Journey Beyond Full Abstraction

Exploring Robust Property Preservation for Secure Compilation

Carmine Abate[1] Roberto Blanco[1] Deepak Garg[2] Cătălin Hriţcu[1] Marco Patrignani[3,4] Jérémy Thibault[1]

[1]Inria Paris [2]MPI-SWS [3]Stanford University [4]CISPA Helmholz Center for Information Security

***Abstract*—Good programming languages provide helpful abstractions for writing secure code, but the security properties of the source language are generally not preserved when compiling a program and linking it with adversarial code in a low-level target language (e.g., a library or a legacy application). Linked target code that is compromised or malicious may, for instance, read and write the compiled program's data and code, jump to arbitrary memory locations, or smash the stack, blatantly violating any source-level abstraction. By contrast, a fully abstract compilation chain protects source-level abstractions all the way down, ensuring that linked adversarial target code cannot observe more about the compiled program than what some linked source code could about the source program. However, while research in this area has so far focused on preserving observational equivalence, as needed for achieving full abstraction, there is a much larger space of security properties one can choose to preserve against linked adversarial code. And the precise class of security properties one chooses crucially impacts not only the supported security goals and the strength of the attacker model, but also the kind of protections a secure compilation chain has to introduce.**

We are the first to thoroughly explore a large space of formal secure compilation criteria based on robust property preservation, i.e., the preservation of properties satisfied against arbitrary adversarial contexts. We study robustly preserving various classes of trace properties such as safety, of hyperproperties such as noninterference, and of relational hyperproperties such as trace equivalence. This leads to many new secure compilation criteria, some of which are easier to practically achieve and prove than full abstraction, and some of which provide strictly stronger security guarantees. For each of the studied criteria we propose an equivalent "property-free" characterization that clarifies which proof techniques apply. For relational properties and hyperproperties, which relate the behaviors of multiple programs, our formal definitions of the property classes themselves are novel. We order our criteria by their relative strength and show several collapses and separation results. Finally, we adapt existing proof techniques to show that even the strongest of our secure compilation criteria, the robust preservation of all relational hyperproperties, is achievable for a simple translation from a statically typed to a dynamically typed language.

1 Introduction

Good programming languages provide helpful abstractions for writing secure code. Even in unsafe low-level languages like C, safe programs have structured control flow and obey the procedure call and return discipline. Languages such as Java, C#, ML, Haskell, or Rust provide type and memory safety for all programs and additional abstractions such as modules and interfaces. Languages for efficient cryptography such as qhasm [21], Jasmin [11], and Low⋆ [80] enforce a "constant-time" coding discipline to rule out certain side-channel attacks. Finally, verification languages such as Coq and F⋆ [80, 93] provide abstractions such as dependent types, logical pre- and postconditions, and tracking side-effects, e.g., distinguishing pure from stateful computations. Such abstractions make reasoning about security more tractable and have, for instance, enabled developing high-assurance libraries in areas such as cryptography [11, 33, 43, 102].

However, such abstractions are not enforced all the way down by mainstream compilation chains. The security properties a program satisfies in the source language are generally not preserved when compiling the program and linking it with adversarial target code. High-assurance cryptographic libraries, for instance, get linked into real applications such as web browsers [24, 43] and web servers, which include millions of lines of legacy C/C++ code. Even if the abstractions of the source language ensure that the API of a TLS library cannot leak the server's private key [33], such guarantees are completely lost when compiling the library and linking it into a C/C++ application that can get compromised via a buffer overflow, simply allowing the adversary to read the private key from memory [39]. A compromised or malicious application that links in a high-assurance library can easily read and write its data and code, jump to arbitrary memory locations, or smash the stack, blatantly violating any source-level abstraction and breaking any security guarantee obtained by source-level reasoning.

An idea that has been gaining increasing traction recently is that it should be possible to build secure compilation chains that protect source-level abstractions even against linked adversarial target code, which is generally represented by target language *contexts*. Research in this area has so far focused on achieving *full abstraction* [2, 3, 6, 7, 8, 9, 35, 47, 52, 55, 64, 71, 76, 77, 78], whose security-relevant direction ensures that even an adversarial target context cannot observe more about the compiled program than some source context could about the source program. In order to achieve full abstraction, the various parts of the secure compilation chain—including, e.g., the compiler, linker, loader, runtime, system, and hardware—have to work together to provide enough protection to the compiled program, so that whenever two programs are *observationally equivalent* in the source language (i.e., no source context can distinguish them), the two programs obtained by compiling them are observationally equivalent in the target language (i.e., no target context can distinguish them).

Observational equivalences are, however, not the only class of security properties one may want to *robustly preserve*, i.e., preserve against arbitrary adversarial contexts. One could instead be interested in robustly preserving, for instance, classes of trace properties such as safety [63] or liveness [12], or of hyperproperties [31] such as hypersafety, including variants of noninterference [13, 48, 68, 85, 86, 101], which cover data confidentiality and integrity. However, full abstraction is generally not strong enough on its own to imply the robust preservation of any of these properties (as we show in §5, and as was also argued by others [74]). At the same time, the kind of protections one has to put in place for achieving full abstraction seem like overkill if all one wants is to robustly preserve safety or hypersafety. Indeed, it is significantly harder to hide the differences between two programs that are observationally equivalent but otherwise arbitrary, than to protect the internal invariants and the secret data of a single program. Thus, a secure compilation chain for robust safety or hypersafety can likely be more efficient than one for observational equivalence. Moreover, hiding the differences between two observationally equivalent programs is hopeless in the presence of any side-channels, while robustly preserving safety is not a problem and even robustly preserving noninterference seems possible in specific scenarios [19]. Finally, even when efficiency is not a concern (e.g., when security is enforced by *static* restrictions on target contexts [1, 7, 8, 9, 71]), proving full abstraction is notoriously challenging even for simple languages, and conjectures have survived for decades before being settled [37].

Convinced that there is no "one-size-fits-all" criterion for secure interoperability with linked target code, we explore, for the first time, a large space of secure compilation criteria based on robust property preservation. Some of the criteria we introduce are strictly stronger than full abstraction and, moreover, immediately imply the robust preservation of well-studied property classes such as safety and hypersafety. Other criteria we introduce seem easier to practically achieve and prove than full abstraction. In general, the richer the class of security properties one tries to robustly preserve, the harder efficient enforcement becomes, so the best one can hope for is to strike a pragmatic balance between security and efficiency that matches each application domain.

For informing such difficult design decisions, we explore robustly preserving classes of trace properties (§2), of hyperproperties (§3), and of relational hyperproperties (§4). All these property notions are phrased in terms of execution traces, which for us are (finite or infinite) sequences of events such as inputs from and outputs to an external environment [60, 65]. Trace properties such as safety [63] restrict what happens along individual program traces, while hyperproperties [31] such as noninterference generalize this to predicates over multiple traces of a program. In this work we generalize this further to a new class we call *relational hyperproperties*, which relate the traces of *different* programs. An example of relational hyperproperty is trace equivalence, which requires that two programs produce the same set of traces. We work out many interesting subclasses that are also novel, such as *relational trace properties*, which relate *individual* traces of multiple programs. For instance, "On every input, program A's output is less than program B's" is a relational trace property.

We order the secure compilation criteria we introduce by their relative strength as illustrated by the partial order in Figure 1. In this Hasse diagram edges represent logical implication from higher criteria to lower ones, so the higher a criterion is, the harder it is to achieve and prove. Intuitively, the criteria based on the robust preservation of trace properties (in the yellow area) only require sandboxing the context (i.e., linked adversarial code) and protecting the internal invariants of the program from it, i.e., *only data integrity*. The criteria based on hyperproperties (in the red area) require additionally hiding the data of the program from the context, i.e., *data confidentiality*. Finally, the criteria based on relational hyperproperties (in the blue area) require additionally hiding the code of the program from the context, i.e., *code confidentiality*.

While most implications in the diagram follow directly from the inclusion between the property classes [31], *strict* inclusion between property classes does not imply strict implication between criteria. Robustly preserving two distinct property classes can in fact lead to equivalent criteria, as happens in general for hyperliveness and hyperproperties (§3.5) and, in the presence of source-level reflection or internal nondeterminism, for many criteria involving hyperproperties and relational hyperproperties (§4.5). To show the absence of more collapses, we also prove various separation results, for instance that *Robust Safety Property Preservation* (RSP) is *strictly* weaker than *Robust Trace Property Preservation* (RTP). For this, we design (counterexample) compilation chains that satisfy the weaker criterion but not the stronger one.

For each introduced secure compilation criterion we also discovered an *equivalent "property-free" characterization* that is generally better tailored for proofs and that provides important insights into what kind of techniques one can use to prove the criterion. For instance, for proving RSP and RTP we can produce a different source context to explain *each* attack trace, while for proving stronger criteria such as *Robust Hyperproperty Preservation* (RHP) we have to produce a single source context that works for *any* attack trace.

We also formally study the relation between our new security criteria and full abstraction (§5) proxied by the robust preservation of trace equivalence (RTEP), which in determinate languages—i.e., languages without internal nondeterminism—was shown to coincide with observational equivalence [27, 42]. In one direction, RTEP follows unconditionally from *Robust 2-relational Hyperproperty Preservation*, which is one of our stronger criteria. However, if the source and target languages are determinate and we make some mild extra assumptions (such as input totality [46, 100]) RTEP follows even from the weaker *Robust 2-relational relaXed safety Preservation* (R2rXP). Here, the challenge was identifying these extra assumptions and showing that they are sufficient to establish RTEP. In the other direction, we adapt a counterexample proposed by Patrignani and Garg [74] to show

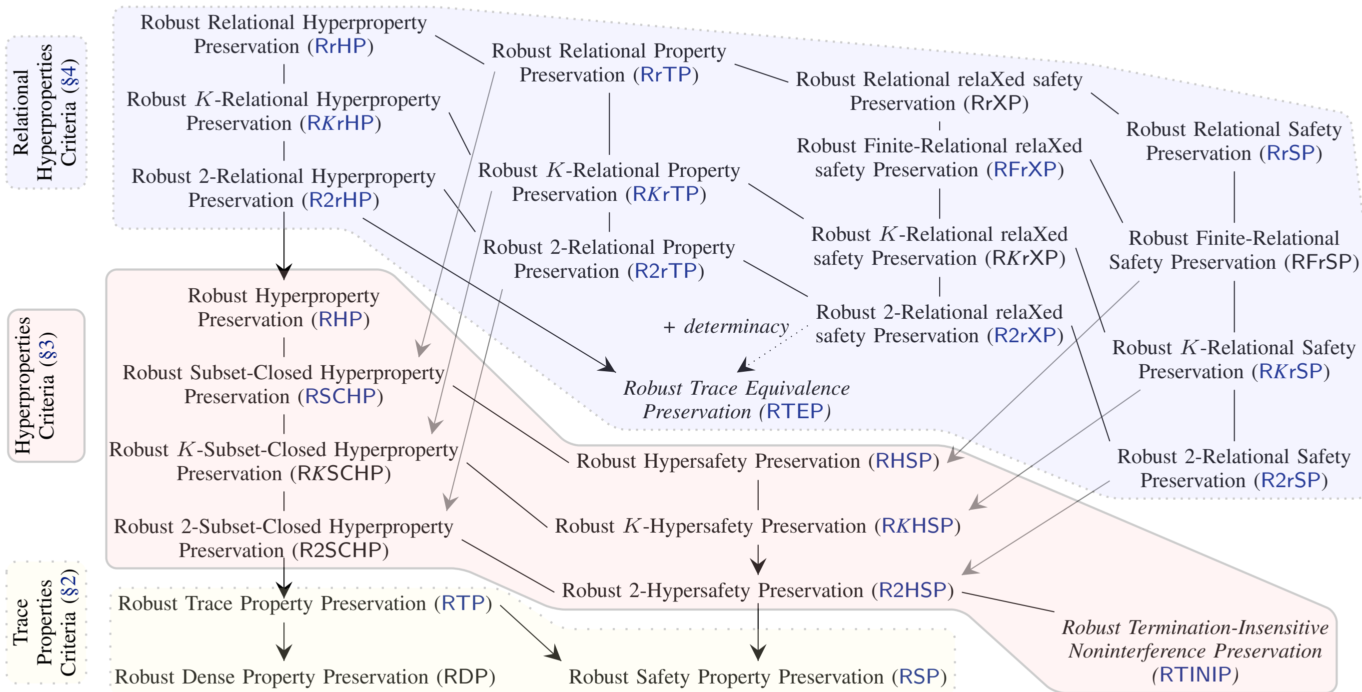


Fig. 1: Partial order with the secure compilation criteria studied in this paper. Criteria higher in the diagram imply the lower ones to which they are connected by edges. Criteria based on trace properties are grouped in a yellow area, those based on hyperproperties are in a red area, and those based on relational hyperproperties are in a blue area. Criteria with an *italics name* preserve a *single* property that belongs to the class they are connected to; the dotted edge requires an additional determinacy assumption. Finally, each edge with a thick arrow denotes a *strict* implication that we have proved as a separation result.

that RTEP (and thus full abstraction), even in conjunction with compositional compiler correctness, does *not* imply even the weakest of our criteria, RSP, RDP, and RTINIP.

Finally, we show that two proof techniques originally developed for full abstraction can be readily adapted to prove our new secure compilation criteria (§6). First, we use a "universal embedding" [71] to prove that the strongest of our secure compilation criteria, Robust Relational Hyperproperty Preservation (RrHP), is achievable for a simple translation from a statically typed to a dynamically typed first-order language with first-order functions and I/O. Second, we use the same simple translation to illustrate that for proving *Robust Finite-relational relaXed safety Preservation* (RFrXP) we can employ a "trace-based back-translation" [53, 76], a slightly less powerful but more generic technique that we extend to back-translate a finite set of finite execution prefixes into a source context. This second technique is applicable to all criteria implied by RFrXP, which includes robust preservation of safety, of hypersafety, and in a determinate setting also of trace (and thus observational) equivalence.

In summary, our paper makes five **contributions**:

C1. We phrase the formal security guarantees obtained by protecting compiled programs from adversarial contexts in terms of robustly preserving classes of properties. We are the first to explore a large space of security criteria based on this idea, including criteria that provide strictly stronger security guarantees than full abstraction, and also criteria that are easier to practically achieve and prove, which is important for building more realistic secure compilation chains.

C2. We carefully study each new secure compilation criterion and the non-trivial relations between them. For each criterion we propose a property-free characterization that clarifies which proof techniques apply. For relating the criteria, we order them by their relative strength, show several interesting collapses, and prove several challenging separation results.

C3. We introduce *relational* properties and hyperproperties, which are new property classes of independent interest, even outside of secure compilation.

C4. We formally study the relation between our security criteria and full abstraction. In one direction, we show that determinacy is enough for robustly preserving classes of relational properties and hyperproperties to imply preservation of observational equivalence. In the other direction, we show that, even when assuming compiler correctness, full abstraction does not imply even our weakest criteria.

C5. We show that two existing proof techniques originally developed for full abstraction can be readily adapted to our new criteria, which is important since good proof techniques are difficult to find in this space [36, 71, 78].

The paper closes with discussions of related (§7) and future work (§8). The appendix contains omitted technical details. Many of the theorems formally or informally mentioned in the paper were also mechanized in the Coq proof assistant and are marked with ; this development has around 4400 lines of code and is available at `https://github.com/secure-compilation/exploring-robust-property-preservation`

2 Robustly Preserving Trace Properties

In this section we look at robustly preserving classes of *trace properties*, and first study the robust preservation of *all* trace properties and its relation to correct compilation (§2.1). We then look at robustly preserving *safety properties* (§2.2), which are the trace properties that can be falsified by a finite trace prefix (e.g., a program never performs a certain dangerous system call). These criteria are grouped in the Trace Properties yellow area in Figure 1. We also carefully studied the robust preservation of *liveness properties*, but it turns out that the very definition of liveness is highly dependent on the specifics of the program execution traces, which makes that part more technical. For saving space and avoiding a technical detour, we relegate to the appendix (§B) the details of our CompCert-inspired trace model, as well as the part about liveness.

2.1 Robust Trace Property Preservation (RTP)

Like all secure compilation criteria we study in this paper, the RTP criterion below is a *generic* property of an arbitrary *compilation chain*, which includes a source and a target language, each with a notion of partial programs (P) and contexts (C) that can be linked together to produce whole programs ($C[P]$), and each with a trace-producing semantics for whole programs ($C[P] \rightsquigarrow t$). The sets of partial programs and of contexts of the source and target languages are unconstrained parameters of our secure compilation criteria; our criteria make no assumptions about their structure, or whether the program or the context gets control initially once linked and executed (e.g., the context could be an application that embeds a library program or the context could be a library that is embedded into an application program).[1] The traces produced by the source and target semantics[2] are arbitrary for RTP, but for RSP we have to consider traces with a specific structure (finite or infinite sequences of events drawn from an arbitrary set). Intuitively, traces capture the interaction between a whole program and its external environment, including for instance user input, output to a terminal, network communication, system calls, etc. [60, 65]. As opposed to a context, which is just a piece of a program, the environment's behavior is not (and often *cannot* be) modeled by the programming language, beyond the (often nondeterministic) interaction events that we store in the trace. Finally, a compilation chain includes a compiler: the compilation of a partial source program P is a partial target program we write $\mathsf{P}\downarrow$.[3]

The responsibility of enforcing secure compilation does not have to rest just with the compiler, but may be freely shared by various parts of the compilation chain. In particular, to help enforce security, the target-level *linker* could disallow linking with a suspicious context (e.g., one that is not well-typed [1, 7, 8, 9, 71]) or could always allow linking but introduce protection barriers between the program and the context (e.g., by instrumenting the program [35, 71] or the context [5, 95, 96] to introduce dynamic checks). Similarly, the semantics of the target language can include various protection mechanisms (e.g., processes with different virtual address spaces [25, 50, 59, 81, 82], protected enclaves [76], capabilities [30, 40, 90, 98], tags [5, 14]). Finally, the compiler might have to refrain from aggressive optimizations that would break security [19, 38, 89]. Our secure compilation criteria are agnostic to the concrete enforcement mechanism used by the compilation chain to protect the compiled program from the adversarial target context.

Trace properties are defined simply as sets of allowed traces [63]. A whole program $C[P]$ *satisfies* a trace property π when the set of traces produced by $C[P]$ is included in the set π or, formally, $\{t \mid C[P] \rightsquigarrow t\} \subseteq \pi$. More interestingly, we say that a partial program P *robustly satisfies* [49, 61, 94] a trace property π when P linked with *any* (adversarial) context C satisfies π. Armed with this, *Robust Trace Property Preservation* (RTP) is defined as the preservation of robust satisfaction of all trace properties. So if a partial source program P robustly satisfies a trace property $\pi \in 2^{\mathit{Trace}}$ (wrt. all source contexts) then its compilation $\mathsf{P}\downarrow$ must also robustly satisfy π (wrt. all target contexts). If we unfold all intermediate definitions, a compilation chain satisfies RTP iff:

$$\begin{aligned}\mathsf{RTP}: \quad & \forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ & \qquad (\forall \mathbf{C_T}\ t.\ \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)\end{aligned}$$

This definition directly captures which properties (specifically, all trace properties) of the source are robustly preserved by the compilation chain. However, in order to prove that a compilation chain satisfies RTP we propose an equivalent () "property-free" characterization, which we call RTC (for "RTP Characterization"):

$$\mathsf{RTC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \ \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

RTC requires that, given a compiled program $\mathsf{P}\downarrow$ and a target context $\mathbf{C_T}$ which together produce an attack trace t, we can generate a source context $\mathsf{C_S}$ that causes trace t to be produced by P. When proving that a compilation chain satisfies RTC we can pick a different context $\mathsf{C_S}$ for each t and, in fact, try to construct $\mathsf{C_S}$ from trace t or from the execution $\mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t$.

We present similar property-free characterizations for each of our criteria (Figure 1). However, for criteria stronger than RTP, a single context $\mathsf{C_S}$ will have to work for more than

[1] One limitation of our formal setup, is that for simplicity we assume that any partial program can be linked with any context, irrespective of their interfaces (e.g., types or specs). One can extend our criteria to take interfaces into account, as we illustrate in §G for the example in §6.

[2] In this paper we assume for simplicity that traces are exactly the same in both the source and target language, as is also the case in the CompCert verified C compiler [65]. We hope to lift this restriction in the future (§8).

[3] For easier reading, we use a blue, sans-serif font for source elements, an **orange, bold** font for **target** elements and a *black, italic* font generically for elements of either language.

one trace. In general, the shape of the property-free characterization explains what information can be used to produce the source context $\mathsf{C_S}$ when proving a compilation chain secure.

Relation to compiler correctness RTC is similar to "backward simulation" (TC), a standard compiler *correctness* criterion [65]. Let W denote a whole program.

$$\mathsf{TC}: \quad \forall \mathsf{W}.\ \forall t.\ \mathsf{W}{\downarrow} \rightsquigarrow t \Rightarrow \mathsf{W} \rightsquigarrow t$$

Maybe slightly less known is that this property-free characterization of correct compilation also has an equivalent property-full characterization as the preservation of all trace properties:

$$\mathsf{TP}: \quad \forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{W}.$$
$$(\forall t.\ \mathsf{W} \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow (\forall t.\ \mathsf{W}{\downarrow} \rightsquigarrow t \Rightarrow t \in \pi)$$

The major difference compared to RTP is that TP only preserves the trace properties of whole programs and does not consider adversaries. In contrast, RTP allows linking a compiled partial program with arbitrary target contexts and protects the program so that all *robustly satisfied* trace properties are preserved. In general, RTP and TP are incomparable. However, RTP strictly implies TP when whole programs (W) are a subset of partial programs (P) and, additionally, the semantics of *whole* programs is independent of any linked context (i.e., $\forall W\ t\ C.\ W \rightsquigarrow t \iff C[W] \rightsquigarrow t$, which happens, intuitively, when the whole program starts execution and, being whole, never calls into the context).

More compositional criteria for compiler correctness have also been proposed [56, 70, 79, 92]. At a minimum such criteria allow linking with contexts that are the compilation of source contexts [56], which can be formalized as follows:

$$\mathsf{SCC}: \quad \forall \mathsf{P}.\ \forall \mathsf{C_S}.\ \forall t.\ \mathsf{C_S}{\downarrow}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

More permissive criteria allow linking with any target context that behaves like some source context [70], which in our setting can be written as:

$$\mathsf{CCC}: \ \forall \mathsf{P}\ \mathbf{C_T}\ \mathsf{C_S}\ t.\ \mathbf{C_T}{\approx}\mathsf{C_S} \wedge \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

Here $\approx$ relates equivalent partial programs in the target and the source, and could, for instance, be instantiated with a cross-language logical relation [7, 70]. RTP is incomparable to SCC and CCC. On the one hand, RTP allows linking with *arbitrary* target-level contexts, which is not allowed by SCC and CCC, and requires inserting strong protection barriers. On the other hand, in RTP all source-level reasoning has to be done with respect to an *arbitrary* source context, while with SCC and CCC one can reason about a known source context.

2.2 Robust Safety Property Preservation (RSP)

Robust safety preservation is an interesting criterion for secure compilation because it is easier to achieve and prove than most criteria of Figure 1, while still being quite expressive [49, 94].

Recall that a trace property is a safety property if, within any (possibly infinite) trace that violates the property, there exists a finite "bad prefix" that violates it. We write $m \leq t$ for the prefix relation between a finite trace prefix m and a trace t(and give a precise definition in §B). Using this we define safety properties in the usual way [12, 63, 88]:

$$\mathit{Safety} \triangleq \{\pi \in 2^{\mathit{Trace}} \mid \forall t \notin \pi.\ \exists m \leq t.\ \forall t' \geq m.\ t' \notin \pi\}$$

The definition of RSP simply restricts the preservation of robust satisfaction from all trace properties in RTP to only safety properties; otherwise the definition is exactly the same:

$$\mathsf{RSP}: \quad \forall \pi \in \mathit{Safety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow$$
$$(\forall \mathbf{C_T}\ t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)$$

One might wonder how safety properties can be *robustly* satisfied in the source, given that execution traces can contain events emitted not just by the partial program but also by the adversarial context, which could trivially emit "bad events" and, hence, violate any safety property. A first alternative is for the semantics of the source language to simply prevent the context from producing any events, maybe other than termination, or, at least, prevent the context from producing any events the safety properties of interest consider bad. The compilation chain has then to "sandbox" the context to restrict the events it can produce [95, 96]. A second alternative is for the source semantics to record enough information in the trace so that one can determine the origin of each event—the partial program or the context. Then, safety properties in which the context's events are never bad can be robustly satisfied. With this second alternative, the obtained global guarantees are weaker, e.g., one cannot enforce that the whole program never makes a dangerous system call, but only that the partial program cannot be tricked by the context into making it.

The equivalent ( ) property-free characterization for RSP requires one to back-translate a program (P), a target context ($\mathbf{C_T}$), and a *finite* bad trace prefix ($\mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m$) into a source context ($\mathsf{C_S}$) producing the same finite trace prefix (m) in the source ($\mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow m$):

$$\mathsf{RSC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall m.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow m$$

Syntactically, the only change with respect to RTC is the switch from whole traces t to finite trace prefixes m. As for RTC, we can pick a different context $\mathsf{C_S}$ for each execution $\mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow m$. (In our formalization we define $W \rightsquigarrow m$ generically as $\exists t {\geq} m.\ W \rightsquigarrow t$.) The fact that for RSC these are *finite* execution prefixes can significantly simplify the back-translation into source contexts (as we show in §6.4).

It is trivially true that RTP implies RSP, since the former robustly preserves all trace properties while the latter only robustly preserves safety properties. We have also proved that RTP *strictly implies* RSP.

Theorem 2.1. RTP $\Rightarrow$ RSP, but RSP $\not\Rightarrow$ RTP

Proof sketch. As explained above, RTP $\Rightarrow$ RSP is trivial. Showing strictness requires constructing a counterexample compilation chain to the reverse implication. We take any target language that can produce infinite traces. We take the source language to be a variant of the target with the same partial programs, but where we extend whole programs and contexts with a bound on the number of events they can produce before being terminated. Compilation simply erases this bound. (While this construction might seem artificial, languages with a fuel bound are gaining popularity [99].) This compilation chain satisfies RSP (equivalently, RSC) but not RTP. To show that it satisfies RSC, we simply back-translate a target context $\mathbf{C_T}$ and a finite trace prefix m to a source context $(\mathbf{C_T}, \mathit{length}(m))$ that uses the length of m as the allowed bound, so this context can still produce m in the source without being prematurely terminated. However, this compilation chain does not satisfy RTP, since in the source all executions are finite and, hence, no infinite target trace can be simulated by *any* source context. $\square$

3 Robustly Preserving Hyperproperties

So far, we have studied the robust preservation of trace properties, which are properties of *individual* traces of a program. In this section we generalize this to *hyperproperties*, which are properties of *multiple* traces of a program [31]. A well-known hyperproperty is noninterference [13, 48, 68, 85, 101], which usually requires considering two traces of a program that differ on secret inputs. Another hyperproperty is bounded mean response time over all executions. We study robust preservation of many subclasses of hyperproperties: all hyperproperties (§3.1), subset-closed hyperproperties (§3.2), hypersafety and K-hypersafety (§3.3), and hyperliveness (§3.5). These criteria are in the red area in Figure 1.

3.1 Robust Hyperproperty Preservation (RHP)

While trace properties are sets of traces, hyperproperties are sets of sets of traces [31]. We call the set of traces of a whole program W the *behavior* of W: $\texttt{Behav}\,(W) = \{t \mid W \rightsquigarrow t\}$. A hyperproperty is a set of allowed behaviors. Program W satisfies hyperproperty H if the behavior of W is a member of H, i.e., $\texttt{Behav}\,(W) \in H$, or, equivalently, $\{t \mid W \rightsquigarrow t\} \in H$. Contrast this to W satisfying trace property π, which holds if the behavior of W is a subset of the set π, i.e., $\texttt{Behav}\,(W) \subseteq \pi$, or, equivalently, $\forall t.\ W \rightsquigarrow t \Rightarrow t \in \pi$. So while a trace property determines whether each individual trace of a program should be allowed or not, a hyperproperty determines whether the set of traces of a program, its behavior, should be allowed or not. For instance, the trace property $\pi_{123} = \{t_1, t_2, t_3\}$ is satisfied by programs with behaviors such as $\{t_1\}$, $\{t_2\}$, $\{t_2, t_3\}$, and $\{t_1, t_2, t_3\}$, but a program with behavior $\{t_1, t_4\}$ does not satisfy π_{123}. A hyperproperty like $H_{1+23} = \{\{t_1\}, \{t_2, t_3\}\}$ is satisfied only by programs with behavior $\{t_1\}$ or with behavior $\{t_2, t_3\}$. A program with behavior $\{t_2\}$ does not satisfy H_{1+23}, so hyperproperties can express that if some traces (e.g., t_2) are possible then some other traces (e.g., t_3) should also be possible. A program with behavior $\{t_1, t_2, t_3\}$ also does not satisfy H_{1+23}, so hyperproperties can express that if some traces (e.g., t_2 and t_3) are possible then some other traces (e.g., t_1) should not be possible. Finally, trace properties can be easily lifted to hyperproperties: A trace property π becomes the hyperproperty $[\pi] = 2^{\pi}$, the powerset of π.

We say that a partial program P *robustly satisfies* a hyperproperty H if it satisfies H for any context C. Given this we define RHP as the preservation of robust satisfaction of arbitrary hyperproperties:

$$\mathsf{RHP}: \quad \forall H \in 2^{2^{\mathit{Trace}}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \texttt{Behav}\,(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow (\forall \mathbf{C_T}.\ \texttt{Behav}\,(\mathbf{C_T}[\mathsf{P}{\downarrow}]) \in H)$$

The equivalent (🐓) characterization of RHP is RHC :

$$\mathsf{RHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \texttt{Behav}\,(\mathbf{C_T}[\mathsf{P}{\downarrow}]) = \texttt{Behav}\,(\mathsf{C_S}[\mathsf{P}])$$

$$\mathsf{RHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \iff \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

This requires that, for every partial program P and target context $\mathbf{C_T}$, there is a (back-translated) source context $\mathsf{C_S}$ that perfectly preserves the set of traces of $\mathbf{C_T}[\mathsf{P}{\downarrow}]$ when linked to P. There are two differences from RTP: (1) the $\exists \mathsf{C_S}$ and $\forall t$ quantifiers are swapped, so the back-translated $\mathsf{C_S}$ must work for all traces t, and (2) the implication in RTC ($\Rightarrow$) became a two-way implication in RHC ($\iff$), so compilation has to perfectly preserve the set of traces. In particular the compiler cannot refine behavior (remove traces), e.g., it cannot implement nondeterministic scheduling via a deterministic scheduler.

In the following subsections we study restrictions of RHP to various subclasses of hyperproperties. To prevent duplication we define $\mathsf{RHP}(X)$ to be the robust satisfaction of a class X of hyperproperties (so RHP above is simply $\mathsf{RHP}(2^{2^{\mathit{Trace}}})$):

$$\mathsf{RHP}(X): \quad \forall H \in X.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \texttt{Behav}\,(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow (\forall \mathbf{C_T}.\ \texttt{Behav}\,(\mathbf{C_T}[\mathsf{P}{\downarrow}]) \in H)$$

3.2 Robust Subset-Closed Hyperproperty Preservation (RSCHP)

If one restricts robust preservation to only subset-closed hyperproperties then refinement of behavior is again allowed. A hyperproperty H is subset-closed, written $H \in \mathit{SC}$, if for any two behaviors $b_1 \subseteq b_2$, if $b_2 \in H$ then $b_1 \in H$. For instance, the lifting $[\pi]$ of any trace property π is subset-closed, but the hyperproperty H_{1+23} above is not. It can be made subset-closed by allowing all smaller behaviors: $H^{\mathit{SC}}_{1+23} = \{\varnothing, \{t_1\}, \{t_2\}, \{t_3\}, \{t_2, t_3\}\}$ is subset-closed.

Robust Subset-Closed Hyperproperty Preservation (RSCHP) is simply defined as $\mathsf{RHP}(\mathit{SC})$. The equivalent (🐓) property-free characterization of RSCHC simply gives up the $\Leftarrow$ direction of RHC:

$$\mathsf{RSCHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

The most interesting subclass of subset-closed hyperproperties is hypersafety, which we discuss next. The appendix (§C.2.3)also studies K-subset-closed hyperproperties [67], which can be seen as generalizing K-hypersafety below.

3.3 Robust Hypersafety Preservation (RHSP)

Hypersafety is a generalization of safety that is very important in practice, since several important notions of noninterference are hypersafety, such as termination-insensitive noninterference [13, 45, 86], observational determinism [68, 83, 101], and nonmalleable information flow [26].

According to Alpern and Schneider [12], the "bad thing" that a safety property disallows must be *finitely observable* and *irremediable*. For safety the "bad thing" is a finite trace prefix that cannot be extended to any trace satisfying the safety property. For hypersafety, Clarkson and Schneider [31] generalize the "bad thing" to a finite set of finite trace prefixes that they call an *observation*, drawn from the set $\mathit{Obs} = 2^{\mathit{FinPref}}_{\mathit{Fin}}$, which denotes the set of all finite subsets of finite prefixes. They then lift the prefix relation to sets: an observation $o \in \mathit{Obs}$ is a prefix of a behavior $b \in 2^{\mathit{Trace}}$, written $o \leq b$, if $\forall m \in o.\ \exists t \in b.\ m \leq t$. Finally, they define hypersafety analogously to safety, but the domains involved include an extra level of sets:

$$\mathit{Hypersafety} \triangleq \{H \mid \forall b \notin H.\ (\exists o \in \mathit{Obs}.\ o \leq b \wedge (\forall b' \geq o.\ b' \notin H))\}$$

Here the "bad thing" is an observation o that cannot be extended to a behavior b' satisfying the hypersafety property H. We use this to define *Robust Hypersafety Preservation* (RHSP) as RHP(*Hypersafety*) and propose the following equivalent (🐓) characterization for it:

$$\mathsf{RHSC}: \ \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall o \in \mathit{Obs}.$$
$$o \leq \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) \Rightarrow \exists \mathsf{C_S}.\ o \leq \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}])$$

This says that to prove RHSP one needs to be able to back-translate a partial program P, a context $\mathbf{C_T}$, and a prefix o of the behavior of $\mathbf{C_T}\,[\mathsf{P}{\downarrow}]$, to a source context $\mathsf{C_S}$ so that the behavior of $\mathsf{C_S}\,[\mathsf{P}]$ extends o. It is possible to use the finite set of finite executions corresponding to observation o to drive this back-translation (as we do in §6.4).

For hypersafety the involved observations are finite sets but their cardinality is otherwise unrestricted. In practice though, most hypersafety properties can be falsified by very small sets: counterexamples to termination-insensitive noninterference [13, 45, 86] and observational determinism [68, 83, 101] are observations containing 2 finite prefixes, while counterexamples to nonmalleable information flow [26] are observations containing 4 finite prefixes. To account for this, Clarkson and Schneider [31] introduce K-hypersafety as a restriction of hypersafety to observations of a fixed cardinality K. Given $\mathit{Obs}_K = 2^{\mathit{FinPref}}_{\mathit{Fin}(K)}$, the set of observations with cardinality K, all definitions and results above can be ported to K-hypersafety by simply replacing Obs with Obs_K. Specifically, we denote by RKHSP the criterion RHP(K-*Hypersafety*).

The set of lifted safety properties, $\{[\pi] \mid \pi \in \mathit{Safety}\}$, is precisely the same as 1-hypersafety, since the counterexample for them is a single finite prefix. For a more interesting example, termination-insensitive noninterference (*TINI*) [13, 45, 86] can be defined as follows in our setting:

$$\begin{aligned}\mathit{TINI} \triangleq \{b \mid \forall t_1\ t_2 \in b.\ (&t_1\ \mathit{terminating} \wedge t_2\ \mathit{terminating} \\ &\wedge \mathit{pub\text{-}inputs}(t_1) = \mathit{pub\text{-}inputs}(t_2)) \\ &\Rightarrow \mathit{pub\text{-}events}(t_1) = \mathit{pub\text{-}events}(t_2)\}\end{aligned}$$

This requires that trace events are either inputs or outputs, each of them associated with a security level: public or secret. *TINI* ensures that for any two terminating traces of the program behavior for which the two sequences of public inputs are the same, the two sequences of public events—inputs and outputs—are also the same. *TINI* is 2-hypersafety, since $b \notin \mathit{TINI}$ implies that there exist finite traces t_1 and t_2 that agree on the public inputs but not on all public events, so we can simply take $o = \{t_1, t_2\}$. Since the traces in o are already terminated, any extension b' of o can only add extra traces, i.e., $\{t_1, t_2\} \subseteq b'$, so $b' \notin \mathit{TINI}$ as needed to conclude that *TINI* is in 2-hypersafety. In Figure 1, we write *Robust Termination-Insensitive Noninterference Preservation* (RTINIP) for RHP($\{\mathit{TINI}\}$).

3.4 Separation Between Properties and Hyperproperties

Enforcing RHSP is strictly more demanding than enforcing RSP. Because even R2HSP (robust 2-hypersafety preservation) implies RTINIP, a compilation chain satisfying R2HSP has to make sure that a target-level context cannot infer more information about the internal data of $\mathsf{P}{\downarrow}$ than a source context could infer about the data of P. By contrast, a RSP compilation chain can allow arbitrary *reads* of $\mathsf{P}{\downarrow}$'s internal data, even if P's data is private at the source level. Intuitively, for proving RSC, the source context produced by back-translation can guess any secret $\mathsf{P}{\downarrow}$ receives in the *single* considered execution, but for R2HSP the single source context needs to work for *two* different executions, potentially with two different secrets, so guessing is no longer an option. We use this idea to prove a more general separation result RTP $\not\Rightarrow$ RTINIP, by exhibiting a toy compilation chain in which private variables are readable in the target language, but not in source.

Theorem 3.1. RTP $\not\Rightarrow$ RTINIP

This implies a strict separation between all criteria based on hyperproperties (the red area in Figure 1, having RTINIP as the bottom) and all the ones based on trace properties (the yellow area in Figure 1 having RTP as the top).

Using a more complex counterexample involving a system of K linear equations, we have also shown that, for any K, robust preservation of K-hypersafety, does not imply robust preservation of $(K{+}1)$-hypersafety.

Theorem 3.2. $\forall K.$ RKHSP $\not\Rightarrow$ R(K+*1*)HSP

3.5 Where Is Robust Hyperliveness Preservation?

Robust Hyperliveness Preservation (RHLP) does not appear in Figure 1, because it is provably equivalent to RHP (or, equivalently, RHC). We define RHLP as RHP(*Hyperliveness*) for the following standard definition of *Hyperliveness* [31]:

$$\mathit{Hyperliveness} \triangleq \{H \mid \forall o \in \mathit{Obs}.\ \exists b \geq o.\ b \in H\}$$

The proof that RHLP implies RHC (🐓) involves showing that $\{b \mid b \neq \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}])\}$, the hyperproperty allowing all behaviors other than $\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}])$, is hyperliveness. Another way to obtain this result is from the fact that, as in previous models [12], each hyperproperty can be decomposed as the intersection of two hyperliveness properties. This collapse of *preserving* hyperliveness and *preserving* all hyperproperties happens irrespective of the adversarial contexts.

4 Robustly Preserving Relational Hyperproperties

Trace properties and hyperproperties are predicates on the behavior of a single program. However, we may be interested in showing that compilation robustly preserves *relations* between the behaviors of two or more programs. For example, suppose we optimize a partial source program P_1 to P_2 such that P_2 runs faster than P_1 in any source context. We may want compilation to preserve this "runs faster than" *relation* between the two program behaviors against arbitrary target contexts. Similarly, in any source context, the behaviors of P_1 and P_2 may be equal and we may want the compiler to preserve such trace equivalence [17, 28, 32] in arbitrary target contexts. This last criterion, which we call *Robust Trace Equivalence Preservation* (RTEP) in Figure 1, is interesting because in various determinate settings [27, 42] it coincides with preserving observational equivalence, the security-relevant part of full abstraction (see §5).

In this section, we study the robust preservation of such *relational hyperproperties* and several interesting subclasses, still relating the behaviors of multiple programs. Unlike hyperproperties and trace properties, relational hyperproperties have not been defined as a general concept in the literature, so even their definitions are new. We describe relational hyperproperties and their robust preservation in §4.1, then look at subclasses induced by what we call *relational properties* (§4.2) and *relational safety properties* (§4.3). The appendix (§C.3)presents a few other subclasses. The corresponding secure compilation criteria are grouped in the blue area in Figure 1. In §4.4 we show that, in general, none of these relational criteria are implied by any non-relational criterion (from §2 and §3), while in §4.5 we show two specific situations in which most relational criteria collapse to non-relational ones.

4.1 Relational Hyperproperty Preservation (RrHP)

We define a *relational hyperproperty* as a predicate (relation) on a sequence of behaviors of some length. A sequence of programs of the same length is then said to have the relational hyperproperty if their behaviors collectively satisfy the predicate. Depending on the arity of the predicate, we get different subclasses of relational hyperproperties. For arity 1, the resulting subclass describes relations on the behavior of individual programs, which coincides with hyperproperties (§3). For arity 2, the resulting subclass consists of relations on the behaviors of two programs. Both examples described at the beginning of this section lie in this subclass. This generalizes to any finite arity K (predicates on behaviors of K programs), and to the infinite arity.

Next, we define the robust preservation of these subclasses. For arity 2, *robust 2-relational hyperproperty preservation*, R2rHP, is defined as follows:

$$\begin{aligned}&\mathsf{R2rHP}:\ \forall R \in 2^{(\mathsf{Behavs}^2)}.\ \forall \mathsf{P}_1\ \mathsf{P}_2.\\ &\quad (\forall \mathsf{C_S}.\ (\mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_1]),\ \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_2])) \in R) \Rightarrow\\ &\quad (\forall \mathbf{C_T}.\ (\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]),\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_2{\downarrow}])) \in R)\end{aligned}$$

R2rHP says that for any binary relation R on behaviors of programs, if the behaviors of P_1 and P_2 satisfy R in every source context, then so do the behaviors of $\mathsf{P}_1{\downarrow}$ and $\mathsf{P}_2{\downarrow}$ in every target context. In other words, a compiler satisfies R2rHP iff it preserves any relation between pairs of program behaviors that hold in all contexts. In particular, such a compilation chain preserves trace equivalence in all contexts (i.e., RTEP), which we obtain by instantiating R with equality in the above definition. If execution time is recorded on program traces, then such a compilation chain also preserves relations like "the average execution time of P_1 across all inputs is no more than the average execution time of P_2 across all inputs" and "P_1 runs faster than P_2 on all inputs" (i.e., P_1 is an improvement of P_2). This last property can also be described as a relational predicate on pairs of traces (rather than behaviors); we return to this point in §4.2.

R2rHP has an equivalent () property-free variant that does not mention relations R:

$$\begin{aligned}\mathsf{R2rHC}:\ \forall \mathsf{P}_1\,\mathsf{P}_2\,\mathbf{C_T}.\exists \mathsf{C_S}.\ &\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_1])\\ \wedge\ &\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_2{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_2])\end{aligned}$$

R2rHC is a generalization of RHC from §3.1, but now the same source context $\mathsf{C_S}$ has to simulate the behaviors of *two* target programs, $\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]$ and $\mathbf{C_T}\,[\mathsf{P}_2{\downarrow}]$.

R2rHP generalizes to any finite arity K in the obvious way, yielding RKrHP. Finally, this also generalizes to the infinite arity. We call this *Robust Relational Hyperproperty Preservation* (RrHP):

$$\begin{aligned}&\mathsf{RrHP}:\ \forall R \in 2^{(\mathsf{Behavs}^{\omega})}.\ \forall \mathsf{P}_1, .., \mathsf{P}_K, \ldots\\ &\quad (\forall \mathsf{C_S}.\ (\mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_1]), .., \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_K]), ..) \in R) \Rightarrow\\ &\quad (\forall \mathbf{C_T}.\ (\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]), .., \mathtt{Behav}\,(\mathbf{C_S}\,[\mathsf{P}_K{\downarrow}]), ..) \in R)\end{aligned}$$

RrHP is the strongest criterion we study and, hence, it is the highest point in Figure 1. This includes robustly preserving predicates on all programs of the language, although we have not yet found practical uses for this. More interestingly, RrHP has a very natural equivalent property-free characterization, RrHC, requiring for every target context $\mathbf{C_T}$, a source context $\mathsf{C_S}$ that can simulate the behavior of $\mathbf{C_T}$ for *any* program:

$$\mathsf{RrHC}:\quad \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall \mathsf{P}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}])$$

It is instructive to compare the property-free characterizations of the preservation of robust trace properties (RTC), hyperproperties (RHC), and relational hyperproperties (RrHC). In RTC, the source context $\mathsf{C_S}$ may depend on the target context $\mathbf{C_T}$, the source program P and a given trace t. In RHC, $\mathsf{C_S}$ may depend only on $\mathbf{C_T}$ and P. In RrHC, $\mathsf{C_S}$ may depend only on $\mathbf{C_T}$. This directly reflects the increasing expressive power of trace properties, hyperproperties, and relational hyperproperties, as predicates on traces, behaviors (set of traces), and sequences of behaviors, respectively.

4.2 Relational Trace Property Preservation (RrTP)

Relational (trace) properties are the subclass of relational hyperproperties that are fully characterized by relations on *individual* traces of multiple programs. For example, the relation "$\mathsf{P_1}$ runs faster than $\mathsf{P_2}$ on every input" is a 2-ary relational property characterized by pairs of traces, one from $\mathsf{P_1}$ and the other from $\mathsf{P_2}$, which either differ in the input or where the execution time in $\mathsf{P_1}$'s trace is less than that in $\mathsf{P_2}$'s trace. Formally, relational properties of arity K are a subclass of relational hyperproperties of the same arity. A K-ary relational hyperproperty is a relational (trace) property if there is a K-ary relation R on *traces* such that $P_1, .., P_K$ are related by the relational hyperproperty iff $(t_1, \ldots, t_k) \in R$ for any $t_1 \in \mathtt{Behav}\,(P_1), \ldots, t_k \in \mathtt{Behav}\,(P_K)$. Next, we define the robust preservation of relational properties of different arities. For arity 1, this coincides with RTP from §2.1. For arity 2, we define *Robust 2-relational Property Preservation*:

$$\mathsf{R2rTP} : \forall R \in 2^{(\mathit{Trace}^2)}.\ \forall \mathsf{P_1}\ \mathsf{P_2}.$$
$$\big(\forall \mathsf{C_S}\, t_1\, t_2.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow t_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow t_2) \Rightarrow (t_1,t_2) \in R\big) \Rightarrow$$
$$\big(\forall \mathbf{C_T}\, t_1\, t_2.\, (\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow t_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow t_2) \Rightarrow (t_1,t_2) \in R\big)$$

R2rTP is weaker than its relational hyperproperty counterpart, R2rHP (§4.1): Unlike R2rHP, R2rTP does not imply the robust preservation of relations like "the average execution time of $\mathsf{P_1}$ across all inputs is no more than the average execution time of $\mathsf{P_2}$ across all inputs" (a relation between average execution times of $\mathsf{P_1}$ and $\mathsf{P_2}$ cannot be characterized by any relation between individual traces of $\mathsf{P_1}$ and $\mathsf{P_2}$).

R2rTP also has an equivalent (🐓) characterization:

$$\mathsf{R2rTC} :\ \forall \mathsf{P_1}\ \mathsf{P_2}\ \mathbf{C_T}\ t_1\ t_2.$$
$$(\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow t_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow t_2) \Rightarrow$$
$$\exists \mathsf{C_S}.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow t_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow t_2)$$

Establishing R2rTC requires constructing a source context $\mathsf{C_S}$ that can simultaneously simulate a given trace of $\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}]$ and a given trace of $\mathbf{C_T}\,[\mathsf{P_2}{\downarrow}]$. R2rTP generalizes from arity 2 to any finite arity K (yielding RKrTP) and the infinite one (yielding RrTP) in the obvious way.

4.3 Robust Relational Safety Preservation (RrSP)

Relational safety properties are a natural generalization of safety and hypersafety properties to multiple programs, and an important subclass of relational trace properties. Several interesting relational trace properties are actually relational safety properties. For instance, if we restrict the earlier relational trace property "$\mathsf{P_1}$ runs faster than $\mathsf{P_2}$ on all inputs" to terminating programs it becomes a relational safety property, characterized by pairs of bad terminating prefixes, where both prefixes have the same input, and the left prefix shows termination no earlier than the right prefix.

Formally, a relation $R \in 2^{(\mathit{Trace}^K)}$ is *K-relational safety* if for every K "bad" traces $(t_1, \ldots, t_K) \notin R$, there exist K "bad" finite prefixes $m_1, \ldots, m_k$ such that $\forall i.\ m_i \leq t_i$, and any K traces $(t'_1, \ldots, t'_K)$ pointwise extending $m_1, \ldots, m_k$ are also not in the relation, i.e., $\forall i.\ m_i \leq t'_i$ implies $(t'_1, \ldots, t'_K) \notin R$. Then, *Robust 2-relational Safety Preservation* (R2rSP) is simply defined by restricting R2rTP to only 2-relational safety properties. The equivalent (🐓) property-free characterization for R2rSP is the following:

$$\mathsf{R2rSC} :\ \forall \mathsf{P_1}\ \mathsf{P_2}\ \mathbf{C_T}\ m_1\ m_2.$$
$$(\mathbf{C_T}\,[\mathsf{P_1}{\downarrow}] \rightsquigarrow m_1 \wedge \mathbf{C_T}\,[\mathsf{P_2}{\downarrow}] \rightsquigarrow m_2) \Rightarrow$$
$$\exists \mathsf{C_S}.\, (\mathsf{C_S}\,[\mathsf{P_1}] \rightsquigarrow m_1 \wedge \mathsf{C_S}\,[\mathsf{P_2}] \rightsquigarrow m_2)$$

The only difference from the stronger R2rTC (§4.2) is between considering full traces and only finite prefixes. Again, R2rSP generalizes to any finite arity K (yielding RKrSP) and the infinite one (yielding RrSP) in the obvious way.

4.4 Separation Between Relational and Non-Relational

Relational (hyper)properties (§4.1, §4.2) and hyperproperties (§3) are different but both have a "relational" nature: relational (hyper)properties are relations on the behaviors or traces of multiple programs, while hyperproperties are relations on multiple traces of the same program. So one may wonder whether there is any case in which the *robust preservation* of a class of relational (hyper)properties is equivalent to that of a class of hyperproperties. Could a compiler that robustly preserves all hyperproperties (RHP, §3.1) also robustly preserves at least some class of 2-relational (hyper)properties? In §4.5 we show special cases in which this is indeed the case, while here we now show that *in general* RHP does not imply the robust preservation of any subclass of relational properties that we have described so far (except, of course, relational properties of arity 1, that are just hyperproperties). Since RHP is the strongest non-relational robust preservation criterion that we study, this also means that no non-relational robust preservation criterion implies any relational robust preservation criterion in Figure 1. So, all edges from relational to non-relational criteria in Figure 1 are strict implications.

To prove this, we build a compilation chain satisfying RHP, but not R2rSP, the weakest relational criterion in Figure 1.

Theorem 4.1. $\mathsf{RHP} \not\Rightarrow \mathsf{R2rSP}$

Proof sketch. Consider a source language that lacks code introspection, and a target language that is exactly the same, but additionally has a primitive with which the context can read the code of the compiled program as data [91]. Consider the trivial compiler that is syntactically the identity. It is clear that this compiler satisfies RHP since the added operation of code introspection offers no advantage to the context when we consider properties of a single program, as is the case in RHP. More precisely, in establishing RHC, the property-free characterization of RHP, given a target context $\mathbf{C_T}$ and a program P, we can construct a simulating source context $\mathsf{C_S}$ by modifying $\mathbf{C_T}$ to hard-code P wherever $\mathbf{C_T}$ performs code introspection. This works as $\mathsf{C_S}$ can depend on P in RHC.

Now consider two programs that differ only in some dead code, that both read a value from the context and write it back verbatim to the output. These two programs satisfy the relational safety property "the outputs of the two programs are equal" in any *source* context. However, there is a trivial *target* context that causes the compiled programs to break this relational property. This context reads the code of the program it is linked to, and provides 1 as input if it happens to be the first of our two programs and 2 otherwise. Consequently, in this target context, the two programs produce outputs 1 and 2 and do not have this relational safety property in all contexts. Hence, this compiler does not satisfy R2rSP. Technically, the trick of hard-coding the program in $\mathsf{C_S}$ no longer works since there are two different programs here. □

This proof provides a fundamental insight: To robustly preserve any subclass of relational (hyper)properties, compilation must ensure that target contexts cannot learn anything about the *syntactic program* they interact with beyond what source contexts can also learn. When the target language is low-level, hiding code attributes can be difficult: it may require padding the code segment of the compiled program to a fixed size, and cleaning or hiding any code-layout-dependent data like code pointers from memory and registers when passing control to the context. These complex protections are not necessary for any non-relational preservation criteria (even RHP), but are already known to be necessary for fully abstract compilation to low-level code [55, 57, 76, 77]. They can also be trivially circumvented if the context has access to any side-channels, e.g., it can measure time via a different thread. In fact, in such settings trying to hide the source code can be seen as a hopeless attempt at "security through obscurity", which is widely rejected by cryptographers since the early days [58].

4.5 Composing Contexts Using Full Reflection or Internal Nondeterminism in the Source Language

The proof of the previous separation theorem strongly relies on the absence of code introspection in the source language. However, if source contexts can obtain *complete* intrinsic information about the programs they are linked with, then RHP implies R2rHP. Such "full reflection" facilities are available in languages such as Lisp [91] and Smalltalk. [4]For proving this collapse we inspect the alternative characterizations, RHC and R2rHC. The main difference between these two criteria, as explained in §4.1, is that the source context $\mathsf{C_S}$ obtained by R2rHC depends on two, possibly distinct programs $\mathsf{P_1}$ and $\mathsf{P_2}$ and a target context $\mathbf{C_T}$, while every possible source context obtained by RHC depends on one single program. Hence, by applying RHC once for $\mathsf{P_1}$ and once for $\mathsf{P_2}$, with the same context $\mathbf{C_T}$, we obtain two source contexts $\mathsf{C_{S_1}}$ and $\mathsf{C_{S_2}}$ that are a priori unrelated. Without further hypotheses, one cannot show R2rHC. However, with full reflection we can define a source context $\mathsf{C'_S}$ that behaves exactly like $\mathsf{C_{S_1}}$ when linked with $\mathsf{P_1}$, and like $\mathsf{C_{S_2}}$ otherwise. We can use this construction to show not only that RHP implies R2rHP, but also that robust preservation of each class of finite-relational properties collapses to the corresponding hyperproperty-based criterion:

Theorem 4.2. If the source language has full reflection then RHP⇒RKrHP, RSCHP⇒RKrTP, and R2HSP⇒RFrSP.

One may wonder whether some other condition exists that makes robust preservation of relational hyperproperty classes collapse even to the corresponding *trace-property-based* criteria (§2). This is indeed the case when the source language has an *internal nondeterministic choice operator* $\oplus$, such that the behavior of $\mathsf{P_1} \oplus \mathsf{P_2}$ is at least the union of the behaviors of $\mathsf{P_1}$ and $\mathsf{P_2}$. Such an operator is standard in process calculi [87]. To illustrate this we show that RTC implies R2rTC. Note that R2rTC produces a source context $\mathsf{C_S}$ that depends on a target context, two source programs $\mathsf{P_1}$ and $\mathsf{P_2}$ and two, possibly incomparable, traces t_1 and t_2. RTC produces a context depending only on a single trace of a single source program. We can apply RTC twice: once for t_1 and $\mathsf{P_1}$ obtaining $\mathsf{C_{S_1}}$ and once for t_2 and $\mathsf{P_2}$ obtaining $\mathsf{C_{S_2}}$. To prove R2rTC we need to build a source context that over-approximates the behaviors of both $\mathsf{C_{S_1}}$ and $\mathsf{C_{S_2}}$. This context can be $\mathsf{C_{S_1}} \oplus \mathsf{C_{S_2}}$. Hence, in this setting RTC (RTP) implies R2rTC (R2rTP). This result generalizes to any finite arity.

Theorem 4.3. If the source language has an internal nondeterministic choice operator on contexts then RTP ⇒ RKrTP, RSCHP ⇒ RFrSCHP, and RSP ⇒ RFrSP.

Notice that since contexts are finite objects, the techniques above only produce collapses in cases where finitely many source contexts need to be composed. Criteria relying on infinite-arity relations such as RrHP and RrTP are thus not impacted by these collapses. The appendix (§F)has more details and collapsed variants of Figure 1.

5 Where Is Full Abstraction?

Full abstraction—the preservation and reflection of observational equivalence—is a well-studied criterion for secure compilation (§7). The security-relevant direction of full abstraction is *Observational Equivalence Preservation* (OEP) [35, 78]:

$$\mathsf{OEP} : \forall \mathsf{P_1}\ \mathsf{P_2}.\ \mathsf{P_1} \approx \mathsf{P_2} \Rightarrow \mathsf{P_1}{\downarrow} \approx \mathsf{P_2}{\downarrow}$$

One natural question is how OEP relates to our criteria of robust preservation.

Here we answer this question for languages without internal nondeterminism. In such *determinate* [42, 65] settings observational equivalence coincides with trace equivalence in all contexts [27, 42] and, hence, OEP coincides with robust trace-equivalence preservation (RTEP). As explained in §4.1, it is obvious that RTEP is an instance of R2rHP, obtained by choosing equality as the relation R. However, for determinate languages with *input totality* [46, 100] (if the program accepts one input value, it has to also accept any other input value) we have proved that even the weaker R2rTP implies RTEP ( ). This proof also requires that if a whole program can produce every finite prefix of an infinite trace then it can also produce the complete trace, but we have showed that this holds for the infinite traces produced in a standard way by *any* determinate small-step semantics. Under these assumptions, we have in fact proved that RTEP follows from the even weaker *Robust 2-relational relaXed safety Preservation* (R2rXP). The class *2-relational relaXed safety* is a variant of *2-relational Safety* from §4.3; with this relaxed variant "bad" prefixes x_1 and x_2

[4]Full reflection was shown to cause observational equivalence to degenerate to syntactical identity [44, 97].

are allowed to end with silent divergence (denoted as *XPref*):

$$R \in \textit{2-relational relaXed safety} \iff \forall(t_1,t_2)\notin R.\ \exists x_1\, x_2 \in \mathit{XPref}.\ \forall t_1' \geq x_1\, t_2' \geq x_2.\ (t_1',t_2')\notin R$$

Theorem 5.1. Assuming a determinate source language and a determinate and input total small-step semantics for the target language, R2rXP ⇒ RTEP.

In the other direction, we adapt an existing counterexample [74] to show that RTEP (and, hence, for determinate languages also OEP) does *not* imply RSP or any of the criteria above it in Figure 1. Fundamentally, RTEP only requires preserving *equivalence* of behavior. Consequently, an RTEP compiler can insert code that violates any security property, as long as it doesn't alter these equivalences [74]. Worse, even when the RTEP compiler is also required to be correct (i.e., TP, SCC, and CCC from §2.1), the compiled program only needs to properly deal with interactions with target contexts that behave like source ones, and can behave insecurely when interacting with target contexts that have no source equivalent.

Theorem 5.2. There exists a compiler between two deterministic languages that satisfies RTEP, TP, SCC, and CCC, but that does not satisfy RSP.

Proof. Consider a source language where a partial program receives a natural number or boolean from the context, and produces a number output, which is the only event. We compile to a restricted language that only has numbers by mapping booleans true and false to $\mathbf{0}$ and $\mathbf{1}$ respectively. The compiler's only interesting aspect is that it translates a source function $\mathsf{P} = \mathsf{f(x{:}Bool)} \mapsto \mathsf{e}$ that inputs booleans to $\mathsf{P}{\downarrow} = \mathbf{f(x{:}Nat)} {\mapsto} \mathbf{if\ x{<}2\ then\ } \mathsf{e}{\downarrow}\ \mathbf{else\ if\ x{<}3\ then\ f(x)\ else\ 42}$. The compiled function checks if its input is a valid boolean ($\mathbf{0}$ or $\mathbf{1}$). If so, it executes $\mathsf{e}{\downarrow}$. Otherwise, it behaves insecurely, silently diverging on input 2 and outputting 42 on inputs 3 or more. This compiler does not satisfy RSP since the source program $\mathsf{f(x{:}Bool)}{\mapsto}\mathsf{0}$ robustly satisfies the safety property "never output 42", but the program's compilation does not.

On the other hand, it is easy to see that this compiler is correct since a compiled program behaves exactly like its source counterpart on correct inputs. It is also easily seen to satisfy RTEP, since the additional behaviors added by the compiler (silently diverging on input 2 and outputting 42 on inputs 3 or more) are independent of the source code (they only depend on the type), so these cannot be used by any target context to distinguish two compiled programs.

In the appendix(§E.5), we use the same counterexample compilation chain to also show that RTEP does not imply the robust preservation of (our variant of) liveness properties. We also use a simple extension of this compilation chain to show that RTEP does not imply RTINIP either. The idea is similar: we add a secret external input to the languages and when receiving an out of bounds argument the compiled code simply leaks the secret input, which breaks RTINIP, but not RTEP. □

6 Proof Techniques for RrHP and RFrXP

This section demonstrates that the criteria we introduce can be proved by adapting existing back-translation techniques. We introduce a statically typed source language and a similar dynamically typed target one (§6.1), as well as a simple translation between the two (§6.2). We then describe the essence of two very different secure compilation proofs for this compilation chain, both based on techniques originally developed for showing fully abstract compilation. The first proof shows (a typed variant of) RrHP (§6.3), the strongest criterion from Figure 1, using a *context-based* back-translation, which provides a "universal embedding" of a target context into a source context [71]. The second proof shows a slightly weaker criterion, *Robust Finite-relational relaXed safety Preservation* (RFrXP; §6.4), but which is still very useful, as it implies robust preservation of arbitrary safety and hypersafety properties as well as RTEP. This second proof relies on a *trace-based* back-translation [53, 76], extended to produce a context from *a finite set* of finite execution prefixes. These finiteness restrictions are offset by a more generic proof technique that only depends on the context-program interaction (e.g., calls and returns), while ignoring all other language details. For space reasons, we leave the details of the proofs for §G.

6.1 Source and Target Languages

The two languages we consider are simple first-order languages with named procedures and boolean and natural values. The source language L^{τ} is typed while the target language $\mathbf{L^u}$ is untyped. A program in either language is a collection of function definitions, each function body is a pure expression that can perform comparison and natural operations (⊕), conditional branching, recursive calls, and use let-in bindings. Expressions can also read naturals from the environment and write naturals to the environment, both of which generate trace events. $\mathbf{L^u}$ has all the features of L^{τ} and adds a primitive $\mathbf{e\ has\ }\tau$ to dynamically check whether an expression $\mathbf{e}$ has type τ. A context $\mathbb{C}$ can call functions and perform general computation on the returned values, but it cannot directly generate *read* and *write e* events, as those are security-sensitive. Since contexts are single expressions, we disallow callbacks from the program to the context: thus calls go from context to program, and returns from program to context.

$$\begin{aligned}
&\textit{Programs}\ \mathsf{P} ::= \overline{\mathsf{I}}; \overline{\mathsf{F}} \qquad \textit{Contexts}\ \mathbb{C} ::= \mathsf{e}\\
&\textit{Types}\ \tau ::= \mathsf{Bool} \mid \mathsf{Nat} \qquad \textit{Interfaces}\ \mathsf{I} ::= \mathsf{f} : \tau \to \tau\\
&\textit{Functions}\ \mathsf{F} ::= \mathsf{f}(\mathsf{x} : \tau) : \tau \mapsto \mathsf{ret\ e}\\
&\textit{Expressions}\ \mathsf{e} ::= \mathsf{x} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \in \mathbb{N} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \geq \mathsf{e}\\
&\qquad \mid \mathsf{let\ x} : \tau = \mathsf{e\ in\ e} \mid \mathsf{if\ e\ then\ e\ else\ e}\\
&\qquad \mid \mathsf{call\ f\ e} \mid \mathsf{read} \mid \mathsf{write\ e} \mid \mathsf{fail}
\end{aligned}$$

$$\begin{aligned}
&\textit{Programs}\ \mathbf{P} ::= \overline{\mathbf{I}}; \overline{\mathbf{F}} \qquad \textit{Contexts}\ \mathbb{C} ::= \mathbf{e}\\
&\textit{Types}\ \tau ::= \mathbf{Bool} \mid \mathbf{Nat} \qquad \textit{Interfaces}\ \mathbf{I} ::= \mathbf{f}\\
&\textit{Functions}\ \mathbf{F} ::= \mathbf{f(x)} \mapsto \mathbf{ret\ e}\\
&\textit{Expressions}\ \mathbf{e} ::= \mathbf{x} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{n} \in \mathbb{N} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{e} \geq \mathbf{e}
\end{aligned}$$

$$\mid \mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e}$$
$$\mid \mathbf{call}\ \mathbf{f}\ \mathbf{e} \mid \mathbf{read} \mid \mathbf{write}\ \mathbf{e} \mid \mathbf{fail} \mid \mathbf{e}\ \mathbf{has}\ \tau$$
$$\textit{Labels}\ \lambda ::= \epsilon \mid \alpha$$
$$\textit{Actions}\ \alpha ::= \texttt{read}\ n \mid \texttt{write}\ n \mid \Downarrow \mid \Uparrow \mid \bot$$

Each language has a standard small-step operational semantics (omitted for brevity), as well as a big-step trace semantics ($\Omega \rightsquigarrow \overline{\alpha}$, as in previous sections). The initial state of a program P plugged into a context $\mathbb{C}$ is denoted as $P \triangleright \mathbb{C}$ and the behavior of such a program is the set of traces that can be produced by the semantics:

$$\texttt{Behav}\,(\mathbb{C}[P]) = \{\overline{\alpha} \mid P \triangleright \mathbb{C} \rightsquigarrow \overline{\alpha}\}$$

6.2 Compiler

The compiler $\cdot\downarrow$ takes programs of L^{τ} and generates programs of $\mathbf{L}^{\mathbf{u}}$, by replacing static type annotations with dynamic type checks of function arguments upon function invocation:

$$\mathsf{I}_1, \cdots, \mathsf{I}_m; \mathsf{F}_1, \cdots, \mathsf{F}_n\downarrow = \mathsf{I}_1\downarrow, \cdots, \mathsf{I}_m\downarrow; \mathsf{F}_1\downarrow, \cdots, \mathsf{F}_n\downarrow$$
$$\mathsf{f} : \tau \to \tau'\big\downarrow = \mathbf{f}$$
$$\left.\begin{array}{l}\mathsf{f}(\mathsf{x}:\tau):\tau' \mapsto \\ \mathsf{ret\ e}\end{array}\right\downarrow = \left(\begin{array}{l}\mathbf{f(x)} \mapsto \mathbf{ret}\ \ \mathbf{if\ x\ has}\ \tau\downarrow \\ \qquad\qquad \mathbf{then\ e}\downarrow\ \mathbf{else\ fail}\end{array}\right)$$

$$\mathsf{Nat}\downarrow = \mathbf{Nat} \qquad \mathsf{Bool}\downarrow = \mathbf{Bool}$$
$$\mathsf{true}\downarrow = \mathbf{true} \qquad \mathsf{false}\downarrow = \mathbf{false}$$
$$\mathsf{n}\downarrow = \mathbf{n} \qquad \mathsf{x}\downarrow = \mathbf{x}$$
$$\mathsf{e} \oplus \mathsf{e}'\big\downarrow = \mathsf{e}\downarrow \oplus \mathsf{e}'\big\downarrow \qquad \mathsf{e} \geq \mathsf{e}'\big\downarrow = \mathsf{e}\downarrow \geq \mathsf{e}'\big\downarrow$$
$$\mathsf{read}\downarrow = \mathbf{read} \qquad \mathsf{write\ e}\downarrow = \mathbf{write\ e}\downarrow$$
$$\mathsf{call\ f\ e}\downarrow = \mathbf{call\ f\ e}\downarrow$$
$$\left.\begin{array}{l}\mathsf{let\ x}:\tau{=}\mathsf{e} \\ \mathsf{in\ e}'\end{array}\right\downarrow = \begin{array}{l}\mathbf{let\ x{=}e}\downarrow \\ \mathbf{in\ e}'\big\downarrow\end{array} \qquad \left.\begin{array}{l}\mathsf{if\ e\ then\ e}' \\ \mathsf{else\ e}''\end{array}\right\downarrow = \begin{array}{l}\mathbf{if\ e}\downarrow\ \mathbf{then\ e}'\big\downarrow \\ \mathbf{else\ e}''\big\downarrow\end{array}$$

6.3 Proof of RrHP by Context-Based Back-Translation

To prove that $\cdot\downarrow$ attains RrHP, we need a way to back-translate target contexts into source contexts. To this end we use a universal embedding, a technique previously proposed for proving fully abstract compilation [71]. The back-translation needs to generate a source context that respects source-level constraints; in this case, the resulting source context must be well-typed. To ensure this, we use Nat as an *universal back-translation type* in the produced source contexts. The intuition of the back-translation is that it will encode $\mathbf{true}$ as $\mathsf{0}$, $\mathbf{false}$ as $\mathsf{1}$ and an arbitrary natural number $\mathbf{n}$ as $\mathsf{n}+\mathsf{2}$. Based on this encoding, we translate values between regular source types and the back-translation type. Specifically, we define the following shorthand for the back-translation: $\mathsf{inject}_{\tau}(\mathsf{e})$ takes an expression e of type τ and returns an expression of back-translation type; $\mathsf{extract}_{\tau}(\mathsf{e})$ takes an expression e of the back-translation type and returns an expression of type τ.

$$\mathsf{inject}_{\mathsf{Nat}}(\mathsf{e}) = \mathsf{e}+\mathsf{2}$$
$$\mathsf{inject}_{\mathsf{Bool}}(\mathsf{e}) = \mathsf{if\ e\ then\ 1\ else\ 0}$$
$$\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}) = (\mathsf{let\ x{=}e\ in\ if\ x} \geq \mathsf{2\ then\ x}-\mathsf{2\ else\ fail})$$
$$\mathsf{extract}_{\mathsf{Bool}}(\mathsf{e}) = \left(\begin{array}{l}\mathsf{let\ x{=}e\ in\ if\ x} \geq \mathsf{2\ then\ fail} \\ \quad \mathsf{else\ if\ x}+\mathsf{1} \geq \mathsf{2\ then\ true\ else\ false}\end{array}\right)$$

$\mathsf{inject}_{\tau}(\mathsf{e})$ never incurs runtime errors, but $\mathsf{extract}_{\tau}(\mathsf{e})$ may. This mimics the ability of target contexts to write ill-typed code (e.g., $\mathbf{3}+\mathbf{true}$) which we must be able to back-translate and whose semantics we must preserve (see Example 6.1).

Concretely, the back-translation is defined inductively on the structure of target contexts:

$$\mathbf{true}\uparrow = \mathsf{1} \qquad \mathbf{false}\uparrow = \mathsf{0} \qquad \mathbf{n}\uparrow = \mathsf{n}+\mathsf{2} \qquad \mathbf{x}\uparrow = \mathsf{x}$$
$$\mathbf{e} \geq \mathbf{e}'\big\uparrow = \mathsf{let\ x1 : Nat{=}extract_{Nat}(\mathbf{e}\uparrow)}$$
$$\qquad \mathsf{in\ let\ x2 : Nat{=}extract_{Nat}(\mathbf{e}'\big\uparrow)}$$
$$\qquad \mathsf{in\ inject_{Bool}(x1 \geq x2)}$$
$$\mathbf{e} \oplus \mathbf{e}'\big\uparrow = \mathsf{let\ x1 : Nat{=}extract_{Nat}(\mathbf{e}\uparrow)}$$
$$\qquad \mathsf{in\ let\ x2 : Nat{=}extract_{Nat}(\mathbf{e}'\big\uparrow)}$$
$$\qquad \mathsf{in\ inject_{Nat}(x1 \oplus x2)}$$
$$\mathbf{let\ x{=}e\ in\ e}'\big\uparrow = \mathsf{let\ x : Nat{=}}\mathbf{e}\uparrow\ \mathsf{in}\ \mathbf{e}'\big\uparrow$$
$$\left.\left(\begin{array}{l}\mathbf{if\ e\ then} \\ \ \mathbf{e}'\ \mathbf{else\ e}''\end{array}\right)\right\uparrow = \mathsf{if\ extract_{Bool}(\mathbf{e}\uparrow)\ then}\ \mathbf{e}'\big\uparrow\ \mathsf{else}\ \mathbf{e}''\big\uparrow$$
$$\mathbf{e\ has\ Bool}\uparrow = \mathsf{let\ x : Nat{=}}\mathbf{e}\uparrow\ \mathsf{in\ if\ x} \geq \mathsf{2\ then\ 0\ else\ 1}$$
$$\mathbf{e\ has\ Nat}\uparrow = \mathsf{let\ x : Nat{=}}\mathbf{e}\uparrow\ \mathsf{in\ if\ x} \geq \mathsf{2\ then\ 1\ else\ 0}$$
$$\mathbf{call\ f\ e}\uparrow = \mathsf{inject}_{\tau'}(\mathsf{call\ f\ extract}_{\tau}(\mathbf{e}\uparrow))$$
$$\text{if } \mathsf{f} : \tau \to \tau' \in \bar{\mathsf{I}}$$
$$\mathbf{fail}\uparrow = \mathsf{fail}$$

Example 6.1 (Back-Translation)**.** Through the back-translation of two simple target contexts we explain why $\cdot\uparrow$ is correct and why it needs $\mathsf{inject}.$ and $\mathsf{extract}.$.

Consider the context $\mathbb{C}_1 = \mathbf{3} * \mathbf{5}$, which reduces to $\mathbf{15}$ irrespective of the program it links against. The back-translation must intuitively ensure that $\mathbb{C}_1\uparrow$ reduces to $\mathsf{17}$, which is the back-translation of $\mathbf{15}$. If we unfold the definition of $\mathbb{C}_1\uparrow$ we have the following (given that $\mathbf{3}\uparrow{=}\mathsf{5}$ and $\mathbf{5}\uparrow{=}\mathsf{7}$):

$$\mathsf{let\ x1 : Nat{=}extract_{Nat}(5)}$$
$$\quad \mathsf{in\ let\ x2 : Nat{=}extract_{Nat}(7)\ in\ inject_{Nat}(x1 * x2)}$$

By examining the code of $\mathsf{extract}_{\mathsf{Nat}}$ we see that in both cases it will just perform a subtraction by 2, turning $\mathsf{5}$ and $\mathsf{7}$ respectively into $\mathsf{3}$ and $\mathsf{5}$. So after some reduction steps we arrive at the following term: $\mathsf{inject}_{\mathsf{Nat}}(3 * 5)$. The inner multiplication then returns $\mathsf{15}$ and its injection returns $\mathsf{17}$, which is also the result of $\mathbf{15}\uparrow$.

Let us now consider a different context, $\mathbb{C}_2 = \mathbf{false} + \mathbf{3}$. We know that no matter what program links against it, it will reduce to $\mathbf{fail}$. Its statically well-typed back-translation is:

$$\mathsf{let\ x1 : Nat{=}extract_{Nat}(0)}$$
$$\quad \mathsf{in\ let\ x2 : Nat{=}extract_{Nat}(7)\ in\ inject_{Nat}(x1 * x2)}$$

By looking at its code we can see that the execution of $\mathsf{extract}_{\mathsf{Nat}}(0)$ will indeed result in fail, which is what we want and expect, as that is precisely the back-translation of $\mathbf{fail}$. ⊡

The RrHP proof for this compilation chain uses a simple logical relation that includes cases for both terms of source type (intuitively used for compiler correctness) and for terms of back-translation type [35, 71].

6.4 Proof of RFrXP by Trace-Based Back-Translation

Proving that this simple compilation chain attains RFrXC does not require back-translating a target context, as we only need to build a source context that can reproduce a finite set of finite trace prefixes, but that is not necessarily equivalent to the original target context. We describe this back-translation on an example leaving again details to §G.

Example 6.2 (Back-Translation of Traces)**.** Consider the following two programs:

$$\mathsf{P_1} = (\mathsf{f(x{:}Nat) : Nat \mapsto ret\ x, g(x{:}Nat) : Bool \mapsto ret\ true})$$
$$\mathsf{P_2} = (\mathsf{f(x{:}Nat) : Nat \mapsto ret\ read, g(x{:}Nat) : Bool \mapsto ret\ true})$$

Their compiled counterparts are almost identical, with the only addition of dynamic type checks on function arguments:

$$\mathsf{P_1}\!\downarrow = \mathbf{f(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ x\ else\ fail)},$$
$$\qquad \mathbf{g(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}$$
$$\mathsf{P_2}\!\downarrow = \mathbf{f(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ read\ else\ fail)},$$
$$\qquad \mathbf{g(x) \mapsto ret\ (if\ x\ has\ Nat\ then\ true\ else\ fail)}$$

Now, consider the following target context:

$$\mathbb{C} = \mathbf{let\ x1{=}call\ f\ 5}$$
$$\qquad \mathbf{in\ if\ x1 \geq 5\ then\ call\ g\ (x1)\ else\ call\ g\ (false)}$$

The two programs plugged into this context can generate (at least) the following traces (where $\Downarrow$ indicates termination and $\bot$ indicates failure):

$$\mathbb{C}[\mathsf{P_1}\!\downarrow] \rightsquigarrow \Downarrow \quad \mathbb{C}[\mathsf{P_2}\!\downarrow] \rightsquigarrow \texttt{read}\ 5; \Downarrow \quad \mathbb{C}[\mathsf{P_2}\!\downarrow] \rightsquigarrow \texttt{read}\ 0; \bot$$

In the execution of $\mathbb{C}[\mathsf{P_1}\!\downarrow]$, the program executes completely and terminates, producing no side effects. In the first execution of $\mathbb{C}[\mathsf{P_2}\!\downarrow]$, the program reads 5, and the *then* branch of the context's conditional is executed. In the second execution of $\mathbb{C}[\mathsf{P_2}\!\downarrow]$, the program reads 0, the *else* branch of the context's conditional is executed and the program fails in $\mathbf{g}$ after detecting a type error.

These traces alone are not enough to construct a source context since they do not record information about the control flow of program executions, specifically on which function produces which input or output. To recover this information we enrich execution prefixes with information about calls (from context to program) and returns (from program to context). The enriched rules on calls and returns now generate events to model these control flows. If a call or return occurs internally within the program, no trace event is generated since they are not relevant for back-translating the context. The revised semantics is almost identical to the original, and allows exactly the same program executions, only producing more informative traces. Hence, the original execution can be enriched in a valid way for the new semantics.

$$\textit{Labels}\ \lambda ::= \cdots \mid \beta \quad \textit{Interactions}\ \beta ::= \texttt{call}\ f\ v \mid \texttt{ret}\ v$$

The traces produced by the compiled programs plugged into the context become:

$$\mathbb{C}[\mathsf{P_1}\!\downarrow] \rightsquigarrow \texttt{call}\ f\ 5; \qquad \texttt{ret}\ 5; \texttt{call}\ g\ 5; \texttt{ret}\ \textit{true}; \Downarrow$$
$$\mathbb{C}[\mathsf{P_2}\!\downarrow] \rightsquigarrow \texttt{call}\ f\ 5; \texttt{read}\ 5; \texttt{ret}\ 5; \texttt{call}\ g\ 5; \texttt{ret}\ \textit{true}; \Downarrow$$
$$\mathbb{C}[\mathsf{P_2}\!\downarrow] \rightsquigarrow \texttt{call}\ f\ 5; \texttt{read}\ 0; \texttt{ret}\ 0; \texttt{call}\ g\ \textit{false}; \bot$$

In our languages, reads and writes can only be performed by programs, while the context only performs a sequence of calls to the program, possibly performing some computation and branching on return values. Thus, the role of the back-translated source is to perform the appropriate calls to the program, depending of the values returned. The inner workings of the programs, that is inputs, outputs, and internal calls and returns, are not a concern of the back-translation and are obtained through compiler correctness. Furthermore, the context is shared by all executions, but each execution has its own program. Hence, since I/O occurs only in the program, the only source of variation among all executions come from the program.

From this, one can conclude that the context is a deterministic expression, calling the program, and branching on the returned values. This can be seen in the way traces are organized: ignoring the I/O, the traces form a tree (Figure 2, on the left). This tree can be translated to a source context using nested conditionals as depicted below (Figure 2, on the right, dotted lines indicated what the back-translation generates for each action in the tree). When additional branches are missing (e.g., there is no third trace that analyzes the first return or no second trace that analyses the second return on the left execution), the back-translation inserts fail in the code – they are dead code branches (marked with a **).

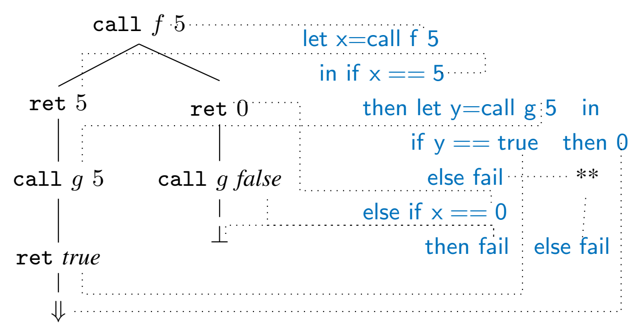


Fig. 2: Example of a back-translation of traces.

To prove RFrXP we show correctness of the back-translation, which ensures that the back-translated source context produces exactly the original non-informative traces. This is, however, not completely true of informative traces (that track calls and returns). Since calling g with a boolean is ill-typed, our back-translation shifts the failure from the program to the context, so the picture links $\texttt{call}\ g\ \textit{false}$ action to a fail. The call is never executed at the source level. ⊡

7 Related Work

Full Abstraction was originally used as a criterion for secure compilation in the seminal work of Abadi [1] and has since received a lot of attention [78]. Abadi [1] and, later, Kennedy [57] identified failures of full abstraction in the Java to JVM and C# to CIL compilers, some of which were fixed, but also others for which fixing was deemed too costly compared to the perceived practical security gain. Abadi et al. [3] proved full abstraction of secure channel implementations using cryptography, but to prevent network traffic attacks they had to introduce noise in their translation, which in practice would consume network bandwidth. Ahmed et al. [7, 8, 9, 71] proved the full abstraction of type-preserving compiler passes for simple functional languages. Abadi and Plotkin [2] and Jagadeesan et al. [52] expressed the protection provided by address space layout randomization as a probabilistic variant of full abstraction. Fournet et al. [47] devised a fully abstract compiler from a subset of ML to JavaScript. Patrignani et al. [64, 76, 77] studied fully abstract compilation to machine code, starting from single modules written in simple, idealized object-oriented and functional languages and targeting a hardware isolation mechanism similar to Intel's SGX [51].

Until recently, most formal work on secure interoperability with linked target code was focused only on fully abstract compilation. The goal of our work is to explore a diverse set of secure compilation criteria, some of them formally stronger than (the interesting direction of) full abstraction at least in various determinate settings, and thus potentially harder to achieve and prove, some of them apparently easier to achieve and prove than full abstraction, but most of them not directly comparable to full abstraction. This exploration clarifies the trade-off between security guarantees and efficient enforcement for secure compilation: On one extreme, RTP robustly preserves only trace properties, but does not require enforcing confidentiality; on the other extreme, robustly preserving relational properties gives very strong guarantees, but requires enforcing that both the private data and the code of a program remain hidden from the context, which is often much harder to achieve. The best criterion to apply depends on the application domain, but our framework can be used to address interesting design questions such as the following: *(1) What secure compilation criterion, when violated, would the developers of practical compilers be willing to fix at least in principle?* The work of Kennedy [57] indicates that fully abstract compilation is not such a good answer to this question, and we wonder whether RTP or RHP could be better answers. *(2) What secure compilation criterion would the translations of Abadi et al. [3] still satisfy if they did not introduce (inefficient) noise to prevent network traffic analysis?* Abadi et al. [3] explicitly leave this problem open in their paper, and we believe one answer could be RTP, since it does not require preserving any confidentiality.

We also hope that our work can help eliminate common misconceptions about the security guarantees provided (or not) by full abstraction. For instance, Fournet et al. [47] illustrate the difficulty of achieving security for JavaScript code using a simple example policy that (1) restricts message sending to only correct URLs and (2) prevents leaking certain secret data. Then they go on to prove full abstraction apparently in the hope of preventing contexts from violating such policies. However, part (1) of this policy is a safety property and part (2) is hypersafety, and as we showed in §4.5 fully abstract compilation does not imply the robust preservation of such properties. In contrast, proving RHSP would directly imply this, without putting any artificial restrictions on code introspection, which are unnecessarily required by full abstraction. Unfortunately, this is not the only work in the literature that uses full abstraction even when it is not the right hammer.

Development of RSP Two pieces of concurrent work have examined more carefully how to attain and prove one of the weakest of our criteria, RSP (§2.2). Patrignani and Garg [75] show RSP for compilers from simple sequential and concurrent languages to capabilities [98]. They observe that if the source language has a verification system for robust safety and compilation is limited to verified programs, then RSP can be established without directly resorting to back-translation. (This observation has also been made independently by Dave Swasey in private communication to us.) Abate et al. [5] aim at devising secure compilation chains for protecting mutually distrustful components written in an unsafe language like C. They show that by moving away from the full abstraction variant used in earlier work [55] to a variant of our RSP criterion from §2.2, they can support a more realistic model of dynamic component compromise, while at the same time obtaining a criterion that is easier to achieve and prove than full abstraction.

Hypersafety Preservation The high-level idea of specifying secure compilation as the preservation of properties and hyperproperties goes back to the work of Patrignani and Garg [74]. However, that work's technical development is limited to one criterion—the preservation of finite prefixes of program traces by compilation. Superficially, this is similar to one of our criteria, RHSP, but there are several differences even from RHSP. First, Patrignani and Garg [74] do not consider adversarial contexts explicitly. This might suffice for their setting of closed reactive programs, where traces are inherently fully abstract (so considering the adversarial context is irrelevant), but not in general. Second, they are interested in designing a criterion that accommodates specific fail-safe like mechanisms for low-level enforcement, so the preservation of hypersafety properties is not perfect, and one has to show, for every relevant property, that the criterion is meaningful. However, Patrignani and Garg [74] consider translations of trace symbols induced by compilation, an extension that would also be interesting for our criteria (§8).

Proof techniques New et al. [71] present a back-translation technique based on a universal type embedding in the source for the purpose of proving full abstraction of translations from typed to untyped languages. In §6.3 we adapted the same technique to show RrHP for a simple translation from a statically typed to a dynamically typed language with first-order functions and I/O. Devriese et al. [35] show that even when a precise universal type does not exist in the source, one can use an approximate embedding that only works for a certain number of execution steps. They illustrate such an approximate back-translation by proving full abstraction for a compiler from the simply-typed to the untyped λ-calculus.

Jeffrey and Rathke [53, 54] introduced a "trace-based" back-translation technique. They were interested in proving full abstraction for so-called trace semantics. This technique was then adapted to show full abstraction of compilation chains to low-level target languages [6, 73, 76, 77]. In §6.4, we showed how these trace-based techniques can be extended to prove all the criteria below RFrXP in Figure 1, which includes robust preservation of safety, of noninterference, and in a determinate setting also of observational equivalence.

While many other proof techniques have been previously proposed [2, 3, 8, 9, 47, 52], proofs of full abstraction remain notoriously difficult, even for simple translations, with apparently simple conjectures surviving for decades before being finally settled [37]. It will be interesting to investigate which existing full abstraction techniques can be repurposed to show the stronger criteria from Figure 1. For instance, it will be interesting to determine the strongest criterion from Figure 1 for which an approximate back-translation [35] can be used.

Source-level verification of robust satisfaction While this paper studies the *preservation* of robust properties in compilation chains, formally verifying that a partial source program robustly satisfies a specification is a challenging problem too. So far, most of the research has focused on techniques for proving observational equivalence [4, 28, 32, 53, 54] or trace equivalence [17, 27]. Robust satisfaction of trace properties has been model checked for systems modeled by nondeterministic Moore machines and properties specified by branching temporal logic [61]. Robust safety, the robust satisfaction of safety properties, was studied for the analysis of security protocols [15, 16, 49], and more recently for compositional verification [94]. Verifying the *robust* satisfaction of relational hyperproperties beyond observational equivalence and trace equivalence seems to be an open research problem. For addressing it, one can hopefully take inspiration in extensions of relational Hoare logic [20] for dealing with cryptographic adversaries represented as procedures parameterized by oracles [18].

Other Kinds of Secure Compilation In this paper we investigated the various kinds of security guarantees one can obtain from a compilation chain that protects the compiled program against linked adversarial low-level code. While this is an instance of *secure compilation* [10], this emerging area is much broader. Since there are many ways in which a compilation chain can be "more secure", there are also many different notions of secure compilation, with different security goals and attacker models. A class secure compilation chains is aimed at providing a "safer" semantics for unsafe low-level languages like C and C++, for instance ensuring memory safety [29, 41, 69]. Other secure compilation work is targeted at closing down side-channels: for instance by preserving the secret independence guarantees of the source code [19], or making sure that the code erasing secrets is not simply optimized away by the unaware compilers [22, 34, 38, 89]. Closer to our work is the work on building compartmentalizing compilation chains [5, 23, 50, 98] for unsafe languages like C and C++. In particular, as mentioned above, Abate et al. [5] have recently showed how RSP can be extended to express the security guarantees obtained by protecting mutually distrustful components against each other.

8 Conclusion and Future Work

This paper proposes a foundation for secure interoperability with linked target code by exploring many different criteria based on robust property preservation (Figure 1). Yet the road to building *practical* secure compilation chains achieving any of these criteria remains long and challenging. Even for RSP, scaling up to realistic programming languages and efficiently enforcing protection of the compiled program without restrictions on the linked context is challenging [5, 75]. For R2HSP the problem is even harder, because one also needs to protect the secrecy of the program's data, which is especially challenging in a realistic model in which the context can observe side-channels like timing. Here, an RTINIP-like property might be the best one can hope for in practice.

In this paper we assumed for simplicity that traces are exactly the same in both the source and target language, and while this assumption is currently true for other work like CompCert [65] as well, it is a restriction nonetheless. We plan to lift this restriction in the future.

Acknowledgments

We are grateful to Akram El-Korashy, Arthur Azevedo de Amorim, Ştefan Ciobâcă, Dominique Devriese, Guido Martínez, Marco Stronati, Dave Swasey, Éric Tanter, and the anonymous reviewers for their valuable feedback and in many cases also for participating in various discussions. This work was in part supported by the ERC under ERC Starting Grant SECOMP (715753), by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), and by DARPA grant SSITH/HOPE (FA8650-15-C-7558).

Appendix A
Notations

We use blue, sans-serif font for *source* elements, **red, bold** font for *target* elements and *black, italic* for elements common to both languages (to avoid repeating similar definitions twice). Thus, P is a source-level program, $\mathbf{P}$ is a target-level program and P is generic notation for either a source-level or a target-level program.

Whole Programs	W
Partial Programs	P
Contexts	C
Termination Events	ε
Events	e
Finite Trace Prefixes	$m \triangleq$
(*terminated*)	$e_1 \cdot \cdots \cdot e_n ⓔ$
(*not yet terminated*)	$e_1 \cdot \cdots \cdot e_n \circ$
relaXed Trace Prefixes	$x \triangleq$
(*terminated*)	$e_1 \cdot \cdots \cdot e_n ⓔ$
(*not yet terminated*)	$e_1 \cdot \cdots \cdot e_n \circ$
(*silent divergence*)	$e_1 \cdot \cdots \cdot e_n \circlearrowleft$
Traces	$t \triangleq$
(*program termination*)	$e_1 \cdot \cdots \cdot e_n ⓔ$
(*silent divergence*)	$e_1 \cdot \cdots \cdot e_n \circlearrowleft$
(*infinitely reactive*)	$e_1 \cdot \cdots \cdot e_n \cdot \cdots$
Prefix relation	$m \leq t$
The set of all traces	*Trace*
The set of all finite trace prefixes	*FinPref*
The set of all relaxed trace prefixes	*XPref*
Semantics of W	$W \rightsquigarrow t$
Behavior of W	$\mathtt{Behav}\,(W) = \{t \mid W \rightsquigarrow t\}$
Set with elements from X	2^X
Set of size K *with elements from* X	2^X_K
Set literal	$\widehat{x} \triangleq \{x_1, x_2, \cdots\}$
Property	$\pi \in 2^{\mathit{Trace}}$
Behavior(*the set of traces of a program*)	$b \in 2^{\mathit{Trace}}$
Hyperproperty	$H \in 2^{2^{\mathit{Trace}}}$
Cardinality	$\lVert\cdot\rVert$

In addition to trace-based, whole-program semantics, we also define a finite prefix-based semantics, $W \rightsquigarrow m$ as $\exists t \geq m.\, W \rightsquigarrow t$. The notations introduced for finite prefixes ($\leq$, $\geq$, $\rightsquigarrow$, etc.) are used not only for finite trace prefixes, but also for relaxed trace prefixes.

Appendix B
Safety and Dense Properties with Event-Based Traces

We start by presenting the CompCert-inspired model for program execution traces we use in this work (§B.1). In this model *safety properties* are defined in the standard way as the trace properties that can be falsified by a finite trace prefix (§2.2). Perhaps more surprisingly, in this trace model the role generally played by liveness is taken by what we call *dense properties*, which we define simply as the trace properties that can only be falsified by non-terminating traces (e.g., a reactive program that runs forever eventually answers every network request it receives). Next, to validate the claim that dense properties indeed play the same role that liveness plays in previously proposed trace models [12, 63, 72, 88], we prove several related properties (§B.2), including the fact that every trace property is the intersection of a safety property and a dense property (this is our variant of a standard decomposition result [12]), and the fact that our definition of dense properties is unique (§B.2). Finally, we study the robust preservation of dense properties (RDP; §B.3).

B.1 Event-Based Trace Model for Safety and Liveness

For defining safety and liveness, traces need a bit of structure, and for this we use a variant of CompCert's *realistic* trace model [65].[5] This model is different from the trace models generally used for studying safety and liveness of reactive systems [12, 31, 62, 63, 66, 88] (e.g., in a transition system or a process calculus). A first important difference is that in CompCert's model, traces are built from events, not from states. This is important for efficient compilation, since taking these events to be relatively coarse-grained gives the compiler more freedom to perform program optimizations. For instance, CompCert is inspired by the C programming language standard and defines the outcome of the program to be a trace of all I/O and volatile operations it performs, plus an indication of whether and how it terminates.

The *events* in our traces are drawn from an arbitrary nonempty set. Intuitively, traces t are finite or infinite lists of events, where a finite trace means that the program terminates (possibly with some related information recording the cause of termination, such as an exit code) or enters an unproductive infinite loop after producing all the events in the list. This kind of trace model is natural for usual programming languages where most programs do indeed terminate and is standard for formally correct compilers [60, 65]. It is different, however, from the trace model usually considered for abstract modeling of reactive systems, which considers only infinite traces [31, 62, 66, 88] and where a common trick to force all traces to be infinite is to use stuttering on the final state of an execution to represent termination [31]. In our model, however, events are observable and infinitely repeating the last event would result in a trace of a non-terminating execution, so we have to be honest about the fact that terminating executions produce finite traces. Moreover, working with traces of events also means that execution steps can be silent (i.e., add no events to the trace) and one has to distinguish termination from silent divergence (a non-terminating execution), although both of them produce a finite number of events. So in our model terminating traces are those that end in an explicit termination event and can thus no longer be extended; all other traces, whether silently divergent or infinite, are non-terminating. The proper treatment of program termination and silent divergence distinguishes the realistic trace model we use here from previous theoretical work that extends safety and liveness to finite and infinite traces [72, 84].

Using this realistic trace model directly impacts the meaning of safety, which we try to keep as standard and natural as possible, and also created the need for a new definition of *dense properties* to take the place of liveness.

Safety Properties The main component of the characterization of safety properties is a definition of *finite* trace prefixes, which capture the finite observations that can be made about an execution, for instance by a reference monitor. We take the stance that a reference monitor can observe that the program has terminated. To reflect this, in our trace model finite trace prefixes are lists of events in which it is observable whether a prefix is terminated and can no longer be extended, or whether it is not yet terminated and can still be extended with further events. Moreover, while termination and silent divergence are two different terminal trace events, no monitor can distinguish between the two in finite time, since one cannot tell whether a program that seems to be looping will *eventually* terminate. Technically, in our model finite trace prefixes m are lists with two different final constructors: ε for a prefix terminated with final event ε (which for instance distinguishes successful from erroneous termination) and $\circ$ for not yet terminated prefixes. In contrast, traces can end either with ε if the program terminates or with $\circlearrowleft$ if the program silently diverges, or they can go on infinitely. The prefix relation $m \leq t$ is defined between a finite prefix m and a trace t according to the intuition above: $\varepsilon \leq \varepsilon$, $\circ \leq t$, and $e \cdot m' \leq e \cdot t'$ whenever $m' \leq t'$ (where $\cdot$ is concatenation).

The definition of safety properties is then unsurprising (as already seen in §2.2):

$$\mathit{Safety} \triangleq \{\pi \in 2^{\mathit{Trace}} \mid \forall t \notin \pi.\ \exists m \leq t.\ \forall t' \geq m.\ t' \notin \pi\}$$

A trace property π is *Safety* if, within any trace t that violates π, there exists a finite "bad prefix" m that can only be extended to traces t' that also violate π.

[5] Our trace model is close to that of CompCert, but as opposed to CompCert, in this paper we use the word "trace" for the result of a single program execution and later "behavior" for the set of all traces of a program (§3).

Example B.1. For instance, the trace property $\pi_{\Box\neg e} = \{t \mid e \notin t\}$, stating that the bad event e never occurs in the trace, is *Safety*, since for every trace t violating $\pi_{\Box\neg e}$ there exists a finite prefix $m = m'{\cdot}e{\cdot}\circ$ (some prefix m' followed by e and then by the unfinished prefix symbol $\circ$) that is a prefix of t, and every trace extending m still contains e, so it continues to violate $\pi_{\Box\neg e}$. ⊡

Example B.2. Consider the property $\pi_{\Box\neg\varepsilon} = \{t \mid \forall\varepsilon.\textcircled{\varepsilon} \notin t\}$ that rejects all terminating traces and accepts all non-terminating traces. This is a safety property, the justification of which crucially relies on allowing $\textcircled{\varepsilon}$ in the finite trace prefixes. For any finite trace $t = e_1 \cdot \ldots \cdot e_n\textcircled{\varepsilon}$ rejected by $\pi_{\Box\neg\varepsilon}$, there exists a bad prefix $m = e_1 \cdot \ldots \cdot e_n\textcircled{\varepsilon}$ such that all extensions of m are also rejected by $\pi_{\Box\neg\varepsilon}$. This last condition is trivial since the prefix m is terminating (i.e., ends with $\textcircled{\varepsilon}$) and can thus only be extended to t itself. ⊡

Example B.3. The trace property $\pi^{term}_{\Diamond e} = \{t \mid t \textit{ terminating} \Rightarrow e \in t\}$ states that in every *terminating* trace the event e must eventually happen. This is also a safety property in our model, since for each terminating trace $t = e_1 \cdot \ldots \cdot e_n\textcircled{\varepsilon}$ violating $\pi^{term}_{\Diamond e}$ there exists a bad prefix $m = e_1 \cdot \ldots \cdot e_n\textcircled{\varepsilon}$ that can only be extended to traces that also violate $\pi^{term}_{\Diamond e}$, i.e., only to t itself. ⊡

Generally speaking, all trace properties (like $\pi_{\Box\neg\varepsilon}$ and $\pi^{term}_{\Diamond e}$) that only reject terminating traces and therefore allow all non-terminating traces are safety properties in our model. That is, if $\forall t$ *non-terminating*. $t \in \pi$, then π is a safety property. Consequently, for any property π, the derived trace property $\pi_S = \pi \cup \{t \mid t \textit{ non-terminating}\}$ is a safety property.

Dense Properties In our trace model the liveness definition of Alpern and Schneider [12] does not have its intended intuitive meaning, so instead we focus on the main properties that the Alpern and Schneider liveness definition satisfies in the infinite state-based trace model and, in particular, that each trace property can be decomposed as the intersection of a safety property and a liveness property. We discovered that in our model the following simple notion of *dense properties* satisfies the characterizing properties of liveness and is, in fact, uniquely determined by these properties:

$$\textit{Dense} \triangleq \{\pi \in 2^{\textit{Trace}} \mid \forall t \textit{ terminating}.\ t \in \pi\}$$

We say that a trace property π is *Dense* if it allows all terminating traces; or, conversely, it can only be violated by non-terminating traces. For instance, the property $\pi^{\neg term}_{\Diamond e} = \{t \mid t \textit{ non-terminating} \Rightarrow e \in t\}$, stating that the event e will eventually happen along every non-terminating trace is a dense property, since it accepts all terminating traces. The property $\pi_{\Box\neg\circlearrowleft} = \{t \mid \ \circlearrowleft \notin t\} = \{t \mid t \textit{ non-terminating} \Rightarrow t \textit{ infinite}\}$ stating that the program does not silently diverge is also dense. Again, more examples are given below:

Example B.4. The property $\pi^{\neg term}_{\Box\Diamond e} = \{t \mid t \textit{ non-terminating} \Rightarrow t \textit{ infinite} \wedge \forall m \leq t.\ \exists m'.\ m{\cdot}m'{\cdot}e{\leq}t\}$ states that event e happens infinitely often in any non-terminating trace. Because it allows all terminating traces, it is a dense property. ⊡

Example B.5. The property $\pi_{\Diamond\varepsilon} = \{t \mid t \textit{ terminating}\} = \{t \mid \exists\varepsilon.\textcircled{\varepsilon} \in t\}$ contains exactly all terminating traces and rejects all non-terminating traces. It is therefore the minimal dense property of our trace model. ⊡

Trivially, any property becomes dense in our model if we modify it to accept all terminating traces. That is, given any property π, the derived $\pi_L = \pi \cup \{t \mid t \textit{ terminating}\}$ is dense.

Example B.6. Take the safety property $\pi_{\Box\neg e} = \{t \mid e \notin t\}$, which forbids an event e from appearing in traces. The modified dense property $\pi^{\neg term}_{\Box\neg e}$ states that event e never occurs along non-terminating traces: $\pi^{\neg term}_{\Box\neg e} = \{t \mid t \textit{ non-terminating} \Rightarrow e \notin t\}$. ⊡

B.2 Theory of Dense Properties

We have proved that our definition of dense properties satisfies the main properties of Alpern and Schneider's related concept of liveness [12], including its topological characterization, and in particular the following fact.

Theorem B.7. Any trace property can be decomposed into the intersection of a safety property and of a dense property ( ): $\forall\pi.\exists\pi_S \in \textit{Safety}.\exists\pi_D \in \textit{Dense}.\pi = \pi_S \cap \pi_D$.

Proof. The proof of this decomposition theorem is in fact very simple in our model. Given any trace property π, define $\pi_S = \pi \cup \{t \mid t \textit{ non-terminating}\}$ and $\pi_D = \pi \cup \{t \mid t \textit{ terminating}\}$. As discussed above, $\pi_S \in \textit{Safety}$ and $\pi_D \in \textit{Dense}$. Finally, $\pi_S \cap \pi_L = (\pi \cup \{t \mid t \textit{ non-terminating}\}) \cap (\pi \cup \{t \mid t \textit{ terminating}\}) = \pi$. □

Example B.8. In our trace model, the property $\pi_{\Diamond e} = \{t \mid e \in t\}$ is neither safety nor dense. However, it can be decomposed as the intersection of $\pi^{term}_{\Diamond e}$ (a safety property) and $\pi^{\neg term}_{\Diamond e}$ (a dense property). ⊡

Concerning the relation between dense properties and the liveness definition of [12], the two are in fact equivalent *in our model*, but this seems to be a coincidence and only happens because Alpern and Schneider's definition completely loses its original intent in our model, as the following theorem and simple proof suggests.

Theorem B.9. $\forall \pi \in 2^{Trace}.\ \pi \in Dense \iff \forall m. \exists t. m \leq t \wedge t \in \pi$

Proof. We will prove each of the directions in turn.

To show the $\Rightarrow$ direction, take some $\pi \in Dense$ and some finite prefix m. We can construct $t_{m\varepsilon}$ from m by simply replacing any final $\circ$ with ε, for some designated ε. By definition $m \leq t_{m\varepsilon}$ and moreover, since $t_{m\varepsilon}$ is terminating and $\pi \in Dense$, we can conclude that $t \in \pi$.

To show the $\Leftarrow$ direction, take some $\pi \in 2^{Trace}$ and some terminating trace t; since t is terminating we can choose $m = t$ and since this finite prefix extends only to t we immediately obtain $t \in \pi$. □

We now show that our definition of dense properties is uniquely determined given the trace model, the definition of safety, and three conditions (see Theorem B.11) usually satisfied by the class of liveness properties [12]. The key idea consists in looking at safety properties from a topological point of view [12, 31]. Conditions in Theorem B.11 provide a characterization of another topological class of interest, that is shown to be exactly the class we called *Dense* (Theorem B.12).

Definition B.10 (Trace Topology [12, 31])**.** $\mathcal{A}$ is the topology on *Trace* defined by its closed set being all and only the Safety properties.

Theorem B.11. Let $X \subseteq 2^{Trace}$ such that

$$\begin{array}{llr} i) & Safety \cap X = \{True\} & (\textit{trivial intersection}) \\ ii) & \forall \pi \in 2^{Trace}.\ \exists S \in Safety\ \exists x \in X.\ \pi = S \cap x & (\textit{decomposition}) \\ iii) & \forall x_1 x_2 \in X.\ \forall S \in Safety.\ x_1 = x_2 \cap S \Rightarrow x_2 = x_1 \wedge S = True & (\textit{unique decomposition for } X) \end{array}$$

Then X is the class of the dense sets in $\mathcal{A}$.

Proof. See file TopologyTrace.v, Theorem X_dense_class. □

Theorem B.12. *Dense* is the class of the dense sets in the topology $\mathcal{A}$.

Proof. See file TopologyTrace.v, Lemma Dense_dense. □

Corollary B.13. Assume $X \subseteq 2^{Trace}$ satisfies the assumptions of Theorem B.11, then $X = Dense$

Proof. See file TopologyTrace.v, X_Dense_class. □

A property of legacy trace models that does not hold in our model is that any trace property can be decomposed as the intersection of two liveness properties [12]. To show it, first recall that if a set is dense, then every set including it is still dense. This means that if the topology allows for two disjoint dense sets $D_1 \cap D_2 = \varnothing$, we can always write an arbitrary property π as intersection of two dense sets.

$$\pi = (D_1 \cup \pi) \cap (D_2 \cup \pi)$$

This happens for instance in the trace model of Clarkson *et al.*, where it is possible to write an arbitrary property as intersection of two liveness properties (that play the role of the dense sets) [12, 31] and is strictly related to the fact that only infinite traces are considered. In our trace model it is not possible to have disjoint dense sets as they must all include the set of all finite traces. It follows that a property discarding some terminating trace cannot have a similar decomposition.

B.3 Robust Dense Property Preservation (RDP)

RDP restricts RTP to only dense properties:

$$\begin{aligned} \mathsf{RDP} : \forall \pi \in Dense.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ (\forall \mathbf{C_T}\ t.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi) \end{aligned}$$

Again, one might wonder how one can get dense properties to be *robustly* satisfied in the source and then preserved by compilation. As for robust safety, one concern is that the context may perform bad events to violate the dense property. This can be handled in the same way as for robust safety (§2.2). An additional concern is that the context may refuse to give back control (but not terminate) or silently diverge, thus violating a dense property such as "along every infinite trace, an infinite number of good outputs are produced". For this, the enforcement mechanism may use time-outs on the context, forcing it to relinquish control to the partial program periodically. Alternatively, we may add information to traces about whether the context or the partial program produces an event, and weaken dense properties of interest to include traces in which the context keeps control forever.

The property-free variant of RDP, called RDC, restricts RTC to only back-translating *non-terminating* traces:

$$\mathsf{RDC} :\ \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t\ \textit{non-terminating}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow$$

$$\exists \mathsf{C_S}.\ \mathsf{C_S}\left[\mathsf{P}\right] \rightsquigarrow t$$

Non-terminating traces are either infinite or silently divergent. We are not aware of good ways to make use of *infinite* executions $\mathbf{C_T}\left[\mathsf{P}{\downarrow}\right] \rightsquigarrow t$ to produce a *finite* context $\mathsf{C_S}$, so, unlike for RSC, back-translation proofs of RDC will likely have to rely only on $\mathbf{C_T}$ and P, not t, to construct $\mathsf{C_S}$.

Finally, we have proved that RTP *strictly implies* RDP (; §E.1). The counterexample compilation chain we use for showing the separation is roughly the inverse of the one we used for RSP (Theorem 2.1). We take the source to be arbitrary, with the sole assumption that there exists a program P_Ω that can produce a single infinite trace w irrespective of the context. We compile programs by simply pairing them with a constant bound on the number of steps, i.e., $\mathsf{P}{\downarrow} = (\mathsf{P}, k)$. On the one hand, RDC holds vacuously, as target programs cannot produce infinite traces. On the other hand, this compilation chain does not have RTP, since the property $\pi = \{w\}$ is robustly satisfied by P_Ω in the source but not by its compilation (P_Ω, k) in the target.

This separation result does not hold in models with only infinite traces, wherein any trace property can be decomposed as the intersection of two liveness properties [12]. In fact, in that model, the analogue of RDP—Robust Liveness Property Preservation—and RTP trivially coincide.

Further, neither RDP nor RSP implies the other. This follows because every property can be written as the intersection of a safety and a dense property (§B.2). So, if RDP implies RSP, then RDP must imply RTP, which we just proved to not hold. By a dual argument, RSP does not imply RDP. More details are given in §E.1.

Appendix C
Secure Compilation Criteria

This appendix describes all the new secure compilation criteria considered in this work, depending on what class of properties they robustly preserve: arbitrary trace properties (Section C.1.1), safety properties (Section C.1.2), dense properties (Section C.1.3); arbitrary hyperproperties (Section C.2.1), subset-closed hyperproperties (Section C.2.2), including K-subset-closed hyperproperties (Section C.2.3), hypersafety (Section C.2.4), including K-hypersafety (Section C.2.5), hyperliveness (Section C.2.6); and arbitrary relational hyperproperties (Section C.4.1) and properties (Section C.3.2), their K- and 2-relational variants (Section C.4.2, Section C.3.1), and safety relational properties (Section C.3.3), including the finite, K-, and 2-relational variants (Section C.3.4). We also describe the relaxed (X) variants of relational safety (Section C.3.5).

Each of these sections gives two definitions: a criterion that is explicit about the class of properties it robustly preserves, and an equivalent characterization that is *property free*, and is thus better suited for proofs.

As in the introduction, we organize these criteria in the diagram from Figure 3, where criteria above imply criteria below, and arrows indicate the strict separation between the two criteria, that is the existence of a compilation chain satisfying the lower criterion but not the higher. These separation results are described in Section E.

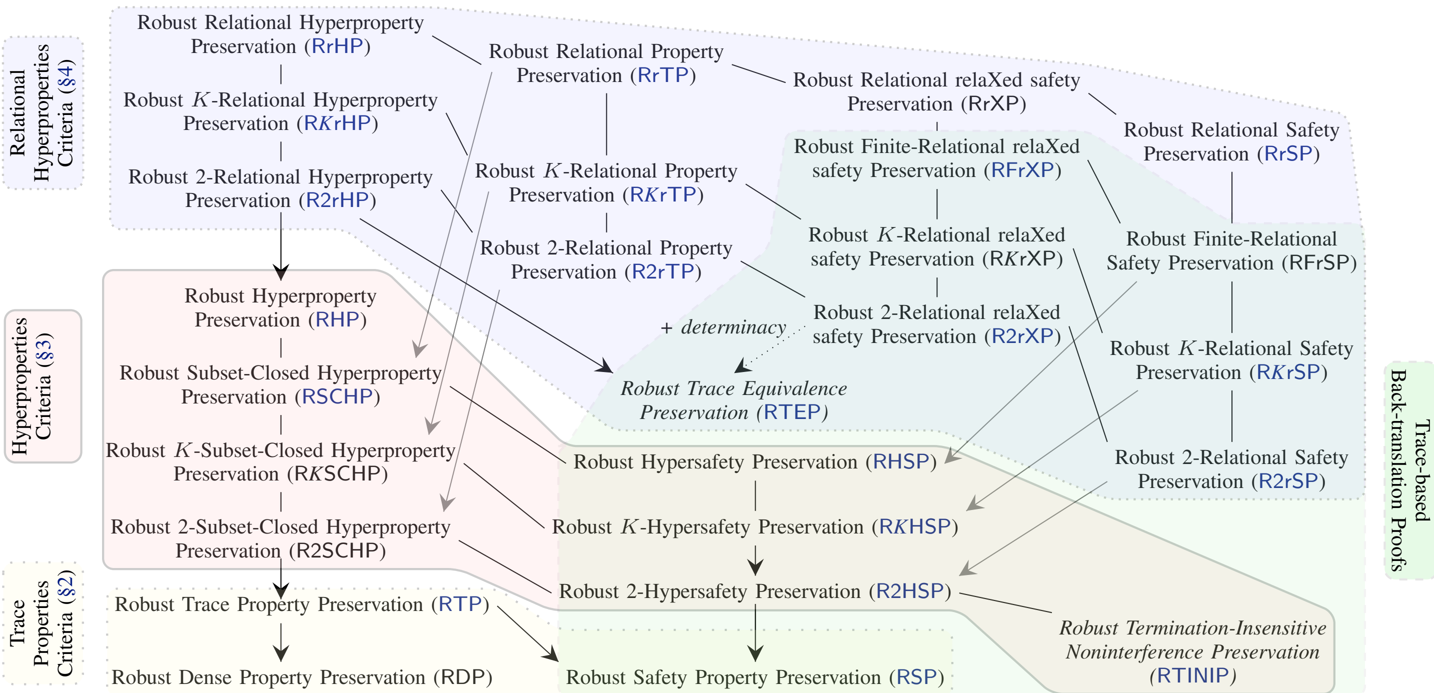


Fig. 3: Partial order of the secure compilation criteria studied in this paper.

C.1 Trace Property-Based Criteria

We start by describing the three criteria at the bottom of the lattice, RTP, RSP, and RDP, corresponding to the robust preservation of *trace properties*, defined as sets of allowed traces.

These criteria state that for any property π in the class they preserve, if a source program can only produce traces belonging to π, when linked with any source context, then the same is true of the compiled program linked with any target context.

C.1.1 Robust Trace Property Preservation

The first of these criteria is called *Robust Trace Property Preservation*, or RTP, and corresponds to the robust preservation of all trace properties.

Definition C.1 (Robust Trace Property Preservation (RTP))**.**

$$\begin{aligned}\mathsf{RTP}: \quad & \forall \pi \in 2^{\mathit{Trace}}. \forall \mathsf{P}. (\forall \mathsf{C_S}\ t. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ & \qquad (\forall \mathbf{C_T}\ t. \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)\end{aligned}$$

The property-free characterization of RTP is RTC. This characterization captures the fact that if a target program can produce a given trace, then the source can also produce this trace. Intuitively, this corresponds to the fact that any violation of a trace property in the target can be explained by the same violation in the source.

Definition C.2 (Equivalent Characterization of RTP (RTC))**.**

$$\mathsf{RTC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \\ \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

Theorem C.3 (RTP and RTC are equivalent)**.**

$$\mathsf{RTP} \iff \mathsf{RTC}$$

Proof. See file Criteria.v, theorem RTC_RTP for a Coq proof. The proof is simple, but still illustrative for how such proofs work in general:

$\Rightarrow$ Let P be arbitrary. We need to show that $\forall \mathbf{C_T}. \forall t. \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S}. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$. We can directly conclude this by applying RTP to P and the property $\pi = \{t \mid \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t\}$; for this application to be possible we need to show that $\forall \mathsf{C_S}\ t. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S'}. \mathsf{C_S'}[\mathsf{P}] \rightsquigarrow t$, which is trivial if taking $\mathsf{C_S'} = \mathsf{C_S}$.

$\Leftarrow$ Given a compilation chain that satisfies RTC and some P and π so that $\forall \mathsf{C_S}\ t. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi\ (H)$ we have to show that $\forall \mathbf{C_T}\ t. \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow t \in \pi$. Let $\mathbf{C_T}$ and t so that $\mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t$, we still have to show that $t \in \pi$. We can apply RTC to obtain $\exists \mathsf{C_S}. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$, which we can use to instantiate H to conclude that $t \in \pi$. □

C.1.2 Robust Safety Property Preservation

Robust Safety Property Preservation is the criterion corresponding to the robust preservation of *safety* properties, i.e., properties that can be finitely refuted: for any safety property, and any trace not in the property, there exists a finite *bad prefix* of the trace that can not be extended to belong to the property.

Definition C.4 (Safety Property)**.** We define the set of safety properties, *Safety*:

$$\mathit{Safety} \triangleq \left\{\pi \in 2^{\mathit{Trace}} \;\middle|\; \forall t \notin \pi.\ \exists m \leq t.\ \forall t' \geq m.\ t' \notin \pi\right\}$$

A property π is a safety property if and only if $\pi \in \mathit{Safety}$.

Definition C.5 (Robust Safety Property Preservation (RSP))**.**

$$\mathsf{RSP}: \quad \forall \pi \in \mathit{Safety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ (\forall \mathbf{C_T}\ t. \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

The equivalent property-free characterization, RSC, captures the fact that a safety property can be refuted by one finite bad prefix m: all finite violation of a safety property at the target level can be explained by the same finite violation at the source level.

Definition C.6 (Equivalent Characterization of RSP (RSC))**.**

$$\mathsf{RSC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall m.\ \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow m \Rightarrow \\ \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow m$$

Theorem C.7 (RSP and RSC are equivalent)**.**

$$\mathsf{RSP} \iff \mathsf{RSC}$$

Proof. See file Criteria.v, theorem RSC_RSP. □

C.1.3 Robust Dense Property Preservation

Robust Dense Property Preservation is the criterion corresponding to the robust preservation of *dense* properties. Dense properties are the properties that include all finite traces, and roughly correspond to *liveness* in our model. See Section B and Section B.2 for more details.

A more detailed view of Robust Dense Property Preservation is given in Section B.3.

Definition C.8 (Dense Property)**.** We define the set of dense property, *Dense*:

$$\mathit{Dense} \triangleq \left\{\pi \in 2^{\mathit{Trace}} \;\middle|\; \forall t\ \mathit{terminating}.\ t \in \pi\right\}$$

A property π is a dense property if and only if $\pi \in \mathit{Dense}$.

Definition C.9 (Robust Dense Property Preservation (RDP))**.**

$$\mathsf{RDP}: \quad \forall \pi \in \mathit{Dense}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t. \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow \\ (\forall \mathbf{C_T}\ t. \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

The property-free characterization RDC captures the fact that dense properties can be violate only by infinite traces.

Definition C.10 (Equivalent Characterization of RDP (RDC))**.**

$$\mathsf{RDC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t \text{ infinite}.\ \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \\ \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

Theorem C.11 (RDP and RDC are equivalent)**.**

$$\mathsf{RDP} \iff \mathsf{RDC}$$

Proof. See file Criteria.v, theorem RDC_RDP. ☐

C.2 Hyperproperty-Based Criteria

The criteria in this section describe the robust preservation of *hyperproperty*, that is sets of allowed program behaviors. Formally, a hyperproperty is an element H of the set $2^{2^{\mathit{Trace}}}$, and a program P satisfies this hyperproperty H if and only if $\mathtt{Behav}\,(P) \in H$. Hyperproperties allow to express more security properties than trace properties, such as noninterference for instance.

Again, these criteria state that for any hyperproperty H in the class they preserve, if a source program's behavior when linked with any any source context belongs to H, then the same is true for the compiled program.

Note that the behavior being considered is not the set of traces generated by the program when linked with all source contexts, but only the set of traces generated when linked with a particular context.

C.2.1 Robust Hyperproperty Preservation

Robust Hyperproperty Preservation is the criterion corresponding to the robust preservation of all hyperproperties.

Definition C.12 (Robust Hyperproperty Preservation (RHP))**.**

$$\mathsf{RHP}: \quad \forall H \in 2^{2^{\mathit{Trace}}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\,(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow \\ (\forall \mathbf{C_T}.\ \mathtt{Behav}\,(\mathbf{C_T}[\mathsf{P}\downarrow]) \in H)$$

The equivalent characterization, RHC, states that all behaviors of the compiled program are behaviors of the source program: if the compiled program violate a hyperproperty with a particular behavior, then the source program does too.

Definition C.13 (Equivalent Characterization of RHP (RHC))**.**

$$\mathsf{RHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \mathtt{Behav}\,(\mathbf{C_T}[\mathsf{P}\downarrow]) = \mathtt{Behav}\,(\mathsf{C_S}[\mathsf{P}])$$

Unfolding this definition gives:

$$\mathsf{RHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ (\mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \iff \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t)$$

Theorem C.14 (RHP and RHC are equivalent)**.**

$$\mathsf{RHP} \iff \mathsf{RHC}$$

Proof. See file Criteria.v, theorem RHC_RHP. ☐

C.2.2 Robust Subset-Closed Hyperproperty Preservation

In general a program satisfies a certain hyperproperty if its set of traces, its behavior, is in the hyperproperty. With subset-closed hyperproperties (§3.2), if a set of traces is accepted then so are all smaller sets of traces. Subset closed hyperproperties can therefore be used to formalize the notion of *refinement* [31].

Definition C.15 (Subset-Closed Hyperproperties)**.** We define the set of Subset-Closed Hyperproperties, *SC*:

$$\mathit{SC} \triangleq \{H \mid \forall b_1 \subseteq b_2. b_2 \in H \Rightarrow b_1 \in H\}$$

A hyperproperty H is subset-closed if and only if $H \in \mathit{SC}$.

Definition C.16 (Robust Subset-Closed Hyperproperty Preservation (RSCHP))**.**

$$\mathsf{RSCHP}: \quad \forall H \in \mathit{SC}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\,(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow \\ (\forall \mathbf{C_T}.\ \mathtt{Behav}\,(\mathbf{C_T}[\mathsf{P}\downarrow]) \in H)$$

The equivalent characterization of RSCHP states that the behaviors of a compiled program in an arbitrary target context are the refinement of the behaviors of the original program in some source context.

Definition C.17 (Equivalent Characterization of RSCHP (RSCHC))**.**

$$\mathsf{RSCHC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

Theorem C.18 (RSCHP and RSCHC are equivalent)**.**

$$\mathsf{RSCHP} \iff \mathsf{RSCHC}$$

Proof. See file Criteria.v, RSCHC_RSCHP. □

C.2.3 Robust K-Subset-Closed Hyperproperty Preservation

While for K-Hypersafety a set of K bad finite prefixes is enough to refuse a behavior, for K-subset-closed hyperproperties, K complete traces could be necessary.

Definition C.19 (K-Subset-Closed Hyperproperties [67])**.** We define the set of K-Subset-Closed Hyperproperties, *KSC*:

$$\mathit{KSC} \triangleq \{H \mid \forall b.\ b \notin H \iff (\exists T_K \subseteq b.\ (|T_K| \leq K \wedge T_K \notin H))\}$$

A hyperproperty H is K-subset-closed if and only if $H \in \mathit{KSC}$.

Definition C.20 (Robust K-Subset-Closed Hyperproperty Preservation (RKSCHP))**.**

$$\begin{aligned}\mathsf{R}K\mathsf{SCHP}: \quad & \forall H \in \mathit{KSC}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow \\ & (\forall \mathbf{C_T}.\ \mathtt{Behav}(\mathbf{C_T}[\mathsf{P}{\downarrow}]) \in H)\end{aligned}$$

Definition C.21 (Equivalent Characterization of RKSCHP (RKSCHC))**.**

$$\begin{aligned}\mathsf{R}K\mathsf{SCHC}: \quad & \forall \mathsf{P}, \mathbf{C_T}.\forall \hat{t}.\|\hat{t}\| = K. \\ & (\hat{t} \subseteq \mathtt{Behav}(\mathbf{C_T}[\mathsf{P}{\downarrow}])) \Rightarrow \exists \mathsf{C_S}.(\hat{t} \subseteq \mathtt{Behav}(\mathsf{C_S}[\mathsf{P}]))\end{aligned}$$

Theorem C.22 (RKSCHP and RKSCHC are equivalent)**.**

$$\mathsf{R}K\mathsf{SCHP} \iff \mathsf{R}K\mathsf{SCHC}$$

Proof. Analogous to that of Theorem C.23 below. □

R2SCHC is an instance of Definition C.21 with $\|\hat{t}\| = 2$. Similarly, R2SCHP is an instance of Definition C.20 for *2SC*.

Theorem C.23 (R2SCHP and R2SCHC are equivalent)**.**

$$\mathsf{R2SCHP} \iff \mathsf{R2SCHC}$$

Proof. See file Criteria.v, theorem R2SCHC_R2SCHPC. □

C.2.4 Robust Hypersafety Preservation

Robust Hypersafety Preservation (§3.3) is the criterion corresponding to the robust preservation of hypersafety properties (aka. safety hyperproperties), i.e., hyperproperties that can be refuted by a finite number of finite trace prefixes.

Hypersafety is a generalization of safety that captures many important security properties, such as noninterference. Informally, a hypersafety property disallows a certain finite observation, i.e., a finite set of finite prefixes. This observation $o \in \mathit{Obs}$ is a "bad observation", and all its extensions cannot satisfy the hyperproperty.

Definition C.24 (Observations)**.** We define the set of observations, *Obs*:

$$\mathit{Obs} \triangleq 2^{\mathit{FinPref}}_{\mathit{Fin}}$$

Definition C.25 (Hypersafety Property)**.** We define the set of hypersafety properties, *Hypersafety*:

$$\mathit{Hypersafety} \triangleq \{H \mid \forall b \notin H.\ (\exists o \in \mathit{Obs}.\ o{\leq}b \wedge (\forall b'{\geq}o.\ b' \notin H))\}$$

A hyperproperty H is a safety hyperproperty, or a hypersafety property, if and only if $H \in \mathit{Hypersafety}$.

Definition C.26 (Robust Hypersafety Preservation (RHSP))**.**

$$\begin{aligned}\mathsf{RHSP}: \quad & \forall H \in \mathit{Hypersafety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}(\mathsf{C_S}[\mathsf{P}]) \in H) \Rightarrow \\ & (\forall \mathbf{C_T}.\ \mathtt{Behav}(\mathbf{C_T}[\mathsf{P}{\downarrow}]) \in H)\end{aligned}$$

The property-free characterization captures the fact that if a compiled program can produce an observation o (in the sense that it contains a prefix of each trace of the program) that refutes a hypersafety property, then the same observation can also be produced by the source program.

Definition C.27 (Equivalent Characterization of RHSP (RHSC)).

$$\mathsf{RHSC}: \quad \forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall o \in \mathit{Obs}.\ o \leq \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right) \Rightarrow \exists \mathsf{C_S}.\ o \leq \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)$$

Theorem C.28 (RHSP and RHSC are equivalent).

$$\mathsf{RHSP} \iff \mathsf{RHSC}$$

Proof. See file Criteria.v, theorem RHSC_RHSP. □

C.2.5 Robust K-Hypersafety Preservation

K-hypersafety properties are hypersafety properties that can be refuted by observations of size at most K, that is one need only K appropriately chosen finite prefixes to prove a program doesn't satisfy the hyperproperty.

Definition C.29 (K-Observations). We define the set of K-observations, i.e., observations of cardinal at most K:

$$\mathit{Obs}_K \triangleq 2^{\mathit{FinPref}}_{\mathit{Fin}(K)}$$

Definition C.30 (K-Hypersafety Property). We define the set of K-hypersafety properties:

$$\mathit{KHypersafety} \triangleq \{H \mid \forall b \notin H.\ (\exists o \in \mathit{Obs}_K.\ o{\leq}b \wedge (\forall b'{\geq}o.\ b' \notin H))\}$$

A hyperproperty H is K-hypersafety if and only if $H \in \mathit{KHypersafety}$.

Definition C.31 (Robust K-Hypersafety Preservation (RKHSP)).

$$\mathsf{R}K\mathsf{HSP}: \quad \forall H \in \mathit{KHypersafety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\, \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right) \in H) \Rightarrow (\forall \mathbf{C_T}.\, \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right) \in H)$$

The property-free characterization has the same intuition, except it is restricted to behaviors of size K.

Definition C.32 (Equivalent Characterization of RKHSP (RKHSC)).

$$\mathsf{R}K\mathsf{HSC}: \quad \forall \mathsf{P}, \mathbf{C_T}.\ \forall \widehat{m}.\ \|\widehat{m}\| = K \Rightarrow\ \widehat{m} \leq \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right) \Rightarrow \exists \mathsf{C_S}.\widehat{m} \leq \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)$$

Theorem C.33 (RKHSP and RKHSC are equivalent).

$$\mathsf{R}K\mathsf{HSP} \iff \mathsf{R}K\mathsf{HSC}$$

Proof. Analogous to Theorem C.34 below. □

R2HSP is an instance of Definition C.31 for $K = 2$. Similarly, R2HSC is an instance of Definition C.32 for $K = 2$.

Theorem C.34 (R2HSP and R2HSC are equivalent).

$$\mathsf{R2HSP} \iff \mathsf{R2HSC}$$

Proof. See file Criteria.v, theorem R2HSC_R2HSP. □

A particular instance of R2HSP is RTINIP (§3.3).

C.2.6 Robust Hyperliveness Preservation

Definition C.35 (Hyperliveness Property). We define the set of hyperliveness properties (or liveness hyperproperties) *Hyperliveness*:

$$\mathit{Hyperliveness} \triangleq \{H \mid \forall o \in \mathit{Obs}.\ \exists b{\geq}o.\ b \in H\}$$

A hyperproperty H is a hyperliveness property if and only if $H \in \mathit{Hyperliveness}$.

Definition C.36 (Robust Hyperliveness Preservation (RHLP)).

$$\mathsf{RHLP}: \quad \forall H \in \mathit{Hyperliveness}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\, \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right) \in H) \Rightarrow (\forall \mathbf{C_T}.\, \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right) \in H)$$

We give no property-free characterization for RHLP, since RHLP collapses with RHP, as was pointed out in §3.5:

Theorem C.37 (RHP and RHLP are equivalent).

$$\mathsf{RHP} \iff \mathsf{RHLP}$$

Proof. See file Criteria.v, theorem RHLP_RHP. □

C.3 Relational Trace Property-Based Criteria

Relational trace properties are a generalization of trace properties to allow comparing individual runs of different programs. For instance, relational trace properties allow expressing properties such as "program P_1 runs faster than P_2 on every input".

C.3.1 Robust K-Relational Trace Property Preservation

A K-relational trace property is a relational trace property of arity K, that is a relation R between K traces.
Given K programs, this programs are said to satisfy the K-relation R if and only if for any traces $t_1, \ldots, t_K$ they can produce when linked with the same context, $(t_1, \ldots, t_K) \in R$. Here, we only give an explicit definition in the case of 2-relations. These definitions can be lifted trivially to the case of K-relations.

Definition C.38 (Robust 2-Relational Trace Property Preservation (R2rTP))**.**

$$\begin{aligned}\mathsf{R2rTP}: \quad & \forall R \in 2^{(\mathit{Trace}^2)}.\ \forall \mathsf{P}_1\ \mathsf{P}_2.\ (\forall \mathsf{C_S}\ t_1\ t_2.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow t_1 \wedge \mathsf{C_S}[\mathsf{P}_2] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R) \Rightarrow \\ & (\forall \mathbf{C_T}\ t_1\ t_2.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow t_1 \wedge \mathbf{C_T}[\mathsf{P}_2{\downarrow}] \rightsquigarrow t_2) \Rightarrow (t_1, t_2) \in R)\end{aligned}$$

The equivalent characterization captures the following intuition: if compiled programs are unrelated by a relation R because of certain traces, they the source programs are also unrelated because of the same traces.

Definition C.39 (Equivalent Characterization of R2rTP (R2rTC))**.**

$$\begin{aligned}\mathsf{R2rTC}: \quad & \forall \mathsf{P}_1\ \mathsf{P}_2.\ \forall \mathbf{C_T}.\ \forall t_1\ t_2.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow t_1 \wedge \mathbf{C_T}[\mathsf{P}_2{\downarrow}] \rightsquigarrow t_2) \Rightarrow \\ & \exists \mathsf{C_S}.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow t_1 \wedge \mathsf{C_S}[\mathsf{P}_2] \rightsquigarrow t_2)\end{aligned}$$

Theorem C.40 (R2rTP and R2rTC are equivalent)**.**

$$\mathsf{R2rTP} \iff \mathsf{R2rTC}$$

Proof. See file Criteria.v, theorem R2rTC_R2rTP. □

The definitions of RKrTP and RKrTC are an easy generalization.

Theorem C.41 (RKrTP and RKrTC are equivalent)**.**

$$\mathsf{R}K\mathsf{rTP} \iff \mathsf{R}K\mathsf{rTC}$$

Proof. Analogous to Theorem C.40. □

C.3.2 Robust Relational Trace Property Preservation

Relational trace properties (§4.2) are a generalization of the previous relational trace properties, allowing comparing individual runs of countably many programs. They are defined as predicate over (infinite) sequence of programs.

Definition C.42 (Robust Relational Trace Property Preservation (RrTP))**.**

$$\begin{aligned}&\mathsf{RrTP}: \forall R \in 2^{(\mathit{Trace}^\omega)}.\ \forall \mathsf{P}_1, .., \mathsf{P}_\mathsf{K}, ... \\ &(\forall \mathsf{C_S}.\ \forall t_1, .., t_k, ..(\forall i. \mathsf{C_S}[\mathsf{P_i}] \rightsquigarrow t_i) \Rightarrow (t_1, .., t_k, ..) \in R) \Rightarrow \\ &(\forall \mathbf{C_T}.\ \forall t_1, .., t_k, ..(\forall i. \mathbf{C_T}[\mathbf{P_i}] \rightsquigarrow t_i) \Rightarrow (t_1, .., t_k, ..) \in R)\end{aligned}$$

Definition C.43 (Equivalent Property-Full Characterization of (RrTP′))**.**

$$\begin{aligned}\mathsf{RrTP}': \quad & \forall R \in 2^{(\mathsf{Progs} \rightarrow \mathit{Trace})}.\ (\forall \mathsf{C_S}. \forall f.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P})) \Rightarrow f \in R) \Rightarrow \\ & (\forall \mathbf{C_T}. \forall f.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow f \in R)\end{aligned}$$

Definition C.44 (Equivalent Property-Free Characterization of RrTP (RrTC))**.**

$$\begin{aligned}\mathsf{RrTC}: \quad & \forall f : \mathsf{Progs} \rightarrow \mathit{Trace}.\ \forall \mathbf{C_T}.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow \\ & \exists \mathsf{C_S}.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P}))\end{aligned}$$

Theorem C.45 (RrTP′ and RrTC are equivalent)**.**

$$\mathsf{RrTP}' \iff \mathsf{RrTC}$$

Proof. See file Criteria.v, theorem RrTC_RrTP'. □

Theorem C.46 (RrTP′ and RrTP)**.** Assuming the set Progs is countable,

$$\mathsf{RrTP}' \iff \mathsf{RrTP}$$

Proof. Same argument used in Theorem C.77. □

Theorem C.47 (RrTP and RrTC)**.** Assuming the set Progs is countable,

$$\mathsf{RrTP} \iff \mathsf{RrTC}$$

Proof. Follows from Theorem C.46 and Theorem C.45. ☐

C.3.3 Robust Relational Safety Preservation

See Section 4.3 for a more detailed account of robust relational safety preservation.

C.3.4 Robust Finite-relational Safety Preservation

A relation $R \in 2^{Trace^K}$ is a K-ary relational safety property if for every "bad" K-trace $(t_1, \ldots, t_K) \notin R$, there exists a set of prefixes $m_1, \ldots, m_k \in \mathit{FinPref}$ such that $m_i \leq t_i,\ i = 1, \ldots, K$, and every K-trace $(t'_1, \ldots, t'_K)$ that extends the set of "bad" prefixes pointwise is also not in the relation, i.e., $m_i \leq t'_i,\ i = 1, \ldots, K$ implies $(t'_1, \ldots, t'_K) \notin R$.
We provide the definition for preservation of the robust satisfaction of relational safety of arity 2 (Definition C.48), the reader can easily deduce the definition for arity K, that we denote by RKrSP.

At arity 2, we define *Robust 2-relational Safety Preservation* (R2rSP) as follows (cf. Definition C.48).

Definition C.48 (Robust 2-Relational Safety Preservation (R2rSP))**.**

$$\begin{aligned}\mathsf{R2rSP}:\ & \forall R \in 2\text{-}\mathit{relational\ Safety}.\ \forall \mathsf{P_1}\ \mathsf{P_2}.\\ & (\forall \mathsf{C_S}\ t_1\ t_2.(\mathsf{C_s}[\mathsf{P_1}] \rightsquigarrow t_1 \wedge \mathsf{C_s}[\mathsf{P_2}] \rightsquigarrow t_2) \implies (t_1, t_2) \in R) \implies\\ & (\forall \mathbf{C_T}\ t_1\ t_2.(\mathbf{C_T}[\mathsf{P_1}{\downarrow}] \rightsquigarrow t_1 \wedge \mathbf{C_T}[\mathsf{P_2}{\downarrow}] \rightsquigarrow t_2) \implies (t_1, t_2) \in R)\end{aligned}$$

We show that R2rSP can be written in the following form, more convenient to work with.

Definition C.49 (Equivalent Characterization of R2rSP (R2rSC))**.**

$$\begin{aligned}\mathsf{R2rSC}:\quad & \forall \mathsf{P_1}\ \mathsf{P_2}.\ \forall \mathbf{C_T}.\ \forall m_1\ m_2.\ (\mathbf{C_T}[\mathsf{P_1}{\downarrow}] \rightsquigarrow m_1 \wedge \mathbf{C_T}[\mathsf{P_2}{\downarrow}] \rightsquigarrow m_2) \Rightarrow\\ & \qquad \exists \mathsf{C_S}.\ (\mathsf{C_S}[\mathsf{P_1}] \rightsquigarrow m_1 \wedge \mathsf{C_S}[\mathsf{P_2}] \rightsquigarrow m_2)\end{aligned}$$

Theorem C.50 (R2rSP and R2rSC are equivalent)**.**

$$\mathsf{R2rSP} \iff \mathsf{R2rSC}$$

Proof. See Criteria.v, theorem R2rSC_R2rSP. ☐

The following theorem gives us an alternative formulation of R2rSP, in terms of preservation of robust satisfaction of relations over finite prefixes. Theorem C.51 allows us to define in a more elegant way the criteria for arbitrary (but finite) relational safety (Definition C.53) as well infinite ones (Definition C.56). A similar theorem holds for RKrSP.

Theorem C.51 (Characterization of R2rSP using finite prefixes)**.**

$$\begin{aligned}\mathsf{R2rSP} \iff & \forall R \in 2^{(\mathit{FinPref}^2)}.\ \forall \mathsf{P_1}\ \mathsf{P_2}.\\ & (\forall \mathsf{C_S}\ m_1\ m_2.(\mathsf{C_s}[\mathsf{P_1}] \rightsquigarrow m_1 \wedge \mathsf{C_s}[\mathsf{P_2}] \rightsquigarrow m_2) \implies (m_1, m_2) \in R) \implies\\ & (\forall \mathbf{C_T}\ m_1\ m_2.(\mathbf{C_T}[\mathsf{P_1}{\downarrow}] \rightsquigarrow m_1 \wedge \mathbf{C_T}[\mathsf{P_2}{\downarrow}] \rightsquigarrow m_2) \implies (m_1, m_2) \in R)\end{aligned}$$

Proof. See Criteria.v, Theorem R2rSP_R2rSC'. ☐

Notice that in the second script we quantify over arbitrary relations over finite prefixes. This captures the main difference between this criterion and the stronger R2rTP: it considers finite prefixes rather than full traces. This is also the case in the equivalent property free characterization, R2rSC.

The definitions of RKrSP and RKrSC are an easy generalization.

Theorem C.52 (RKrSP and RKrSC are equivalent)**.**

$$\mathsf{R}K\mathsf{rSP} \iff \mathsf{R}K\mathsf{rSC}$$

Proof. Analogous to Theorem C.50. ☐

Finally, we define RFrSP as the union over all K of the RKrSPs.

Definition C.53 (Robust Finite-relational Safety Preservation (RFrSP))**.**

$$\begin{aligned}\mathsf{RFrSP}:\ & \forall K, \mathsf{P_1}, \cdots, \mathsf{P_k}, R \in 2^{(\mathit{FinPref}^k)}.\\ & (\forall \mathsf{C_S}, m_1, \cdots, m_k, (\mathsf{C_S}[\mathsf{P_1}] \rightsquigarrow m_1 \wedge \cdots \wedge \mathsf{C_S}[\mathsf{P_k}] \rightsquigarrow m_k)\\ & \quad \Rightarrow (m_1, \cdots, m_k) \in R) \Rightarrow\end{aligned}$$

$$(\forall \mathbf{C_T}.(\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow m_1 \wedge \cdots \wedge \mathbf{C_T}[\mathsf{P}_k{\downarrow}] \rightsquigarrow m_k) \Rightarrow (m_1, \cdots, m_k) \in R)$$

The intuition for this property-free criterion is the same as for finite-relational properties, except it only requires considering finite prefixes.

Definition C.54 (Equivalent Characterization of RFrSP (RFrSC))**.**

$$\mathsf{RFrSC}: \quad \forall K.\ \forall \mathsf{P}_1 \ldots \mathsf{P}_K.\ \forall \mathbf{C_T}.\ \forall m_1 \ldots m_K.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow m_1 \wedge \ldots \wedge \mathbf{C_T}[\mathsf{P}_K{\downarrow}] \rightsquigarrow m_K) \Rightarrow \exists \mathsf{C_S}.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow m_1 \wedge \ldots \wedge \mathsf{C_S}[\mathsf{P}_K] \rightsquigarrow m_K)$$

Theorem C.55 (RFrSP and RFrSC are equivalent)**.**

$$\mathsf{RFrSP} \iff \mathsf{RFrSC}$$

Proof. Analogous to Theorem C.50. □

Definition C.56 (Robust relational Safety Preservation (RrSP))**.**

$$\mathsf{RrSP}: \quad \forall R \in 2^{(\mathit{FinPref})^\omega}.\forall P_1, .., P_k, ..(\forall \mathsf{C_S}.\forall m_1, .., m_k, ... (\forall i.\ \mathsf{C_S}[\mathsf{P_i}] \rightsquigarrow m_i) \Rightarrow (m_1, .., m_k, ..) \in R) \Rightarrow (\forall \mathbf{C_T}.\forall m_1, .., m_k, ... (\forall i.\ \mathbf{C_T}[\mathsf{P_i}{\downarrow}] \rightsquigarrow m_i) \Rightarrow (m_1, .., m_k, ..) \in R)$$

Definition C.57 (Equivalent Property-Full Characterization of RrSP′ (RrSP′))**.**

$$\mathsf{RrSP}': \quad \forall R \in 2^{(\mathsf{Progs} \to \mathit{FinPref})}.\ (\forall \mathsf{C_S}.\forall f.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P})) \Rightarrow R(f)) \Rightarrow (\forall \mathbf{C_T}.\forall f.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow R(f))$$

Definition C.58 (Equivalent Property-Free Characterization of RrSP (RrSC))**.**

$$\mathsf{RrSC}: \quad \forall f : \mathsf{Progs} \to \mathit{FinPref}.\ \forall \mathbf{C_T}.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow \exists \mathsf{C_S}.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P}))$$

Theorem C.59 (RrSC and RrSP′ are equivalent)**.**

$$\mathsf{RrSP}' \iff \mathsf{RrSC}$$

Proof. See file Criteria.v, theorem RrSC_RrSP'. □

Theorem C.60 (RrSP′ and RrSP)**.** Assuming the set Progs is countable,

$$\mathsf{RrSP}' \iff \mathsf{RrSP}$$

Proof. Same argument used in Theorem C.77. □

Theorem C.61 (RrSP and RrSC)**.** Assuming the set Progs is countable,

$$\mathsf{RrSP} \iff \mathsf{RrSC}$$

Proof. Follows from Theorem C.60 and Theorem C.59. □

C.3.5 Robust Relational Relaxed Safety Preservation

Relational Relaxed Safety properties generalize relational safety properties as they consider *XPref* instead of *FinPref*. The semantics of a programming language can capture more than just finite prefixes of complete traces justifying the definition of *Xpref* (see Section A). For instance, silent divergence is not finitely observable, but can be represented and produced by small-step semantics, for instance with a rule such as $\dfrac{\forall n, e \xrightarrow{\epsilon}{}^{n} e'}{e \rightsquigarrow \Uparrow}$ (Silent-Div) where $\Uparrow$ represents silent divergence.

The criteria in this section are defined exactly as the criteria in the previous section, except they deal with extended prefixes instead of finite prefixes.

Definition C.62 (Robust relational relaXed safety Preservation (RrXP))**.**

$$\mathsf{RrSP}: \quad \forall R \in 2^{(\mathit{XPref})^\omega}.\forall P_1, .., P_k, ..(\forall \mathsf{C_S}.\forall x_1, .., x_k, ... (\forall i.\ \mathsf{C_S}[\mathsf{P_i}] \rightsquigarrow x_i) \Rightarrow (x_1, .., x_k, ..) \in R) \Rightarrow (\forall \mathbf{C_T}.\forall x_1, .., x_k, ... (\forall i.\ \mathbf{C_T}[\mathsf{P_i}{\downarrow}] \rightsquigarrow x_i) \Rightarrow (x_1, .., x_k, ..) \in R)$$

Definition C.63 ((RrXP′))**.**

$$\mathsf{RrXP}: \quad \forall R \in 2^{(\mathsf{Progs} \to \mathit{XPref})}.\ (\forall \mathsf{C_S}.\forall f.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P})) \Rightarrow R(f)) \Rightarrow (\forall \mathbf{C_T}.\forall f.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow R(f))$$

Definition C.64 (Equivalent Characterization of RrXP' (RrXC))**.**

$$\mathsf{RrXC}: \quad \forall f : \mathsf{Progs} \to \mathit{XPref}.\ \forall \mathbf{C_T}.\ (\forall \mathsf{P}.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow f(\mathsf{P})) \Rightarrow \\ \exists \mathsf{C_S}.\ (\forall \mathsf{P}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow f(\mathsf{P}))$$

Theorem C.65 (RrXC and RrXP' are equivalent)**.**

$$\mathsf{RrXP} \iff \mathsf{RrXC}$$

Proof. See file Criteria.v, theorem RrXC_RrXP'. □

Theorem C.66 (RrXP' and RrXP)**.** Assuming the set Progs is countable,

$$\mathsf{RrXP}' \iff \mathsf{RrXP}$$

Proof. Same argument used in Theorem C.77. □

Theorem C.67 (RrXP and RrXC)**.** Assuming the set Progs is countable,

$$\mathsf{RrXP} \iff \mathsf{RrXC}$$

Proof. Follows from Theorem C.66 and Theorem C.65. □

C.3.6 Robust Finite-Relational Relaxed Safety Preservation

Definition C.68 (Robust Finite-relational relaXed safety Preservation (RFrXP))**.**
RFrXP :

$$\begin{aligned}
&\forall K, \mathsf{P}_1, \cdots, \mathsf{P}_\mathsf{K}, R \in 2^{(\mathit{XPref}^K)}.\\
&(\forall \mathsf{C_S}, x_1, \cdots, x_K, (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \cdots \wedge \mathsf{C_S}[\mathsf{P}_\mathsf{K}] \rightsquigarrow x_K)\\
&\quad \Rightarrow (x_1, \cdots, x_K) \in R) \Rightarrow\\
&(\forall \mathbf{C_T}.(\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \cdots \wedge \mathbf{C_T}[\mathsf{P}_\mathsf{K}{\downarrow}] \rightsquigarrow x_K)\\
&\quad \Rightarrow (x_1, \cdots, x_K) \in R)
\end{aligned}$$

Definition C.69 (Equivalent Characterization of RFrXP (RFrXC))**.**

$$\mathsf{RFrXC}: \quad \forall K.\ \forall \mathsf{P}_1 \ldots \mathsf{P}_\mathsf{K}.\ \forall \mathbf{C_T}.\ \forall x_1 \ldots x_K.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \ldots \wedge \mathbf{C_T}[\mathsf{P}_\mathsf{K}{\downarrow}] \rightsquigarrow x_K) \Rightarrow \\ \exists \mathsf{C_S}.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \ldots \wedge \mathsf{C_S}[\mathsf{P}_\mathsf{K}] \rightsquigarrow x_K)$$

Theorem C.70 (RFrXP and RFrXC are equivalent)**.**

$$\mathsf{RFrXP} \iff \mathsf{RFrXC}$$

Proof. Analogous to Theorem C.73. □

Definition C.71 (Robust 2-relational relaXed safety Preservation ($\mathsf{R2rXP}$))**.**

$$\mathsf{R2rXP}: \quad \forall R \in 2^{(\mathit{XPref}^2)}.\ \forall \mathsf{P}_1\ \mathsf{P}_2.\ (\forall \mathsf{C_S}\ x_1\ x_2.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \mathsf{C_S}[\mathsf{P}_2] \rightsquigarrow x_2) \Rightarrow (x_1, x_2) \in R) \Rightarrow \\ (\forall \mathbf{C_T}\ x_1\ x_2.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \mathbf{C_T}[\mathsf{P}_2{\downarrow}] \rightsquigarrow x_2) \Rightarrow (x_1, x_2) \in R)$$

Definition C.72 (Equivalent Characterization of $\mathsf{R2rXP}$ ($\mathsf{R2rXC}$))**.**

$$\mathsf{R2rXC}: \quad \forall \mathsf{P}_1\ \mathsf{P}_2.\ \forall \mathbf{C_T}.\ \forall x_1\ x_2.\ (\mathbf{C_T}[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \mathbf{C_T}[\mathsf{P}_2{\downarrow}] \rightsquigarrow x_2) \Rightarrow \\ \exists \mathsf{C_S}.\ (\mathsf{C_S}[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \mathsf{C_S}[\mathsf{P}_2] \rightsquigarrow x_2)$$

Theorem C.73 ($\mathsf{R2rXP}$ and $\mathsf{R2rXC}$ are equivalent)**.**

$$\mathsf{R2rXP} \iff \mathsf{R2rXC}$$

Proof. See file Criteria.v, R2rXC_R2rXP'. □

Theorem C.74 ($\mathsf{R}K\mathsf{rXP}$ and $\mathsf{R}K\mathsf{rXC}$ are equivalent)**.**

$$\mathsf{R}K\mathsf{rXP} \iff \mathsf{R}K\mathsf{rXC}$$

Proof. Analogous to Theorem C.73. □

C.4 Relational Hyperproperty-Based Criteria

C.4.1 Robust Relational Hyperproperty Preservation

Definition C.75 (Robust Relational Hyperproperty Preservation (RrHP))**.**

$$\mathsf{RrHP}: \quad \forall R \in 2^{(\mathsf{Behavs}^\omega)}.\ \forall \mathsf{P}_1, .., \mathsf{P}_\mathsf{K}, ...\ (\forall \mathsf{C_S}.\ (\mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_1\right]\right), .., \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_\mathsf{K}\right]\right), ..) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_1\downarrow\right]\right), .., \mathsf{Behav}\left(\mathbf{C_S}\left[\mathsf{P}_\mathsf{K}\downarrow\right]\right), ..) \in R)$$

RrHP has an equivalent definition (RrHP′ below) that is also *not* property-free. We introduce this definition for technical reasons as we use it in proofs later.

Definition C.76 ((RrHP'))**.**

$$\mathsf{RrHP}': \quad \forall R \in 2^{(\mathsf{Progs} \to \mathsf{Behavs})}.\ (\forall \mathsf{C_S}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right)) \in R)$$

Theorem C.77 (RrHP and RrHP′ are equivalent)**.** Assuming the set Progs is countable,

$$\mathsf{RrHP} \iff \mathsf{RrHP}'$$

Proof. (⇒) Following the definition of RrHP′, assume $R \in 2^{(\mathsf{Progs} \to \mathsf{Behavs})}$. We need to prove:

$$(\forall \mathsf{C_S}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right)) \in R)$$

Let $\mathtt{G}$ be a bijective function from source programs to $\mathbb{N}$.[6] Define $R' \in 2^{(\mathsf{Behavs}^\omega)}$ as follows:

$$R' = \{(b_1, .., b_k, ..) \mid (\lambda \mathsf{P}. b_{\mathtt{G}(\mathsf{P})}) \in R\}$$

For $i \in \mathbb{N}$, let $\mathsf{Q_i} = \mathtt{G}^{-1}(i)$. Instantiate RrHP to R' and $\mathsf{Q}_1, .., \mathsf{Q}_\mathsf{K}, ...$ We get:

$$(\forall \mathsf{C_S}.\ (\mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{Q}_1\right]\right), .., \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{Q}_\mathsf{K}\right]\right), ..) \in R') \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{Q}_1\downarrow\right]\right), .., \mathsf{Behav}\left(\mathbf{C_S}\left[\mathsf{Q}_\mathsf{K}\downarrow\right]\right), ..) \in R')$$

Plugging in the definition of R' above, this becomes:

$$(\forall \mathsf{C_S}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{Q}_{\mathtt{G}(\mathsf{P})}\right]\right)) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{Q}_{\mathtt{G}(\mathsf{P})}\downarrow\right]\right)) \in R)$$

However, by definition, $\mathsf{Q}_{\mathtt{G}(\mathsf{P})} = \mathsf{P}$. So, the above is equal to

$$(\forall \mathsf{C_S}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right)) \in R)$$

which is what we had to prove.

(⇐) Following the definition of RrHP, assume $R \in 2^{(\mathsf{Behavs}^\omega)}$ and some infinite sequence $\mathsf{P}_1, .., \mathsf{P}_\mathsf{K}, ...$ We have to show:

$$(\forall \mathsf{C_S}.\ (\mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_1\right]\right), .., \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_\mathsf{K}\right]\right), ..) \in R) \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_1\downarrow\right]\right), .., \mathsf{Behav}\left(\mathbf{C_S}\left[\mathsf{P}_\mathsf{K}\downarrow\right]\right), ..) \in R)$$

Define $R' \in 2^{(\mathsf{Progs} \to \mathsf{Behavs})}$ as follows:

$$R' = \{f \mid (f(\mathsf{P}_1), .., f(\mathsf{P}_\mathsf{K}), ..) \in R\}$$

Instantiating RrHP′ to R', we get:

$$(\forall \mathsf{C_S}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)) \in R') \Rightarrow$$
$$(\forall \mathbf{C_T}.\ (\lambda \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right)) \in R')$$

Expanding the definition of R', this immediately reduces to what we wanted to show. □

Definition C.78 (Equivalent Characterization of RrHP (RrHC))**.**

$$\mathsf{RrHC}: \quad \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall \mathsf{P}.\ \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\downarrow\right]\right) = \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)$$

Theorem C.79 (RrHP′ and RrHC are equivalent)**.**

$$\mathsf{RrHP}' \iff \mathsf{RrHC}$$

Proof. See file Criteria.v, theorem RrHC_RrHP'. □

[6]When the source language has fewer programs than ω, the proof isn't very different.

Theorem C.80 (RrHP and RrHC)**.** Assuming the set Progs is countable,

$$\mathsf{RrHP} \iff \mathsf{RrHC}$$

Proof. Follows from Theorem C.77 and Theorem C.79. □

C.4.2 Robust K-Relational Hyperproperty Preservation

K-relational hyperproperties are relations between the behaviors of several programs. K programs satisfy a K-relational hyperproperty if and only if, when plugged into any same context, their behaviors are related.

The criteria are as expected, generalizing the intuition of hyperproperties for multiple programs.

Definition C.81 (Robust 2-Relational Hyperproperty Preservation (R2rHP))**.**

$$\begin{aligned}\mathsf{R2rHP}: \quad & \forall R \in 2^{(\mathtt{Behavs}^2)}.\ \forall \mathsf{P}_1\ \mathsf{P}_2.\ (\forall \mathsf{C_S}.\ (\mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_1]), \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_2])) \in R) \Rightarrow \\ & (\forall \mathbf{C_T}.\ (\mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]), \mathtt{Behav}\,(\mathbf{C_S}\,[\mathsf{P}_2{\downarrow}])) \in R)\end{aligned}$$

Definition C.82 (Equivalent Characterization of R2rHP (R2rHC))**.**

$$\begin{aligned}\mathsf{R2rHC}: \quad & \forall \mathsf{P}_1\ \mathsf{P}_2.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_1]) \wedge \\ & \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}_2{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}_2])\end{aligned}$$

Theorem C.83 (R2rHP and R2rHC are equivalent)**.**

$$\mathsf{R2rHP} \iff \mathsf{R2rHC}$$

Proof. See file Criteria.v, theorem R2rHC_R2rHP. □

To obtain RKrHP and RKrHC, take the definitions of R2rHP and R2rHC above and replace $\forall \mathsf{P}_1, \mathsf{P}_2$ with $\forall \mathsf{P}_1, \cdots, \mathsf{P}_K$.

Theorem C.84 (RKrHP and RKrHC are equivalent)**.**

$$\mathsf{R}K\mathsf{rHP} \iff \mathsf{R}K\mathsf{rHC}$$

Proof. Analogous to Theorem C.83. □

C.5 Comparison of Proof Obligations

We briefly compare the robust preservation of (variants of) relational hyperproperties (RrHP), relational trace properties (RrTP), and relational safety properties (RrSP, this subsection) in terms of the difficulty of back-translation proofs. For this, it is instructive to look at the property-free characterizations. In a proof of RrSP or any of its variants, we must construct a source context $\mathsf{C_S}$ that can induce a given set of *finite prefixes of traces*, one from each of the programs being related. In RrTP and its variants, this obligation becomes harder—now the constructed $\mathsf{C_S}$ must be able to induce a given set of *full traces*. In RrHP and its variants, the obligation is even harder—$\mathsf{C_S}$ must be able to induce entire behaviors (sets of traces) from each of the programs being related. Thus, the increasing strength of RrSP, RrTP and RrHP is directly reflected in their corresponding proof obligations.

Furthermore, looking just at the different variants of relational safety, we note that the number of trace prefixes the constructed context $\mathsf{C_S}$ must simultaneously induce in the source programs is exactly the arity of the corresponding relational property. Constructing $\mathsf{C_S}$ from a finite number of prefixes is much easier than constructing $\mathsf{C_S}$ from an infinite number of prefixes. Consequently, it is meaningful to define a special point in the partial order of Figure 3 that is the join of RKrSP for all finite Ks. This point is the criterion we call *Robust Finite-Relational Safety Preservation* (see Section C.3.4), or RFrSP.

Appendix D
Which of our Criteria Imply Robust Trace Equivalence Preservation?

This section extends Theorem 4.1 from §5. While RTEP is always implied by R2rHP, we show that in many cases, RTEP is a consequence of weaker relational criteria.

D.1 Relational Criteria and Robust Trace Equivalence Preservation

We start by recalling the definition of RTEP, which is an instance of R2rHP:

Definition D.1 (Robust Trace Equivalence Preservation (RTEP))**.**

$$\begin{aligned}\mathsf{RTEP}: \quad & \forall \mathsf{P}_1\ \mathsf{P}_2.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_1\right]\right) = \mathtt{Behav}\left(\mathsf{C_S}\left[\mathsf{P}_2\right]\right)) \Rightarrow \\ & (\forall \mathbf{C_T}.\ \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_1\!\downarrow\right]\right) = \mathtt{Behav}\left(\mathbf{C_S}\left[\mathsf{P}_2\!\downarrow\right]\right))\end{aligned}$$

In general, as explained in §5, RTEP is implied by R2rHP.

Theorem D.2. R2rHP $\Rightarrow$ RTEP.

Proof. The thesis immediately follows by instantiating R2rHP with the equality relation, see file Robustdef.v, theorem R2rHP_RTEP for a formal proof. ☐

Similarly, in a deterministic setting, RTEP is implied by R2rTP.

Theorem D.3. For deterministic source languages R2rTP $\Rightarrow$ RTEP.

Proof. See file Criteria.v, theorem R2rTP_RTEP. ☐

The determinism of the source language is a strong assumption though. We show that R2rTP (and even the weaker R2rXP) imply RTEP even if we weaken the determinism assumption to just determinacy, if we add two more assumptions on the target language: input totality, and "safety-like" behavior.

Definition D.4 (Determinate Languages)**.** We say a language is determinate *iff*

$$\forall W.\ \forall t_1\ t_2.\ W \rightsquigarrow t_1 \wedge W \rightsquigarrow t_1 \Rightarrow t_1\ \mathcal{R}\ t_2$$

where

$$\begin{aligned} t_1\ \mathcal{R}\ t_2 \iff & t_1 = t_2\ \vee \\ & \exists m.\ \exists e_1 e_2 \in \mathit{Input}.\ e_1 \neq e_2\ \wedge\ m\ ::\ e_1 \leq t_1\ \wedge\ m\ ::\ e_2 \leq t_2 \end{aligned}$$

Intuitively, determinacy states that a language has no internal non-determinism, or equivalently that the only source of non-determinism is the inputs from the environment.

Definition D.5 (Input Totality)**.** We say a language satisfies input totality *iff*

$$\forall W.\ \forall m.\ \forall e_1 e_2 \in \mathit{Input}.\ W \rightsquigarrow^*\ m\ ::\ e_1 \Rightarrow W \rightsquigarrow^*\ m\ ::\ e_2$$

Intuitively, input totality states that whenever a program receives an input from the environment, then it could have received any other input as well. Both determinacy [27, 42] and input totality [46, 100] are standard assumptions and are for instance satisfied by the CompCert compiler [65].

Definition D.6 ("Safety-like" semantics)**.** Given a language $\mathcal{L}$, it semantics is "safety-like" *iff*

$$\forall W. \forall t \text{ infinite}. W \not\rightsquigarrow t \Rightarrow \exists m. \exists e. W \rightsquigarrow m \wedge m :: e \leq t \wedge W \not\rightsquigarrow m :: e$$

Intuitively, any infinite trace that cannot not produced by a program can be explained as a finite prefix of that trace that *can* produced by the program, but after which the next event can no longer be produced by it. While this property is non-trivial, in §D.2 we show that any small-step semantics satisfying a particular kind of determinacy always satisfies this property.

We can now state the following theorems:

Theorem D.7. If the following assumptions hold

1) The source language is determinate.
2) The target language satisfies input totality.
3) The target language is "safety-like".

then R2rTP $\Rightarrow$ RTEP.

Proof. We give a sketch of the proof here, see file R2rTP_RTEP.v, theorem R2rTP_RTEP for a complete proof.

Two contextually equivalent programs $\mathsf{P}_1, \mathsf{P}_2$ have the same behavior in any source context. This means that for any two traces $t_1 \in \mathtt{Behav}\left(\mathsf{C_s}\left[\mathsf{P}_1\right]\right)$ and $t_2 \in \mathtt{Behav}\left(\mathsf{C_s}\left[\mathsf{P}_2\right]\right)$, since $\mathtt{Behav}\left(\mathsf{C_s}\left[\mathsf{P}_1\right]\right) = \mathtt{Behav}\left(\mathsf{C_s}\left[\mathsf{P}_2\right]\right)$ we can use the determinacy of the source language (1) to obtain that $t_1 \mathcal{R}\ t_2$, for the relation $\mathcal{R}$ used to define determinacy. This allows us to instantiate R2rTP with the relational property $\mathcal{R}$ and deduce that for an arbitrary target context $\mathbf{C_T}$, programs $\mathbf{C_T}\left[\mathsf{P}_1\downarrow\right]$ and $\mathbf{C_T}\left[\mathsf{P}_2\downarrow\right]$ can only produce traces also related by $\mathcal{R}$. Together with hypotheses 2 and 3 this is enough to show mutual inclusion of the target behaviors. We show that for an arbitrary $\mathbf{C_T}$, $\mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_1\downarrow\right]\right) \subseteq \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_2\downarrow\right]\right)$, the other inclusion is symmetric. Assume by contradiction that there exists $t_1 \in \mathtt{Behav}\left(\mathbf{C}\left[\mathsf{P}_1\downarrow\right]\right) \setminus \mathtt{Behav}\left(\mathbf{C}\left[\mathsf{P}_2\downarrow\right]\right)$. Let $m_{max} \leq t_1$ be given by hypothesis 3. Therefore there exists $t_2 \in \mathtt{Behav}\left(\mathbf{C_T}\left[\mathsf{P}_2\downarrow\right]\right)$ such that $m_{max} \leq t_2$, with $t_1 \neq t_2$ but still $t_1 \mathcal{R}\ t_2$. By determinacy, t_1 and t_2 have a common prefix m, and there exist two input events $e_1 \neq e_2$ such that $m :: e_i \leq t_i,\ i = 1, 2$. By maximality of m_{max} it must be $m \leq m_{max}$. The inequality cannot be strict, otherwise both $m :: e_1,\ m :: e_2 \leq m_{max}$. In case $m = m_{max}$ apply input totality and deduce that $\mathbf{C_T}\left[\mathsf{P}_2\downarrow\right] \rightsquigarrow^* m_{max} :: e_2$ contradicting the maximality of m_{max}. □

The next result was discussed previously as Theorem 5.1.

Theorem D.8. Under the same assumptions of Theorem D.7, R2rXP ⇒ RTEP.

Proof. See file R2rXP_RTEP.v, theorem R2rXP_RTEP for a complete proof. The argument is very similar to the one in Theorem D.7, the relation $\mathcal{R}$ is adapted to *XPref* as following.

$$x_1\ \mathcal{R}_X\ x_2 \iff x_1 \leq x_2\ \vee x_2\ \leq x_1\ \vee \\ \exists m.\ \exists e_1 e_2 \in \mathit{Input}.\ e_1 \neq e_2\ \wedge\ m\ ::\ e_1 \leq x_1\ \wedge\ m\ ::\ e_2 \leq x_2$$

$\mathcal{R}$ holds for traces produces by contextually equivalent source programs $\mathsf{P}_1, \mathsf{P}_2$, and by unfolding $\mathsf{C_S}\left[\mathsf{P_i}\downarrow\right] \rightsquigarrow^* x_i,\ i = 1, 2$, $x_1 \mathcal{R}_X\ x_2$ holds as well.
Proceed as in Theorem D.7 to show mutual inclusion of behaviors. Determinacy ensures that if $t_1 \neq t_2$ then there exist two non comparable x_1, x_2 such that $x_1 \leq t_1,\ x_2 \leq t_2$ both with m_{max} as common prefix and still $x_1 \mathcal{R}_X\ x_2$ and we can conclude with the same argument as in Theorem D.7. It is crucial to observe that considering *XPref* instead of *FinPref*, x_1, x_2 can be considered non comparable. This can be proved by case analysis on the two traces, in particular if $t_1 = m :: \circlearrowleft$ and $t_2 = m :: \varepsilon$, $x_1 = m :: \circlearrowleft$ and $x_2 = m :: \varepsilon$ are two non comparable x-prefixes still related by $\mathcal{R}_X$ but all finite prefixes will be comparable, so that input totality is not useful to reach a contradiction. □

While under the rather liberal condition above R2rSP does **not** imply RTEP, it does imply RTEP in the very special case that target programs cannot produce any silently diverging traces, for instance because in the target language is terminating. This is a technical result that we use in a later proof (Theorem E.10).

Theorem D.9. Under the following assumptions:

1) The source language is determinate.
2) The target language satisfies input totality.
3) The target language is "safety-like".
4) Target programs cannot produce silently diverging traces.

then R2rSP ⇒ RTEP.

Proof. See file R2rSP_RTEP.v, theorem R2rSP_RTEP for a complete direct proof. Here we just highlight that R2rXP and R2rSP are equivalent under the very strong hypothesis 4, so we can simply apply Theorem D.8 above. □

While assumption (4) above is very strong, it does hold for strictly terminating languages, and in all other cases one can use Theorem D.8.

D.2 Safety-Like Small-Step Semantics

In this section we state and prove the property that many small-step semantics have the previous "safety-like" behavior, in the sense that we can determine whether an infinite trace cannot be produced by a program after a finite number of steps.

First, we state our semantic model and its basic constituents.

Definition D.10 (Small-step semantics)**.** A small-step semantics is defined in terms of the following abstract components:

- Program states are represented by *configurations*, c.
- An *initial relation* characterizes initial program states.
- A *step relation*, $c \xrightarrow{e} c'$ between pairs of states, producing an event. Its reflexive and transitive closure is denoted $\xrightarrow{e_1 \cdots e_n}{}^*$.
- A well-founded *order relation* on elements of a type of "measures."

Events can be either *visible* or *silent*. A configuration is *stuck* when there is no configuration it can step to; it can *loop silently* if there is an infinite sequence of silent steps starting from it.

A small-step semantics relates program configurations and the traces they produce; the relation is moreover parameterized by an element of the type of measures. In our trace model, there are four possible cases, starting from a configuration c:

- If c is stuck, the semantics produces the terminating trace ε with some associated information ε.
- If c can loop silently, the semantics produces the silently diverging trace $\circlearrowleft$.
- If c can silently step $c \xrightarrow{\varnothing}^{*} c'$ to a c' while decreasing its ordering measure with respect to c, the semantics recurses on c'.
- If c can step with some visible events $c \xrightarrow{m}^{*} c'$, the semantics emits m and recurses on c'.

The addition of the well-founded order relation between measures is used to avoid the usual problem of infinite stuttering on silent events, which is properly captured by silent divergence. Between two visible events there must mediate a finite number of silent events. This requirement is enforced by having the ordered measure decrease when silent steps are taken (there are no restrictions on ordering between states connected by visible events). A similar device is used, for example, in the CompCert verified compiler.

The final result holds for a wide class of reasonable languages. The following determinacy condition is sufficient to prove the result.

Definition D.11 (Weak determinacy)**.** Two program configurations are related if they produce the same traces under the semantics; we write $c_1 R c_2$ for this.

Under weak determinacy, if a pair of states is related and each element of this initial pair steps to another state producing the same sequence of events, the pair of final states is also related:

$$\forall c_1.\forall c_1'.\forall m.\forall c_2.\forall c_2'.c_1 R c_1' \Rightarrow c_1 \xrightarrow{m}^{*} c_2 \Rightarrow c_1' \xrightarrow{m}^{*} c_2' \Rightarrow c_2 R c_2'$$

Thus stated, the "safety-like" quality of small-step semantics follows easily.

Theorem D.12. Assuming weak determinacy holds, all small-step semantics (that can be encoded by the scheme of Definition D.10) are "safety-like."

Proof. See file SemanticsSafetyLike.v, theorem tgt_sem. □

Appendix E
Separation Results

The implications represented by arrows in Figure 3 are strict, that is, the two criteria linked by an arrow are *not* equivalent. This section justifies these separation results by giving, for each of them, counterexample compilation chains that satisfy the criterion occurring lower in the diagram (pointed to by the arrow), but not the upper one. Finally, in §E.5 we prove that RTEP does not imply even the weakest criteria in our diagram (RSP and RDP), even when also assuming compiler correctness (TP, SCC, and CCC).

E.1 RSP and RDP Do Not Imply RTP

In this section, we show that the robust preservation of *either* all safety properties (Lemma E.1) *or* of all dense properties (Lemma E.2) is not enough to guarantee the robust preservation of all trace properties. (Note that, as a corollary to the decomposition result in Theorem B.7, a compiler that preserves all safety properties *and* all dense properties preserves all properties.) The two compilation chains in this section have been formalized in the Coq; see file Separation.v for more details. This section expands upon the description from §2.2 (for safety properties) and §B.3 (for dense properties).

Take an arbitrary language $\mathcal{L}$ described by a small-step semantics. Assume it is possible to write a non-terminating program in $\mathcal{L}$, e.g., a program that produces some infinite trace. Assume moreover that such a program is independent from the context with which it is linked (for instance, it is already whole). To keep things concrete, we consider a standard *while* language as our $\mathcal{L}$ and the following non-terminating program P_Ω, where n∈ℕ:

```
while (true) {
  output(n);
}
```

Next, define a language transformer $\phi(\mathcal{L})$, which produces a new language that is identical to $\mathcal{L}$, except that it bounds program executions by a certain number of steps (its *"fuel"*). In particular:

- If C is a context in $\mathcal{L}$, then for every $n \in \mathbb{N}$, (n, C) is a context in $\phi(\mathcal{L})$ with fuel n.
- Plugging in $\phi(\mathcal{L})$ is defined by $(n, C)[P]_{\phi(\mathcal{L})} \equiv (n, C[P]_{\mathcal{L}})$. Subscripts will be omitted when doing so introduces no ambiguities.
- The semantics of $\phi(\mathcal{L})$ extends the semantics of $\mathcal{L}$ as follows. If the amount fuel is 0, no steps are allowed. Otherwise, every time a step would be taken in $\mathcal{L}$, the same step is taken in $\phi(\mathcal{L})$ and the amount of fuel is decremented by one.

Lemma E.1. RSP $\not\Rightarrow$ RDP

Proof. Take $\phi(\mathcal{L})$ as source language, $\mathcal{L}$ as target, and the compiler to be the projection of contexts of $\phi(\mathcal{L})$ on their second component. We are going to show that all safety properties that are robustly satisfied in the source are also robustly satisfied in the target, but not all dense properties are preserved.

Let $S \in \mathit{Safety}$. Assume that all safety properties are robustly preserved, i.e., that for every program P, every source context (n, C) and every trace t,

$$(n, C[P]) \rightsquigarrow t \Rightarrow t \in S$$

In addition, assume for contradiction that there exists some target context C' and trace t' such that

$$C'[P\downarrow] \rightsquigarrow t' \wedge t' \notin S$$

where $P\downarrow = P$. By definition of safety, there exists $m \leq t'$ such that every continuation t'' of m violates the property,

$$\forall t''.\ m \leq t'' \Rightarrow t'' \notin S$$

Consider the source context $(|m|, C')$ where $|m|$ is the length of m. Denote by t_m the trace that contains the events of m followed by a termination marker. Since $m \leq t_m$ we have that $t_m \notin S$. However, $(|m|, C') \rightsquigarrow t_m$, which implies that $t_m \in S$, a contradiction.

Next, we produce a dense property that is not robustly preserved by this compiler. Consider

$$L = \{t |\ t \textit{ is finite} \vee t = \mathit{output}(42)^\omega\}$$

Observe that L is a dense property as it includes all finite traces. Since programs in the source can produce only finite traces, these will be in L. In the target, however, the program $P = P\downarrow$

```
while (true) {
  output(41);
}
```

is no longer forced to stop after a finite number of steps, and produces an infinite trace different from $\mathit{output}(42)^\omega$. □

Lemma E.2. RDP $\not\Rightarrow$ RSP

Proof. Take $\mathcal{L}$ as source language, $\phi(\mathcal{L})$ as target, and the compiler to be the identity. We are going to show that all dense properties are robustly preserved but not all safety properties are robustly preserved.
Let L be a dense property. Every trace t produced by a program in the target is finite, so that by definition of *Dense*, $t \in L$. Consider now the following property:

$$S = \{\mathit{output}(42)^\omega\}$$

S is a safety property because for every trace $t \notin S$, t starts with a number (possibly zero) of $\mathit{output}(42)$ events, followed either by some other event $e \neq \mathit{output}(42)$ or terminated by ε for some ε, i.e.,

$$\mathit{output}(42)^n; e \leq t \vee \mathit{output}(42)^n; \varepsilon \leq t$$

Here, every continuation of $\mathit{output}(42)^n; e$ is different from $\mathit{output}(42)^\omega$, and different from every finite trace. Finally, consider the program $P = P\downarrow$

```
while (true) {
  output(42);
}
```

which, in the source, produces the infinite trace $\mathit{output}(42)^\omega \in S$ regardless of the context. In the target, only traces of length k can be produced, which are not in S. □

Theorem E.3. Neither RSP nor RDP separately imply RTP.

Proof. Follows directly from Lemma E.1 and Lemma E.2 and Theorem B.7. □

In our previous discussion, Theorem 2.1 corresponds to the non-trivial direction of Theorem E.3.

E.2 RTP Does Not Imply RTINIP

In this section we prove Theorem 3.1 from §3.4:

Theorem E.4. There is a compiler that satisfies RTP but not RTINIP.

Proof. We consider the following source language that works over integers; has traces with exactly two events, one input followed by one output; and has exactly one program P:

```
x = input;
y = f();
output y;
```

where f() is a pure function provided by the context. The target language is the same as the source language, except the context has the ability to directly read the program **P**'s private variables, like **x**. The compiler is the identity.

This compiler satisfies Robust Trace Property Preservation (RTP). The reason is that in the source, program P can generate every possible trace given an appropriate source context: to generate the trace $t = [\texttt{input}\ i, \texttt{output}\ o]$, take the context whose f() returns the integer o. Basically, this source context simply guesses the output values from the single trace t.

However, this compiler does not satisfy RTINIP. If we take the input to be private and the output to be public, then for our language *TINI* is equivalent to the following 2-hypersafety property H:

$$H = \{b \mid \forall i_1, o_1, i_2, o_2.\ [\texttt{input}\ i_1, \texttt{output}\ o_1] \in b \wedge [\texttt{input}\ i_2, \texttt{output}\ o_2] \in b \Rightarrow o_1 = o_2\}$$

In the source, any f() defined by the context must be a constant function. This is because the context is purely functional and has no access to the input stream or P's local variables, hence f()'s result cannot depend on any changeable quantity. If f() returns a constant c and we look at two source traces $[\texttt{input}\ i_1, \texttt{output}\ o_1]$ and $[\texttt{input}\ i_2, \texttt{output}\ o_2]$, then $o_1 = o_2 = c$ and thus the source program satisfies the hyperproperty H.

However, in the target, it is possible to write a context function **f()** that breaks H: **f(){return(x);}**. This function reads **P**'s local variable **x** (which the target context is capable of accessing) and returns its value. Hence, with this context, the program's outputs depend on its inputs. In particular, $[\texttt{input}\ 1, \texttt{output}\ 1]$ and $[\texttt{input}\ 2, \texttt{output}\ 2]$ are two traces where the output vary, so this context (and consequently the compilation chain) breaks the 2-hypersafety property H. □

E.3 R*K*HSP Does Not Imply R(*K*+*1*)HSP

In this section, we prove Theorem 3.2 from §3.4 by exhibiting a counterexample compiler, parametric in K, that has Robust K-Hypersafety Preservation, R*K*HSP, but not Robust $(K+1)$-Hypersafety Preservation, R(*K*+*1*)HSP, for an arbitrary K.

Our source language is a standard *while* language with read and write events to standard I/O. It has traces of length exactly two: one input (read) event followed by one output (write) event. This language's inputs are always in the natural range $[1, \ldots, K+1]$, while its internal values and outputs are real numbers. The language has exactly one program, P, shown below. The context provides the functions $\mathsf{f_1}(), \ldots \mathsf{f_K}()$.

```
x = read();
switch (x) {
  case x = i where 1 <= i <= K:
     y = x + (sum {f_j() | 1 <= j <= K && i <> j});
     break;
  case x = K + 1:
     y = K + f_1();
}
write (y)
```

Our target language is identical to the source, with the exception that the context now has access to the private state of the program, so it can read the local variable **x**. In the source language, the context lacks this capability.

The compiler under consideration, $\cdot\downarrow$, is the identity, i.e., it maps P to its identical counterpart **P**.

Lemma E.5. The compiler $\cdot\downarrow$ satisfies R*K*HSP.

Proof. We prove this by showing that for any finite K-set of prefixes $\{[\texttt{read}\ a_1, \texttt{write}\ b_1], ..., [\texttt{read}\ a_K, \texttt{write}\ b_K]\}$ that the program $\mathbf{C_T}\,[\mathbf{P}]$ can produce (for some target context $\mathbf{C_T}$), there is some source context $\mathsf{C_S}$ that produces these K prefixes as well. This property immediately implies that the compiler has R*K*HSP.

To prove this property, note that if all K prefixes $[\texttt{read}\ a_1, \texttt{write}\ b_1], \ldots, [\texttt{read}\ a_K, \texttt{write}\ b_K]$ can be produced by the target, then, since the target is still deterministic, we must have: $\forall i, \forall j, a_i = a_j \implies b_i = b_j$. Thus, we can assume without loss of generality that all a_is are distinct. It follows that $I = \{a_1, \ldots, a_K\}$ is a K-subset of $\{1, \ldots, K+1\}$, so I must be missing exactly one element in the set of allowed inputs $\{1, \ldots, K+1\}$.

We proceed by case analysis on the missing element:

- The missing element is $K+1$. We can assume without loss of generality (by reordering I if needed) that $a_i = i$, and therefore $I = \{a_1, \ldots, a_K\} = \{1, \ldots, K\}$. We now set up a system of linear equations whose solution characterizes the source context $\mathsf{C_S}$. Let x_i be a variable that represents the value of $\mathsf{f_i}()$ (note that $\mathsf{f_i}()$ must be a constant function in the source context). Then, we formulate the system the equations:
$$\begin{aligned} x_2 + \cdots + x_{K-1} + x_K &= b_1 - a_1 \\ x_1 + \quad + \cdots + x_{K-1} + x_K &= b_2 - a_2 \\ &\ldots \\ x_1 + x_2 + \cdots + x_{K-1} + \quad &= b_K - a_K \end{aligned}$$
This system has a unique solution. To see this, first add all the equations. This yields:
$$(K-1)(x_1 + \cdots + x_K) = (b_1 + \cdots + b_K) - (a_1 + \cdots + a_K).$$
This yields an equation $x_1 + \cdots + x_K = c$ for some c. Subtracting the first equation in the system (corresponding to $b_1 - a_1$) from this sum gives us x_1. Subtracting the second equation gives x_2, and so on. Hence we obtain a value x_i that each $\mathsf{f_i}()$ must return in the source in order to produce the required outcome. This is the required $\mathsf{C_S}$.
- The missing element is 1. Then, $I = \{a_1, \ldots, a_K\} = \{2, \ldots, K+1\}$. Assume without loss of generality that $a_1 = 2, \ldots, a_K = K+1$. Then, as before, we get the equations:
$$\begin{aligned} x_1 + \quad + \cdots + x_{K-1} + x_K &= b_1 - a_1 \\ &\ldots \\ x_1 + x_2 + \cdots + x_{K-1} + \quad &= b_{K-1} - a_{K-1} \\ x_1 + \quad\quad\quad\quad\quad &= b_K - a_K \end{aligned}$$
This set of equations also has a solution. First, the last equation directly gives x_1. Now subtract the last equation from all the previous $K-1$ equations. This yields exactly $K-1$ *cyclic* equations in $K-1$ variables $x_2, \ldots, x_K$. These can be solved exactly as in the previous case.
- The missing element is between 2 and K (both inclusive). Without loss of generality, assume that it is K. Then, $I = \{a_1, \ldots, a_K\} = \{1, \ldots, K-1, K+1\}$. Again, assume that $a_1 = 1, \ldots, a_{K-1} = K-1$, and $a_K = K+1$. Then, we get the equations:
$$\begin{aligned} + x_2 + \cdots + x_{K-2} + x_{K-1} + x_K &= b_1 - a_1 \\ x_1 + \quad + \cdots + x_{K-2} + x_{K-1} + x_K &= b_2 - a_2 \\ &\ldots \\ x_1 + x_2 + \cdots + x_{k-2} + \quad\quad + x_K &= b_{K-1} - a_{K-1} \\ x_1 + \quad\quad\quad\quad\quad\quad\quad &= b_K - a_K \end{aligned}$$
Solving these equations is also easy. x_1 is determined by the last equation. Adding the first and last equations gives the value of $x_1 + \cdots + x_K$. Subtracting the remaining equations from this one, one by one, yields $x_2, \ldots, x_{K-1}$. Then, x_K follows from the first equation.

□

Lemma E.6. The compiler $\cdot\!\downarrow$ is not $\mathsf{R}(K{+}1)\mathsf{HSP}$.

Proof. We construct a concrete $K+1$ prefix set S and a $\mathbf{C_T}$ such $\mathbf{C_T}\,[\mathsf{P}\!\downarrow]$ can produce all prefixes of S, but no $\mathsf{C_S}\,[\mathsf{P}]$) can do the same. The proof relies on the fact that, in the source, each of $\mathsf{f_1}(), \ldots, \mathsf{f_K}()$ must be a constant function, so we can have $k+1$ inconsistent equations for these k constants. In the target, the equations are not required to be constant since any $\mathbf{f_i}$ can return a value based on the private input $\mathbf{x}$.

Let c be the constant $K-1$. Consider now the following falsifying prefix set:
$$\begin{aligned} S = \{&[1, 1+c], \\ &[2, 2+c], \\ &\ldots \\ &[K, K+c], \\ &[K+1, K+1]\} \end{aligned}$$
So, for inputs $x = 1, \ldots, K$, the output is the input value plus c (i.e., $K-1$), but for input $K+1$ the output is the input value $K+1$ itself.

The following target context $\mathbf{C_T}$ generates this prefix set S:

$$\begin{aligned}
\mathbf{f_1}() &= \mathbf{if\ P.x = K+1\ then\ 0\ else\ 1}\\
\mathbf{f_2}() &= \mathbf{1}\\
&\cdots\\
\mathbf{f_K}() &= \mathbf{1}
\end{aligned}$$

where $\mathbf{P.x}$ is the private value $\mathbf{x}$ of $\mathbf{P}$.

The function $\mathbf{f_1}()$ returns 1 except when the private input $\mathbf{x}$ is $K+1$, when it returns 0. It is easy to see that for inputs $\mathbf{x}$ between 1 and K the output of $\mathbf{P}$ is exactly $x+(K-1)$, whereas for input $x = K+1$ the output of $\mathbf{P}$ is $K+1+\mathbf{f_1}() = K+1+0 = K+1$. Hence, $\mathbf{C_T}[\mathsf{P}\downarrow]$ generates the entire prefix set S.

On the other hand, $\mathsf{C_S}[\mathsf{P}]$ cannot generate all prefixes of S for any $\mathsf{C_S}$. To see this, suppose that there exists some $\mathsf{C_S}[\mathsf{P}]$ can actually generate all prefixes of S. Let $\mathsf{f_i}() = x_i$. We get the equations:

$$\begin{aligned}
x_2 + \cdots + x_{K-1} + x_K &= (1+K-1)-1 = K-1\\
x_1 + \quad + \cdots + x_{K-1} + x_K &= (2+K-1)-2 = K-1\\
&\cdots\\
x_1 + x_2 + \cdots + x_{K-1} + \quad &= (K+K-1)-K = K-1\\
x_1 + \quad &= (K+1)-(K+1) = 0
\end{aligned}$$

However, these equations are inconsistent. The first K equations (which are cyclic) force that $x_1 = \cdots = x_K = 1$, while the last equation requires $x_1 = 0$. This contradicts our hypothesis on the existence of $\mathsf{C_S}[\mathsf{P}]$. □

Theorem E.7. For any K, there is a compiler that satisfies RKHSP but not R(K+1)HSP.

Proof. It follows from Lemma E.5 and Lemma E.6 that $\cdot\downarrow$ is RKHSP but not R(K+1)HSP. □

E.4 Robust Non-Relational Property Preservation Does Not Imply Robust Relational Property Preservation

In this section we prove that, as stated in Theorem 4.1 from 4.4, no non-relational preservation criterion implies any relational preservation criterion. We do this constructively, by showing a source language, a target language, and a compiler between them such that:

- The compiler satisfies the *strongest* non-relational preservation criterion (Robust Hyperproperty Preservation, RHP).
- The compiler does not satisfy the *weakest* relational preservation criterion (Robust 2-Relational Safety Preservation, R2rSP). Because the languages will satisfy the conditions that make R2rSP imply Robust Trace Equivalence Preservation (RTEP), we shall simply show that the compiler does not satisfy RTEP, and use the result from the next .

The source language we shall consider is a standard *while* language with read and write events to standard I/O. It has traces comprising exactly two events: one input (read) followed by one output (write). This language works over integers (*not* natural numbers) and has exactly two programs, P1 and P2, shown below, that are only different in that the second adds some dead code to the first:

```
P1:
x = read();
y = f();
write (x + y)

P2:
x = read();
y = f();
... some dead code here ...
write (x + y)
```

Here, $\mathsf{f}()$ is a function provided by the context.

The target language is the same, but additionally allows the context to read the compiled code as a value.

The compiler under consideration, $\cdot\downarrow$, is the identity.

Lemma E.8. The compiler $\cdot\downarrow$ satisfies RHP.

Proof. We need to show that $\forall \mathbf{C_T}\ \mathsf{P}.\exists \mathsf{C_S}.\texttt{Behav}(\mathsf{C_S}[\mathsf{P}]) = \texttt{Behav}(\mathbf{C_T}[\mathsf{P}\downarrow])$. For this, pick a $\mathbf{C_T}$ and a P. Note that $\mathsf{P}\downarrow = \mathsf{P}$ by definition of the compiler. To produce $\mathsf{C_S}$, modify the $\mathbf{C_T}$ so that wherever $\mathbf{C_T}$ reads the code of $\mathsf{P}\downarrow$, P (which is the same as $\mathsf{P}\downarrow$) is hard-coded in $\mathsf{C_S}$ instead.

It is trivial to see that $\mathsf{C_S}[\mathsf{P}]$ and $\mathbf{C_T}[\mathsf{P}\downarrow]$ have exactly the same behaviors. □

Lemma E.9. The compiler $\cdot\downarrow$ does not satisfy RTEP.

Proof. Since $\mathsf{P1}$ and $\mathsf{P2}$ differ only in the presence of some dead code, which a source context cannot examine, it is trivially the case that $\forall \mathsf{C_S}.\texttt{Behav}(\mathsf{C_S}[\mathsf{P1}]) = \texttt{Behav}(\mathsf{C_S}[\mathsf{P2}])$.

On the other hand, we can construct a target context $\mathbf{C_T}$ whose $\mathbf{f}()$ checks whether the compiled code is $\mathbf{P1}$ or $\mathbf{P2}$, and returns either 0 or 1, respectively. Then, $\mathbf{C_T}[\mathbf{P1}]$ produces $[\texttt{read}\ 0, \texttt{write}\ 0]$ as a trace, while $\mathbf{C_T}[\mathbf{P2}]$ does not have this trace. Hence, the compiler is not RTEP. □

Theorem E.10. There exists a compiler between two languages that satisfy the assumptions of Theorem D.9 that has RHP, but not RTEP.

Proof. Both language clearly satisfy determinacy and input totality. Furthermore, given an infinite trace t not produced by some whole program $\mathbf{W}$, the prefix needed is either the trace produced by the program, or the empty prefix.

The theorem follows immediately from Lemma E.8 and Lemma E.9. □

Theorem E.11. There exists a compiler that satisfies RHP but not R2rSP.

Proof. Follows directly from Theorem E.10. □

The Full Story More generally, if we take any source language in which the context cannot examine the code and compile it to a target language that is similar, but where the context can examine the code as an added capability, then the identity compiler satisfies every non-relational criterion including RHP, since *for a single program*, the target context's additional ability to observe the code is inconsequential. More formally, non-relational preservation criteria, including RHP, allow the simulating source context $\mathsf{C_S}$ to depend on the compiled program P, so that program code can be hard-coded into $\mathsf{C_S}$ wherever $\mathbf{C_T}$ examines the code. However, it is extremely unlikely that this compiler satisfies any relational preservation criterion since the target context can branch on the program being executed and provide different values to each of the programs.

E.5 RTEP Does Not Imply RSP or RDP

In this section, we give a counterexample compilation chain for showing a generalization of Theorem 5.2 from §5.

First, we recall three notions of correctness, from §2.1. For CCC we explicitly mention the condition that $\mathsf{C_S}$ should be linkable with P, which is a technical hypothesis that we omitted in the main paper text.

Definition E.12 (Backward Simulation (TP))**.**

$$\mathsf{TP}: \quad \forall \mathsf{W}.\ \mathsf{W}\downarrow \rightsquigarrow t \Rightarrow \mathsf{W} \rightsquigarrow t$$

Definition E.13 (Separate Compiler Correctness (SCC))**.**

$$\mathsf{SCC}: \quad \forall \mathsf{P}.\ \forall \mathsf{C_S}.\ \forall t.\ \mathsf{C_S}\downarrow[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

Definition E.14 (Compositional Compiler Correctness (CCC))**.**

$$\mathsf{CCC}: \forall \mathsf{P}\ \mathbf{C_T}\ \mathsf{C_S}\ t.\ \mathbf{C_T}{\approx}\mathsf{C_S} \wedge \mathsf{C_S} \text{ is linkable with } \mathsf{P} \wedge \mathbf{C_T}[\mathsf{P}\downarrow] \rightsquigarrow t \Rightarrow \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t$$

Theorem E.15. There exists a compiler between two deterministic languages that satisfies RTEP, TP, SCC, and CCC but that satisfies neither RSP nor RDP.

Trace Model We consider languages where exactly one event is produced containing a natural number that represents the final result of the computation. Allowed traces are final result singletons and silent divergence.

Source Language A source language program consists of one function obtaining one input from the context (a natural number or a boolean), perfoming basic computations on it, and returning a natural number as a result.

$$\begin{aligned} \textit{Program}\ \mathsf{P} &::= \mathsf{f(x : Nat)} \mapsto \mathsf{e} \mid \mathsf{f(x : Bool)} \mapsto \mathsf{e} \\ \textit{Expression}\ \mathsf{e} &::= \mathsf{if\ x\ then\ e\ else\ e} \mid \mathsf{if\ x < n\ then\ e\ else\ e} \mid \mathsf{n} \mid \mathsf{f(e)} \\ \textit{Context}\ \mathsf{C} &::= \mathsf{f(n)} \mid \mathsf{f(b)} \end{aligned}$$

In this example, the composition of a program and a context of incompatible types is statically disallowed. We do not consider them in the criteria we prove, and implicitely assume that the criteria only apply when the operations $\cdot[\cdot]$ and $\cdot[\cdot]$ are defined. See Section G for a example where we take into account the fact that not all components are linkable by using simple variants of our criteria.

Target Language The target language is identical to the source, but it only admits natural numbers as inputs.

$$\textit{Program } \mathbf{P} ::= \mathbf{f}(\mathbf{x} : \mathsf{Nat}) \mapsto \mathbf{e}$$
$$\textit{Expression } \mathbf{e} ::= \mathbf{if\ x < n\ then\ e\ else\ e} \mid \mathbf{n} \mid \mathbf{f}(\mathbf{e})$$
$$\textit{Context } \mathbf{C} ::= \mathbf{f}(\mathbf{n})$$

Compiler We first define the compilation of expressions:

$$(\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)\big\downarrow = \mathbf{if\ x < 1\ then}\ \mathsf{e}_1\!\downarrow \mathbf{else}\ \mathsf{e}_2\!\downarrow$$
$$(\mathsf{if}\ \mathsf{x} < \mathsf{n}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)\big\downarrow = \mathbf{if\ x < n\ then}\ \mathsf{e}_1\!\downarrow \mathbf{else}\ \mathsf{e}_2\!\downarrow$$
$$\mathsf{n}\!\downarrow = \mathbf{n}$$
$$\mathsf{true}\!\downarrow = \mathbf{0}$$
$$\mathsf{false}\!\downarrow = \mathbf{1}$$
$$\mathsf{f}(\mathsf{e})\big\downarrow = \mathbf{f}(\mathsf{e}\!\downarrow)$$

Then, we define the compilation of partial programs:

$$(\mathsf{f}(\mathsf{x} : \mathsf{Nat}) \mapsto \mathsf{e})\big\downarrow = \mathbf{f}(\mathbf{x} : \mathsf{Nat}) \mapsto \mathsf{e}\!\downarrow$$
$$(\mathsf{f}(\mathsf{x} : \mathsf{Bool}) \mapsto \mathsf{e})\big\downarrow = \mathbf{f}(\mathbf{x} : \mathsf{Nat}) \mapsto \mathbf{if\ x < 2\ then}\ \mathsf{e}\!\downarrow \mathbf{else\ if\ x < 3\ then\ f(2)\ else\ 42}$$

We can trivially extend this compiler to contexts, and whole programs as well.

Lemma E.16. $\cdot\!\downarrow$ satisfies TP, SCC, and CCC.

Proof. We start by noting that for this compilation chain, TP is implied by SCC. Indeed, let W be a source whole program. Then, by definition, there exists C and P such that $\mathsf{W} = \mathsf{C}[\mathsf{P}]$. Furthermore, $\mathsf{W}\!\downarrow = \mathsf{C}\!\downarrow[\mathsf{P}\!\downarrow]$ by definition of the compiler. Hence, to show TP, one can simply apply SCC to this decomposition of W into two components.

We then show that for this compilation chain CCC is also implied by SCC. To show this, we need to explicit the instantiation of $\approx$ and of the notion of linkable components in the definition of CCC:

- $\approx$ is defined by $\mathbf{C_T} \approx \mathsf{C_S}$ iff $\mathbf{C_T} = \mathsf{C_S}\!\downarrow$.
- P and $\mathsf{C_S}$ are linkable iff they agree on the argument type of the program's function.

Then, by unfolding these definitions in CCC, and substituting equalities, we obtain exactly SCC.

So all that is left to prove is SCC. Suppose $\mathsf{C}\!\downarrow[\mathsf{P}\!\downarrow] \rightsquigarrow t$. We will show that $\mathsf{C}[\mathsf{P}] \rightsquigarrow t$, proceeding by case analysis on P:

- $\mathsf{P} = \mathsf{f}(\mathsf{x} : \mathsf{Nat}) \mapsto \mathsf{e}$. By induction on e, we can prove that $\mathsf{e}\!\downarrow = \mathsf{e}$: indeed, because the function argument x must be a natural number, and because x is the only "variable" ever in scope, e cannot contain a subexpression $\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2$. Otherwise, it would mean that x is a boolean, this would be a contradiction.
- $\mathsf{P} = \mathsf{f}(\mathsf{x} : \mathsf{Bool}) \mapsto \mathsf{e}$. In this case, $\mathsf{P}\!\downarrow = \mathbf{f}(\mathbf{x} : \mathsf{Nat}) \mapsto \mathbf{if\ x < 2\ then}\ \mathsf{e}\!\downarrow \mathbf{else\ if\ x < 3\ then\ f(2)\ else\ 42}$. Since the target context is a compiled context, this steps to the expression $\mathsf{e}\!\downarrow$. Now, f cannot be called recursively in e, because expressions are natural numbers, but f expects a boolean. Hence, the same is true in the compiled version. Now, we can conclude by proving the thesis by induction on e. In particular, note that $(\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2)\big\downarrow = \mathbf{if\ x < 1\ then}\ \mathsf{e}_1\!\downarrow \mathbf{else}\ \mathsf{e}_2\!\downarrow$, that if $\mathsf{x} = \mathsf{true}$ then $\mathbf{x < 1}$ is true, and if $\mathsf{x} = \mathsf{false}$ then $\mathbf{x < 1}$ evaluates to false, where $\mathbf{x}$ is the compilation of x.

$\square$

Lemma E.17. $\cdot\!\downarrow$ satisfies RTEP.

Proof. We prove the contrapositive form of the statement of RTEP.

Let P_1 and P_2 be two programs and suppose their compilations are not observationally equivalent; let $\mathbf{C} = \mathbf{f}(\mathbf{n})$ be the distinguishing context. We consider three cases:

- P_1 and P_2 both expect a natural number. Take $\mathsf{C} = \mathsf{f}(\mathsf{n})$. Since the compiler is the identity for programs that expect a natural number, we obtain the desired result.
- P_1 and P_2 both expect a boolean. If $\mathsf{n} = 0$ or $\mathsf{n} = 1$, then by taking $\mathsf{C} = \mathsf{f}(\mathsf{true})$ in the first case, and $\mathsf{C} = \mathsf{f}(\mathsf{false})$ in the second case, we obtain the desired result by compiler correctness. Otherwise, this case is discharged by contradiction: $\mathbf{C}[\mathsf{P}_1\!\downarrow]$ and $\mathbf{C}[\mathsf{P}_2\!\downarrow]$ have the same behavior, by definition of the compiler.
- P_1 and P_2 have different input types. This is a case we do not consider, because then the source context cannot be linked with both programs.

$\square$

Lemma E.18. $\cdot\!\downarrow$ does not satisfy RSP.

Proof. Consider the program $\mathsf{P} = \mathsf{f(x : Bool) \mapsto 1}$. This program satisfies the safety property "never outputs 42," but its compilation does not (it violates it with input $\mathbf{3}$, for instance). ☐

Lemma E.19. $\cdot\downarrow$ does not satisfy RDP.

Proof. Consider the same program $\mathsf{P} = \mathsf{f(x : Bool) \mapsto 1}$. This program satisfies the safety property "never silently diverge," but its compilation does not (it violates it with input $\mathbf{2}$). ☐

The proof of Theorem E.15 is immediate from the previous lemmas.

We now extend this compilation chain to show that RTEP does not imply RTINIP either. We introduce a new command at the target level, **leak**, to model information leakage. The semantics of this new instruction is simple: **leak** reduces to a non-deterministically chosen natural number. Hence, this command can model looking-up the value of a secret inside memory, and outputting it publicly. All outputs are considered public.

Then, we also modify the compilation of partial programs having a boolean argument:

$$(\mathsf{f(x : Bool) \mapsto e})\downarrow = \mathbf{f(x : Nat) \mapsto if\ x < 2\ then\ } \mathsf{e}\!\downarrow \mathbf{\ else\ if\ x < 3\ then\ f(2)\ else\ if\ x < 4\ then\ 42\ else\ leak()}$$

It is easy to show that the previous lemmas still hold. Now is left to show that the compiler does not satisfy RTINIP.

- Every source whole program trivially satisfies termination-insenstive noninterference, because whole source programs are completely deterministic.
- Now, consider a source program P with a boolean argument, and the target context $\mathbf{C_T} = \mathbf{f(3)}$. Then, $\mathbf{C_T}[\mathsf{P}\!\downarrow] \rightsquigarrow 0$, and $\mathbf{C_T}[\mathsf{P}\!\downarrow] \rightsquigarrow 1$. The public inputs are identical, but not the public output. Hence the compiler does not satisfy RTINIP.

Appendix F
Context Composition by Full Reflection or Internal Nondeterminism in the Source Language

In this section we prove the theorems from §4.5, where we analyzed how certain features of the source language can greatly influence the partial order in Figure 3. In Section F.1 we assume source programs can completely examine their own code, a mechanism that is sometimes called *full reflection*. First of all we need to introduce relational subset-closed hyperproperties, the classes of relational hyperproperties that are downward-closed in each of its arguments. Then in Section F.2, we assume it is possible to build a source context C whose behaviors approximate two given source contexts $\mathsf{C_1}$ and $\mathsf{C_2}$. This is the case when an operator for internal nondeterministic choice is available.

Definition F.1 (*2rSCH*)**.** Given $R \in 2^{(2^{Trace} \times\ 2^{Trace})}$

$$R \in \mathit{2rSCH} \iff \forall (b_1, b_2) \in R. \forall s_1 \subseteq b_1,\ \ s_2 \subseteq b_2. (s_1, s_2) \in R$$

Definition F.2 (R2rSCHP)**.**

$$\begin{aligned}\mathsf{R2rSCHP}: \quad & \forall \mathsf{P_1 P_2}\ R \in \mathit{2rSCH}\ \forall \mathsf{C_s}. (\mathtt{Behav}(\mathsf{C_s}[\mathsf{P_1}]), \mathtt{Behav}(\mathsf{C_s}[\mathsf{P_2}])) \in R \\ & \qquad\qquad \forall \mathbf{C_t}. (\mathtt{Behav}(\mathbf{C_T}[\mathsf{P_1}\!\downarrow]), \mathtt{Behav}(\mathbf{C_T}[\mathsf{P_1}\!\downarrow])) \in R\end{aligned}$$

Definition F.3 (R2rSCHC)**.**

$$\begin{aligned}\mathsf{R2rSCHC}: \forall \mathsf{P_1 P_2} \mathbf{C_T}.\ \exists \mathsf{C_S}.\ & \mathtt{Behav}(\mathsf{C_s}[\mathsf{P_1}]) \subseteq (\mathtt{Behav}(\mathbf{C_T}[\mathsf{P_1}\!\downarrow]) \wedge \\ & \mathtt{Behav}(\mathsf{C_s}[\mathsf{P_2}]) \subseteq (\mathtt{Behav}(\mathbf{C_T}[\mathsf{P_2}\!\downarrow])\end{aligned}$$

Lemma F.4. $\mathsf{R2rSCP} \iff \mathsf{R2rSCC}$

Proof. See file Criteria.v, theorem R2rSCHC_R2rSCHP. ☐

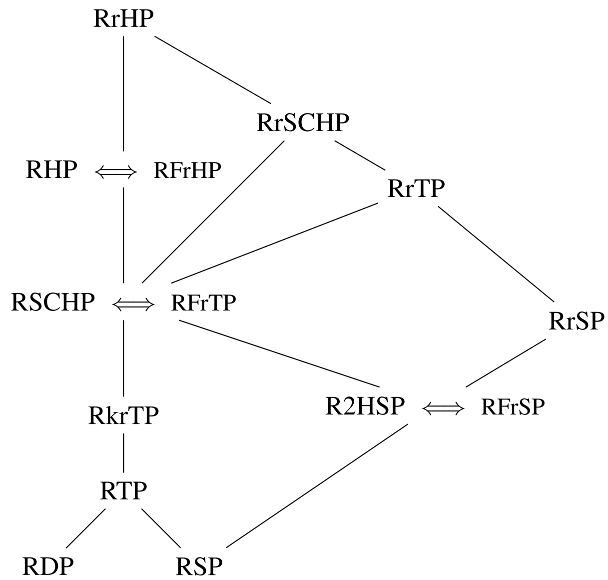
As usual, it is possible to generalize these definitions from binary relations to relations of finite or arbitrary arities.

Definition F.5 (RrSCHP)**.**

$$\begin{aligned}\mathsf{RrSCHP}:\ \forall R \in 2^{(\mathsf{Progs} \to \mathit{SCH})}.\ & (\forall \mathsf{C_S}. (\lambda \mathsf{P}. \mathtt{Behav}(\mathsf{C_S}[\mathsf{P}])) \in R) \Rightarrow \\ & (\forall \mathbf{C_T}. (\lambda \mathsf{P}. \mathtt{Behav}(\mathbf{C_T}[\mathsf{P}\!\downarrow])) \in R)\end{aligned}$$

F.1 Context Composition by Full Reflection

In this section we discuss our criteria assuming source programs can fully examine their own code, as is enabled by the use of *full reflection* mechanisms in languages Lisp [91] and Smalltalk. More precisely, details we assume that given two distinct source programs $\mathsf{P_1}, \mathsf{P_2}$ it is possible to compose two source contexts $\mathsf{C_1}, \mathsf{C_2}$, we write $\mathsf{C} = \mathsf{C_1} \otimes \mathsf{C_2}$ such that $\mathtt{Behav}(\mathsf{C}[\mathsf{P_i}]) = \mathtt{Behav}(\mathsf{C_i}[\mathsf{P_i}]),\ \ i = 1, 2$. Figure 3 reduces to the following diagram:



The file FullReflection.v contains proofs of the following collapses

- R2HSP $\Rightarrow$ R2rSP (theorem R2HSP_R2rSP)
- RHP $\Rightarrow$ R2rHP (theorem RHP_R2rHP)
- RSCHP $\Rightarrow$ R2rSCHP (theorem RSCHP_R2rSCHP)

To sketch a proof of RHP $\Rightarrow$ R2rHP, consider their *property-free* characterizations. For P_1, P_2 distinct and C apply twice RHC and get two source contexts C_1, C_2. Then $C_1 \otimes C_2$ satisfies the thesis. We can generalize these facts to finitary relations.

Theorem F.6. R2HSP $\Rightarrow$ RFrSP

Proof. Same argument used in reflection.v, theorem R2HSP_R2rSP. ☐

Theorem F.7. RHP $\Rightarrow$ RFrHP

Proof. Same argument used in reflection.v, theorem RHP_R2rHP. ☐

Theorem F.8. RSCHP $\Rightarrow$ RFrSCHP

Proof. Same argument used in reflection.v, theorem RSCHP_R2rSCHP. ☐

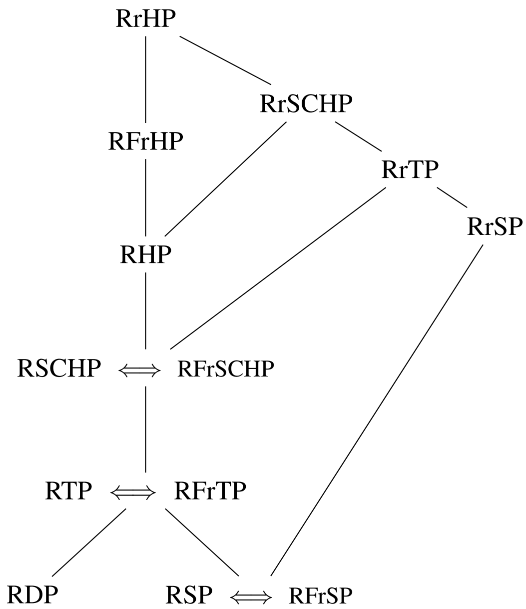
Some of the variants of the results in this section where previously stated in Theorem 4.2.

F.2 Context Composition by Internal Nondeterministic Choice

In this section we discuss our criteria in presence of source contexts that can nondeterministically behave like one of two already existing source contexts. Many criteria, in general stronger, become equivalent to weaker ones. For instance an RSC compiler preserves much more than the robust satisfaction of safety properties, including 2-hypersafety. Formally we assume to have an operator $\oplus : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ such that

$$\forall C_1 C_2 P.\ \texttt{Behav}\,(C_1 \oplus C_2)[P]) \supseteq \texttt{Behav}\,(C_1\,[P]) \cup \texttt{Behav}\,(C_2\,[P])$$

Figure 3 reduces to the following diagram:



The file InternalNondet.v contains proofs of the following binary collapses

- $\mathsf{RSCHP} \Rightarrow \mathsf{R2rSCHP}$ (theorem RSCHP_R2rSCHP)
- $\mathsf{RTP} \Rightarrow \mathsf{R2rTP}$ (theorem RTP_R2rTP)
- $\mathsf{RSP} \Rightarrow \mathsf{R2rSP}$ (theorem RSP_R2rSP)

To sketch a proof for $\mathsf{RSP} \Rightarrow \mathsf{R2rSP}$ consider their *property-free* characterizations, and assume $\mathbf{C}\left[\mathsf{P}{\downarrow}\right] \rightsquigarrow m_1, m_2$. Apply twice RSC and get two, possibly different, source contexts $\mathsf{C}_1, \mathsf{C}_2$, then $\mathsf{C} = \mathsf{C}_1 \oplus \mathsf{C}_2$ satisfies the thesis. We can generalize these facts to finitary relations.

Theorem F.9. $\mathsf{RSCHP} \Rightarrow \mathsf{RFrSCHP}$

Proof. Same argument used in nd_ctxs.v, theorem RSCHP_R2rSCHP. □

Theorem F.10. $\mathsf{RTP} \Rightarrow \mathsf{RFrTP}$

Proof. Same argument used in nd_ctxs.v, theorem RTP_R2rTP. □

Theorem F.11. $\mathsf{RSP} \Rightarrow \mathsf{RFrSP}$

Proof. Same argument used in nd_ctxs.v, theorem RSP_R2rSP. □

Some of the variants of the results in this section where previously stated in Theorem 4.3.

Appendix G
Proof Techniques for $\mathsf{RrHC}_{\bowtie}$ and $\mathsf{RFrXC}_{\bowtie}$

This section presents the formal details of §6. As explained in the main paper, we use two different proof techniques, one that is "context-based", and the other "trace-based", to prove two different security criteria for the same compilation chain. We argue that one of these techniques, the trace-based one, while less powerful, still gives us an interesting criterion, and should be more generic, as it relies less on the details of the languages.

A remark on the security criteria used in this section In the languages used in this example, not all programs and contexts can be linked together. In order for it to be the case, they have to satisfy some interfacing constraints. Here, these constraints are the existence of functions called but not defined by the context, and, in the source language, also agreement on the types of these functions. We introduce the operators $\bowtie$ and $\bowtie$ to represent these constraints. For instance, $\mathsf{P} \bowtie \mathsf{C}$ means that C is linkable with P. We prove two (§C.4.1) and RFrXC (§C.3.6), named $\mathsf{RrHC}_{\bowtie}$ (Definition G.6) and $\mathsf{RFrXC}_{\bowtie}$ (Definition G.45), that take into account these linkability predicates.

G.1 The Source Language L^{τ}

A list of elements $e_1, \cdots, e_n$ is indicated as $\overline{e}$, the empty list is $\varnothing$.

G.1.1 Syntax

$$
\begin{aligned}
\textit{Program}\ \mathsf{P} &::= \bar{\mathsf{I}}; \bar{\mathsf{F}} \\
\textit{Contexts}\ \mathsf{C} &::= \mathsf{e} \\
\textit{Interfaces}\ \mathsf{I} &::= \mathsf{f} : \tau \to \tau \\
\textit{Functions}\ \mathsf{F} &::= \mathsf{f}(\mathsf{x} : \tau) : \tau \mapsto \mathsf{return}\ \mathsf{e} \\
\textit{Types}\ \tau &::= \mathsf{Bool} \mid \mathsf{Nat} \\
\textit{Operations}\ \oplus &::= + \mid - \\
\textit{Values}\ \mathsf{v} &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \in \mathbb{N} \\
\textit{Expressions}\ \mathsf{e} &::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{let}\ \mathsf{x} : \tau = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e} \mid \mathsf{e} \geq \mathsf{e} \\
&\quad \mid \mathsf{call}\ \mathsf{f}\ \mathsf{e} \mid \mathsf{read} \mid \mathsf{write}\ \mathsf{e} \mid \mathsf{fail} \\
\textit{Runtime Expr.}\ \mathsf{e} &::= \cdots \mid \mathsf{return}\ \mathsf{e} \\
\textit{Eval. Ctxs.}\ \mathbb{E} &::= [\cdot] \mid \mathsf{e} \oplus \mathbb{E} \mid \mathbb{E} \oplus \mathsf{n} \mid \mathsf{let}\ \mathsf{x} = \mathbb{E}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{if}\ \mathbb{E}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e} \mid \mathsf{e} \geq \mathbb{E} \mid \mathbb{E} \geq \mathsf{n} \\
&\quad \mid \mathsf{call}\ \mathsf{f}\ \mathbb{E} \mid \mathsf{write}\ \mathbb{E} \mid \mathsf{return}\ \mathbb{E} \\
\textit{Substitutions}\ \rho &::= [\mathsf{v}/\mathsf{x}] \\
\textit{Prog. States}\ \Omega &::= \mathsf{P} \triangleright \mathsf{e} \mid \mathsf{fail} \\
\textit{Environments}\ \Gamma &::= \varnothing \mid \Gamma; (\mathsf{x} : \tau) \\
\textit{Labels}\ \lambda &::= \epsilon \mid \alpha \\
\textit{Actions}\ \alpha &::= \mathtt{read}\ n \mid \mathtt{write}\ n \mid \Downarrow \mid \Uparrow \mid \bot \\
\textit{Interactions}\ \gamma &::= \mathtt{call}\ f\ v? \mid \mathtt{ret}\ v! \\
\textit{Behaviors}\ \beta &::= \overline{\alpha} \\
\textit{Traces}\ \sigma &::= \varnothing \mid \sigma\alpha \mid \sigma\gamma
\end{aligned}
$$

G.1.2 Static Semantics

The static semantics follows these typing judgements.

$\vdash \mathsf{P}$	Program P is well-typed.
$\mathsf{P} \vdash \mathsf{F} : \tau \to \tau$	Function F has type $\tau \to \tau$ in program P.
$\Gamma \vdash \diamond$	Environment Γ is well-formed.
$\mathsf{P}; \Gamma \vdash \mathsf{e} : \tau$	Expression e has type τ in Γ and P.

$\boxed{\vdash \mathsf{P}}$

$$\frac{\mathsf{P} \equiv \bar{\mathsf{I}}; \bar{\mathsf{F}} \qquad \mathsf{P} \vdash \bar{\mathsf{F}} : \tau \to \tau \qquad \mathtt{dom}\left(\bar{\mathsf{F}}\right) \subseteq \bar{\mathsf{I}}}{\vdash \mathsf{P}}\ (\mathsf{TL}^{\tau}\text{-component})$$

$\boxed{\mathsf{P} \vdash \mathsf{F} : \tau \to \tau}$

$$\frac{\mathsf{F} \equiv \mathsf{f}(\mathsf{x} : \tau) : \tau' \mapsto \mathsf{return}\ \mathsf{e} \qquad \mathsf{P}; \mathsf{x} : \tau \vdash \mathsf{e} : \tau'}{\mathsf{P} \vdash \mathsf{F} : \tau \to \tau'}\ (\mathsf{TL}^{\tau}\text{-function})$$

$\boxed{\mathsf{P}; \Gamma \vdash \mathsf{e} : \tau}$

$$\frac{\Gamma \vdash \diamond}{\mathsf{P}; \Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ (\mathsf{TL}^{\tau}\text{-true}) \qquad \frac{\Gamma \vdash \diamond}{\mathsf{P}; \Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ (\mathsf{TL}^{\tau}\text{-false}) \qquad \frac{\Gamma \vdash \diamond}{\mathsf{P}; \Gamma \vdash \mathsf{n} : \mathsf{Nat}}\ (\mathsf{TL}^{\tau}\text{-nat})$$

$$\frac{\mathsf{x} : \tau \in \Gamma}{\mathsf{P}; \Gamma \vdash \mathsf{x} : \tau}\ (\mathsf{TL}^{\tau}\text{-var}) \qquad \frac{\begin{array}{c}\mathsf{P}; \Gamma \vdash \mathsf{e} : \mathsf{Nat} \\ \mathsf{P}; \Gamma \vdash \mathsf{e}' : \mathsf{Nat}\end{array}}{\mathsf{P}; \Gamma \vdash \mathsf{e} \oplus \mathsf{e}' : \mathsf{Nat}}\ (\mathsf{TL}^{\tau}\text{-op})$$

$$\frac{\begin{array}{c}\mathsf{P}; \Gamma \vdash \mathsf{e} : \mathsf{Nat} \\ \mathsf{P}; \Gamma \vdash \mathsf{e}' : \mathsf{Nat}\end{array}}{\mathsf{P}; \Gamma \vdash \mathsf{e} \geq \mathsf{e}' : \mathsf{Bool}}\ (\mathsf{TL}^{\tau}\text{-geq}) \qquad \frac{\begin{array}{c}\mathsf{P}; \Gamma \vdash \mathsf{e} : \tau \\ \mathsf{P}; \Gamma; \mathsf{x} : \tau \vdash \mathsf{e}' : \tau'\end{array}}{\mathsf{P}; \Gamma \vdash \mathsf{let}\ \mathsf{x} : \tau = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' : \tau'}\ (\mathsf{TL}^{\tau}\text{-letin})$$

$$\frac{\begin{array}{c}\mathsf{P}; \Gamma \vdash \mathsf{e} : \mathsf{Bool} \\ \mathsf{P}; \Gamma \vdash \mathsf{e_t} : \tau \qquad \mathsf{P}; \Gamma \vdash \mathsf{e_f} : \tau\end{array}}{\mathsf{P}; \Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e_t}\ \mathsf{else}\ \mathsf{e_f} : \tau}\ (\mathsf{TL}^{\tau}\text{-if})$$

$$\frac{(f(x:\tau):\tau' \mapsto \text{return } e \in \overline{F}) \qquad P \equiv \overline{I};\overline{F} \qquad P;\Gamma \vdash e:\tau}{P;\Gamma \vdash \text{call } f\ e : \tau'}\ (\text{TL}^{\tau}\text{-function-call})$$

$$\frac{(f(x:\tau):\tau' \mapsto \text{return } e \in \overline{I}) \qquad P \equiv \overline{I};\overline{F} \qquad P;\Gamma \vdash e:\tau}{P;\Gamma \vdash \text{call } f\ e : \tau'}\ (\text{TL}^{\tau}\text{-context-function-call})$$

$$\frac{}{P;\Gamma \vdash \text{read} : \text{Nat}}\ (\text{TL}^{\tau}\text{-read})$$

$$\frac{P;\Gamma \vdash e : \text{Nat}}{P;\Gamma \vdash \text{write } e : \text{Nat}}\ (\text{TL}^{\tau}\text{-write}) \qquad \frac{}{P;\Gamma \vdash \text{fail} : \tau}\ (\text{TL}^{\tau}\text{-fail})$$

$\boxed{P \bowtie C}$

$$\frac{P \equiv \overline{I};\overline{F} \qquad C \equiv e \qquad \vdash P \qquad P;\Gamma \vdash e:\tau \qquad \text{fail} \notin P \qquad \text{read}, \text{write } \_ \notin C \qquad \forall \text{call } f \in C, f \in \overline{I}}{P \bowtie C}\ (\text{Link-L}^{\tau})$$

G.1.3 Dynamic Semantics

$\Omega \xrightarrow{\lambda} \Omega'$ Program state Ω steps to Ω' emitting action λ.

$\Omega \xRightarrow{\beta} \Omega'$ Program state Ω steps to Ω' with behavior β.

$\boxed{P \triangleright e \xrightarrow{\lambda} P \triangleright e'}$

$$\frac{n \oplus n' = n''}{P \triangleright n \oplus n' \xrightarrow{\epsilon} P \triangleright n''}\ (\text{EL}^{\tau}\text{-op}) \qquad \frac{n \geq n'}{P \triangleright n \geq n' \xrightarrow{\epsilon} P \triangleright \text{true}}\ (\text{EL}^{\tau}\text{-geq-true}) \qquad \frac{n < n'}{P \triangleright n \geq n' \xrightarrow{\epsilon} P \triangleright \text{false}}\ (\text{EL}^{\tau}\text{-geq-false})$$

$$\frac{}{P \triangleright \text{if true then } e \text{ else } e' \xrightarrow{\epsilon} P \triangleright e}\ (\text{EL}^{\tau}\text{-if-true}) \qquad \frac{}{P \triangleright \text{if false then } e \text{ else } e' \xrightarrow{\epsilon} P \triangleright e'}\ (\text{EL}^{\tau}\text{-if-false})$$

$$\frac{}{P \triangleright \text{let } x = v \text{ in } e \xrightarrow{\epsilon} P \triangleright e[v/x]}\ (\text{EL}^{\tau}\text{-let}) \qquad \frac{f(x:\tau_1):\tau_2 \mapsto \text{return } e \in P}{P \triangleright_{\overline{f}} \text{call } f\ v \xrightarrow{\epsilon} P \triangleright_{\overline{f},f} \text{return } e[v/x]}\ (\text{EL}^{\tau}\text{-call-internal})$$

$$\frac{f(x:\tau_1):\tau_2 \mapsto \text{return } e \in P}{P \triangleright_{\epsilon} \text{call } f\ v \xrightarrow{\texttt{call f v?}} P \triangleright_{f} \text{return } e[v/x]}\ (\text{EL}^{\tau}\text{-call-in}) \qquad \frac{}{P \triangleright_{\overline{f},f,f'} \text{return } v \xrightarrow{\epsilon} P \triangleright_{\overline{f},f} v}\ (\text{EL}^{\tau}\text{-ret-internal})$$

$$\frac{}{P \triangleright_{f} \text{return } v \xrightarrow{\texttt{ret v!}} P \triangleright v}\ (\text{EL}^{\tau}\text{-ret-out}) \qquad \frac{}{P \triangleright \text{read} \xrightarrow{\texttt{read n}} P \triangleright n}\ (\text{EL}^{\tau}\text{-read}) \qquad \frac{}{P \triangleright \text{write } n \xrightarrow{\texttt{write n}} P \triangleright n}\ (\text{EL}^{\tau}\text{-write})$$

$$\frac{P \triangleright e \xrightarrow{\epsilon} P \triangleright e'}{P \triangleright \mathbb{E}[e] \xrightarrow{\epsilon} P \triangleright \mathbb{E}[e']}\ (\text{EL}^{\tau}\text{-ctx}) \qquad \frac{}{P \triangleright \text{fail} \xrightarrow{\perp} \text{fail}}\ (\text{EL}^{\tau}\text{-fail})$$

$\boxed{P \triangleright e \xRightarrow{\beta} P \triangleright e'}$

$$\frac{}{\Omega \Rightarrow \Omega}\ (\text{EL}^{\tau}\text{-refl}) \qquad \frac{\Omega \not\Rightarrow \_}{\Omega \xRightarrow{\Downarrow} \Omega}\ (\text{EL}^{\tau}\text{-terminate}) \qquad \frac{\forall n.\ \Omega \xrightarrow{\epsilon}{}^{n}\ \Omega'_n}{\Omega \xRightarrow{\Uparrow} \Omega}\ (\text{EL}^{\tau}\text{-diverge})$$

$$\frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Rightarrow \Omega'}\ (\text{EL}^{\tau}\text{-silent}) \qquad \frac{\Omega \xrightarrow{\alpha} \Omega'}{\Omega \xRightarrow{\alpha} \Omega'}\ (\text{EL}^{\tau}\text{-single}) \qquad \frac{\Omega \xrightarrow{\gamma} \Omega'}{\Omega \Rightarrow \Omega'}\ (\text{EL}^{\tau}\text{-silent-act})$$

$$\frac{\Omega \xRightarrow{\beta} \Omega'' \qquad \Omega'' \xRightarrow{\beta'} \Omega'}{\Omega \xRightarrow{\beta\beta'} \Omega'}\ (\text{EL}^{\tau}\text{-cons})$$

$\boxed{P \triangleright e \xRightarrow{t}\!\!\!\Rightarrow P \triangleright e'}$

$$\frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Longrightarrow \Omega'}\ (\text{EL}^{\tau}\text{-silent}) \qquad \frac{\Omega \overset{\alpha}{\Longrightarrow} \Omega'}{\Omega \overset{\alpha}{\Longrightarrow} \Omega'}\ (\text{EL}^{\tau}\text{-action}) \qquad \frac{\Omega \xrightarrow{\gamma} \Omega'}{\Omega \overset{\gamma}{\Longrightarrow} \Omega'}\ (\text{EL}^{\tau}\text{-single}) \qquad \frac{\Omega \overset{\sigma}{\Longrightarrow} \Omega'' \quad \Omega'' \overset{\sigma'}{\Longrightarrow} \Omega'}{\Omega \overset{\sigma\sigma'}{\Longrightarrow} \Omega'}\ (\text{EL}^{\tau}\text{-cons})$$

G.1.4 Auxiliaries and Definitions

Helpers

$$\frac{\mathsf{P} \bowtie \mathsf{C} \qquad \mathsf{C} \equiv \mathsf{e}}{\Omega_0(\mathsf{C}[\mathsf{P}]) = \mathsf{P} \triangleright \mathsf{e}}\ (\text{L}^{\tau}\text{-Initial State})$$

Definition G.1 (Program Behaviors)**.**

$$\texttt{Behav}\,(\mathsf{P}) = \left\{\beta \;\middle|\; \exists \Omega'.\Omega_0(\mathsf{P}) \overset{\beta}{\Longrightarrow} \Omega'\right\}$$

Theorem G.2 (Progress)**.** If $\mathsf{P};\Gamma \vdash \mathsf{e} : \tau$ then either $\mathsf{e} \equiv \mathsf{v}$ or $\exists \mathsf{e}'.\mathsf{P} \triangleright \mathsf{e} \hookrightarrow \mathsf{P} \triangleright \mathsf{e}'$.

Theorem G.3 (Preservation)**.** If $\mathsf{P};\Gamma \vdash \mathsf{e} : \tau$ and $\mathsf{P} \triangleright \mathsf{e} \hookrightarrow \mathsf{P} \triangleright \mathsf{e}'$ then $\mathsf{P};\Gamma \vdash \mathsf{e}' : \tau$.

G.2 The Target Language $\mathbf{L^u}$

G.2.1 Syntax

$$\begin{aligned}
\textit{Program}\ \mathbf{P} &::= \overline{\mathbf{I}}; \overline{\mathbf{F}} \\
\textit{Contexts}\ \mathbf{C} &::= \mathbf{e} \\
\textit{Interfaces}\ \mathbf{I} &::= \mathbf{f} \\
\textit{Functions}\ \mathbf{F} &::= \mathbf{f(x)} \mapsto \mathbf{return\ e} \\
\textit{Types}\ \tau &::= \mathbf{Bool} \mid \mathbf{Nat} \\
\textit{Operations}\ \oplus &::= + \mid - \\
\textit{Values}\ \mathbf{v} &::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{n} \in \mathbb{N} \\
\textit{Expressions}\ \mathbf{e} &::= \mathbf{x} \mid \mathbf{v} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{let\ x = e\ in\ e} \mid \mathbf{if\ e\ then\ e\ else\ e} \mid \mathbf{e} \geq \mathbf{e} \\
&\quad\ \mid \mathbf{call\ f\ e} \mid \mathbf{read} \mid \mathbf{write\ e} \mid \mathbf{fail} \mid \mathbf{e\ has}\ \tau \\
\textit{Runtime Expr.}\ \mathbf{e} &::= \cdots \mid \mathbf{return\ e} \\
\textit{Eval. Ctxs.}\ \mathbb{E} &::= [\cdot] \mid \mathbf{e} \oplus \mathbb{E} \mid \mathbb{E} \oplus \mathbf{n} \mid \mathbf{let\ x} = \mathbb{E}\ \mathbf{in\ e} \mid \mathbf{if}\ \mathbb{E}\ \mathbf{then\ e\ else\ e} \mid \mathbf{e} \geq \mathbb{E} \mid \mathbb{E} \geq \mathbf{n} \\
&\quad\ \mid \mathbf{call\ f}\ \mathbb{E} \mid \mathbf{write}\ \mathbb{E} \mid \mathbf{return}\ \mathbb{E} \mid \mathbb{E}\ \mathbf{has}\ \tau \\
\textit{Substitutions}\ \rho &::= [\mathbf{v}/\mathbf{x}] \\
\textit{Prog. States}\ \mathbf{\Omega} &::= \mathbf{P} \triangleright_{\overline{\mathbf{f}}} \mathbf{e} \mid \mathbf{fail} \\
\textit{Labels}\ \lambda &::= \epsilon \mid \alpha \mid \gamma \\
\textit{Actions}\ \alpha &::= \texttt{read}\ n \mid \texttt{write}\ n \mid \Downarrow \mid \Uparrow \mid \bot \\
\textit{Interactions}\ \gamma &::= \texttt{call}\ f\ v? \mid \texttt{ret}\ v! \\
\textit{Behaviors}\ \beta &::= \overline{\alpha} \\
\textit{Traces}\ \sigma &::= \varnothing \mid \sigma\alpha \mid \sigma\gamma
\end{aligned}$$

Program states carry around the stack of called functions (the $\overline{\mathbf{f}}$ subscript) in order to correctly characterise calls and returns that go in Traces. We mostly omit this stack when it just clutters the presentation without itself changing and make it explicit only when it is needed.

We define the linkability operator as follows:

$\mathbf{P} \bowtie \mathbf{C}$

$$\frac{\mathbf{P} \equiv \overline{\mathbf{I}}; \overline{\mathbf{F}} \qquad \mathbf{C} \equiv \mathbf{e} \qquad \mathbf{read}, \mathbf{write}\ \_ \notin \mathbf{C} \qquad \forall \mathbf{call\ f} \in \mathbf{C}, \mathbf{f} \in \overline{\mathbf{I}}}{\mathbf{P} \bowtie \mathbf{C}}\ (\text{Link-}\mathbf{L^u})$$

G.2.2 Dynamic Semantics

$\Omega \xrightarrow{\lambda} \Omega'$ Program state Ω steps to Ω' emitting action λ.

$\Omega \xRightarrow{\beta} \Omega'$ Program state Ω steps to Ω' with behavior β.

$\Omega \overset{\sigma}{\Rrightarrow} \Omega'$ Program state Ω steps to Ω' with trace σ.

$\boxed{\mathbf{P} \triangleright \mathbf{e} \xrightarrow{\lambda} \mathbf{P} \triangleright \mathbf{e}'}$

$$\text{(EL}^{\mathbf{u}}\text{-op)}\ \frac{n \oplus n' = n''}{\mathbf{P} \triangleright \mathbf{n} \oplus \mathbf{n}' \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{n}''}$$

$$\text{(EL}^{\mathbf{u}}\text{-geq-true)}\ \frac{n \geq n'}{\mathbf{P} \triangleright \mathbf{n} \geq \mathbf{n}' \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{true}}$$

$$\text{(EL}^{\mathbf{u}}\text{-geq-false)}\ \frac{n < n'}{\mathbf{P} \triangleright \mathbf{n} \geq \mathbf{n}' \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{false}}$$

$$\text{(EL}^{\mathbf{u}}\text{-if-true)}\ \frac{}{\mathbf{P} \triangleright \mathbf{if\ true\ then\ e\ else\ e}' \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{e}}$$

$$\text{(EL}^{\mathbf{u}}\text{-if-false)}\ \frac{}{\mathbf{P} \triangleright \mathbf{if\ false\ then\ e\ else\ e}' \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{e}'}$$

$$\text{(EL}^{\mathbf{u}}\text{-let)}\ \frac{}{\mathbf{P} \triangleright \mathbf{let\ x = v\ in\ e} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{e}[\mathbf{v}/\mathbf{x}]}$$

$$\text{(EL}^{\mathbf{u}}\text{-read)}\ \frac{}{\mathbf{P} \triangleright \mathbf{read} \xrightarrow{\texttt{read n}} \mathbf{P} \triangleright \mathbf{n}}$$

$$\text{(EL}^{\mathbf{u}}\text{-write)}\ \frac{}{\mathbf{P} \triangleright \mathbf{write\ n} \xrightarrow{\texttt{write n}} \mathbf{P} \triangleright \mathbf{n}}$$

$$\text{(EL}^{\mathbf{u}}\text{-ctx)}\ \frac{\mathbf{P} \triangleright \mathbf{e} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{e}'}{\mathbf{P} \triangleright \mathbb{E}[\mathbf{e}] \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbb{E}[\mathbf{e}']}$$

$$\text{(EL}^{\mathbf{u}}\text{-check-bool-true)}\ \frac{\mathbf{v} \equiv \mathbf{true} \vee \mathbf{v} \equiv \mathbf{false}}{\mathbf{P} \triangleright \mathbf{v}\ \texttt{has Bool} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{true}}$$

$$\text{(EL}^{\mathbf{u}}\text{-check-bool-false)}\ \frac{}{\mathbf{P} \triangleright \mathbf{n}\ \texttt{has Bool} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{false}}$$

$$\text{(EL}^{\mathbf{u}}\text{-check-nat-true)}\ \frac{}{\mathbf{P} \triangleright \mathbf{n\ has}\ \mathbb{N} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{true}}$$

$$\text{(EL}^{\mathbf{u}}\text{-check-nat-false)}\ \frac{\mathbf{v} \equiv \mathbf{true} \vee \mathbf{v} \equiv \mathbf{false}}{\mathbf{P} \triangleright \mathbf{v\ has}\ \mathbb{N} \xrightarrow{\epsilon} \mathbf{P} \triangleright \mathbf{false}}$$

$$\text{(EL}^{\mathbf{u}}\text{-call-internal)}\ \frac{\mathbf{f}(\mathbf{x}) \mapsto \mathbf{return\ e} \in \mathbf{P}}{\mathbf{P} \triangleright_{\overline{\mathbf{f}}} \mathbf{call\ f\ v} \xrightarrow{\epsilon} \mathbf{P} \triangleright_{\overline{\mathbf{f}},\mathbf{f}} \mathbf{return\ e}[\mathbf{v}/\mathbf{x}]}$$

$$\text{(EL}^{\mathbf{u}}\text{-call-in)}\ \frac{\mathbf{f}(\mathbf{x}) \mapsto \mathbf{return\ e} \in \mathbf{P}}{\mathbf{P} \triangleright_{\epsilon} \mathbf{call\ f\ v} \xrightarrow{\texttt{call f v?}} \mathbf{P} \triangleright_{\mathbf{f}} \mathbf{return\ e}[\mathbf{v}/\mathbf{x}]}$$

$$\text{(EL}^{\mathbf{u}}\text{-ret-internal)}\ \frac{}{\mathbf{P} \triangleright_{\overline{\mathbf{f}},\mathbf{f},\mathbf{f}'} \mathbf{return\ v} \xrightarrow{\epsilon} \mathbf{P} \triangleright_{\overline{\mathbf{f}},\mathbf{f}} \mathbf{v}}$$

$$\text{(EL}^{\mathbf{u}}\text{-ret-out)}\ \frac{}{\mathbf{P} \triangleright_{\mathbf{f}} \mathbf{return\ v} \xrightarrow{\texttt{ret v!}} \mathbf{P} \triangleright \mathbf{v}}$$

$$\text{(EL}^{\mathbf{u}}\text{-op-fail)}\ \frac{\mathbf{v} \equiv \mathbf{true} \vee \mathbf{v} \equiv \mathbf{false} \vee \mathbf{v}' \equiv \mathbf{true} \vee \mathbf{v}' \equiv \mathbf{false}}{\mathbf{P} \triangleright \mathbf{v} \oplus \mathbf{v}' \xrightarrow{\perp} \mathbf{fail}}$$

$$\text{(EL}^{\mathbf{u}}\text{-geq-fail)}\ \frac{\mathbf{v} \equiv \mathbf{true} \vee \mathbf{v} \equiv \mathbf{false} \vee \mathbf{v}' \equiv \mathbf{true} \vee \mathbf{v}' \equiv \mathbf{false}}{\mathbf{P} \triangleright \mathbf{v} \geq \mathbf{v}' \xrightarrow{\perp} \mathbf{fail}}$$

$$\text{(EL}^{\mathbf{u}}\text{-if-fail)}\ \frac{}{\mathbf{P} \triangleright \mathbf{if\ n\ then\ e\ else\ e}' \xrightarrow{\perp} \mathbf{fail}}$$

$$\text{(EL}^{\mathbf{u}}\text{-fail)}\ \frac{}{\mathbf{P} \triangleright \mathbf{fail} \xrightarrow{\perp} \mathbf{fail}}$$

$\boxed{\mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta} \mathbf{P} \triangleright \mathbf{e}'}$

$$\text{(EL}^{\mathbf{u}}\text{-refl)}\ \frac{}{\Omega \Rightarrow \Omega}$$

$$\text{(EL}^{\mathbf{u}}\text{-terminate)}\ \frac{\Omega \not\xrightarrow{\epsilon} \_}{\Omega \xRightarrow{\Downarrow} \Omega}$$

$$\text{(EL}^{\mathbf{u}}\text{-diverge)}\ \frac{\forall \mathbf{n}.\ \Omega \xrightarrow{\epsilon}{}^{\mathbf{n}} \Omega'_{\mathbf{n}}}{\Omega \xRightarrow{\Uparrow} \Omega}$$

$$\text{(EL}^{\mathbf{u}}\text{-silent)}\ \frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Rightarrow \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-silent-act)}\ \frac{\Omega \xrightarrow{\gamma} \Omega'}{\Omega \Rightarrow \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-single)}\ \frac{\Omega \xrightarrow{\alpha} \Omega'}{\Omega \xRightarrow{\alpha} \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-cons)}\ \frac{\Omega \xRightarrow{\beta} \Omega'' \quad \Omega'' \xRightarrow{\beta'} \Omega'}{\Omega \xRightarrow{\beta\beta'} \Omega'}$$

$\boxed{\mathbf{P} \triangleright \mathbf{e} \overset{\sigma}{\Rrightarrow} \mathbf{P} \triangleright \mathbf{e}'}$

$$\text{(EL}^{\mathbf{u}}\text{-silent)}\ \frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Rrightarrow \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-action)}\ \frac{\Omega \xRightarrow{\alpha} \Omega'}{\Omega \overset{\alpha}{\Rrightarrow} \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-single)}\ \frac{\Omega \xrightarrow{\gamma} \Omega'}{\Omega \overset{\gamma}{\Rrightarrow} \Omega'}$$

$$\text{(EL}^{\mathbf{u}}\text{-cons)}\ \frac{\Omega \overset{\sigma}{\Rrightarrow} \Omega'' \quad \Omega'' \overset{\sigma'}{\Rrightarrow} \Omega'}{\Omega \overset{\sigma\sigma'}{\Rrightarrow} \Omega'}$$

G.2.3 Auxiliaries and Definitions

Helpers

$$\frac{\mathbf{P} \bowtie \mathbf{C} \qquad \mathbf{C} \equiv \mathbf{e}}{\mathbf{\Omega_0}(\mathbf{C}[\mathbf{P}]) = \mathbf{P} \triangleright \mathbf{e}}\ (\mathbf{L^u}\text{-Initial State})$$

Definition G.4 (Program Behaviors)**.**

$$\mathsf{Behav}\left(\mathbf{P}\right) = \left\{\beta \;\middle|\; \exists \mathbf{\Omega}'.\mathbf{\Omega_0}(\mathbf{P}) \overset{\beta}{\Longrightarrow} \mathbf{\Omega}'\right\}$$

Definition G.5 (Program Traces)**.**

$$\mathsf{TR}(\mathbf{P}) = \left\{\sigma \;\middle|\; \exists \mathbf{\Omega}'.\mathbf{\Omega_0}(\mathbf{P}) \overset{\sigma}{\Longrightarrow\!\!\!\!\!\!\Rightarrow} \mathbf{\Omega}'\right\}$$

G.3 $\cdot\!\downarrow$: A Compiler from L^τ to $\mathbf{L^u}$

$$\mathsf{I_1, \cdots, I_m; F_1, \cdots, F_n}\!\downarrow\ = \mathsf{I_1}\!\downarrow, \cdots, \mathsf{I_m}\!\downarrow; \mathsf{F_1}\!\downarrow, \cdots, \mathsf{F_n}\!\downarrow \qquad (\cdot\!\downarrow\text{-Prog})$$

$$\mathsf{f} : \tau \to \tau'\!\downarrow\ = \mathbf{f} \qquad (\cdot\!\downarrow\text{-Intf})$$

$$\mathsf{f(x : \tau) : \tau' \mapsto return\ e}\!\downarrow\ = \mathbf{f(x) \mapsto return\ if\ x\ has\ \tau\!\downarrow\ then\ e\!\downarrow\ else\ fail} \qquad (\cdot\!\downarrow\text{-Fun})$$

$$\mathsf{n}\!\downarrow\ = \mathbf{n} \qquad (\cdot\!\downarrow\text{-Nat})$$

$$\mathsf{true}\!\downarrow\ = \mathbf{true} \qquad (\cdot\!\downarrow\text{-True})$$

$$\mathsf{false}\!\downarrow\ = \mathbf{false} \qquad (\cdot\!\downarrow\text{-False})$$

$$\mathsf{x}\!\downarrow\ = \mathbf{x} \qquad (\cdot\!\downarrow\text{-Var})$$

$$\mathsf{e \oplus e'}\!\downarrow\ = \mathsf{e}\!\downarrow \mathbf{\oplus}\ \mathsf{e'}\!\downarrow \qquad (\cdot\!\downarrow\text{-Op})$$

$$\mathsf{e \geq e'}\!\downarrow\ = \mathsf{e}\!\downarrow \mathbf{\geq}\ \mathsf{e'}\!\downarrow \qquad (\cdot\!\downarrow\text{-Geq})$$

$$\mathsf{let\ x : \tau = e\ in\ e'}\!\downarrow\ = \mathbf{let\ x =}\ \mathsf{e}\!\downarrow\ \mathbf{in}\ \mathsf{e'}\!\downarrow \qquad (\cdot\!\downarrow\text{-Let})$$

$$\mathsf{if\ e\ then\ e'\ else\ e''}\!\downarrow\ = \mathbf{if}\ \mathsf{e}\!\downarrow\ \mathbf{then}\ \mathsf{e'}\!\downarrow\ \mathbf{else}\ \mathsf{e''}\!\downarrow \qquad (\cdot\!\downarrow\text{-If})$$

$$\mathsf{call\ f\ e}\!\downarrow\ = \mathbf{call\ f}\ \mathsf{e}\!\downarrow \qquad (\cdot\!\downarrow\text{-Call})$$

$$\mathsf{read}\!\downarrow\ = \mathbf{read} \qquad (\cdot\!\downarrow\text{-Read})$$

$$\mathsf{write\ e}\!\downarrow\ = \mathbf{write}\ \mathsf{e}\!\downarrow \qquad (\cdot\!\downarrow\text{-Write})$$

$$\mathsf{Nat}\!\downarrow\ = \mathbf{Nat} \qquad (\cdot\!\downarrow\text{-Ty-Nat})$$

$$\mathsf{Bool}\!\downarrow\ = \mathbf{Bool} \qquad (\cdot\!\downarrow\text{-Ty-Bool})$$

G.4 Proof That $\cdot\!\downarrow$ Is $\mathsf{RrHC}_{\bowtie}$

We prove that the compiler satisfies the following variant of RrHC:

Definition G.6 ($\mathsf{RrHC}_{\bowtie}$)**.**

$$\mathsf{RrHC}_{\bowtie} : \quad \forall \bar{\mathsf{I}}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall \mathsf{P} : \bar{\mathsf{I}}.\ \mathsf{P}\!\downarrow \bowtie \mathbf{C_T} \implies \mathsf{P} \bowtie \mathsf{C_S} \wedge \mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P}\!\downarrow\right]\right) = \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P}\right]\right)$$

All programs must satisfy the same interface $\bar{\mathsf{I}}$ in order for the linkability with a single $\mathsf{C_S}$ to be possible.

We also give the following property-full criteria:

Definition G.7 ($\mathsf{RrHP}_{\bowtie}$)**.**

$$\begin{aligned}\mathsf{RrHP}_{\bowtie} : \quad & \forall \bar{\mathsf{I}}.\ \forall R \in 2^{(\mathsf{Behavs}^\omega)}.\ \forall \mathsf{P_1}, .., \mathsf{P_K} : \bar{\mathsf{I}}, \ldots \\ & (\forall \mathsf{C_S}.\ (\forall i, \mathsf{P_i} \bowtie \mathsf{C_S}) \implies (\mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P_1}\right]\right), .., \mathsf{Behav}\left(\mathsf{C_S}\left[\mathsf{P_K}\right]\right), ..) \in R) \Rightarrow \\ & (\forall \mathbf{C_T}.\ (\forall i, \mathsf{P_i}\!\downarrow \bowtie \mathbf{C_T}) \implies (\mathsf{Behav}\left(\mathbf{C_T}\left[\mathsf{P_1}\!\downarrow\right]\right), .., \mathsf{Behav}\left(\mathbf{C_S}\left[\mathsf{P_K}\!\downarrow\right]\right), ..) \in R)\end{aligned}$$

The proof of the equivalence of these two criteria is similar to the proof of Theorem C.80.

G.4.1 $\cdot\uparrow$: Backtranslation of Contexts from $\mathbf{L^u}$ to L^τ

Technically, the backtranslation needs one additional parameter to be passed around, the list of functions defined by the compiled component $\bar{\mathsf{I}}$, we elide it for simplicity when it is not necessary.

$$\mathbf{n}\uparrow = \mathsf{n} + 2 \tag{$\cdot\uparrow$-Nat}$$

$$\mathbf{true}\uparrow = 1 \tag{$\cdot\uparrow$-True}$$

$$\mathbf{false}\uparrow = 0 \tag{$\cdot\uparrow$-False}$$

$$\mathbf{x}\uparrow = \mathsf{x} \tag{$\cdot\uparrow$-Var}$$

$$\begin{aligned}\mathbf{e} \oplus \mathbf{e}'\uparrow = \ & \mathsf{let}\ \mathsf{x1} : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathbf{e}\uparrow) \\ & \mathsf{in}\ \mathsf{let}\ \mathsf{x2} : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathbf{e}'\uparrow) \\ & \mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(\mathsf{x1} \oplus \mathsf{x2})\end{aligned} \tag{$\cdot\uparrow$-Op}$$

$$\begin{aligned}\mathbf{e} \geq \mathbf{e}'\uparrow = \ & \mathsf{let}\ \mathsf{x1} : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathbf{e}\uparrow) \\ & \mathsf{in}\ \mathsf{let}\ \mathsf{x2} : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathbf{e}'\uparrow) \\ & \mathsf{in}\ \mathsf{inject}_{\mathsf{Bool}}(\mathsf{x1} \geq \mathsf{x2})\end{aligned} \tag{$\cdot\uparrow$-Geq}$$

$$\mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}'\uparrow = \mathsf{let}\ \mathsf{x} : \mathsf{Nat} = \mathbf{e}\uparrow\ \mathsf{in}\ \mathbf{e}'\uparrow \tag{$\cdot\uparrow$-Let}$$

$$\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}''\uparrow = \mathsf{if}\ \mathsf{extract}_{\mathsf{Bool}}(\mathbf{e}\uparrow)\ \mathsf{then}\ \mathbf{e}'\uparrow\ \mathsf{else}\ \mathbf{e}''\uparrow \tag{$\cdot\uparrow$-If}$$

$$\begin{aligned}\mathbf{call}\ \mathbf{f}\ \mathbf{e}\uparrow = \ & \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\tau}(\mathbf{e}\uparrow)) \\ & \text{if } \mathsf{f} : \tau \to \tau' \in \bar{\mathsf{I}}\end{aligned} \tag{$\cdot\uparrow$-Call}$$

$$\mathbf{e}\ \mathbf{has}\ \tau\uparrow = \begin{cases}\mathsf{let}\ \mathsf{x} : \mathsf{Nat} = \mathbf{e}\uparrow\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ 0\ \mathsf{else}\ 1 & \text{if } \tau \equiv \mathtt{Bool} \\ \mathsf{let}\ \mathsf{x} : \mathsf{Nat} = \mathbf{e}\uparrow\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ 1\ \mathsf{else}\ 0 & \text{if } \tau \equiv \mathbb{N}\end{cases} \tag{$\cdot\uparrow$-Check}$$

Helper functions The back-translation type is Nat but the encoding is not straight from Nat but it is Nat shifted by 2. $\mathsf{inject}_\tau(\mathsf{e})$ takes an expression e of type τ and returns an expression whose type is the back-translation type. $\mathsf{extract}_\tau(\mathsf{e})$ takes an expression e of back-translation type and returns an expression whose type is τ.

$$\begin{aligned}\mathsf{inject}_{\mathsf{Nat}}(\mathsf{e}) &= \mathsf{e} + 2 \\ \mathsf{inject}_{\mathsf{Bool}}(\mathsf{e}) &= \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ 1\ \mathsf{else}\ 0 \\ \mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}) &= \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{x} - 2\ \mathsf{else}\ \mathsf{fail} \\ \mathsf{extract}_{\mathsf{Bool}}(\mathsf{e}) &= \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x} + 1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\end{aligned}$$

G.4.2 Cross-Language Logical Relation

Language De-sugaring

$$\begin{aligned}\mathsf{v} &::= \ldots \mid \mathsf{call}\ \mathsf{f} \\ \mathsf{e} &::= \ldots \mid \mathsf{call}\ \mathsf{f}\ \mathsf{e} \\ \textit{Types}\ \tau &::= \sigma \mid \sigma \to \sigma \\ \textit{Base Types}\ \sigma &::= \mathsf{Nat} \mid \mathsf{Bool}\end{aligned}$$

Replace Rule TL^τ-function-call with these below.

$$\frac{\mathsf{f}(\mathsf{x} : \sigma) : \sigma' \mapsto \mathsf{return}\ \mathsf{e} \in \mathtt{dom}\left(\bar{\mathsf{F}}\right)}{\mathsf{P}; \Gamma \vdash \mathsf{call}\ \mathsf{f} : \sigma \to \sigma'}\ (\mathsf{TL}^\tau\text{-call}) \qquad \frac{\begin{array}{c}\mathsf{P}; \Gamma \vdash \mathsf{call}\ \mathsf{f} : \sigma' \to \sigma \\ \mathsf{P}; \Gamma \vdash \mathsf{e}' : \sigma'\end{array}}{\mathsf{P}; \Gamma \vdash \mathsf{call}\ \mathsf{f}\ \mathsf{e}' : \sigma}\ (\mathsf{TL}^\tau\text{-app})$$

Apply the same changes above to $\mathbf{L^u}$ too.

Context well-formedness ensures that expressions are never turned into $\mathbf{call}\ \mathbf{f}$ values.

$$\mathbf{\Gamma} ::= \varnothing \mid \mathbf{\Gamma}, \mathbf{x}$$

$$\frac{}{\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{true}}\ (\text{Ctx-}\mathbf{L^u}\text{-true}) \qquad \frac{}{\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{false}}\ (\text{Ctx-}\mathbf{L^u}\text{-false}) \qquad \frac{}{\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{n}}\ (\text{Ctx-}\mathbf{L^u}\text{-nat})$$

$$\frac{\mathbf{x} \in \mathtt{dom}\,(\mathbf{\Gamma})}{\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{x}}\ (\text{Ctx-}\mathbf{L^u}\text{-var}) \qquad \frac{\begin{array}{c}\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{e}' \quad \mathbf{e}' \not\equiv \mathbf{call}\ \mathbf{f} \\ \mathbf{f}(\mathbf{x}) \mapsto \mathbf{return}\ \mathbf{e} \in \mathbf{P}\end{array}}{\mathbf{P}; \mathbf{\Gamma} \vdash \mathbf{call}\ \mathbf{f}\ \mathbf{e}'}\ (\text{Ctx-}\mathbf{L^u}\text{-app})$$

$$\frac{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\quad \mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}' \quad \mathbf{e},\mathbf{e}'\not\equiv\mathbf{call\ f}}{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\oplus\ \mathbf{e}'}\ (\text{Ctx-}\mathbf{L^u}\text{-op})\qquad \frac{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\quad \mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}' \quad \mathbf{e},\mathbf{e}'\not\equiv\mathbf{call\ f}}{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\geq\ \mathbf{e}'}\ (\text{Ctx-}\mathbf{L^u}\text{-geq})\qquad \frac{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\quad \mathbf{P};\boldsymbol{\Gamma},\mathbf{x}\vdash\mathbf{e}' \quad \mathbf{e},\mathbf{e}'\not\equiv\mathbf{call\ f}}{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{let\ x=e\ in\ e}'}\ (\text{Ctx-}\mathbf{L^u}\text{-letin})$$

$$\frac{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}\quad \mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}'\quad \mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e}'' \quad \mathbf{e},\mathbf{e}',\mathbf{e}''\not\equiv\mathbf{call\ f}}{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{if\ e\ then\ e}'\ \mathbf{else\ e}''}\ (\text{Ctx-}\mathbf{L^u}\text{-if})\qquad \frac{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e} \quad \mathbf{e}\not\equiv\mathbf{call\ f}}{\mathbf{P};\boldsymbol{\Gamma}\vdash\mathbf{e\ has}\ \tau}\ (\text{Ctx-}\mathbf{L^u}\text{-check})$$

Replace Section G.3 with these below.

$$\mathsf{call\ f}\!\downarrow\; = \mathbf{call\ f} \qquad (\cdot\!\downarrow\text{-Call-v})$$

$$\mathsf{e\ e'}\!\downarrow\; = \mathsf{e}\!\downarrow\ \mathsf{e'}\!\downarrow \qquad (\cdot\!\downarrow\text{-App})$$

Worlds

$$\begin{aligned}
\mathit{World}\ W &::= (n, (\mathsf{P}, \mathbf{P}))\\
\mathit{lev}((n, _)) &= n\\
\mathit{progs}((_, (\mathsf{P}, \mathbf{P}))) &= (\mathsf{P}, \mathbf{P})\\
\mathit{srcprog}((_, (\mathsf{P}, \mathbf{P}))) &= \mathsf{P}\\
\mathit{trgprog}((_, (\mathsf{P}, \mathbf{P}))) &= \mathbf{P}\\
\triangleright((0, _)) &= (0, _)\\
\triangleright((n+1, _)) &= (n, _)\\
W \sqsupseteq W' &= \mathit{lev}(W') \leq \mathit{lev}(W)\\
W \sqsupset_{\triangleright} W' &= \mathit{lev}(W') < \mathit{lev}(W)\\
O(W)_{\lesssim} &\stackrel{\mathsf{def}}{=} \left\{ (\mathsf{e}, \mathbf{e}) \;\middle|\; \begin{array}{l} \text{if } \mathit{lev}(W) = n \text{ and } \mathit{progs}(W) = (\mathsf{P}, \mathbf{P})\\ \text{and } \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta}{}^{\mathsf{n}}\ \mathsf{P} \triangleright \mathsf{e}'\\ \text{then } \exists \mathbf{k}.\ \mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta}{}^{\mathbf{k}}\ \mathbf{P} \triangleright \mathbf{e}' \end{array} \right\}\\
O(W)_{\gtrsim} &\stackrel{\mathsf{def}}{=} \left\{ (\mathsf{e}, \mathbf{e}) \;\middle|\; \begin{array}{l} \text{if } \mathit{lev}(W) = n \text{ and } \mathit{progs}(W) = (\mathsf{P}, \mathbf{P})\\ \text{and } \mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta}{}^{\mathbf{n}}\ \mathbf{P} \triangleright \mathbf{e}'\\ \text{then } \exists \mathsf{k}.\ \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta}{}^{\mathsf{k}}\ \mathsf{P} \triangleright \mathsf{e}' \end{array} \right\}\\
O(W)_{\approx} &\stackrel{\mathsf{def}}{=} O(W)_{\lesssim} \cap O(W)_{\gtrsim}\\
\triangleright R &\stackrel{\mathsf{def}}{=} \{(W, \mathsf{v}, \mathbf{v}) \mid \text{if } \mathit{lev}(W) > 0 \text{ then } (\triangleright(W), \mathsf{v}, \mathbf{v}) \in R\}\\
\nearrow(R) &\stackrel{\mathsf{def}}{=} \{(W, \mathsf{v_1}, \mathbf{v_2}) \mid \forall W' \sqsupseteq W. (W', \mathsf{v_1}, \mathbf{v_2}) \in R\}
\end{aligned}$$

for R a world-values relation

The Back-translation Type and Pseudo Types We index the logical relation by a pseudo type, which captures all the standard types as well as the type of backtranslated stuff.

$$\hat{\tau} ::= \tau \mid \mathsf{EmulTy}$$

Function `toEmul` $(\cdot)$ takes a $\boldsymbol{\Gamma}$ and returns a $\mathsf{\Gamma}$ that has the same domain but where variables all have type Nat.

Value, Context, Expression and Environment relation

$$\begin{aligned}
\mathcal{V}\,[\![\mathsf{Bool}]\!]_{\triangledown} &\stackrel{\mathsf{def}}{=} \{(W, \mathsf{true}, \mathbf{true}), (W, \mathsf{false}, \mathbf{false})\}\\
\mathcal{V}\,[\![\mathsf{Nat}]\!]_{\triangledown} &\stackrel{\mathsf{def}}{=} \{(W, \mathsf{n}, \mathbf{n})\}\\
\mathcal{V}\left[\!\!\left[\hat{\tau} \to \hat{\tau}'\right]\!\!\right]_{\triangledown} &\stackrel{\mathsf{def}}{=} \left\{ (W, \mathsf{call\ f}, \mathbf{call\ f}) \;\middle|\; \begin{array}{l} \mathsf{f(x:\tau):\tau' \mapsto return\ e} \in \mathit{srcprog}(W) \text{ and}\\ \mathbf{f(x) \mapsto return\ e} \in \mathit{trgprog}(W)\\ \forall W', \mathsf{v}', \mathbf{v}'. \text{ if } W' \sqsupset_{\triangleright} W \text{ and } (W', \mathsf{v}', \mathbf{v}') \in \mathcal{V}\,[\![\hat{\tau}]\!]_{\triangledown} \text{ then}\\ \quad (W', \mathsf{return\ e[v/x]}, \mathbf{return\ e[v/x]}) \in \mathcal{E}\left[\!\!\left[\hat{\tau}'\right]\!\!\right]_{\triangledown} \end{array} \right\}\\
\mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\triangledown} &\stackrel{\mathsf{def}}{=} \{(W, \mathsf{n+2}, \mathbf{n}), (W, \mathsf{1}, \mathbf{true}), (W, \mathsf{0}, \mathbf{false})\}
\end{aligned}$$

$$\mathcal{K}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \stackrel{\text{def}}{=} \left\{ (W, \mathbb{E}, \mathbb{E}) \;\middle|\; \begin{array}{l} \forall W', \mathsf{v}, \mathbf{v}.\ \text{if } W' \sqsupseteq W \text{ and } (W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \text{ then} \\ (\mathbb{E}\left[\mathsf{v}\right], \mathbb{E}\left[\mathbf{v}\right]) \in O(W')_{\triangledown} \end{array} \right\}$$

$$\mathcal{E}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \stackrel{\text{def}}{=} \{(W, \mathsf{t}, \mathbf{t}) \mid \forall \mathbb{E}, \mathbb{E}.\ \text{if } (W, \mathbb{E}, \mathbb{E}) \in \mathcal{K}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \text{ then } (\mathbb{E}\left[\mathsf{t}\right], \mathbb{E}\left[\mathbf{t}\right]) \in O(W)_{\triangledown}\}$$

$$\mathcal{G}\left[\!\left[\varnothing\right]\!\right]_{\triangledown} \stackrel{\text{def}}{=} \{(W, \varnothing, \varnothing)\}$$

$$\mathcal{G}\left[\!\left[\hat{\Gamma}, \mathsf{x} : \hat{\tau}\right]\!\right]_{\triangledown} \stackrel{\text{def}}{=} \left\{ (W, \gamma[\mathsf{v}/\mathsf{x}], \gamma[\mathbf{v}/\mathbf{x}]) \;\middle|\; (W, \gamma, \gamma) \in \mathcal{G}\left[\!\left[\hat{\Gamma}\right]\!\right]_{\triangledown} \text{ and } (W, \mathsf{v}, \mathbf{v}) \in \mathcal{V}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \right\}$$

Relation for Open and Closed Terms and Programs

Definition G.8 (Logical relation up to n steps)**.**

$$\begin{aligned} \hat{\Gamma}; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \hat{\tau} \stackrel{\text{def}}{=}\ & \hat{\Gamma}; \mathsf{P} \vdash \mathsf{e} : \hat{\tau} \\ & \text{and } \ \forall W. \\ & \quad \text{if } \ \mathit{lev}(W) \geq n \text{ and } \mathit{progs}(W) = (\mathsf{P}, \mathbf{P}) \\ & \text{then } \ \forall \gamma, \gamma.\ (W, \gamma, \gamma) \in \mathcal{G}\left[\!\left[\hat{\Gamma}\right]\!\right]_{\triangledown}, \\ & \qquad (W, \mathsf{e}\gamma, \mathbf{e}\gamma) \in \mathcal{E}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \end{aligned}$$

Definition G.9 (Logical relation for expressions)**.**

$$\hat{\Gamma}; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown \mathbf{e} : \hat{\tau} \stackrel{\text{def}}{=} \forall n \in \mathbb{N}.\ \hat{\Gamma}; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \hat{\tau}$$

Definition G.10 (Logical relation for programs)**.**

$$\begin{aligned} \vdash \mathsf{P} \triangledown \mathbf{P} \stackrel{\text{def}}{=}\ & \mathsf{f}(\mathsf{x} : \sigma') : \sigma \mapsto \mathsf{return}\ \mathsf{e} \in \mathsf{P} \text{ iff } \mathbf{f}(\mathbf{x}) \mapsto \mathbf{return}\ \mathbf{e} \in \mathbf{P} \\ & \mathsf{x} : \sigma'; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown \mathbf{e} : \sigma \end{aligned}$$

Auxiliary Lemmas from Existing Work

Lemma G.11 (No observation with 0 steps)**.**

$$\begin{aligned} \text{if } \ & \mathit{lev}(W) = 0 \\ \text{then } \ & \forall \mathsf{e}, \mathbf{e}. (\mathsf{e}, \mathbf{e}) \in O(W)_{\triangledown} \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.12 (No steps means relation)**.**

$$\begin{aligned} \text{if } \ & \mathit{lev}(W) = n \\ & \mathsf{P} \triangleright \mathsf{e} \ \xRightarrow{\beta}{}^{\mathsf{n}} \ \_ \\ & \mathbf{P} \triangleright \mathbf{e} \ \xRightarrow{\beta}{}^{\mathbf{n}} \ \_ \\ \text{then } \ & (\mathsf{e}, \mathbf{e}) \in O(W)_{\triangledown} \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.13 (Later preserves monotonicity)**.**

$$\begin{aligned} \text{if } \ & \forall R, R \subseteq \nearrow(R) \\ \text{then } \ & \triangleright R \subseteq \nearrow(\triangleright R) \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.14 (Monotonicity for environment relation)**.**

$$\begin{aligned} \text{if } \ & W' \sqsupseteq W \\ & (W, \gamma, \gamma) \in \mathcal{G}\left[\!\left[\Gamma\right]\!\right]_{\triangledown} \\ \text{then } \ & (W', \gamma, \gamma) \in \mathcal{G}\left[\!\left[\Gamma\right]\!\right]_{\triangledown} \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.15 (Monotonicity for continuation relation)**.**

$$\begin{aligned} \text{if } \ & W' \sqsupseteq W \\ & (W, \mathbb{C}, \mathbb{C}) \in \mathcal{K}\left[\!\left[\hat{\tau}\right]\!\right]_{\triangledown} \end{aligned}$$

$$\text{then} \quad (W', \mathbb{C}, \mathbb{C}) \in \mathcal{K}\llbracket \hat{\tau} \rrbracket_{\triangledown}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.16 (Monotonicity for value relation)**.**

$$\mathcal{V}\llbracket \hat{\tau} \rrbracket_{\triangledown} \subseteq \nearrow(\mathcal{V}\llbracket \hat{\tau} \rrbracket_{\triangledown})$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.17 (Value relation implies term relation)**.**

$$\forall \hat{\tau}, \mathcal{V}\llbracket \hat{\tau} \rrbracket_{\triangledown} \subseteq \mathcal{E}\llbracket \hat{\tau} \rrbracket_{\triangledown}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.18 (Adequacy for $\lesssim$)**.**

$$\begin{aligned} &\text{if } \varnothing; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \lesssim_n \mathbf{e} : \tau \\ &\quad \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta}{}^{\mathsf{m}} \mathsf{P} \triangleright \mathsf{e}' \text{ with } n \geq m \\ &\text{then } \quad \mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta} \mathbf{P} \triangleright \_. \end{aligned}$$

Proof. By Definition G.9 (Logical relation for expressions) we have that $(W, \mathsf{e}, \mathbf{e}) \in \mathcal{E}\llbracket \tau \rrbracket_{\lesssim}$ for a W such that $lev(W) = n$.
By taking $(W, [\cdot], [\cdot]) \in \mathcal{K}\llbracket \tau \rrbracket_{\lesssim}$ we know that $(\mathsf{e}, \mathbf{e}) \in O(W)_{\lesssim}$.
By definition of $O(\cdot)_{\lesssim}$, with the HP of the source reduction, we conclude the thesis. □

Lemma G.19 (Adequacy for $\gtrsim$)**.**

$$\begin{aligned} &\text{if } \varnothing; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \gtrsim_n \mathbf{e} : \tau \\ &\quad \mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta}{}^{\mathbf{m}} \mathbf{P} \triangleright \mathbf{e}'. \text{ with } n \geq m \\ &\text{then } \quad \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta} \mathsf{P} \triangleright \_ \end{aligned}$$

Proof. By Definition G.9 (Logical relation for expressions) we have that $(W, \mathsf{e}, \mathbf{e}) \in \mathcal{E}\llbracket \tau \rrbracket_{\gtrsim}$ for a W such that $lev(W) = n$.
By taking $(W, [\cdot], [\cdot]) \in \mathcal{K}\llbracket \tau \rrbracket_{\gtrsim}$ we know that $(\mathsf{e}, \mathbf{e}) \in O(W)_{\gtrsim}$.
By definition of $O(\cdot)_{\gtrsim}$, with the HP of the target reduction, we conclude the thesis. □

Lemma G.20 (Observation relation is closed under antireduction)**.**

$$\begin{aligned} &\text{if } \quad \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta}{}^{\mathsf{i}} \mathsf{P} \triangleright \mathsf{e}' \\ &\quad \mathbf{P} \triangleright \mathbf{e} \xRightarrow{\beta}{}^{\mathbf{j}} \mathbf{P} \triangleright \mathbf{e}' \\ &\quad (\mathsf{e}', \mathbf{e}') \in O(W')_{\triangledown} \text{ for } W' \sqsupseteq W \\ &\quad progs(W) = progs(W') = (\mathsf{P}, \mathbf{P}) \\ &\quad lev(W') \geq lev(W) - \mathtt{min}\,(i, j) \\ &\quad (\text{ that is: } lev(W) \leq lev(W') + \mathtt{min}\,(i, j)) \\ &\text{then } \quad (\mathsf{e}, \mathbf{e}) \in O(W)_{\triangledown} \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.21 (Closedness under antireduction)**.**

$$\begin{aligned} &\text{if } \quad \mathsf{P} \triangleright \mathbb{C}[\mathsf{e}] \xRightarrow{\beta}{}^{\mathsf{i}} \mathsf{P} \triangleright \mathbb{C}[\mathsf{e}'] \\ &\quad \mathbf{P} \triangleright \mathbb{C}[\mathbf{e}] \xRightarrow{\beta}{}^{\mathbf{i}} \mathbf{P} \triangleright \mathbb{C}[\mathbf{e}'] \\ &\quad (W', \mathsf{e}', \mathbf{e}') \in \mathcal{E}\llbracket \hat{\tau} \rrbracket_{\triangledown} \\ &\quad W' \sqsupseteq W \\ &\quad lev(W') \geq lev(W) - \mathtt{min}\,(i, j) \\ &\quad (\text{ that is } lev(W) \leq lev(W') + \mathtt{min}\,(i, j)) \\ &\text{then } \quad (W, \mathsf{e}, \mathbf{e}) \in \mathcal{E}\llbracket \hat{\tau} \rrbracket_{\triangledown} \end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.22 (Related terms plugged in related contexts are still related)**.**

$$\begin{aligned}
\text{if}\quad & (W, \mathsf{e}, \mathbf{e}) \in \mathcal{E}\left[\!\left[\hat{\tau}'\right]\!\right]_{\triangledown} \\
& \text{and if } W' \sqsupseteq W \\
& (W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}\left[\!\left[\hat{\tau}'\right]\!\right]_{\triangledown} \\
& \text{then } (W', \mathbb{C}[\mathsf{v}], \mathbb{C}[\mathbf{v}]) \in \mathcal{E}[\![\hat{\tau}]\!]_{\triangledown} \\
\text{then}\quad & (W, \mathbb{C}[\mathsf{e}], \mathbb{C}[\mathbf{e}]) \in \mathcal{E}[\![\hat{\tau}]\!]_{\triangledown}
\end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Lemma G.23 (Related functions applied to related arguments are related terms)**.**

$$\begin{aligned}
\text{if}\quad & (W, \mathsf{v}, \mathbf{v}) \in \mathcal{V}\left[\!\left[\hat{\tau}' \to \hat{\tau}\right]\!\right]_{\triangledown} \\
& (W, \mathsf{v}', \mathbf{v}') \in \mathcal{V}\left[\!\left[\hat{\tau}'\right]\!\right]_{\triangledown} \\
\text{then}\quad & (W, \mathsf{v}\ \mathsf{v}', \mathbf{v}\ \mathbf{v}') \in \mathcal{E}[\![\hat{\tau}]\!]_{\triangledown}
\end{aligned}$$

Proof. Trivial adaptation of the same proof in [35, 36]. □

Auxiliary Results

Lemma G.24 (If Extract reduces, it preserves relatedness)**.**

$$\begin{aligned}
\text{if}\quad & (W, \mathsf{v}, \mathbf{v}) \in \mathcal{V}[\![\mathsf{EmulTy}]\!]_{\triangledown} \\
& \mathsf{P} \triangleright \mathsf{extract}_{\sigma}(\mathsf{v}) \hookrightarrow^{*} \mathsf{P} \triangleright \mathsf{v}' \\
\text{then}\quad & (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}[\![\sigma]\!]_{\triangledown}
\end{aligned}$$

Proof. Trivial case analysis:

$\sigma = \mathsf{Bool}$ means that v=$\mathsf{0}$ or $\mathsf{1}$, so by definition of $\mathcal{V}[\![\mathsf{EmulTy}]\!]_{\triangledown}$ $\mathbf{v}$=**false** or **true** (respectively).
Consider the $\mathsf{0}$ and **false** case, the other is analogous.
By definition the reduction of extract goes as follows.

$$\begin{aligned}
& \mathsf{P} \triangleright \mathsf{extract}_{\mathsf{Bool}} \mathsf{0} \\
\equiv & \mathsf{P} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{0}\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x}+1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false} \\
\hookrightarrow\ \hookrightarrow & \mathsf{P} \triangleright \mathsf{if}\ 1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false} \\
\hookrightarrow & \mathsf{P} \triangleright \mathsf{false}
\end{aligned}$$

We need to show that $(W, \mathsf{false}, \mathbf{false}) \in \mathcal{V}[\![\mathsf{Bool}]\!]_{\triangledown}$, which follows from its definition.

$\sigma = \mathsf{Nat}$ means that v=$\mathsf{n}+2$ and $\mathbf{v}$=$\mathbf{n}$
By definition the reduction of extract goes as follows. (we write n+2 as a value, not as an expression to simplify this)

$$\begin{aligned}
& \mathsf{P} \triangleright \mathsf{extract}_{\mathsf{Nat}} \mathsf{n}+2 \\
\equiv & \mathsf{P} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{n}+2\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{x}-2\ \mathsf{else}\ \mathsf{fail} \\
\hookrightarrow & \mathsf{P} \triangleright \mathsf{if}\ \mathsf{n}+2 \geq 2\ \mathsf{then}\ \mathsf{x}-2\ \mathsf{else}\ \mathsf{fail} \\
\hookrightarrow & \mathsf{P} \triangleright \mathsf{n}
\end{aligned}$$

We need to show that $(W, \mathsf{n}, \mathbf{n}) \in \mathcal{V}[\![\mathsf{Nat}]\!]_{\triangledown}$, which follows from its definition.

□

Lemma G.25 (Inject reduces and preserves relatedness)**.**

$$\begin{aligned}
\text{if}\quad & (W, \mathsf{v}, \mathbf{v}) \in \mathcal{V}[\![\sigma]\!]_{\triangledown} \\
& \mathsf{P} \triangleright \mathsf{inject}_{\sigma} \mathsf{v} \hookrightarrow^{*} \mathsf{P} \triangleright \mathsf{v}' \\
\text{then}\quad & (W, \mathsf{v}', \mathbf{v}) \in \mathcal{V}[\![\mathsf{EmulTy}]\!]_{\triangledown}
\end{aligned}$$

Proof. Trivial case analysis on σ.

$\sigma = \mathsf{Bool}$ By definition of $\mathcal{V}[\![\mathsf{Bool}]\!]_{\triangledown}$ we have v=true and $\mathbf{v}$=**true** or false/**false**. We consider the first case only, the second is analogous.
By definition of inject we have:

$$\mathsf{P} \triangleright \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 1\ \mathsf{else}\ 0$$

$$\hookrightarrow \mathsf{P} \triangleright 1$$

So we need to prove that $(W, 1, \mathbf{true}) \in \mathcal{V}\left[\!\left[\mathsf{EmulTy}\right]\!\right]_{\triangledown}$ which follows from its definition.

$\sigma = \mathsf{Nat}$ By definition of $\mathcal{V}\left[\!\left[\mathsf{Nat}\right]\!\right]_{\triangledown}$ we have $\mathsf{v}{=}\mathsf{n}$ and $\mathbf{v}{=}\mathbf{n}$.

By definition of inject, we have:

$$\mathsf{P} \triangleright \mathsf{n} + 2$$
$$\hookrightarrow \mathsf{P} \triangleright \mathsf{n} + 2$$

(we keep the value as a sum for simplicity)

So we need to prove that $(W, \mathsf{n}+2, \mathbf{n}) \in \mathcal{V}\left[\!\left[\mathsf{EmulTy}\right]\!\right]_{\triangledown}$ which follows from its definition.

□

Compatibility Lemmas for τ Types

Lemma G.26 (Compatibility lemma for calls)**.**

$$\begin{aligned} \text{if } & \Gamma, \mathsf{x} : \sigma'; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \sigma \\ & \mathsf{f}(\mathsf{x} : \sigma') : \sigma \mapsto \mathsf{return}\ \mathsf{e} \in \mathsf{P} \\ & \mathbf{f}(\mathbf{x}) \mapsto \mathbf{return\ if\ x\ has}\ \sigma'\ \mathbf{then\ e\ else\ fail} \in \mathbf{P} \\ \text{then } & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{call}\ \mathsf{f} \triangledown_n \mathbf{call\ f} : \sigma' \to \sigma \end{aligned}$$

Proof. We need to prove that

$$\Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{call}\ \mathsf{f} \triangledown_n \mathbf{call\ f} : \sigma' \to \sigma$$

Take W such that $lev(W) \leq n$ and HG $(W, \gamma, \gamma) \in \mathcal{G}\left[\!\left[\mathtt{toEmul}\left(\boldsymbol{\Gamma}\right)\right]\!\right]_{\triangledown}$, the thesis is:

- $(W, \mathsf{call}\ \mathsf{f}, \mathbf{call\ f}) \in \mathcal{E}\left[\!\left[\sigma' \to \sigma\right]\!\right]_{\triangledown}$

By Lemma G.17 (Value relation implies term relation) the thesis is:

- $(W, \mathsf{call}\ \mathsf{f}, \mathbf{call\ f}) \in \mathcal{V}\left[\!\left[\sigma' \to \sigma\right]\!\right]_{\triangledown}$

By definition of the $\mathcal{V}\left[\!\left[\cdot\right]\!\right]_{\triangledown}$ we take HV $(W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}\left[\!\left[\sigma'\right]\!\right]_{\triangledown}$ such that $W' \sqsupseteq_{\triangleright} W$ and the thesis is:

- $(W', \mathsf{return}\ \mathsf{e}[\mathsf{v}/\mathsf{x}]\gamma, \mathbf{return\ if\ x\ has}\ \sigma'\ \mathbf{then\ e\ else\ fail}[\mathbf{v}/\mathbf{x}]\gamma) \in \mathcal{E}\left[\!\left[\sigma\right]\!\right]_{\triangledown}$

The reductions proceed as:

$$\begin{aligned} & \mathbf{P} \triangleright \mathbf{return\ if\ x\ has}\ \sigma'\ \mathbf{then\ e\ else\ fail}[\mathbf{v}/\mathbf{x}]\gamma \\ \equiv & \mathbf{P} \triangleright \mathbf{return\ if\ v\ has}\ \sigma'\ \mathbf{then}\ (\mathbf{e}[\mathbf{v}/\mathbf{x}]\gamma)\ \mathbf{else\ fail} \\ \hookrightarrow\ & \mathbf{P} \triangleright \mathbf{return\ if\ true\ then}\ (\mathbf{e}[\mathbf{v}/\mathbf{x}]\gamma)\ \mathbf{else\ fail} \\ \hookrightarrow\ & \mathbf{P} \triangleright \mathbf{return}\ (\mathbf{e}[\mathbf{v}/\mathbf{x}]\gamma) \end{aligned}$$

By Lemma G.21 the thesis becomes:

- $(W', \mathsf{return}\ \mathsf{e}[\mathsf{v}/\mathsf{x}]\gamma, \mathbf{return\ e}[\mathbf{v}/\mathbf{x}]\gamma) \in \mathcal{E}\left[\!\left[\sigma\right]\!\right]_{\triangledown}$

This follows from the definition of logical relation if

- $(W', [\mathsf{v}/\mathsf{x}]\gamma, [\mathbf{v}/\mathbf{x}]\gamma) \in \mathcal{G}\left[\!\left[\Gamma, \mathsf{x} : \sigma'\right]\!\right]_{\triangledown}$

This follows from HG with Lemma G.14 and by HV and Lemma G.16 and by the definition of $\mathcal{G}\left[\!\left[\cdot\right]\!\right]_{\triangledown}$. □

Lemma G.27 (Compatibility lemma for application)**.**

$$\begin{aligned} \text{if } & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \sigma' \to \sigma \\ & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}' \triangledown_n \mathbf{e}' : \sigma' \\ \text{then } & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}\ \mathsf{e}' \triangledown_n \mathbf{e}\ \mathbf{e}' : \sigma \end{aligned}$$

Proof. This is standard using Lemma G.17, Lemma G.16, Lemma G.22 and Lemma G.21. □

Lemma G.28 (Compatibility lemma for op)**.**

$$\begin{aligned} \text{if } & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \mathsf{Nat} \\ & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}' \triangledown_n \mathbf{e}' : \mathsf{Nat} \\ \text{then } & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \oplus \mathsf{e}' \triangledown_n \mathbf{e} \oplus \mathbf{e}' : \mathsf{Nat} \end{aligned}$$

Proof. This is standard and analogous to the proof of Lemma G.27. □

Lemma G.29 (Compatibility lemma for geq)**.**

$$\begin{aligned} \text{if} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \; \nabla_n \; \mathbf{e} : \mathsf{Nat} \\ & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}' \; \nabla_n \; \mathbf{e}' : \mathsf{Nat} \\ \text{then} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \geq \mathsf{e}' \; \nabla_n \; \mathbf{e} \geq \mathbf{e}' : \mathsf{Bool} \end{aligned}$$

Proof. This is standard and analogous to the proof of Lemma G.27. ☐

Lemma G.30 (Compatibility lemma for letin)**.**

$$\begin{aligned} \text{if} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \; \nabla_n \; \mathbf{e} : \sigma \\ & \Gamma, \mathsf{x} : \sigma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}' \; \nabla_n \; \mathbf{e}' : \sigma' \\ \text{then} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e}' \; \nabla_n \; \mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e}' : \sigma' \end{aligned}$$

Proof. This is standard and analogous to the proof of Lemma G.27. ☐

Lemma G.31 (Compatibility lemma for if)**.**

$$\begin{aligned} \text{if} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \; \nabla_n \; \mathbf{e} : \mathsf{Bool} \\ & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}' \; \nabla_n \; \mathbf{e}' : \sigma \\ & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e}'' \; \nabla_n \; \mathbf{e}'' : \sigma \\ \text{then} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}'' \; \nabla_n \; \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'' : \sigma \end{aligned}$$

Proof. This is standard and analogous to the proof of Lemma G.27. ☐

Lemma G.32 (Compatibility lemma for read)**.**

$$\begin{aligned} \text{if} \quad & \\ \text{then} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{read} \; \nabla_n \; \mathbf{read} : \mathsf{Nat} \end{aligned}$$

Proof. By definition of the $O(W)_\triangledown$. ☐

Lemma G.33 (Compatibility lemma for write)**.**

$$\begin{aligned} \text{if} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \; \nabla_n \; \mathbf{e} : \mathsf{Nat} \\ \text{then} \quad & \Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{write}\ \mathsf{e} \; \nabla_n \; \mathbf{write}\ \mathbf{e} : \mathsf{Nat} \end{aligned}$$

Proof. We need to prove that

$$\Gamma; \mathsf{P}; \mathbf{P} \vdash \mathsf{write}\ \mathsf{e} \; \nabla_n \; \mathbf{write}\ \mathbf{e} : \mathsf{Nat}$$

Take W such that $lev(W) \leq n$ and $(W, \gamma, \boldsymbol{\gamma}) \in \mathcal{G}\,[\![\texttt{toEmul}\,(\boldsymbol{\Gamma})]\!]_\triangledown$, the thesis is: (we omit substitutions as they don't play an active role)

- $(W, \mathsf{write}\ \mathsf{e}, \mathbf{write}\ \mathbf{e}) \in \mathcal{E}\,[\![\mathsf{Nat}]\!]_\triangledown$

By Lemma G.22 (Related terms plugged in related contexts are still related) with HE, we have that for HW $W' \sqsupseteq W$, and HV $(W', \mathsf{n}, \mathbf{n}) \in \mathcal{V}\,[\![\mathsf{Nat}]\!]_\triangledown$, the thesis becomes:

- $(W', \mathsf{write}\ \mathsf{n}, \mathbf{write}\ \mathbf{n}) \in \mathcal{E}\,[\![\mathsf{Nat}]\!]_\triangledown$

The reductions proceed as:

$$\mathsf{P} \triangleright \mathsf{write}\ \mathsf{n} \xrightarrow{\texttt{write n}} \mathsf{P} \triangleright \mathsf{n}$$

and

$$\mathbf{P} \triangleright \mathbf{write}\ \mathbf{n} \xrightarrow{\texttt{write n}} \mathbf{P} \triangleright \mathbf{n}$$

By Lemma G.21 (Closedness under antireduction) the thesis is:

- $(W', \mathsf{n}, \mathbf{n}) \in \mathcal{E}\,[\![\mathsf{Nat}]\!]_\triangledown$

So the theorem holds by Lemma G.17 (Value relation implies term relation) with HV. ☐

Semantic Preservation Results

Theorem G.34 ($\cdot\!\downarrow$ is semantics preserving for expressions)**.**

$$\begin{aligned} \text{if} \quad & \mathsf{P}; \Gamma \vdash \mathsf{e} : \tau \\ & \vdash \mathsf{P} \; \nabla_n \; \mathbf{P} \end{aligned}$$

$$\text{then} \quad \forall n.\ \Gamma; P; \mathbf{P} \vdash e \ \nabla_n \ e\!\downarrow : \tau$$

Proof. The proof proceeds by induction on the type derivation.

true, false, nat By definition of $\mathcal{V}[\![\cdot]\!]_\nabla$.
var By definition of $\mathcal{G}[\![\cdot]\!]_\nabla$.
call By Lemma G.26 (Compatibility lemma for calls).
app By IH with Lemma G.27 (Compatibility lemma for application).
op By IH with Lemma G.28 (Compatibility lemma for op).
geq By IH with Lemma G.29 (Compatibility lemma for geq).
letin By IH with Lemma G.30 (Compatibility lemma for letin).
if By IH with Lemma G.31 (Compatibility lemma for if).
read By Lemma G.32 (Compatibility lemma for read).
write By IH with Lemma G.33 (Compatibility lemma for write).

□

Theorem G.35 ($\cdot\!\downarrow$ is semantics preserving for programs)**.**

$$\begin{aligned} &\text{if} && \vdash P \\ &\text{then} && \vdash P \ \nabla \ P\!\downarrow \end{aligned}$$

Proof. By induction on the size of P and then Section G.3 and with Theorem G.34 ($\cdot\!\downarrow$ is semantics preserving for expressions) on each function body. □

Compatibility Lemmas for Pseudo Types

Lemma G.36 (Compatibility lemma for backtranslation of op)**.**

$$\begin{aligned} &\text{if} && (HE)\ \texttt{toEmul}(\mathbf{\Gamma}); P; \mathbf{P} \vdash e \ \nabla_n \ \mathbf{e} : \mathsf{EmulTy} \\ & && (HEP)\ \texttt{toEmul}(\mathbf{\Gamma}); P; \mathbf{P} \vdash e' \ \nabla_n \ \mathbf{e'} : \mathsf{EmulTy} \\ &\text{then} && \texttt{toEmul}(\mathbf{\Gamma}); P; \mathbf{P} \vdash \begin{array}{l}\mathsf{let}\ x1 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e) \\ \mathsf{in}\ \mathsf{let}\ x2 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e') \\ \mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(x1 \oplus x2)\end{array} \ \nabla_n \ \mathbf{e} \oplus \mathbf{e'} : \mathsf{EmulTy} \end{aligned}$$

Proof. We need to prove that

$$\texttt{toEmul}(\mathbf{\Gamma}); P; \mathbf{P} \vdash \begin{array}{l}\mathsf{let}\ x1 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e) \\ \mathsf{in}\ \mathsf{let}\ x2 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e') \\ \mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(x1 \oplus x2)\end{array} \ \nabla \ \mathbf{e} \oplus \mathbf{e'} : \mathsf{EmulTy}$$

Take W such that $lev(W) \leq n$ and $(W, \gamma, \boldsymbol{\gamma}) \in \mathcal{G}[\![\texttt{toEmul}(\mathbf{\Gamma})]\!]_\nabla$, the thesis is:

- $\left(W, \begin{array}{l}\mathsf{let}\ x1 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e) \\ \mathsf{in}\ \mathsf{let}\ x2 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e') \\ \mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(x1 \oplus x2)\end{array}, \mathbf{e} \oplus \mathbf{e'}\right) \in \mathcal{E}[\![\mathsf{EmulTy}]\!]_\nabla$

By Lemma G.22 (Related terms plugged in related contexts are still related) with HE we need to prove that $\forall W' \sqsupseteq W$, given IHV $(W', v, \mathbf{v}) \in \mathcal{V}[\![\mathsf{EmulTy}]\!]_\nabla$

- $\left(W', \begin{array}{l}\mathsf{let}\ x1 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(v) \\ \mathsf{in}\ \mathsf{let}\ x2 : \mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(e') \\ \mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(x1 \oplus x2)\end{array}, \mathbf{v} \oplus \mathbf{e'}\right) \in \mathcal{E}[\![\mathsf{EmulTy}]\!]_\nabla$

By IHV we perform a case analysis on $\mathbf{v}$:

- **true**/ **false** and thus v is 1/0 respectively.
  We show the case for **true**, 1 the other is analogous.
  In this case we have:

  $$\mathbf{P} \triangleright \mathbf{true} \oplus \mathbf{e'} \xRightarrow{\perp} \mathbf{fail}$$

  and

  $$\begin{aligned} &P \triangleright \mathsf{extract}_{\mathsf{Nat}}(1) \\ &\equiv \mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{if}\ x \geq 2\ \mathsf{then}\ x - 2\ \mathsf{else}\ \mathsf{fail} \\ &\hookrightarrow \mathsf{if}\ 1 \geq 2\ \mathsf{then}\ x - 2\ \mathsf{else}\ \mathsf{fail} \end{aligned}$$

$$\stackrel{\perp}{\Longrightarrow} \mathsf{fail}$$

So this case follows from the definition of $O(W')_{\triangledown}$ as both terms perform the same visible action ($\perp$).

- $\mathbf{n}$ and thus v is $\mathsf{n}+2$.
In this case we have:

$$\begin{aligned}
&\mathsf{P} \triangleright \mathsf{extract}_{\mathsf{Nat}}(\mathsf{n}+2)\\
\equiv&\mathsf{let}\ \mathsf{x}=\mathsf{n}+2\ \mathsf{in}\ \mathsf{if}\ \mathsf{x}\geq 2\ \mathsf{then}\ \mathsf{x}-2\ \mathsf{else}\ \mathsf{fail}\\
\hookrightarrow&\ \mathsf{if}\ \mathsf{n}+2\geq 2\ \mathsf{then}\ \mathsf{x}-2\ \mathsf{else}\ \mathsf{fail}\\
\hookrightarrow&\ \mathsf{n}
\end{aligned}$$

And by Lemma G.24 (If Extract reduces, it preserves relatedness) with IHV we know that IHN $(W', \mathsf{n}, \mathbf{n}) \in \mathcal{V}[\![\mathsf{Nat}]\!]_{\triangledown}$.
Analogously, e' and $\mathbf{e}'$ follow the same treatment. So we apply Lemma G.22 (Related terms plugged in related contexts are still related) with HEP, perform a case analysis, in one case they fail and in the other they reduce to $\mathsf{n}'/\mathbf{n}'$ such that IHNP $(W', \mathsf{n}', \mathbf{n}') \in \mathcal{V}[\![\mathsf{Nat}]\!]_{\triangledown}$.
So the reductions are :

$$\begin{aligned}
&\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x1}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e})\ \mathsf{in}\ \mathsf{let}\ \mathsf{x2}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}')\\
&\quad\mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(\mathsf{x1}\oplus\mathsf{x2})\\
\hookrightarrow^*\ &\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x1}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{n})\ \mathsf{in}\ \mathsf{let}\ \mathsf{x2}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}')\\
&\quad\mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(\mathsf{x1}\oplus\mathsf{x2})\\
\hookrightarrow\ &\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x2}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}')\\
&\quad\mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}\oplus\mathsf{x2})\\
\hookrightarrow^*\ &\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x2}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{n}')\\
&\quad\mathsf{in}\ \mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}\oplus\mathsf{x2})\\
\hookrightarrow\ &\mathsf{P}\triangleright\mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}\oplus\mathsf{n}')
\end{aligned}$$

and

$$\mathbf{P}\triangleright\mathbf{e}\oplus\mathbf{e}' \hookrightarrow^* \mathbf{P}\triangleright\mathbf{n}\oplus\mathbf{e}' \hookrightarrow^* \mathbf{P}\triangleright\mathbf{n}\oplus\mathbf{n}'$$

By Lemma G.21 (Closedness under antireduction) the thesis becomes:
– $(W', \mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}\oplus\mathsf{n}'), \mathbf{n}\oplus\mathbf{n}') \in \mathcal{E}[\![\mathsf{EmulTy}]\!]_{\triangledown}$
If the $lev(W')=0$ the thesis follows from Lemma G.12 (No steps means relation), otherwise:
By Rule EL$^{\tau}$-op and Rule EL$^{\mathbf{u}}$-op we can apply Lemma G.21 (Closedness under antireduction) (with IHN and IHNP in the term relation by Lemma G.17 (Value relation implies term relation)) and the thesis becomes:
– $(W', \mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}''), \mathbf{n}'') \in \mathcal{E}[\![\mathsf{EmulTy}]\!]_{\triangledown}$
The reductions proceed as follows:

$$\mathsf{P}\triangleright\mathsf{inject}_{\mathsf{Nat}}(\mathsf{n}'') \hookrightarrow \mathsf{P}\triangleright\mathsf{n}''+2$$

By Lemma G.21 (Closedness under antireduction) and then Lemma G.17 (Value relation implies term relation) the thesis becomes:
– $(W', \mathsf{n}'''+2, \mathbf{n}'') \in \mathcal{V}[\![\mathsf{EmulTy}]\!]_{\triangledown}$
By Lemma G.25 (Inject reduces and preserves relatedness) the thesis becomes:
– $(W', \mathsf{n}'', \mathbf{n}'') \in \mathcal{V}[\![\mathsf{Nat}]\!]_{\triangledown}$
which follows from the definition of $\mathcal{V}[\![\mathsf{Nat}]\!]_{\triangledown}$.

□

Lemma G.37 (Compatibility lemma for backtranslation of geq)**.**

$$\begin{aligned}
\text{if}\quad &\mathtt{toEmul}\,(\mathbf{\Gamma});\mathsf{P};\mathbf{P}\vdash\mathsf{e}\ \triangledown_n\ \mathbf{e}:\mathsf{EmulTy}\\
&\mathtt{toEmul}\,(\mathbf{\Gamma});\mathsf{P};\mathbf{P}\vdash\mathsf{e}'\ \triangledown_n\ \mathbf{e}':\mathsf{EmulTy}\\
\text{then}\quad &\mathtt{toEmul}\,(\mathbf{\Gamma});\mathsf{P};\mathbf{P}\vdash\mathsf{let}\ \mathsf{x1}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e})\qquad \triangledown_n\ \mathbf{e}\geq\mathbf{e}':\mathsf{EmulTy}\\
&\qquad\mathsf{in}\ \mathsf{let}\ \mathsf{x2}:\mathsf{Nat}{=}\mathsf{extract}_{\mathsf{Nat}}(\mathsf{e}')\\
&\qquad\mathsf{in}\ \mathsf{inject}_{\mathsf{Bool}}(\mathsf{x1}\geq\mathsf{x2})
\end{aligned}$$

Proof. Analogous to the proof of Lemma G.36. □

Lemma G.38 (Compatibility lemma for backtranslation of letin)**.**

$$\begin{aligned}\text{if}\quad & \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{e}\ \nabla_n\ \mathbf{e}:\mathsf{EmulTy}\\ & \texttt{toEmul}\,(\Gamma),\mathsf{x}:\mathsf{Nat};\mathsf{P};\mathbf{P}\vdash \mathsf{e}'\ \nabla_n\ \mathbf{e}':\mathsf{EmulTy}\\ \text{then}\quad & \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{let}\ \mathsf{x}:\mathsf{Nat}=\mathsf{e}\ \mathsf{in}\ \mathsf{e}'\ \nabla_n\ \mathbf{let}\ \mathbf{x}=\mathbf{e}\ \mathbf{in}\ \mathbf{e}':\mathsf{EmulTy}\end{aligned}$$

Proof. This is a trivial application of Lemma G.22 (Related terms plugged in related contexts are still related) and Lemma G.21 (Closedness under antireduction) and definitions. □

Lemma G.39 (Compatibility lemma for backtranslation of if)**.**

$$\begin{aligned}\text{if}\quad & (HE)\ \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{e}\ \nabla_n\ \mathbf{e}:\mathsf{EmulTy}\\ & (HEP)\ \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{e}'\ \nabla_n\ \mathbf{e}':\mathsf{EmulTy}\\ & \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{e}''\ \nabla_n\ \mathbf{e}'':\mathsf{EmulTy}\\ \text{then}\quad & \texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{if}\ \mathsf{extract_{Bool}}(\mathsf{e})\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}''\ \nabla_n\ \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'':\mathsf{EmulTy}\end{aligned}$$

Proof. We need to prove that

$$\texttt{toEmul}\,(\Gamma);\mathsf{P};\mathbf{P}\vdash \mathsf{if}\ \mathsf{extract_{Bool}}(\mathsf{e})\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}''\ \nabla\ \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'':\mathsf{EmulTy}$$

Take W such that $lev(W)\leq n$ and $(W,\gamma,\gamma)\in\mathcal{G}\,[\![\texttt{toEmul}\,(\Gamma)]\!]_{\triangledown}$, the thesis is: (we omit substitutions as they don't play an active role)

- $(W,\mathsf{if}\ \mathsf{extract_{Bool}}(\mathsf{e})\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}'',\mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'')\in\mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

By Lemma G.22 (Related terms plugged in related contexts are still related) with HE, we have that for HW $W'\sqsupseteq W$, and HV $(W',\mathsf{v},\mathbf{v})\in\mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$, the thesis becomes:

- $(W',\mathsf{if}\ \mathsf{extract_{Bool}}(\mathsf{v})\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}'',\mathbf{if}\ \mathbf{v}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'')\in\mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

We perform a case analysis based on HV:

- $\mathbf{v}$=$\mathbf{true}$/$\mathbf{false}$ and v=$\mathsf{1}$/$\mathsf{0}$
  We consider the case $\mathbf{true}$/$\mathsf{1}$ the other is analogous.
  The reductions proceed as follows:
  $$\begin{aligned}&\mathsf{P}\triangleright\mathsf{extract_{Bool}}(1)\\ \equiv&\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x}=1\ \mathsf{in}\ \mathsf{if}\ \mathsf{x}\geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x}+1\geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\ \hookrightarrow\ &\mathsf{P}\triangleright\mathsf{if}\ 1\geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ 1+1\geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\ \hookrightarrow\ &\mathsf{P}\triangleright\mathsf{if}\ 1+1\geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\ \hookrightarrow\quad\hookrightarrow\ &\mathsf{P}\triangleright\mathsf{true}\end{aligned}$$
  By Lemma G.21 (Closedness under antireduction) the thesis becomes:
  - $(W',\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{e}'\ \mathsf{else}\ \mathsf{e}'',\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}'')\in\mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

  If the $lev(W')=0$ the thesis follows from Lemma G.12 (No steps means relation), otherwise:
  We can reduce based on Rule EL^{τ}-if-true and Rule $\mathbf{EL^{u}}$-if-true. By Lemma G.21 (Closedness under antireduction) the thesis becomes:
  - $(W',\mathsf{e}',\mathbf{e}')\in\mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

  If the $lev(W')=0$ the thesis follows from Lemma G.12 (No steps means relation), otherwise by HEP.
- $\mathbf{v}$=$\mathbf{n}$ and v=$\mathsf{n}+2$
  In this case we have that:
  $$\begin{aligned}&\mathsf{P}\triangleright\mathsf{extract_{Bool}}(\mathsf{n}+2)\\ \equiv&\mathsf{P}\triangleright\mathsf{let}\ \mathsf{x}=\mathsf{n}+2\ \mathsf{in}\ \mathsf{if}\ \mathsf{x}\geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x}+1\geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\ \hookrightarrow\ &\mathsf{P}\triangleright\mathsf{if}\ \mathsf{n}+2\geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x}+1\geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}\\ \overset{\perp}{\Longrightarrow}\ &\mathsf{fail}\end{aligned}$$
  and
  $$\mathbf{P}\triangleright\mathbf{if}\ \mathbf{n}\ \mathbf{then}\ \mathbf{e}'\ \mathbf{else}\ \mathbf{e}''\ \overset{\perp}{\Longrightarrow}\ \mathbf{fail}$$
  So this case holds by definition of $O(W')_{\triangledown}$.

□

Lemma G.40 (Compatibility lemma for backtranslation of application)**.**

$$\begin{aligned}
\text{if}\;\; & \texttt{toEmul}\,(\Gamma); \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \mathrel{\nabla_n} \mathbf{e} : \mathsf{EmulTy} \\
& \mathsf{f}(\mathsf{x} : \sigma') : \sigma \mapsto \mathsf{return}\ \mathsf{e} \in \mathsf{P} \\
& (HP)\ \mathsf{P}; \mathbf{P} \vdash \mathsf{call}\ \mathsf{f} \mathrel{\nabla_n} \mathbf{call}\ \mathbf{f} : \sigma' \to \sigma \\
\text{then}\;\; & \texttt{toEmul}\,(\Gamma); \mathsf{P}; \mathbf{P} \vdash \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\tau}(\mathsf{e})) \mathrel{\nabla_n} \mathbf{call}\ \mathbf{f}\ \mathbf{e} : \mathsf{EmulTy}
\end{aligned}$$

Proof. We need to prove that

$$\texttt{toEmul}\,(\Gamma); \mathsf{P}; \mathbf{P} \vdash \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\tau}(\mathsf{e})) \mathrel{\nabla_n} \mathbf{call}\ \mathbf{f}\ \mathbf{e} : \mathsf{EmulTy}$$

Take W such that $lev(W) \leq n$ and $(W, \gamma, \gamma) \in \mathcal{G}\,[\![\texttt{toEmul}\,(\Gamma)]\!]_{\nabla}$, the thesis is: (we omit substitutions as they don't play an active role)

- $(W, \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\tau}(\mathsf{e})), \mathbf{call}\ \mathbf{f}\ \mathbf{e}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$

By Lemma G.22 (Related terms plugged in related contexts are still related) with HE we have that for HW $W' \sqsupseteq W$, and HV $(W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$, the thesis becomes:

- $(W, \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\tau}(\mathsf{v})), \mathbf{call}\ \mathbf{f}\ \mathbf{v}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$

We perform a case analysis based on HV:

- $\mathbf{v}$=$\mathbf{true}$/$\mathbf{false}$ and v=$\mathsf{1}$/$\mathsf{0}$ (respectively).
  We consider the first case only, the other is analogous.
  We perform a case analysis on τ:
  - τ=Bool
    The thesis is:
    * $(W', \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\mathsf{Bool}}(\mathsf{v})), \mathbf{call}\ \mathbf{f}\ \mathbf{v}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$

    By definition of $\mathsf{extract}_{\mathbf{Bool}}$ we have

    $$\begin{aligned}
    & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\mathsf{Bool}}(1)) \\
    \equiv & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{let}\ \mathsf{x} = 1\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ \mathsf{x} + 1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}) \\
    \hookrightarrow\ & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{if}\ 1 \geq 2\ \mathsf{then}\ \mathsf{fail}\ \mathsf{else}\ \mathsf{if}\ 1 + 1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}) \\
    \hookrightarrow\ & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{if}\ 1 + 1 \geq 2\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}) \\
    \hookrightarrow\ & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{true})
    \end{aligned}$$

    So by Lemma G.21 (Closedness under antireduction) the thesis becomes:
    * $(W', \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{true}), \mathbf{call}\ \mathbf{f}\ \mathbf{true}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$

    If the $lev(W') = 0$ the thesis follows from Lemma G.12 (No steps means relation), otherwise:
    By HP and by the Hs on the function bodies, and by the relatedness of true and $\mathbf{true}$ and by the Lemma G.16 (Monotonicity for value relation) we have that HF:

    $$(W', \mathsf{return}\ \mathsf{e}[\mathsf{true}/\mathsf{x}], \mathbf{return}\ \mathbf{e}[\mathbf{true}/\mathbf{x}]) \in \mathcal{E}\left[\!\left[\hat{\tau'}\right]\!\right]_{\nabla}$$

    By Lemma G.22 (Related terms plugged in related contexts are still related) with HF we have that for HW $W'' \sqsupseteq W'$, and HV $(W'', \mathsf{v}', \mathbf{v}') \in \mathcal{V}\,[\![\tau']\!]_{\nabla}$, the thesis becomes:
    * $(W', \mathsf{inject}_{\tau'}(\mathsf{v}'), \mathbf{v}') \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\nabla}$

    This case follows from Lemma G.17 (Value relation implies term relation) and by Lemma G.25 (Inject reduces and preserves relatedness) with HV.
  - τ = Nat
    By definition of $\mathsf{extract}_{\mathsf{Nat}}$ we have:

    $$\begin{aligned}
    & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{extract}_{\mathsf{Nat}}(1)) \\
    \equiv & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{let}\ \mathsf{x} = 1\ \mathsf{in}\ \mathsf{if}\ \mathsf{x} \geq 2\ \mathsf{then}\ \mathsf{x} - 2\ \mathsf{else}\ \mathsf{fail}) \\
    \hookrightarrow\ & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{if}\ 1 \geq 2\ \mathsf{then}\ 1 - 2\ \mathsf{else}\ \mathsf{fail}) \\
    \hookrightarrow\ & \mathsf{P} \triangleright \mathsf{inject}_{\tau'}(\mathsf{call}\ \mathsf{f}\ \mathsf{fail}) \\
    \hookrightarrow\ & \mathsf{fail}
    \end{aligned}$$

    and by definition of the function bodies and Section G.3:

    $$\begin{aligned}
    & \mathbf{P} \triangleright \mathbf{call}\ \mathbf{f}\ \mathbf{true} \\
    \hookrightarrow\ & \mathbf{P} \triangleright \mathbf{return}\ \mathbf{if}\ \mathbf{true}\ \mathbf{has}\ \mathsf{Nat}\!\downarrow\ \mathbf{then}\ \mathsf{e}\!\downarrow\ \mathbf{else}\ \mathbf{fail}
    \end{aligned}$$

$$\equiv \mathbf{P} \triangleright \mathbf{return\ if\ true\ has}\ \mathbb{N}\ \mathbf{then}\ \mathsf{e}\!\downarrow\ \mathbf{else\ fail}$$
$$\hookrightarrow \mathbf{P} \triangleright \mathbf{return\ if\ false\ then}\ \mathsf{e}\!\downarrow\ \mathbf{else\ fail}$$
$$\hookrightarrow \mathbf{P} \triangleright \mathbf{return\ fail}$$
$$\hookrightarrow \mathbf{fail}$$

So this case holds by definition of $O(W')_{\triangledown}$.

- $\mathbf{v}$=$\mathbf{n}$ and v=$\mathsf{n}+2$
  Case analysis on τ
  - τ=Bool
    This is analogous to the case for naturals above.
  - τ = Nat
    This is analogous to the case for booleans above.

□

Lemma G.41 (Compatibility lemma for backtranslation of check)**.**

$$\text{if } (HE)\ \texttt{toEmul}\,(\mathbf{\Gamma}); \mathsf{P}; \mathbf{P} \vdash \mathsf{e} \triangledown_n \mathbf{e} : \mathsf{EmulTy}$$
$$\text{then } 1\ \texttt{toEmul}\,(\mathbf{\Gamma}); \mathsf{P}; \mathbf{P} \vdash \mathsf{let\ x : Nat = e\ in\ if\ x \geq 2\ then\ 0\ else\ 1} \triangledown_n \mathbf{e\ has\ Bool} : \mathsf{EmulTy}$$
$$2\ \texttt{toEmul}\,(\mathbf{\Gamma}); \mathsf{P}; \mathbf{P} \vdash \mathsf{let\ x : Nat = e\ in\ if\ x \geq 2\ then\ 1\ else\ 0} \triangledown_n \mathbf{e\ has}\ \mathbb{N} : \mathsf{EmulTy}$$

Proof. We need to prove that

$$1\ \texttt{toEmul}\,(\mathbf{\Gamma}); \mathsf{P}; \mathbf{P} \vdash \mathsf{let\ x : Nat = e\ in\ if\ x \geq 2\ then\ 0\ else\ 1} \triangledown_n \mathbf{e\ has\ Bool} : \mathsf{EmulTy}$$
$$2\ \texttt{toEmul}\,(\mathbf{\Gamma}); \mathsf{P}; \mathbf{P} \vdash \mathsf{let\ x : Nat = e\ in\ if\ x \geq 2\ then\ 1\ else\ 0} \triangledown_n \mathbf{e\ has}\ \mathbb{N} : \mathsf{EmulTy}$$

We only show case 1, the other is analogous.

Take W such that $lev(W) \leq n$ and $(W, \gamma, \gamma) \in \mathcal{G}\,[\![\texttt{toEmul}\,(\mathbf{\Gamma})]\!]_{\triangledown}$, the thesis is: (we omit substitutions as they don't play an active role)

1) $(W, \mathsf{let\ x : Nat = e\ in\ if\ x \geq 2\ then\ 0\ else\ 1}, \mathbf{e\ has\ Bool}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

By Lemma G.22 (Related terms plugged in related contexts are still related) with HE we have that for HW $W' \sqsupseteq W$, and HV $(W', \mathsf{v}, \mathbf{v}) \in \mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$, the thesis becomes:

- $(W', \mathsf{let\ x : Nat = v\ in\ if\ x \geq 2\ then\ 0\ else\ 1}, \mathbf{v\ has\ Bool}) \in \mathcal{E}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$

We perform a case analysis based on HV:

- $\mathbf{v}$=$\mathbf{true}$/$\mathbf{false}$ and v=$\mathsf{1}$/$\mathsf{0}$ (respectively).
  We consider only the first case, the other is analogous.
  We have that

$$\mathsf{P} \triangleright \mathsf{let\ x : Nat = 1\ in\ if\ x \geq 2\ then\ 0\ else\ 1}$$
$$\hookrightarrow \mathsf{P} \triangleright \mathsf{if\ 1 \geq 2\ then\ 0\ else\ 1}$$
$$\hookrightarrow \mathsf{P} \triangleright \mathsf{1}$$

  and

$$\mathbf{P} \triangleright \mathbf{true\ has\ Bool} \hookrightarrow \mathbf{P} \triangleright \mathbf{true}$$

  This case holds by Lemma G.21 (Closedness under antireduction) and Lemma G.17 (Value relation implies term relation) and by the definition of $\mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$.

- $\mathbf{v}$=$\mathbf{n}$ and v=$\mathsf{n}+2$
  In this case we have that:

$$\mathsf{P} \triangleright \mathsf{let\ x : Nat = n+2\ in\ if\ x \geq 2\ then\ 0\ else\ 1}$$
$$\hookrightarrow \mathsf{P} \triangleright \mathsf{if\ n+2 \geq 2\ then\ 0\ else\ 1}$$
$$\hookrightarrow \mathsf{P} \triangleright \mathsf{0}$$

  and

$$\mathbf{P} \triangleright \mathbf{n\ has\ Bool} \hookrightarrow \mathbf{P} \triangleright \mathbf{false}$$

  This case holds by Lemma G.21 (Closedness under antireduction) and Lemma G.17 (Value relation implies term relation) and by the definition of $\mathcal{V}\,[\![\mathsf{EmulTy}]\!]_{\triangledown}$.

□

Semantic Preservation of Backtranslation

Theorem G.42 ($\cdot\uparrow$ is semantics preserving)**.**

$$\begin{aligned}&\text{if } \Gamma \vdash \mathbf{e}\\ &\quad (HP)\ \vdash \mathsf{P} \triangledown \mathbf{P}\\ &\text{then } \texttt{toEmul}(\Gamma);\mathsf{P};\mathbf{P} \vdash \langle\!\langle \mathbf{e} \rangle\!\rangle \triangledown_n \mathbf{e} : \mathsf{EmulTy}\end{aligned}$$

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash \mathbf{e}$.

Base cases **true,false,nat** By definition of the $\mathcal{V}[\![\mathsf{EmulTy}]\!]_{\triangledown}$
  var By definition of the $\mathcal{G}[\![\cdot]\!]_{\triangledown}$.
  call This case cannot arise.

Inductive cases **app** By IH and HP and Lemma G.40 (Compatibility lemma for backtranslation of application).
  op By IH and Lemma G.36 (Compatibility lemma for backtranslation of op).
  geq Analogous to the case above.
  if By IH and Lemma G.39 (Compatibility lemma for backtranslation of if).
  letin By IH and Lemma G.38 (Compatibility lemma for backtranslation of letin).
  check By IH and Lemma G.41 (Compatibility lemma for backtranslation of check).

□

Theorems that Yield RrHC

Theorem G.43 ($\cdot\downarrow$ preserves behaviors)**.**

$$\begin{aligned}&\text{if } (HT)\ \mathsf{P}\downarrow \triangleright \mathbf{e} \xRightarrow{\beta} \mathsf{P}\downarrow \triangleright \mathbf{e}'\\ &\text{then } \mathsf{P} \triangleright \mathbf{e}\uparrow \xRightarrow{\beta} \mathsf{P} \triangleright \mathsf{e}'\end{aligned}$$

Proof. By Theorem G.35 ($\cdot\downarrow$ is semantics preserving for programs) we have HPP:

- $\vdash \mathsf{P} \triangledown \mathsf{P}\downarrow$

Given that $\varnothing \vdash \mathbf{e}$, by Theorem G.42 ($\cdot\uparrow$ is semantics preserving) with HPP we have HPE:

- $\texttt{toEmul}(\Gamma);\mathsf{P};\mathsf{P}\downarrow \vdash \langle\!\langle \mathbf{e} \rangle\!\rangle \triangledown_n \mathbf{e} : \mathsf{EmulTy}$

The thesis follows by Lemma G.19 (Adequacy for $\gtrsim$) with HT. □

Theorem G.44 ($\cdot\downarrow$ reflects behaviors)**.**

$$\begin{aligned}&\text{if } (HS)\ \mathsf{P} \triangleright \mathbf{e}\uparrow \xRightarrow{\beta} \mathsf{P} \triangleright \mathsf{e}'\\ &\text{then } \mathsf{P}\downarrow \triangleright \mathbf{e} \xRightarrow{\beta} \mathsf{P}\downarrow \triangleright \mathbf{e}'\end{aligned}$$

Proof. By Theorem G.35 ($\cdot\downarrow$ is semantics preserving for programs) we have HPP:

- $\vdash \mathsf{P} \triangledown \mathsf{P}\downarrow$

Given that $\varnothing \vdash \mathbf{e}$, by Theorem G.42 ($\cdot\uparrow$ is semantics preserving) with HPP we have HPE:

- $\texttt{toEmul}(\Gamma);\mathsf{P};\mathsf{P}\downarrow \vdash \langle\!\langle \mathbf{e} \rangle\!\rangle \triangledown_n \mathbf{e} : \mathsf{EmulTy}$

The thesis follows by Lemma G.18 (Adequacy for $\lesssim$) with HS. □

G.4.3 Proof That $\cdot\downarrow$ Satisfies Definition G.6 ($\mathsf{RrHC}_{\bowtie}$)

$$\begin{aligned}&\forall \mathbf{e}.\exists \mathsf{e}.\forall \mathsf{P} \text{ such that } \mathsf{P} \bowtie \mathsf{e}, \forall \beta\\ &\quad \mathsf{P}\downarrow \triangleright \mathbf{e} \xRightarrow{\beta} \mathsf{P}\downarrow \triangleright \mathbf{e}'\\ &\iff \mathsf{P} \triangleright \mathsf{e} \xRightarrow{\beta} \mathsf{P} \triangleright \mathsf{e}'\end{aligned}$$

We instantiate e with $\mathbf{e}\uparrow$ then two cases arise.

$\Rightarrow$ **direction** By Theorem G.43 ($\cdot\downarrow$ preserves behaviors)
$\Leftarrow$ **direction** By Theorem G.44 ($\cdot\downarrow$ reflects behaviors).

$$\begin{tikzcd}
\forall i, \mathsf{C_S}[\mathsf{P}_i] \rightsquigarrow m_i & \forall i, \mathsf{C_S}[\mathsf{P}_i] \hookrightarrow \mu_i^s \arrow[l, "\text{Uninf}"'] & \forall i, \mathsf{P}_i \hookrightarrow_{\text{prg}} \mu_i \arrow[l, "\text{Comp}"'] & \forall i, \mathsf{C_S} \hookrightarrow_{\text{ctx}} \mu_i^s \arrow[ll, bend right, "\text{Comp}"'] \\
\forall i, \mathbf{C_T}\,[\mathsf{P}_i{\downarrow}] \rightsquigarrow m_i \arrow[r, "\text{Inf}"] & \forall i, \exists \mu_i \sqsupseteq m_i, \mathbf{C_T}[\mathsf{P}_i{\downarrow}] \hookrightarrow \mu_i \arrow[r, "\text{Dec}"] \arrow[rr, bend right, "\text{Dec}"'] & \forall i, \mathsf{P}_i{\downarrow} \hookrightarrow_{\text{prg}} \mu_i \arrow[u, "\text{BCC}"] & \forall i, \mathbf{C_T} \hookrightarrow_{\text{ctx}} \mu_i \arrow[u, "\text{BT}"]
\end{tikzcd}$$

Fig. 4: Proposed proof technique

G.5 Proof That $\cdot\!\downarrow$ Is $\mathsf{RFrXC}_{\bowtie}$

This section focuses on giving a high-level overview of the proof technique that we use to prove that our compiler satisfies the following variant of the criterion RFrXC:

Definition G.45 ($\mathsf{RFrXC}_{\bowtie}$)**.**

$$\begin{aligned}
\mathsf{RFrXC}_{\bowtie}:\quad & \forall K.\ \forall \mathsf{P}_1 \dots \mathsf{P}_\mathsf{K} : \bar{\mathsf{I}}.\ \forall \mathbf{C_T}.\ \forall x_1 \dots x_K. \\
& (\mathsf{P}_1{\downarrow} \bowtie \mathbf{C_T} \wedge \dots \wedge \mathsf{P}_\mathsf{K}{\downarrow} \bowtie \mathbf{C_T}) \implies \\
& (\mathbf{C_T}\,[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \dots \wedge \mathbf{C_T}\,[\mathsf{P}_\mathsf{K}{\downarrow}] \rightsquigarrow x_K) \implies \\
& \exists \mathsf{C_S}.\, \mathsf{P}_1 \bowtie \mathsf{C_S} \wedge \dots \wedge \mathsf{P}_\mathsf{K} \bowtie \mathsf{C_S} \wedge \\
& \qquad \mathsf{C_S}\,[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \dots \wedge \mathsf{C_S}\,[\mathsf{P}_\mathsf{K}] \rightsquigarrow x_K
\end{aligned}$$

This criterion is equivalent to the following property-full criterion:

Definition G.46 ($\mathsf{RFrXP}_{\bowtie}$)**.**

$$\begin{aligned}
\mathsf{RFrXP}_{\bowtie}:\quad & \forall \bar{\mathsf{I}}.\ \forall K, \mathsf{P}_1, \cdots, \mathsf{P}_\mathsf{k} : \bar{\mathsf{I}}, R \in 2^{(XPref^K)}. \\
& (\forall \mathsf{C_S}, x_1, \cdots, x_K, (\mathsf{P}_1 \bowtie \mathsf{C_S} \wedge \mathsf{C_S}\,[\mathsf{P}_1] \rightsquigarrow x_1 \wedge \cdots \wedge \mathsf{P}_\mathsf{K} \bowtie \mathsf{C_S} \wedge \mathsf{C_S}\,[\mathsf{P}_\mathsf{K}] \rightsquigarrow x_K) \\
& \quad \Rightarrow (x_1, \cdots, x_K) \in R) \Rightarrow \\
& (\forall \mathbf{C_T}, x_1, \cdots, x_K (\mathsf{P}_1{\downarrow} \bowtie \mathbf{C_T} \wedge \mathbf{C_T}\,[\mathsf{P}_1{\downarrow}] \rightsquigarrow x_1 \wedge \cdots \wedge \mathsf{P}_\mathsf{K}{\downarrow} \bowtie \mathbf{C_T} \wedge \mathbf{C_T}\,[\mathsf{P}_\mathsf{K}{\downarrow}] \rightsquigarrow x_K) \\
& \quad \Rightarrow (x_1, \cdots, x_K) \in R)
\end{aligned}$$

The proof of the equivalence of these two criteria is very similar to Theorem C.70.

G.5.1 Overview of the Proof Technique

Our proof technique for this is described in Figure 4. At the heart of this technique is the back-translation of a finite set of finite trace prefixes into a source context. In particular, this back-translation technique do not inspect the code of the target context. The first steps consist in transforming the trace prefixes into prefixes that can be back-translated easily, and separating the target context from the compiled programs. Then, we build a back-translation that provides us with a source context that can be composed with the initial source programs to generate the initial traces.

The reason for requiring all programs to share the same interface $\bar{\mathsf{I}}$ is that it allows us to produce a well-typed context. Otherwise, two programs could contain the same function, but one returning a natural number and the other a boolean. The back-translation would be immediatly impossible.

G.5.2 Informative Traces

The first step of the proof is to augment the existing operational semantics with new events that allow to precisely track the behavior of the program and of the context. This new semantics is called *informative semantics* and produce *informative traces*. They are defined at both the source level and the target level. The relations $\hookrightarrow$ are the equivalent of $\rightsquigarrow$ for these informative semantics, and are defined as:

$$\mathsf{C}[\mathsf{P}] \hookrightarrow \mu \iff \exists \mathsf{e}, \mathsf{P} \triangleright \mathsf{C} \xRightarrow{\mu} \mathsf{P} \triangleright \mathsf{e}$$

$$\mathbf{C}[\mathbf{P}] \hookrightarrow \mu \iff \exists \mathbf{e}, \mathbf{P} \triangleright \mathbf{C} \xRightarrow{\mu} \mathbf{P} \triangleright \mathbf{e}$$

We can state the theorem for passing to informative traces as follow

Theorem G.47 (Informative traces)**.** Let $\mathbf{C_T}$ be a target context and $\mathbf{P_T}$ a target program that are linkable. Then,

$$\forall m, \mathbf{C_T}[\mathbf{P_T}] \rightsquigarrow m \implies \exists \mu \sqsupseteq m, \mathbf{C_T}[\mathbf{P_T}] \hookrightarrow \mu$$

where

$$\mu \sqsupseteq m \iff |\mu|_{\text{I/O/termination}} = m.$$

Proof. Let $\mathbf{C_T}$ be a target context, $\mathbf{P_T}$ a target program and m a finite prefix. We are going to show that if there exists $\mathbf{e}$ such that $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mathbf{m}} \mathbf{P_T} \triangleright \mathbf{e}$, then there exists μ such that $|\mu|_{\text{I/O}} = m$ and $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mu} \mathbf{P_T} \triangleright \mathbf{e}$.

Let us proceed by induction on the relation $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mathbf{m}} \mathbf{P_T} \triangleright \mathbf{e}$.

Rule $\mathbf{EL^u}$-refl Immediate.

Rule $\mathbf{EL^u}$-terminate This is true by taking $\mu = \Downarrow$, because the informative semantics can progress if and only if the non-informative semantics can.

Rule $\mathbf{EL^u}$-diverge This is true by taking $\mu = \Uparrow$, because the informative semantics can only diverge when executing the program part (the context can not loop or do recursion), and calls from the program part do not generate any event.

Rule $\mathbf{EL^u}$-silent Then $\mathbf{P_T} \triangleright \mathbf{C_T} \xrightarrow{\epsilon} \mathbf{P_T} \triangleright \mathbf{e}$ according to the non-informative semantics. Since the semantics only differ on the events that are generated, we have two cases. Either $\mathbf{P_T} \triangleright \mathbf{C_T} \xrightarrow{\epsilon} \mathbf{P_T} \triangleright \mathbf{e}$ according to the informative semantics, in which case we can take $\mu = \epsilon$. Or $\mathbf{P_T} \triangleright \mathbf{C_T} \xrightarrow{\alpha} \mathbf{P_T} \triangleright \mathbf{e}$ according to the informative semantics, in which case we can take $\mu = \alpha$. This α must be a call or return event by definition of the informative semantics, hence the result.

Rule $\mathbf{EL^u}$-single Since $\mathbf{P_T} \triangleright \mathbf{C_T} \xrightarrow{\alpha} \mathbf{P_T} \triangleright \mathbf{e}$ according to the non-informative semantics, this is also the case according to the informative semantics, hence the result.

Rule $\mathbf{EL^u}$-cons Then $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mathbf{m_1}} \mathbf{P_T} \triangleright \mathbf{e}'$ and $\mathbf{P_T} \triangleright \mathbf{e}' \xRightarrow{\mathbf{m_2}} \mathbf{P_T} \triangleright \mathbf{e}$ with $m = m_1 m_2$. By applying the induction hypothesis, there exists μ_1 and μ_2 such that $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mu_1} \mathbf{e}'$, $\mathbf{P_T} \triangleright \mathbf{e}' \xRightarrow{\mu_2} \mathbf{e}$, $|\mu_1|_{\text{I/O/termination}} = m_1$, and $|\mu_2|_{\text{I/O/termination}} = m_2$.

Therefore by applying Rule $\mathbf{EL^u}$-cons, $\mathbf{P_T} \triangleright \mathbf{C_T} \xRightarrow{\mu_1 \mu_2} \mathbf{e}$. It is easy to see that $|\mu_1 \mu_2|_{\text{I/O/termination}} = m_1 m_2$. We are done. ∎

G.5.3 Decomposition

This decomposition step relies on the definition of *partial semantics*, one for programs and one for contexts. These partial semantics describe the possible behaviors of a program in any context and of a context with respect to any program. Partial semantics can often be defined by abstracting away one part of the whole program (the context for the partial semantics of programs, and the program for the partial semantics of contexts), by introducing non-determinism for modeling the abstracted part.

We index our relations by either "ctx" or "prg" to denote the partial semantics. The partial semantics for contexts defined as:

$$\frac{}{\mathsf{call}\ \mathsf{f}\ \mathsf{v} \xrightarrow{\texttt{call f v?}}_{\text{ctx}} \mathsf{return}\ \mathsf{e}}\ (\mathsf{EL}^{\tau}\text{-ctx-call}) \qquad \frac{}{\mathbf{call\ f\ v} \xrightarrow{\texttt{call f v?}}_{\text{ctx}} \mathbf{return\ e}}\ (\mathsf{E}\mathbf{L^u}\text{-ctx-call})$$

$$\frac{}{\mathsf{return}\ \mathsf{v} \xrightarrow{\epsilon}_{\text{ctx}} \mathsf{v}}\ (\mathsf{EL}^{\tau}\text{-ctx-ret}) \qquad \frac{}{\mathbf{return\ v} \xrightarrow{\epsilon}_{\text{ctx}} \mathbf{v}}\ (\mathsf{E}\mathbf{L^u}\text{-ctx-ret})$$

and the relations $\xRightarrow{\cdot}_{\text{ctx}}$ and $\xRightarrow{\cdot}_{\text{ctx}}$ are defined in the same manner as the complete semantics.

The partial semantics for programs are defined in terms of the complete semantics, and are parameterized by the interface of the program I. Informally, we define $P \hookrightarrow_{\text{prg}} \mu$ to mean that the program P is able to produce each part of the trace μ that comes from the program, i.e. each part that starts with a call event $\texttt{call}\ f\ v?$ and ends before or with the corresponding return event, when it is put into the context that simply calls this function f with this value v. For every "subtrace" μ' of μ starting with a call event $\texttt{call}\ f\ v?$ and stopping at the latest at the next (corresponding) return event, it must be that $P \triangleright \mathit{call}\ f\ v \hookrightarrow \mu'$.

Definition G.48 (Partial semantics for programs)**.** $\mathsf{P} \hookrightarrow_{\text{prg}} \mu$ if and only if:

- for any trace $\mu_{f,v,v'} = \texttt{call}\ f\ v?; \mu'; \texttt{ret}\ v'!$ such that $\mu = \mu_1; \mu_{f,v,v'}; \mu_2$, such that there is no event $\mathit{return}\ \ldots$ in μ', and such that $\mathsf{f} : \tau \to \tau' \in \mathsf{I}$ with $\mathsf{v} \in \tau$, we have
$$\mathsf{P_T} \triangleright \mathsf{call}\ \mathsf{f}\ \mathsf{v} \xRightarrow{\mu_{\mathsf{f,v,v'}}} \mathsf{P} \triangleright \mathsf{v}';$$
- for any trace $\mu_{f,v} = \texttt{call}\ f\ v?; \mu'$ such that $\mu = \mu_1; \mu_{f,v}$, such that there is no event $\mathit{return}\ \ldots$ in μ', and such that $\mathsf{f} : \tau \to \tau' \in \mathsf{I}$ with $\mathsf{v} \in \tau$, there exists e such that
$$\mathsf{P_T} \triangleright \mathsf{call}\ \mathsf{f}\ \mathsf{v} \xRightarrow{\mu_{\mathsf{f,v}}} \mathsf{P} \triangleright \mathsf{e}.$$

$\mathbf{P} \hookrightarrow_{\text{prg}} \mu$ if and only if:

- for any trace $\mu_{f,v,v'} = \texttt{call}\ f\ v?; \mu'; \texttt{ret}\ v'!$ such that $\mu = \mu_1; \mu_{f,v,v'}; \mu_2$, such that there is no event $return\ \ldots$ in μ', and such that $\mathsf{f} \in \bar{\mathsf{I}}$ we have
$$\mathbf{P_T} \triangleright \mathbf{call\ f\ v} \xrightarrow{\mu_{\mathbf{f,v,v'}}}\!\!\!\!\!\!\!\rightarrow \mathbf{P} \triangleright \mathbf{v'};$$
- for any trace $\mu_{f,v} = \texttt{call}\ f\ v?; \mu'$ such that $\mu = \mu_1; \mu_{f,v}$, such that there is no event $return\ \ldots$ in μ', and such that $\mathsf{f} \in \bar{\mathsf{I}}$ there exists $\mathbf{e}$ such that
$$\mathbf{P_T} \triangleright \mathbf{call\ f\ v} \xrightarrow{\mu_{\mathbf{f,v}}}\!\!\!\!\!\!\!\rightarrow \mathbf{P} \triangleright \mathbf{e}.$$

We must restrict this definition to the well-typed calls in the source level: indeed, a badly-typed call does not make sense in the source language.

Our decomposition theorem talks about both programs and contexts:

Theorem G.49 (Decomposition)**.** Let $\mathbf{C_T}$ be a target context and $\mathbf{P_T}$ a target program that are linkable. Then,
$$\forall \mu, \mathbf{C_T}\left[\mathbf{P_T}\right] \hookrightarrow \mu \implies \mathbf{C_T} \hookrightarrow_{\text{ctx}} \mu \wedge \mathbf{P_T} \hookrightarrow_{\text{prg}} \mu$$

We are going to prove two different lemmas, one for contexts and one for programs.

Lemma G.50. Let $\mathbf{C_T}$ be a target context and $\mathbf{P_T}$ be a target program, μ an informative trace and $\mathbf{e}$ a target expression. Then,
$$\mathbf{C_T}\left[\mathbf{P_T}\right] \xrightarrow{\mu}\!\!\!\!\!\!\!\rightarrow \mathbf{e} \implies \mathbf{C_T} \xrightarrow{\mu}\!\!\!\!\!\!\!\rightarrow_{\text{ctx}} \mathbf{e}$$

Proof. By induction on the relation $\mathbf{C_T}[\mathbf{P_T}] \xrightarrow{\mu}\!\!\!\!\!\!\!\rightarrow \mathbf{P_T} \triangleright \mathbf{e}$

Rule $\mathbf{EL^u}$-silent Therefore $\mathbf{C_T}[\mathbf{P_T}] \xrightarrow{\epsilon} \mathbf{P_T} \triangleright \mathbf{e}$. By case analysis, it is also the case that $\mathbf{C_T} \xrightarrow{\epsilon}_{\text{ctx}} \mathbf{e}$ hence the result.

Rule $\mathbf{EL^u}$-action $\mathbf{C_T}[\mathbf{P_T}] \xrightarrow{\alpha} \mathbf{P_T} \triangleright \mathbf{e}$. We proceed by case analysis on this relation: if α is an I/O operation, correct termination or failure event, then we indeed have $\mathbf{C_T} \xrightarrow{\alpha}_{\text{ctx}} \mathbf{e}$.

Otherwise, $\alpha = \Uparrow$. Therefore, $\forall n, \exists \mathbf{e_n}, \mathbf{C_T}[\mathbf{P_T}] \xrightarrow{\epsilon}{}^{\mathbf{n}} \mathbf{P_T} \triangleright \mathbf{e_n}$. Now, by induction on n, we can prove that $\forall n, \exists \mathbf{e_n}, \mathbf{C_T} \xrightarrow{\epsilon}_{\text{ctx}}{}^{\mathbf{n}} \mathbf{e_n}$. Hence the result.

Rule $\mathbf{EL^u}$-single Then $\mathbf{C_T}[\mathbf{P_T}] \xrightarrow{\beta} \mathbf{P_T} \triangleright \mathbf{e}$. We proceed by case analysis on this relation:

- If $\beta = \texttt{call}\ f\ v?$, then $\mathbf{C_T} = \mathbb{E}\left[\mathbf{call\ f\ v}\right]$ and $\mathbf{e} = \mathbb{E}\left[\mathbf{return\ e'}\right]$ for some evaluation context $\mathbb{E}$ and some expression $\mathbf{e'}$. Therefore, $\mathbf{e} \xrightarrow{\texttt{call f v?}}_{\text{ctx}} \mathbb{E}\left[\mathbf{return\ e'}\right]$ by the partial semantics, hence the result.
- If $\beta = \texttt{ret}\ f!v$, then $\mathbf{C_T} = \mathbb{E}\left[\mathbf{return\ v}\right]$ for some evaluation context $\mathbb{E}$. Therefore, $\mathbf{e} \xrightarrow{\texttt{ret f!v}}_{\text{ctx}} \mathbb{E}\left[\mathbf{v}\right]$ according to the partial semantics, hence the result.

Rule $\mathbf{EL^u}$-cons We have that $\mathbf{P_T} \triangleright \mathbf{C_t} \xrightarrow{\mu_1}\!\!\!\!\!\!\!\rightarrow_{\text{ctx}} \mathbf{e'}$ and $\mathbf{P_T} \triangleright \mathbf{e'} \xrightarrow{\mu_2}\!\!\!\!\!\!\!\rightarrow_{\text{ctx}} \mathbf{e}$. Then, by applying the induction hypothesis to the two relations, we are done.

$\square$

Then, we prove a similar lemma for programs:

Lemma G.51. Let $\mathbf{P_T}$ be a target program, $\mathbf{C_T}$ a target context and μ an informative trace. Suppose that $\mathbf{C_T}\left[\mathbf{P_T}\right] \hookrightarrow \mu$. Then:

- for any trace $\mu_{f,v,v'} = \texttt{call}\ f\ v?; \mu'; \texttt{ret}\ v'!$ such that $\mu = \mu_1; \mu_{f,v,v'}; \mu_2$ and such that there is no event $return\ \ldots$ in μ', $\mathbf{P_T} \triangleright \mathbf{call\ f\ v} \xrightarrow{\mu_{\mathbf{f,v,v'}}}\!\!\!\!\!\!\!\rightarrow \mathbf{v'}$
- for any trace $\mu_{f,v} = \texttt{call}\ f\ v?; \mu'$ such that $\mu = \mu_1; \mu_{f,v}$ and such that there is no event $return\ \ldots$ in μ', there exists $\mathbf{e}$ such that $\mathbf{P_T} \triangleright \mathbf{call\ f\ v} \xrightarrow{\mu_{\mathbf{f,v}}}\!\!\!\!\!\!\!\rightarrow \mathbf{e}$.

Proof. Consider the first case for instance. From the fact that $\mu_{f,v,v'}$ appears in μ, we can deduce the fact that there exists an evaluation context $\mathbb{E}$ such that $\mathbf{P} \triangleright \mathbb{E}\left[\mathbf{call\ f\ v}\right] \xrightarrow{\mu_{f,v,v'}}\!\!\!\!\!\!\!\rightarrow_{\text{ctx}} \mathbb{E}\left[\mathbf{v'}\right]$.

From this, we can reason by induction and use Rule $\mathbf{EL^u}$-ctx to obtain the result. $\square$

G.5.4 Backward Compiler Correctness for Programs

Theorem G.52 (Backward Compiler Correctness)**.** Let P be a source program. Then,
$$\forall \mu, \mathsf{P}\!\downarrow\ \hookrightarrow_{\text{prg}} \mu \implies \mathsf{P} \hookrightarrow_{\text{prg}} \mu.$$

Before proving the theorem, we state a preliminary lemma:

Lemma G.53. Suppose that $\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xrightarrow{\texttt{call f v?};\mu}\!\!\!\!\!\!\!\rightarrow \mathsf{P}\!\downarrow \triangleright \mathbf{e'}$ where the call is well-typed.

Then, $\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xrightarrow{\texttt{call f v?}}\!\!\!\!\!\!\!\rightarrow \mathsf{P}\!\downarrow \triangleright \mathsf{e}\!\downarrow\![\mathbf{x}/\mathbf{v}]$ and:

- $\mathsf{P}\!\downarrow \triangleright\ \mathsf{e}\!\downarrow\![\mathbf{x}/\mathbf{v}] \xRightarrow{\mu} \mathsf{P}\!\downarrow \triangleright \mathbf{e}'$,
- or, $\mu = \epsilon$ and $\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xRightarrow{\texttt{call } f\ v?} \mathsf{P}\!\downarrow \triangleright \mathbf{e}'$

where e is the code of the function f in the source program.

Proof. By induction on $\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xRightarrow{\texttt{call } f\ v?;\mu} \mathbf{e}'$.

Rule $\mathbf{EL^u}$-single In this case, $\mu = \epsilon$. The result is obtained by direct application of the semantics.
Rule $\mathbf{EL^u}$-cons There exists μ_1 and μ_2 such that $\mu_1\mu_2 = \mu$ and

$$\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xRightarrow{\texttt{call } f\ v?;\mu_1} \mathbf{e_1}$$

and

$$\mathsf{P}\!\downarrow \triangleright\ \mathbf{e_1} \xRightarrow{\mu_2} \mathbf{e}.$$

By applying the induction hypothesis to the first relation, we obtain the result.
Other cases: these cases are impossible

□

We can now prove the backward compiler correctness theorem:

Theorem G.54 (Backward Compiler Correctness)**.** Let P be a source program. Then,

$$\forall \mu, \mathsf{P}\!\downarrow\ \hookrightarrow_{\text{prg}} \mu \implies \mathsf{P} \hookrightarrow_{\text{prg}} \mu.$$

Proof. Let P be a source program and μ an informative trace. Suppose that $\mathsf{P}\!\downarrow\ \hookrightarrow_{\text{prg}} \mu$, we will prove that $\mathsf{P} \hookrightarrow_{\text{prg}} \mu$.
Let $\mu_{f,v,v'} = \texttt{call}\ f\ v?;\mu';\texttt{ret}\ v'!$ be a trace as defined by the source partial semantics. Let us show that

$$\mathsf{P} \triangleright \mathsf{call\ f\ v} \xRightarrow{\mu_{\mathsf{f,v,v'}}} \mathsf{v}',$$

knowing that

$$\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xRightarrow{\mu_{\mathbf{f,v,v'}}} \mathbf{v}'.$$

By the preliminary lemma, and since $\mu' \neq \epsilon$, we have that

$$\mathsf{P}\!\downarrow \triangleright\ \mathbf{call\ f\ v} \xRightarrow{\texttt{call } f\ v?} \mathsf{e}\!\downarrow\![\mathbf{x}/\mathbf{v}]$$

where e is the source of f in the source program, because the call is well-typed and $\mathsf{P}\!\downarrow \triangleright\ \mathsf{e}\!\downarrow\![\mathbf{x}/\mathbf{v}] \xRightarrow{\mu';\texttt{ret } v'!} \mathbf{v}'$.
Now, we can conclude by induction on e.

□

G.5.5 Back-Translation of a Finite Set of Finite Trace Prefixes

The theorem we wish to prove in this section is the following theorem:

Theorem G.55. Let $\mathbf{C_T}$ be a target context and $\{\mu_i\}$ be a finite set of trace prefixes such that $\forall i, \mathbf{C_T} \hookrightarrow_{\text{ctx}} \mu_i$. Then,

$$\exists \mathsf{C_S}, \forall i, \mathsf{C_S} \hookrightarrow_{\text{ctx}} \mu_i^s$$

where the relation between μ_i and μ_i^s is explicited later.
We will construct a function $\uparrow$ such that if F is a set of finite prefixes, $F\!\uparrow$ is a source context such that:

$$\forall \mu \in F, F\!\uparrow \hookrightarrow_{\text{ctx}} \mu^s.$$

where μ^s, defined later, is the trace μ with the possibility of swapping failure and calls events, as described previously.
We only consider traces that do not have any I/O. Indeed, I/O is produced only by the programs in these languages, hence do not affect the backtranslation of a source context. First, we explicit the tree structure that is found in F by defining the following inductive construction:

$$\begin{aligned} T ::= &\ \epsilon \mid \Downarrow \mid \bot \mid \Uparrow \\ &\mid (\texttt{call}\ f\ v?, (v_1, T_1), (v_2, T_2), \ldots, (v_i, T_i)) \end{aligned}$$

From a set of trace F, we define a relation $F \vDash T$ as follow:

$$\frac{F = \varnothing \vee \forall \mu \in F, \mu = \epsilon}{F \vDash \epsilon}\text{(Tree-Empty)} \qquad \frac{\forall \mu \in F, \mu \neq \epsilon \implies \mu = \Downarrow}{F \vDash \Downarrow}\text{(Tree-Term)} \qquad \frac{\forall \mu \in F, \mu \neq \epsilon \implies \mu = \Uparrow}{F \vDash \Uparrow}\text{(Tree-Divr)}$$

$$\frac{\forall \mu \in F, \mu \neq \epsilon \implies \mu = \bot}{F \vDash \bot}\text{(Tree-Fail)} \qquad \frac{\forall \mu \in F, \mu \neq \epsilon \implies \mu = \texttt{call}\ f\ v?; \mu' \wedge \mathsf{f} : \tau \to \tau' \wedge v \notin \tau}{F \vDash \bot}\text{(Tree-Fail-Type)}$$

$$\frac{\begin{array}{c}\forall i, \exists \mu \in F, \mu = \texttt{call}\ f\ v?; \texttt{ret}\ v_i!; \mu' \\ \{\mu' \mid \texttt{call}\ f\ v?; \texttt{ret}\ v_i!; \mu' \in F\} \vDash T_i \\ \bigcup_{1 \leq j \leq i} \{\texttt{call}\ f\ v?; \texttt{ret}\ v_j!; \mu' \in F\} \cup \{\texttt{call}\ f\ v?; \Uparrow\} \cup \{\texttt{call}\ f\ v?\} \cup \{\epsilon\} \supseteq F\end{array}}{F \vDash (\texttt{call}\ f\ v?, (v_1, T_1), \ldots, (v_i, T_i))}\text{(Tree-Call-Ret)}$$

This relation means that the tree T represents the set of traces F. The first five rules represent the base cases from the point of view of the context: Rule Tree-Empty is the case where every trace is empty or there are no trace in F. Rule Tree-Term represent the case where all traces terminate. Rule Tree-Divr is a case that should never happen, because the context should never diverge. Rule Tree-Fail is the case where all traces fail in the context. Rule Tree-Fail-Type represent the case where all traces call a function with an incorrect argument and must fail.

The last rule, Rule Tree-Call-Ret, represent the case where some traces may be cut, and the others shall call a function. The next event must be either divergence, which is ignored because it is part of the program, or a return event. Then, the remaining traces are separated into groups receiving the same return value: these traces are then considered on their own to construct subtrees T_i. The third condition is required to ensure that no trace is forgotten.

The fact that this object is indeed defined is directly derived from the determinacy of the context. Indeed, let F be a set of informative traces produced by the same context. They must either be empty, or start by the same event, by determinacy, and this event has to be a call event. If this call in not correctly typed, then we are in the fifth case. Otherwise, we are necessarily in the last case, and the T_i exist by induction.

The back-translation of F is defined by induction on the tree T such that $F \vDash T$:

Definition G.56 (Backtranslation of the tree T)**.**

$$T\uparrow = \begin{cases} \mathsf{fail} & \text{if } T = \epsilon \text{ or } T = \bot \\ \mathsf{0} & \text{if } T = \Downarrow \\ \mathsf{fail} & \text{if } T = \Uparrow \\ \mathsf{let\ x = call\ f\ v\ in} \begin{pmatrix} \mathsf{if\ x = v_1\ then\ T_1\uparrow} \\ \mathsf{else\ if\ x = v_2\ then\ \ldots} \\ \mathsf{else\ if\ x = v_i\ then\ T_i\uparrow\ else\ fail} \end{pmatrix} & \begin{array}{l}\text{if } T = (\texttt{call}\ f\ v?, (v_1, T_1), \ldots, (v_i, T_i)) \text{ and} \\ \mathsf{f} : \tau \to \tau' \text{ and } v \in \tau\end{array} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Lemma G.57. The back-translation of a set of traces F generated by a single context is well-typed and linkable.

Proof. By induction on the relation $F \vDash T$. □

We define what it means for a trace to be "part" of such a tree:

Definition G.58 (Trace extract from a tree)**.** We say that a trace μ is extracted from a tree T if:

1) $\mu = \epsilon$
2) $\mu = \Downarrow$ and $T = \Downarrow$
3) $\mu = \bot$ and $T = \bot$
4) $\mu = \texttt{call}\ f\ v? :: \epsilon$, $\text{type}(v) \neq \text{input\_type}(f)$ and $T = \bot$
5) $\mu = \texttt{call}\ f\ v? :: \bot$, $\text{type}(v) \neq \text{input\_type}(f)$ and $T = \bot$
6) $\mu = \texttt{call}\ f\ v? :: \epsilon$ or $\mu = \texttt{call}\ f\ v?; \Uparrow$, $T = (\texttt{call}\ f\ v?, \ldots)$ and $\text{type}(v) = \text{input\_type}(f)$
7) $\mu = \texttt{call}\ f\ v? :: \texttt{ret}\ v'! :: \mu'$, $T = (\texttt{call}\ f\ v?, (v_1, T_1), \ldots, (v_i, T_i))$, and $\exists j$, such that $v_j = v'$ and μ' is extracted from T_j
8) $\mu = \texttt{call}\ f\ v? :: \epsilon$ or $\mu = \texttt{call}\ f\ v?; \bot$, $T = \bot$ and $\text{type}(v) \neq \text{input\_type}(f)$

We are going to prove that any such trace extracted from a tree can be produced by the back-translated context, modulo the behaviors allowed at the target level but not at the source level.

Definition G.59.

$$\mu^s = \begin{cases} \mu'\bot & \text{if } \mu = \mu'\texttt{call}\ f\ v? \text{ such that } \text{input\_type}(f) \neq \text{type}(v) \\ \mu'\bot & \text{if } \mu = \mu'\texttt{call}\ f\ v?\bot \text{ such that } \text{input\_type}(f) \neq \text{type}(v) \\ \mu & \text{otherwise} \end{cases}$$

Theorem G.60 (Correction of the backtranslation)**.** Let T be a tree and μ a trace extracted from T. Then, $T\uparrow\rightsquigarrow \mu^s$.

Proof. We are going to prove by induction on the relation "μ is extracted from T" that there exists e such that $T\uparrow \xRightarrow{\mu^s}\!\!\!\!\gg \mathsf{e}$.

1) $\mu = \epsilon$: OK.
2) $\mu = \Downarrow$ and $T = \Downarrow$: $T\uparrow = 0$. OK.
3) $\mu = \bot$ and $T = \bot$: $T\uparrow = \mathsf{fail}$. OK.
4) $\mu = \texttt{call}\ f\ v?; \epsilon$, $\text{type}(v) \neq \text{input\_type}(f)$ and $T = \bot$ We are in the first case for μ^s: OK.
5) $\mu = \texttt{call}\ f\ v?; \bot$, $\text{type}(v) \neq \text{input\_type}(f)$ and $T = \bot$ We are in the second case for μ^s: OK.
6) $\mu = \texttt{call}\ f\ v?; \epsilon$, $T = (\texttt{call}\ f\ v?, \dots)$ and $\text{type}(v) = \text{input\_type}(f)$: $T\uparrow = \mathsf{let\ x = call\ f\ v\ in\ \dots}$. OK. Idem with $\Uparrow$ instead of ϵ.
7) $t = \texttt{call}\ f\ v?; \texttt{ret}\ v'!; \mu'$, $T = (\texttt{call}\ f\ v?, (v_1, T_1), \dots, (v_i, T_i))$, and $\exists j$, such that $v_j = v'$ and μ' is extracted from T_j: Then:
$$T\uparrow = \mathsf{let\ x = call\ f\ v\ in\ if\ \dots\ then\ if\ x = v_j\ then\ T_j\uparrow\ else\ \dots\ else\ \dots}.$$
By application of the partial semantics:
$$T\uparrow \xRightarrow{\texttt{call}\ f\ v?;\texttt{ret}\ v_j!}_{\text{ctx}} \mathsf{if\ x = v_j\ then\ T_j\uparrow\ else\ \dots [v_j/x]}$$
and therefore by substituting and application of the partial semantics:
$$T\uparrow \xRightarrow{\texttt{call}\ f\ v?;\texttt{ret}\ v_j!}_{\text{ctx}} T_j\uparrow.$$
By induction hypothesis, we are done.
8) $\mu = \texttt{call}\ f\ v? :: \epsilon$ or $\mu = \texttt{call}\ f\ v?; \bot$, $T = \bot$ and $\text{type}(v) \neq \text{input\_type}(f)$. The result is immediate

□

Now, we can prove that any of the initial traces that are used to construct the tree can be found in this tree, and then the theorem applies to them.

Lemma G.61. Let F be a set of traces and T such that $F \vDash T$. Then, any trace $\mu \in F$ is extracted from the tree T.

Proof. Let us prove by induction on T that if there exists F such that $T = T(F)$, then $\forall \mu \in F$, μ is extracted from T. Since the trace ϵ is always extracted from any tree, we ignore this case.

$T = \epsilon$**:** OK.
$T = \Downarrow$**:** Then $\mu = \Downarrow$. OK.
$T = \Uparrow$**:** Then $\mu = \Uparrow$. OK.
$T = (\texttt{call}\ f\ v?, (v_1, T_1), \dots, (v_i, T_i))$**:** By induction hypothesis.

□

G.5.6 Composition

The composition theorem states that if a context and a program can partially produce two related informative traces, then plugging the program into the context gives a whole program that can produce one of the traces. The relation between the two traces captures the fact that the way things fail in the source is not the same as in the target, as seen in the back-translation section. The theorem is stated as follows:

Theorem G.62 (Composition)**.** Let $\mathsf{C_S}$ be a source context, $\mathsf{P_S}$ be a source program, $\mu_i \sim \mu_i^s$ two related traces, and suppose $\mathsf{P_S} \bowtie \mathsf{C_S}$. Then, if $\mathsf{C_S} \hookrightarrow_{\text{ctx}} \mu_i^s$ and $\mathsf{P_S} \hookrightarrow_{\text{prg}} \mu_i$, then $\mathsf{C_S}\,[\mathsf{P_S}] \hookrightarrow \mu_i^s$.

We state a preliminary lemma:

Lemma G.63. If $\mathsf{P} \hookrightarrow_{\text{prg}} \mu_i$, then $\mathsf{P} \hookrightarrow_{\text{prg}} \mu_i^s$.

Proof. This is by definition of μ_i^s. □

Lemma G.64. Let $\mathsf{C_S}$ be a source context, $\mathsf{P_S}$ be a source program, $\mu_i \sim \mu_i^s$ two related traces such that μ_i was produced by $\mathsf{P_S}\!\downarrow$ and some target context, and e an expression. Then, if $\mathsf{C_S} \xRightarrow{\mu_i^s}\!\!\!\!\gg \mathsf{e}$ and $\mathsf{P_s} \hookrightarrow_{\text{prg}} \mu_i^s$, then $\mathsf{P_S} \triangleright \mathsf{C_S} \xRightarrow{\mu_i^s}\!\!\!\!\gg \mathsf{e}'$ where $\mathsf{C_S} \xRightarrow{\mu_i^s}\!\!\!\!\gg \mathsf{e}'$.

Proof. We will prove by induction n that $\forall n, \forall \mu, |\mu| = n, \forall \mathsf{e}, \forall \mathsf{P_S}, \mathsf{e} \hookrightarrow_{\text{ctx}} \mu \wedge \mathsf{P_S} \hookrightarrow_{\text{prg}} \mu \implies \exists \mathsf{e}', \mathsf{e} \xRightarrow{\mu}\!\!\!\!\gg \mathsf{e}' \wedge \mathsf{P_S} \triangleright \mathsf{e} \xRightarrow{\mu}\!\!\!\!\gg \mathsf{e}'$
Base case If $n = 0$, this is trivially true.
Inductive case Let $n \in \mathbb{N}$, μ of length n, e and $\mathsf{P_S}$ such that $\mathsf{e} \hookrightarrow_{\text{ctx}} \mu$ and $\mathsf{P_S} \hookrightarrow_{\text{prg}} \mu$.
We consider only one case, but the other cases are similar:
$$\mu = \mu_1 \mu_2 \mu_3$$

where $\mu_2 = \texttt{call}\ f\ v?\mu_2'\texttt{ret}\ v'!$ is defined as in the definition of $\hookrightarrow_{\text{prg}}$.

- First, $\mathsf{e} \hookrightarrow_{\text{ctx}} \mu_1$ and $\mathsf{e} \hookrightarrow_{\text{prg}} \mu_1$, by definition of these relations. Therefore, by induction hypothesis, $\exists \mathsf{e}', \mathsf{e} \xRightarrow{\mu_1} \mathsf{e}'$ and $\mathsf{P_S} \triangleright \mathsf{e} \xRightarrow{\mu_1} \mathsf{e}'$. In particular, e' is of the form $\mathbb{E}\,[\mathsf{call\ f\ v}]$ by determinism of the execution of the context (since the read/writes are set by the trace), such that $\mathsf{e}' \hookrightarrow_{\text{ctx}} \mu_2\mu_3$.
- We have that $\mathsf{P_S} \triangleright \mathsf{e}' \xRightarrow{\mu_2} \mathbb{E}\,[\mathsf{v}']$ by definition of the partial semantics for programs, and the rules of evaluations inside contexts.
- We can again apply the induction hypothesis to μ_3.

Hence, we obtain the result: $\mathsf{P_S} \triangleright \mathsf{e} \xRightarrow{\mu_1\mu_2\mu_3} \mathsf{e}''$ where $\mathsf{e} \xRightarrow{\mu_1\mu_2\mu_3} \mathsf{e}''$.

□

By using these two lemmas, we can prove the composition theorem.

G.5.7 Back to Non-Informative Traces

The last step of the proof is to go back to the non-informative trace model. In particular, we must take into account that the trace μ_i^s that is generated by the whole program is not exactly equal to the original trace μ_i.

Theorem G.65 (Back to non-informative traces)**.** Let $\mathsf{C_S}$ be a source context, $\mathsf{P_S}$ be a source program, m a non-informative trace and μ an informative trace such that $\mu \sqsupseteq m$.

Then, $\mathsf{C_S}\,[\mathsf{P_S}] \hookrightarrow \mu^s \implies \mathsf{C_S}\,[\mathsf{P_S}] \rightsquigarrow m$.

The proof is immediate by definition of μ^s.

G.5.8 Proving the Secure Compilation Criterion

The proof follows the scheme depicted by Figure 4.

Proof. Let $\mathsf{P}_1 \dots \mathsf{P}_k$ be k programs and $m_1 \dots m_k$ be k finite trace prefixes. Let $\mathbf{C_T}$ be a target context and suppose the following holds:

$$\forall i, \mathbf{C_T}[\mathsf{P}_i\!\downarrow] \rightsquigarrow m_i$$

We can pass to informative traces by applying Theorem G.47 to each m_i

$$\forall i, \exists \mu_i \sqsupseteq m, \mathbf{C_T}[\mathsf{P}_i\!\downarrow] \hookrightarrow \mu_i.$$

From here, we can apply the decomposition theorem (Theorem G.49) to each μ_i:

$$\forall i, \mathbf{C_T} \hookrightarrow_{\text{ctx}} \mu_i \wedge \mathsf{P}_i\!\downarrow \hookrightarrow_{\text{prg}} \mu_i.$$

By the backward compiler correctness theorem (Theorem G.54) for programs applied to each program, we obtain that:

$$\forall i, \mathsf{P}_i \hookrightarrow_{\text{prg}} \mu_i.$$

Also, by applying the back-translation theorem, we can produce a source context:

$$\exists \mathsf{C_S}, \forall i, \mathsf{C_S} \hookrightarrow_{\text{ctx}} \mu_i^s.$$

Moreover, this $\mathsf{C_S}$ is well-typed and linkable with the $\mathsf{P_i}$.

Now, we are able to apply the composition theorem (Theorem G.62) to each program:

$$\forall i, \mathsf{C_S}[\mathsf{P}_i] \hookrightarrow \mu_i^s$$

Finally, we can go back to the non-informative traces by the last theorem (Theorem G.65):

$$\forall i, \mathsf{C_S}[\mathsf{P}_i] \rightsquigarrow m_i.$$

□

Remarks on the proof technique This proof technique should be fairly generic and could be adapted to other languages. if needed, it is possible to change the top-level statement by introducing a more complex relation between source and target, that could for instance model the exchange between failure and calls that might happen in our instance, or to model non-determinism in a non-deterministic language. While decomposition and composition are natural properties that we expect to hold for most languages, and while backward correctness can reasonably be expected from a secure compiler, the back-translation seems to be the hardest part of the proof and the most subject to change between languages.

References

[1] M. Abadi. Protection in programming-language translations. *Secure Internet Programming*. 1999.

[2] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM TISSEC*, 15(2), 2012.

[3] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1), 2002.

[4] M. Abadi, B. Blanchet, and C. Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1), 2018.

[5] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018.

[6] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012.

[7] A. Ahmed. Verified compilers for a multi-language world. *SNAPL*. 2015.

[8] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. *ICFP*. 2008.

[9] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. *ICFP*. 2011.

[10] A. Ahmed, D. Garg, C. Hriţcu, and F. Piessens. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports*, 8(5), 2018.

[11] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. *CCS*. 2017.

[12] B. Alpern and F. B. Schneider. Defining liveness. *IPL*, 21(4), 1985.

[13] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. *ESORICS*. 2008.

[14] A. Azevedo de Amorim, M. Dénès, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. *Oakland S&P*. 2015.

[15] M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. *CCS*. 2008.

[16] M. Backes, C. Hriţcu, and M. Maffei. Union and intersection types for secure protocol implementations. *TOSCA*. 2011.

[17] D. Baelde, S. Delaune, and L. Hirschi. A reduced semantics for deciding trace equivalence. *LMCS*, 13(2), 2017.

[18] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *FOSAD 2012/2013*. 2013.

[19] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". *CSF*. 2018.

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

[21] D. Bernstein. Writing high-speed software. http://cr.yp.to/qhasm.html.

[22] F. Besson, A. Dang, and T. Jensen. Securing compilation against memory probing. *PLAS*. 2018.

[23] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke. Compiling sandboxes: Formally verified software fault isolation. In *ESOP*, 2019.

[24] B. Beurdouche, K. Bhargavan, F. Kiefer, J. Protzenko, E. Rescorla, T. Taubert, M. Thomson, and J.-K. Zinzindohoue. HACL* in Mozilla Firefox: Formal methods and high assurance applications for the web. Real World Crypto Symposium, 2018.

[25] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. *USENIX NSDI*, 2008.

[26] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable information flow control. *CCS*. 2017.

[27] V. Cheval, V. Cortier, and S. Delaune. Deciding equivalence-based properties using constraint solving. *TCS*, 492, 2013.

[28] V. Cheval, S. Kremer, and I. Rakotonirina. DEEPSEC: Deciding equivalence properties in security protocols theory and practice. *S&P*. 2018.

[29] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann. Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. *ASPLOS*. 2015.

[30] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. D. Son, M. Roe, S. W. Moore, P. G. Neumann, B. Laurie, and R. N. M. Watson. CHERI JNI: sinking the Java security model into the C. *ASPLOS*. 2017.

[31] M. R. Clarkson and F. B. Schneider. Hyperproperties. *JCS*, 18(6), 2010.

[32] S. Delaune and L. Hirschi. A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. *JLAMP*, 87, 2017.

[33] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *S&P*. 2017.

[34] C. Deng and K. S. Namjoshi. Securing a compiler transformation. *FMSD*, 53(2), 2018.

[35] D. Devriese, M. Patrignani, and F. Piessens. Fully-abstract compilation by approximate back-translation. *POPL*, 2016.

[36] D. Devriese, M. Patrignani, and F. Piessens. Fully abstract compilation by approximate back-translation: Technical appendix. Technical Report CW 687, Dept. of Computer Science, KU Leuven, 2016.

[37] D. Devriese, M. Patrignani, and F. Piessens. Parametricity versus the universal type. *PACMPL*, 2(POPL), 2018.

[38] V. D'Silva, M. Payer, and D. X. Song. The correctness-security gap in compiler optimization. *S&P Workshops*. 2015.

[39] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. *IMC*. 2014.

[40] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Compiling a secure variant of C to capabilities. Dagstuhl Seminar 18201 on Secure Compilation, 2018.

[41] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked C: Making C safe by extension. SecDev, 2018.

[42] J. Engelfriet. Determinacy implies (observation equivalence = trace equivalence). *TCS*, 36, 1985.

[43] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic – with proofs, without compromises. *S&P*, 2019.

[44] M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1-3), 1991.

[45] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2), 1974.

[46] R. Focardi and R. Gorrieri. A taxonomy of security properties for process algebras. *JCS*, 3(1), 1995.

[47] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*. 2013.

[48] J. A. Goguen and J. Meseguer. Security policies and security models. *S&P*, 1982.

[49] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *JCS*, 12(3-4), 2004.

[50] K. Gudka, R. N. M. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson. Clean application compartmentalization with SOAAP. *CCS*. 2015.

[51] Intel. Software guard extensions (SGX) programming reference, 2014.

[52] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. *CSF*. 2011.

[53] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. *ESOP*. 2005.

[54] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *TCS*, 338(1-3), 2005.

[55] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, B. Eng, and B. C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. *CSF*, 2016.

[56] J. Kang, Y. Kim, C.-K. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. *POPL*, 2016.

[57] A. Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3), 2006.

[58] A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX, 1883.

[59] D. Kilpatrick. Privman: A library for partitioning applications. *USENIX FREENIX*. 2003.

[60] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. *POPL*. 2014.

[61] O. Kupferman and M. Y. Vardi. Robust satisfaction. *CONCUR*, 1999.

[62] L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

[63] L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, 1984.

[64] A. Larmuseau, M. Patrignani, and D. Clarke. A secure compiler for

ML modules. *APLAS*, 2015.
[65] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7), 2009.
[66] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer Science & Business Media, 2012.
[67] I. Mastroeni and M. Pasqua. Verifying bounded subset-closed hyperproperties. *SAS*. Springer, 2018.
[68] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.
[69] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything you want to know about pointer-based checking. *SNAPL*. 2015.
[70] G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. *ICFP*. 2015.
[71] M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*, 2016.
[72] M. Pasqua and I. Mastroeni. On topologies for (hyper)properties. *CEUR*. 2017.
[73] M. Patrignani and D. Clarke. Fully abstract trace semantics for protected module architectures. *CL*, 42, 2015.
[74] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. *CSF*. 2017.
[75] M. Patrignani and D. Garg. Robustly safe compilation. *ESOP*, 2019.
[76] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015.
[77] M. Patrignani, D. Devriese, and F. Piessens. On modular and fully-abstract compilation. *CSF*. 2016.
[78] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.
[79] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. *ESOP*. 2014.
[80] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP), 2017.
[81] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*. 2003.
[82] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. *EuroSys*. 2009.
[83] A. W. Roscoe. CSP and determinism in security modelling. *S&P*. 1995.
[84] G. Rosu. On safety properties and their monitoring. *Sci. Ann. Comp. Sci.*, 22(2), 2012.
[85] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1), 2003.
[86] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *HOSC*, 14(1), 2001.
[87] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.
[88] F. Schneider. *On Concurrent Programming*. Texts in Computer Science. Springer New York, 1997.
[89] L. Simon, D. Chisnall, and R. J. Anderson. What you get is what you C: Controlling side effects in mainstream C compilers. *EuroS&P*. 2018.
[90] L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *PACMPL*, 3(POPL), 2019.
[91] B. C. Smith. Reflection and semantics in Lisp. *POPL*. 1984.
[92] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. *POPL*. 2015.
[93] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.
[94] D. Swasey, D. Garg, and D. Dreyer. Robust and compositional verification of object capability patterns. *PACMPL*, 1(OOPSLA), 2017.
[95] G. Tan. Principles and implementation techniques of software-based fault isolation. *FTSEC*, 1(3), 2017.
[96] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP*, 1993.
[97] M. Wand and D. P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1), 1988.
[98] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *S&P*. 2015.
[99] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151, 2014.
[100] A. Zakinthinos and E. S. Lee. A general theory of security properties. *S&P*. 1997.
[101] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. *CSFW*. 2003.
[102] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*, 2017.