

```
let instrument_MIO
  (a b:Type) (f:a → MIO b)
  (pi: monitorable_prop) (pre: a→ trace→ Type0)
  (post: a → trace → maybe b → trace → Type0) {| d3:checkable4 post |}
  (x:a) : IIO (maybe b) pi (pre x) (post x) =
  let h = get_trace () in
  let tree : iio b = (* MIO.*)reify (f x) h (fun _ _ → True) in
  let r = interpreter tree pi in
  let lt = get_local_trace h in
  if d.check4 x h r lt then (Inl r)
  else (Inr Contract_failure)
```

7.3 Proof of secure interoperability

We show the following property about our model:

$$\forall \pi\ I\ (P : \mathrm{Prog}^S_{I,\pi})\ (C : \mathrm{Ctx}^T_I).\ \mathrm{Beh}(C \bowtie^T_I (P \downarrow)) \subseteq \pi$$

To prove our property is enough to look at the following term: $C \bowtie^T_I P \downarrow^{\mathrm{Prog}^S_{I,\pi}}_{\mathrm{Prog}^T_I}$. We unfold the linking and we obtain: $P \downarrow^{\mathrm{Prog}^S_{I,\pi}}_{\mathrm{Prog}^T_I}$ (instrument_MIO C). Next, we unfold the compilation for the partial program (extract_HO_iio) and we obtain the following piece of code:

```
(fun C → if enforced_globally pi (get_trace ()) && true then
                (Inl (export (P (strengthen_iio C)))
            else (Inr Contract_failure)) (instrument_MIO C))
```

We can normalize the previous piece of code and obtain:

```
if enforced_globally pi (get_trace ()) then
    (Inl (export (P (strengthen_iio (instrument_MIO C)))))
else (Inr Contract_failure)
```

Since, the enforced_globally function used in the if is Tot, it does not produce events, therefore, from the point of the behavior, it is a silent step we can skip.

```
export (P (strengthen_iio (instrument_MIO C)))
```

Furthermore, our partial program P returns a weak type, therefore export here is the identity function (also Tot), which we can also skip.

```
(P (strengthen_iio (instrument_MIO C))
```

We know from the type of P that it respects π. □

8 Future work

There are several directions to which this work can be extended. The first two ideas presented are extensions and the other two are ideas are about where to use next some of the novel contributions presented.

More effects. Our current IO Dijkstra monad supports only the input-output and exceptions effect. Because we use the free monad as the computational monad, it means we can extend the signature to also state and probabilities. The work that remains is to choose how to verify easily such a complex Dijkstra monad that contains 4 effects, especially that state is not usually verified using traces, but using frames. This is the first challenge we would like to take.

Non-termination. To also have non-termination it may be more complicated. There is some work on how to use Interaction Trees to verify impure programs, but their data structure is defined using co-induction. Unfortunately, F* does not support co-induction, therefore we are looking for ways to encode interaction trees without it, but it seems to be a difficult task.

Automatas. A next step would be to try to replace the trace with automatas. This can offer better efficiency, especially for memory consumption, compared to traces. We believe it should not be hard to replace our traces with automatas since there is a lot of related work on this topic, but it may be difficult to define a proper Dijkstra monad for it.

Gradual verification. We believe that since IIO Dijkstra monad supports mixing dynamic and static verification, extending it to support gradual verification would be possible. To reach this goal, our prototype has to be greatly extended to support a smooth continuum between static and dynamic verification. This smooth continuum will be enabled by the concept of imprecise formula, which is a formula that contains a wildcard. Such an imprecise formula is accepted by the verifier if there exists some interpretation of the wildcard that makes the formula valid. This will allow much finer-grained verification, because each piece of code can have verified and unverified parts. This will allow programmers to gradually evolve and refine rich specifications.

Blockchain verification. This is a wild idea, but since programming languages made for blockchains do not have concurrency, non-termination and propabilities, it seems

like our IO Dijkstra monad + state, could be used to verify such programs. Just a thought.

9 Contributions and conclusions

It took us more than expected to reach this point, but we think we put a solid ground for solving secure interoperability between verified and unverified code. Now, we can extend this to multiple directions.

We define a novel Instrumented IO Dijkstra monad that allows the seamless interoperability between static and dynamic checking for IO programs in F*. We present a setup that enables the static verification of partial programs and the hybrid enforcement of trace properties. Using a shallow embedding, we model our mechanism in F* and show that we can write top-level security guarantees about the whole program. We define a system of automatic transformations from IO to Instrumented IO. We also test our ideas doing a nontrivial case study by extending in F* a web server with a ML-plugin mechanism. In the case study, we show we can prevent the web server and an arbitrary plugin to open specific files.

10 Related Work

Chen et al. [2004] presented a framework to enable Monitoring Oriented Programming (MOP) for software development and analysis that builds on the Aspect Oriented Programming (AOP). They also present an environment [Chen and Roşu, 2005] that implements their framework that enables MOP for Java. In their framework, "monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program". It seems, MOP can be used to solve the same problem as us, but our work differs from theirs in one major way. The MOP depends heavily on the powerful Java Virtual Machine with AOP enabled; AOP is well developed in Java, but even if some work to bring this paradigm to different languages exists, it does not seem to be as well developed, therefore MOP, for now, is possible only in Java. Our proposal does not depend on AOP, and in fact, it can work with any language that can be instrumented, therefore our work is more general and does not imply modifying the language.

There is some related work for static verification of IO programs, but which does

not support hybrid verification. Malecha et al. [2011] have a similar setting for static verification because they use properties that are defined over traces of events. Penninckx et al. [2015] present a sound approach to verifying IO programs using Petri Nets instead of traces, implemented in VeriFast. Their approach may be more memory efficient than ours and can handle infinite traces, which at the moment we can not, but we plan to support. Letan and Régis-Gianas [2020] show how the FreeSpec framework for Coq can be used for verifying impure computation by verifying a Mini HTTP Server. This work is very similar to how we statically verify components with IO, but they choose as an internal state to only keep the open file descriptors. A different approach presented by Xia et al. [2020] is using Interaction Trees which supports infinite traces. We stress that none of these works supports mixing static verification with runtime verification.

A related topic is gradual verification. Extending gradual typing to gradual verification is a topic of active research. Bader et al. [2018] and Wise et al. [2020] propose gradual program verification to easily combine dynamic and static verification. They only deal with the state effect, while we deal with the IO effect and with safety properties on traces. Dagand et al. [2018] propose a dependent interoperability framework which has a mechanism to easily export dependently-typed programs to simply-typed applications, but does not discuss interoperability for a type system that contains effects.

One approach that presents interoperability between trusted and untrusted code but in a different context is proposed by Sammler et al. [2020]. They discuss only robust safety related to the memory model by doing low-level sandboxing.

References

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016. ISBN 978-1-4503-3549-2. URL https://www.fstar-lang.org/papers/mumon/.

Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018. doi: 10.1007/978-3-319-75632-5_1. URL https://doi.org/10.1007/978-3-319-75632-5_1.

Leslie Lamport and Fred B. Schneider. Formal foundation for specification and verification. In Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider, editors, *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich, Germany*, volume 190 of *Lecture Notes in Computer Science*, pages 203–285. Springer, 1984. doi: 10.1007/3-540-15216-4_15. URL https://doi.org/10.1007/3-540-15216-4_15.

Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. doi: 10.1145/353323.353382. URL https://doi.org/10.1145/353323.353382.

Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring events that carry data. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 61–102. Springer, 2018. doi: 10.1007/978-3-319-75632-5_3. URL https://doi.org/10.1007/978-3-319-75632-5_3.

Guillaume Pothier and Éric Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Pro-*

ceedings, volume 6813 of *Lecture Notes in Computer Science*, pages 558–582. Springer, 2011. doi: 10.1007/978-3-642-22655-7_26. URL https://doi.org/10.1007/978-3-642-22655-7_26.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975.

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. doi: 10.1145/2491956.2491978. URL https://doi.org/10.1145/2491956.2491978.

Bart Jacobs. Dijkstra and hoare monads in monadic computation. *Theor. Comput. Sci.*, 604:30–45, 2015. doi: 10.1016/j.tcs.2015.03.020. URL https://doi.org/10.1016/j.tcs.2015.03.020.

Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009878.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019. doi: 10.1145/3341708. URL https://doi.org/10.1145/3341708.

Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Layered indexed effects: Foundations and applications of effectful dependently typed programming, 2020. URL https://www.fstar-lang.org/papers/layeredeffects/. In submission.

Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers.

J. Log. Algebraic Methods Program., 84(1):108–123, 2015. doi: 10.1016/j.jlamp.2014.02.001. URL https://doi.org/10.1016/j.jlamp.2014.02.001.

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.*, 33 (1):127–150, 2021. doi: 10.1007/s00165-020-00523-2. URL https://doi.org/10.1007/s00165-020-00523-2.

Feng Chen, Marcelo D'Amorim, and Grigore Roşu. A formal monitoring-based framework for software development and analysis. In *Formal Methods and Software Engineering*, pages 357–372. Springer Berlin Heidelberg, 2004. doi: 10.1007/978-3-540-30482-1_31. URL https://doi.org/10.1007/978-3-540-30482-1_31.

Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550. Springer Berlin Heidelberg, 2005. doi: 10.1007/978-3-540-31980-1_36. URL https://doi.org/10.1007/978-3-540-31980-1_36.

Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46(2):95–118, 2011. doi: 10.1016/j.jsc.2010.08.004. URL https://doi.org/10.1016/j.jsc.2010.08.004.

Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 158–182. Springer, 2015. doi: 10.1007/978-3-662-46669-8_7. URL https://doi.org/10.1007/978-3-662-46669-8_7.

Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 32–46.

ACM, 2020. doi: 10.1145/3372885.3373812. URL https://doi.org/10.1145/3372885.3373812.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 25–46. Springer, 2018. doi: 10.1007/978-3-319-73721-8_2. URL https://doi.org/10.1007/978-3-319-73721-8_2.

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. Gradual verification of recursive heap data structures. *Proc. ACM Program. Lang.*, 4(OOPSLA):228:1–228:28, 2020. doi: 10.1145/3428296. URL https://doi.org/10.1145/3428296.

Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. Foundations of dependent interoperability. *J. Funct. Program.*, 28:e9, 2018. doi: 10.1017/S0956796818000011. URL https://doi.org/10.1017/S0956796818000011.

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.*, 4(POPL):32:1–32:32, 2020. doi: 10.1145/3371100. URL https://doi.org/10.1145/3371100.