Hybrid Enforcement of IO Trace Properties

Cezar-Constantin Andrici (Advisors: Ştefan Ciobâcă, Cătălin Hriţcu)
cezar.andrici@info.uaic.ro
Alexandru Ioan Cuza University of Iaşi and MPI-SP
Iaşi, România

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 verify 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★ [1] 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 or ASM code.

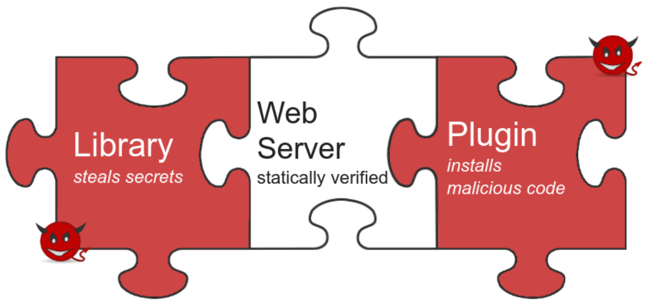

Figure 1: The statically verified web server is linked with a third-party plugin and library that are malicious and try to steal secrets, therefore breaking all the proved guarantees.

2 BACKGROUND

F★ is a functional programing 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. For example, an expression whose inferred type is Tot int is an expression that always terminates, does not have any other effects and returns an int. F★ comes with other primitive effects: Dv (for possibly divergent code), ML (for arbitrary code), etc., but also with multiple mechanisms to define new effects.

We show in Figure 2 the type of the operation read. The operation 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. Be the program webserver1:

```
let webserver1 () :
    IO unit
      (ensures (fun _ _ 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

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

Figure 2: Operation read with pre- and post-conditions enforced statically.

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 occured 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:

```
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
```

where plugin is an unknown function. 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*".

A different type of verification is runtime verification [2–4] and this can help us to verify the program webserver2. The way it 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 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 be linked with another one instrumented. We assume that tools that can instrument programs in the way we want exist and can be used.

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

Figure 3: Operation read exported. The pre-condition was converted to a runtime check.

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 with 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 4);
- It is possible to show that W satisfies the trace property π.

F* does not have a primitive effect to enable verification of IO programs, therefore we define our own. F* supports to define new monadic effects using different mechanisms. Maillard et al. [5] 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

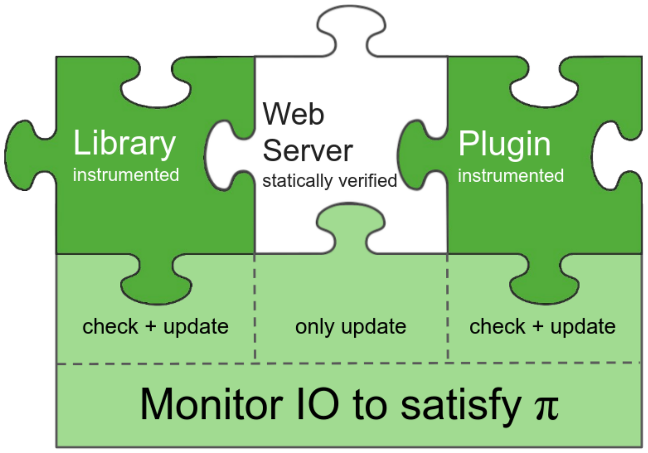

Figure 4: The statically verified web server is linked with a third-party plugin and library instrumented and monitored.

framework for defining Dijkstra monads in F* that we use to define our monadic effect, IO, for use by the trusted code.

For our IO monadic effect, we chose a computational monad that is parametric in the underlying primitive operations by using a free monad [6] that can accept any interface [7]. For simplicity, we define operations for file management and for socket communication, but these account for one possible instantiation of the monad. The approach is general and it is easy to extend. In addition, the free monad can be used to implement other effects such as exceptions, state and non-determinism by extending the interface [6] and we plan to integrate them in future work. 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 throw errors.

The IO monad allows writing specifications about computations written in our computational monad. The properties 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 specification monad we chose allows writing a pre-condition 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. In Figure 2 we present the specification of the operation read.

Up to this point, we defined a monad that allows verifying statically IO programs, but does not support dynamic checks because the trace exists only at the specification level.

To enable dynamic verification, we extend our IO monad with a new silent operation called get_trace obtaining a new monad we call IIO, for "instrumented IO". 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 very easily, creating a seamless interoperability between the two for enforcing IO trace properties. As a future work, we think the monadic 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 export. 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 Figure 3 we present the exported version of the operation read.

4 CASE STUDY: THE WEB SERVER

We report on a nontrivial ongoing case study that aims to use 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 5 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

```
type plugin_type = fd:file_descr →
  IIO 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 5: 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 6: 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.

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 6 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 7. 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 monadic effect that models a safe subset of ML. The instrumentation should bring the unverified code into the monadic 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 monad. Therefore, for now we define a synonym effect of the IO monad that does not have pre- and post-conditions. We call this monadic 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".

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 throws a contract failure. The resulting plugin after instrumentation is conceptually equivalent to the plugin' in Figure 8.

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

The transformation functions export and instrument are part of a system of automatic checks and transformations for which we use multiple type classes to implement.

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 has the liberty to observe only some operations and to store the trace where and how it finds efficient and secure [8].

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.

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

Figure 7: 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 8: The adversarial plugin from Figure 7 instrumented.

4.3 Global 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^S_π and P^T_π. The P^S_π denotes the verified web server in F⋆ and P^T_π the extracted web server. Equivalently, C^T is the plugin before instrumentation and C^S_π is the instrumented plugin. We show in Figure 9 how the partial program and the context are linked in the source and in the target. The $\bowtie^T_\pi$ 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.

$$\forall \pi\ P^S_\pi\ C^T .\ \mathrm{Beh}(C^T \bowtie^T_\pi (P^S_\pi \downarrow)) \subseteq \pi$$

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. □

5 CONTRIBUTIONS AND CONCLUSIONS

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

$$\begin{tikzcd}
\mathrm{P}^S_\pi : \mathrm{IIO}\ \mathrm{a}\ \pi \arrow[d, "\text{extract}"'] & \bowtie^S_\pi & \mathrm{C}^S_\pi : \mathrm{IIO}\ \mathrm{a}\ \pi & = & \mathrm{W}^S_\pi : \mathrm{IIO}\ \mathrm{a}\ \pi \\
\mathrm{P}^T_\pi : \mathrm{MIIO}\ \mathrm{a} & \bowtie^T_\pi & \mathrm{C}^T : \mathrm{MIO}\ \mathrm{a} \arrow[u, "\text{instrument}"'] & = & \mathrm{W}^T : \mathrm{MIIO}\ \mathrm{a}
\end{tikzcd}$$

Figure 9: 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.

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.

6 RELATED WORK

There is some related work for static verification of IO programs, but which does not support hybrid verification. Malecha et al. [9] have a similar setting for static verification because they use properties that are defined over traces of events. Penninckx et al. [10] 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 [11] 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 internal state to only keep the open file descriptors. A different approach presented by Xia et al. [12] 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. [13] and Wise et al. [14] 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. [15] 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. [16]. They discuss only robust safety related to the memory model by doing low-level sandboxing.

7 ACKNOWLEDGMENTS

This work is part of the F* project and I collaborated with my two official advisors, Guido Martínez, Exequeil Rivas and Éric Tanter. I am the main author of the implementation of the presented mechanisms in F*, the verified web server in F* and of the ongoing case study.

REFERENCES

[1] 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/.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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 Proceedings*, 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.