Purity of an ST Monad

Full Abstraction by Semantically Typed Back-Translation

KOEN JACOBS, imec-DistriNet, KU Leuven, Belgium
DOMINIQUE DEVRIESE, imec-DistriNet, KU Leuven, Belgium
AMIN TIMANY, Aarhus University, Denmark

In 1995, Launchbury and Peyton Jones extended Haskell with an ST monad that allows the programmer to use higher-order mutable state. They informally argued that these state computations were safely encapsulated, and as such, that the rich reasoning principles stemming from the *purity* of the language, were not threatened.

In this paper, we give a formal account of the preservation of *purity* after adding an ST monad to a simply-typed call-by-value recursive lambda calculus. We state and prove full abstraction when embedding the pure language into its extension with ST; contextual equivalences from the pure language continue to hold in the presence of ST.

Proving full abstraction of compilers is usually done by emulating or *back-translating* the target features (here: ST computations) into the source language, a well-known challenge in the secure compilation community. We employ a novel proof technique for proving our full abstraction result that allows us to use a semantically (but not syntactically) typed back-translation into an intermediate language. We believe that this technique provides additional insight into our proof and that it is of general interest to researchers studying programming languages and compilers using full abstraction.

The results presented here are fully formalized in the Coq proof assistant using the Iris framework.

CCS Concepts: • **Theory of computation → Program reasoning**; **Logic and verification**; **Type theory**; **Separation logic**; • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: ST monad, Functional Programming Languages, Full Abstraction

ACM Reference Format:
Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST Monad: Full Abstraction by Semantically Typed Back-Translation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 82 (April 2022), 27 pages. https://doi.org/10.1145/3527326

1 INTRODUCTION

The ST monad. Purity is one of the defining characteristics of programming languages like Haskell [Hudak et al. 2007]. Intuitively, it refers to the fact that most side effects are only allowed inside a special IO monad, while other code only allows limited side effects, particularly non-termination. This purity has several benefits: it enables the use of a non-strict evaluation order, and perhaps more importantly, it simplifies informal and formal reasoning about programs. At the same time, this purity can also be a disadvantage, for example when implementing algorithms that make critical use of mutable state. For cases where this mutable state is used only internally in an algorithm with an otherwise functional interface, Launchbury and Peyton Jones [1994] have proposed the

Authors' addresses: Koen Jacobs, imec-DistriNet, KU Leuven, Leuven, Belgium, koen.jacobs@kuleuven.be; Dominique Devriese, imec-DistriNet, KU Leuven, Leuven, Belgium, dominique.devriese@kuleuven.be; Amin Timany, Aarhus University, Aarhus, Denmark, timany@cs.au.dk.

2475-1421/2022/4-ART82
https://doi.org/10.1145/3527326

ST monad. This monad offers primitives for allocating, dereferencing, and updating higher-order reference cells. However, the primitives are assigned cleverly-designed types that encapsulate this use of mutable state behind a pure interface.

Concretely, the ST monad offers the following primitives (for clarity, we write explicit foralls):

$$\begin{aligned}
\texttt{return} &:: \forall s, a.\, a \to \texttt{ST}\ s\ a \\
(\gg\!=) &:: \forall s, a, b.\, \texttt{ST}\ s\ a \to (a \to \texttt{ST}\ s\ b) \to \texttt{ST}\ s\ b \\
\texttt{newSTRef} &:: \forall a, s.\, a \to \texttt{ST}\ s\ (\texttt{STRef}\ s\ a) \\
\texttt{readSTRef} &:: \forall s, a.\, \texttt{STRef}\ s\ a \to \texttt{ST}\ s\ a \\
\texttt{writeSTRef} &:: \forall s, a.\, \texttt{STRef}\ s\ a \to a \to \texttt{ST}\ s\ ()
\end{aligned}$$

Computations in the ST monad have type ST s a for type arguments s and a. The type a corresponds to the result type of the computation, while the type argument s is a kind of type-level token that identifies the so-called state thread of the computation; all references allocated within the stateful computation are annotated with s. Conceptually, s can be regarded as naming an isolated fragment of the heap that is owned by a particular stateful computation; hence the name state thread.

ST computations can be constructed and composed using standard monadic operators return and, bind, $\gg\!=$, respectively constructing a trivial computation that simply returns a value and sequentially composing a stateful computation of type ST s a with a continuation of type $a \to$ ST s b. These two primitives allow us to use the monadic do notation to write ST computations in an imperative style: $cmp1 \gg\!= (\lambda x.\, cmp2)$ can be written as

$$\begin{aligned}
&\textsf{do}\ x \leftarrow cmp1 \\
&\quad cmp2
\end{aligned}$$

Additionally, ST offers methods newSTRef, readSTRef, and writeSTRef for allocating, dereferencing, and updating mutable references. Mutable references of type a within an ST computation have the type STRef s a. The type of newSTRef enforces that this s must correspond to the ST computation within which the reference has been allocated and the types of readSTRef and writeSTRef enforce that they can only be used within this computation.

The crucial idea behind the ST monad is in the type of the runST primitive:

$$\texttt{runST} :: \forall a.\, (\forall s.\texttt{ST}\ s\ a) \to a$$

This primitive turns an ST computation of type ST s a into a pure value of type a. It is this primitive that allows encapsulating a stateful computation behind a pure interface, as the result is a pure value of type a. To make sure this primitive does not break the purity of the language, it has a special type that prevents the mutable state effect from leaking into pure code. Note the order of quantification of a and s in the type of runST. This particular order means that the type s is not in scope when a is introduced and hence the variable s, which any reference allocated within the computation run by runST will be annotated with, cannot appear in a. Hence, no reference can escape out of a computation run by runST. For instance, attempting to leak an STRef s a reference out of an ST computation by invoking runST (newSTRef 42) results in an error about the type argument s escaping its scope. Moreover, the computation c run by runST is polymorphic in its state thread identifier type. Hence, c cannot assume anything about the state thread s that it will be running in and particularly, can only use STRef's that are allocated within c itself. For example, if we attempt to dereference an STRef *ref* from a different ST thread by invoking runST (readSTRef *ref*), we will get an error about a mismatch of the involved type arguments s.

Formally characterizing purity. In these examples, `runST`'s type effectively prevents mutable effects from `ST` from leaking into the pure part of the language. But how can we be sure that the design works in general, i.e. how can we be sure that adding `ST` does not somehow break the purity of the language? Of course, answering this question first requires us to clarify what purity of a language means exactly. Launchbury and Peyton Jones [1994] already suggest an answer in their paper: they say that (quote):

> [After adding `ST`,] all the usual techniques for reasoning about functional programs continue to work.

While they do give an informal justification for this (more on this later), they do not formally specify and prove the claim.

More recently, Timany et al. [2017b] have made this idea more formal in the context of a recursive call-by-value (cbv) System F extended with an `ST` monad. They present and prove a set of equivalences (and refinements) that one would typically expect in a cbv pure language, for instance, the following equivalences for arbitrary expressions e, e_1, and e_2:

$$\texttt{let } x = e_2 \texttt{ in } (e_1, x) \approx_{\text{ctx}} (e_1, e_2) \qquad \text{(commutativity)}$$

$$\texttt{let } x = e \texttt{ in } (x, x) \approx_{\text{ctx}} (e, e) \qquad \text{(idempotency)}$$

The equivalences above express that the order of expressions can be swapped and that expressions can be duplicated. Such properties clearly would not hold in the presence of effects (e.g. mutable references), and as such, they capture a form of purity of the language with `ST`. Note that these equivalences should be interpreted as contextual equivalences, i.e. replacing one expression by the other in a larger program, does not modify the behavior of the program. More formally, two terms t_1 and t_2 are contextually equivalent if embedding them in an arbitrary program context C produces equiterminating terms: $C[t_1] \Downarrow$ iff $C[t_2] \Downarrow$.

While these results are strong, the proven equivalences only cover a specific part of the equational theory of a pure language and it remains uncertain whether this subset fully captures what it means for a programming language to be pure. In this paper, we remove this remaining uncertainty. Essentially, we will prove that adding `ST` does not just preserve certain equivalences, but *all* equivalences that hold in the original language. More precisely, we will establish that terms in the original language (without `ST`) are equivalent if and only if they are equivalent after adding `ST`. Our result has practical benefits: when reasoning about equivalences of pure programs, programmers are justified to pretend `ST` does not exist, and the same applies when reasoning about compiler optimizations. From a more theoretical point of view, our result establishes Abadi's full abstraction property [Abadi 1998, 1999], for the function that embeds the language without `ST` into its extension with `ST`. Full abstraction is commonly used to compare the expressive power of languages [see, e.g. Parrow 2008; Patrignani et al. 2021]. From this perspective, our results imply that adding `ST` does not increase the expressiveness of the base language: contexts in both languages have equal expressive power to distinguish two given programs of a given type.

Proving preservation of equivalence. Proving preservation of equivalence (the forward and hardest direction of full abstraction) is a difficult problem. As mentioned, we need to prove that if no pure context can distinguish between two pure terms, then no context using `ST` can do so either. The standard way to do this is to back-translate surrounding contexts that use `ST` to pure contexts that emulate their behavior.

Fortunately, Launchbury and Peyton Jones [1994] provide us with a hint on how to do this. To justify their argument that `ST` does not jeopardize purity, they note that stateful computations can be *thought of* as being *pure* (even though they are implemented using in-place modifications of a global heap). Specifically, they remark that

> ... a stateful computation is a *state transformer*, that is, a function from an initial state to a final state. It is like a "script", detailing the actions to be performed on its input state.

This intuition of ST computations as state transformers suggests emulating a stateful computation by a pure function with an explicit state-passing style, à la State monad as described by Wadler [1993]. Concretely, a stateful computation of type ST *s Bool* could be back-translated to a function of type $S \to (S \times \text{Bool})$. Unfortunately, it is far from obvious what type S should be. In general, if we want to fully back-translate arbitrary contexts, S should be able to represent an arbitrary heap, as any number of references may be allocated in any order, not to mention that they may have higher-order types.

The insight of our solution. To solve this problem, we start by observing that the challenge lies in the syntactic well-typedness of the back-translation. Producing an untyped back-translation that translates ST to a state monad is not very hard. In fact, when interacting with pure code, this internal use of the hard-to-type state monad would not even be observable. More formally, without a definition of type S, such a back-translation is not syntactically well-typed, but it would be semantically well-typed, i.e., it will *behave* like a term of the right type even though it is not a term of that type — we will clearly and formally explain what we mean by semantic typing later on.

This leads us to the main idea of our paper: we will construct our back-translation in two steps. In a first step, computations with ST are back-translated to semantically typed pure computations. Subsequently, semantically typed contexts are back-translated to syntactically typed contexts by using a universal embedding, as in earlier work [Devriese et al. 2016; Jacobs et al. 2021; New et al. 2016; Patrignani et al. 2021]. We believe this idea can be useful in other settings where back-translations can be constructed but not easily typed. In fact, the second part of our back-translation (from semantically to syntactically typed pure code) leads to an independently valuable full abstraction result (for the embedding of syntactically typed into semantically typed code) that we think might have other applications than our study of ST.

ST but not Haskell. Before we continue to list our contributions and give an overview of the paper, we want to make it clear that our results do not directly apply to Haskell. Instead, we formalize our solution for a pure *call-by-value* recursive simply-typed lambda calculus and its extension with an ST monad. As already mentioned by Timany et al., the ST monad's safe encapsulation of state is orthogonal to the evaluation order of the language, and it is easier to formalize a cbv evaluation strategy. Working with a simply-typed calculus, we furthermore only model the polymorphism over state thread identifiers. We omit System F's type abstraction and application to simplify the problem. We discuss this decision and its impact in more detail in section 6.

Contributions. Our contributions are as follows:

(1) We propose full abstraction as a formal account for the preservation of purity after adding an ST monad.
(2) We prove full abstraction of the embedding from a pure cbv simply-typed lambda-calculus with iso-recursive types into its stateful extension with an ST monad.
(3) Our full abstraction proof uses a novel proof technique that defines an intermediate step using the idea of semantic typing. As we will discuss, this novel proof technique provides valuable new insight into the result, and is potentially applicable to other full abstraction proofs where it is difficult/impossible to have a syntactically typed back-translation.
(4) The formal results that we present here are all fully formalized [Jacobs et al. 2022] in the Coq proof assistant on top of the Iris framework [Jung et al. 2018].

$$
\begin{aligned}
e ::= {} & () \mid e; e \mid z \mid e \circledcirc e \mid \mathit{true} \mid \mathit{false} \mid \mathit{if}\ e\ \mathit{then}\ e\ \mathit{else}\ e \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid \\
& \mathit{inj}_1\ e \mid \mathit{inj}_2\ e \mid \mathit{case}\ e\ \mathit{of}\ \mathit{inj}_1\ x_1 \Rightarrow e, \mathit{inj}_2\ x_2 \Rightarrow e \mid \mathit{fold}\ e \mid \mathit{unfold}\ e \mid \\
& x \mid \mathit{let}\ x = e\ \mathit{in}\ e \mid \lambda x.\, e \mid e\ e \mid \\
& \ell \mid \mathbf{ref\ e} \mid \mathbf{!e} \mid \mathbf{e \leftarrow e} \mid \mathbf{e \gg\!= e} \mid \mathbf{return\ e} \mid \mathbf{runST\ \{e\}} \mid \mathbf{compare\ e\ e} \\
v ::= {} & () \mid z \mid \mathit{true} \mid \mathit{false} \mid (v, v) \mid \mathit{inj}_1\ v \mid \mathit{inj}_2\ v \mid \mathit{fold}\ v \mid \lambda x.\, e \mid \\
& \ell \mid \mathbf{ref\ v} \mid \mathbf{!v} \mid \mathbf{v \leftarrow v} \mid \mathbf{v \gg\!= v} \mid \mathbf{return\ v} \\
\tau ::= {} & 1 \mid Z \mid B \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\, \tau \mid \tau \to \tau \mid \mathbf{STRef\ X\ \tau} \mid \mathbf{ST\ X\ \tau}
\end{aligned}
$$

Fig. 1. Grammars of $\lambda^{\vdash}$ and $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$. Common parts are left in black.

Overview paper. The remainder of the paper is structured as follows. In section 2, we will formally present the relevant three languages: the cbv simply-typed lambda calculus with iso-recursive types, its semantically typed extension, and its extension with an ST monad. Afterwards, in section 3, we will present the properties that we will prove in detail after which we give an overview of the proof, splitting it into the two parts mentioned in this introduction. The proof of the first part will be laid out in section 4, the second in section 5. In the sections that follow, we have some discussions, go over related and future work, and conclude the paper.

2 PRELIMINARIES

2.1 Definitions of $\lambda^{\vdash}$ and $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$

Here, we present $\lambda^{\vdash}$ and $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$. Inspired by [Patrignani 2020], we use a blue, sans−serif font for everything related to $\lambda^{\vdash}$ and a **red**, **bold** one for $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$. The former is a pretty standard curry-style call-by-value simply-typed lambda calculus with unit, integer, boolean, product, sum, arrow types, and iso-recursive types. The latter is the former extended with the same ST monad that is defined by Timany et al. [2017b]. In essence, $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$ is just a simplified version of their STLang (System F_{μ} extended with the ST monad), where we keep only the polymorphism over state thread identifiers.

We first present the grammar of expressions, values, and types of $\lambda^{\vdash}$ and $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$ in Figure 1. Here, we use a *black*, *unformatted* font for the elements common to both languages. These are quite standard. We have a unit value $()$, and the sequencing operator $e; e$ as the "eliminator" for unit values. The expression z is meant to represent an arbitrary integer. The operator $\circledcirc$ ranges over a family of operators (+, −, and =), all to be interpreted as binary operators on integers. Furthermore, we have the standard constructors and destructors for respectively booleans, pairs, sums, recursive expressions, and functions. In $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$, our expressions additionally include (in order of presentation) locations, the allocation primitive, the reading primitive, the writing primitive, the bind operator, the return operator, the **runST**-operator that executes a state computation, and lastly, an additional construct to check whether two references are equal.

Note that all these additional constructors, except **runST** and **compare**, can form values. As such, **ST** computations can be thought of as scripts whose execution is delayed until, as we shall see, they are executed under a **runST**-operator.

Evaluation steps in $\lambda^{\vdash}$ are pure, and thus they are simply of the form $\mathbf{e} \longrightarrow \mathbf{e}'$. Evaluation steps in $\boldsymbol{\lambda}^{\vdash}_{\mathbf{ST}}$ are stateful however, requiring as an additional argument a heap, $h : \mathbf{Loc} \rightharpoonup \mathbf{Val}$, represented by a partial finite map from locations to values. In both languages, evaluation steps are defined in terms of head steps (primitive reduction steps) and evaluation contexts. Evaluation contexts are defined as follows (again, the parts left in black are common to both languages).

$$
\begin{aligned}
K ::= {} & [\,] \mid K; e \mid K \circledcirc e \mid v \circledcirc K \mid \mathit{if}\ K\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid (K, e) \mid (v, K) \mid \pi_1\ K \mid \pi_2\ K \mid \\
& \mathit{inj}_1\ K \mid \mathit{inj}_2\ K \mid \mathit{case}\ K\ \mathit{of}\ \mathit{inj}_1\ x_1 \Rightarrow e_1, \mathit{inj}_2\ x_2 \Rightarrow e_2 \mid \mathit{fold}\ K \mid \mathit{unfold}\ K \mid \\
& \mathit{let}\ x = K\ \mathit{in}\ e \mid K\ e \mid v\ K \mid \\
& \mathbf{compare\ K\ e} \mid \mathbf{compare\ v\ K} \mid \mathbf{ref\ K} \mid \mathbf{!K} \mid \mathbf{K \leftarrow e} \mid \mathbf{v \leftarrow K} \mid \\
& \mathbf{K \gg\!= e} \mid \mathbf{v \gg\!= K} \mid \mathbf{return\ K} \mid \mathbf{runST\ \{K\}}
\end{aligned}
$$

Most of the head steps are entirely standard. We restrict ourselves here to the ones concerned with the ST monad in $\lambda^{\vdash}_{\mathrm{ST}}$.

$$
\frac{\langle h, \mathbf{v}\rangle \rightsquigarrow \langle h', \mathbf{e}\rangle}{\langle h, \mathbf{runST\ \{v\}}\rangle \to_h \langle h', \mathbf{runST\ \{e\}}\rangle} \qquad \langle h, \mathbf{runST\ \{return\ v\}}\rangle \to_h \langle h, \mathbf{v}\rangle
$$

$$
\frac{\ell_1 \neq \ell_2}{\langle h, \mathbf{compare}\ \ell_1\ \ell_2\rangle \to_h \langle h, \mathbf{false}\rangle} \qquad \langle h, \mathbf{compare}\ \ell\ \ell\rangle \to_h \langle h, \mathbf{true}\rangle
$$

The rule on the upper right escapes a returned value out of the stateful computation. On the upper left side, we have a rule in which the assumption features a wiggly arrow that denotes an effectful step. The effectful steps are ones that depend on the state of the heap (and possible change it). As we can see, stateful computations will be evaluated down as much as possible to a value, but no effectful evaluation will actually occur unless it is forced so under a runST-operator. On the lower half, we have two rules for the comparison of two locations. Note that as such a comparison does not really depend on the heap, its execution can happen outside the runST-operator. The effectful steps themselves are in turn defined by a different kind of evaluation contexts; the effectful evaluation contexts as defined below.

$$
\mathcal{K} ::= [\,] \mid \mathcal{K} \mathbf{\gg\!= v}
$$

The effectful head steps are defined below.

$$
\langle h, \mathbf{return\ v \gg\!= v'}\rangle \rightsquigarrow_h \langle h, \mathbf{v'\ v}\rangle \qquad \frac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathbf{ref\ v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathbf{v}\}, \mathbf{return}\ \ell\rangle}
$$

$$
\langle h \uplus \{\ell \mapsto \mathbf{v}\}, \mathbf{!}\ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathbf{v}\}, \mathbf{return\ v}\rangle
$$

$$
\langle h \uplus \{\ell \mapsto \mathbf{v'}\}, \ell \leftarrow \mathbf{v}\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto \mathbf{v}\}, \mathbf{return\ ()}\rangle
$$

Apart from their monadic structure, these rules are quite standard. Consider the evaluation rule for allocation for instance, and note that it does not simply evaluate to ℓ, but to $\mathbf{return}\ \ell$.

Lastly, we will introduce the typing rules. The ones for $\lambda^{\vdash}$ are of the form $\Gamma \vdash \mathbf{e} : \tau$ where we assume every type to be closed. As they are entirely standard [Pierce 2002], we omit them here. Typing rules in $\lambda^{\vdash}_{\mathrm{ST}}$ are of the form $\Xi \mid \Gamma \vdash \mathbf{e} : \tau$, where Ξ is a collection of type variables that keeps track of the free type variables in Γ and τ. The relation $\Xi \vdash \tau$ denotes that all the free variables inside τ are in Ξ. The typing rules related to the stateful computations are defined as follows.

$$\frac{\Xi \mid \Gamma \vdash \mathbf{e} : \mathbf{STRef\ X}\ \tau \qquad \Xi \mid \Gamma \vdash \mathbf{e}' : \mathbf{STRef\ X}\ \tau}{\Xi \mid \Gamma \vdash \mathbf{compare\ e\ e}' : \mathbf{B}} \qquad \frac{\Xi \mid \Gamma \vdash \mathbf{e} : \tau \qquad \Xi \vdash \mathbf{X}}{\Xi \mid \Gamma \vdash \mathbf{ref\ e} : \mathbf{ST\ X\ (STRef\ X}\ \tau)}$$

$$\frac{\Xi \mid \Gamma \vdash \mathbf{e} : \mathbf{STRef\ X}\ \tau}{\Xi \mid \Gamma \vdash\ \mathbf{!e} : \mathbf{ST\ X}\ \tau} \qquad \frac{\Xi \mid \Gamma \vdash \mathbf{e} : \mathbf{STRef\ X}\ \tau \qquad \Xi \mid \Gamma \vdash \mathbf{e}' : \tau}{\Xi \mid \Gamma \vdash \mathbf{e} \leftarrow \mathbf{e}' : \mathbf{ST\ X\ 1}} \qquad \frac{\Xi \mid \Gamma \vdash \mathbf{e} : \tau \qquad \Xi \vdash \mathbf{X}}{\Xi \mid \Gamma \vdash \mathbf{return\ e} : \mathbf{ST\ X}\ \tau}$$

$$\frac{\Xi \mid \Gamma \vdash \mathbf{e} : \mathbf{ST\ X}\ \tau \qquad \Xi \mid \Gamma \vdash \mathbf{e}' : \tau \to \mathbf{ST\ X}\ \tau'}{\Xi \mid \Gamma \vdash \mathbf{e} \gg\!= \mathbf{e}' : \mathbf{ST\ X}\ \tau'} \qquad \frac{\Xi, \mathbf{X} \mid \Gamma \vdash \mathbf{e} : \mathbf{ST\ X}\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathbf{runST}\ \{\mathbf{e}\} : \tau}$$

The typing rules mirror those of Haskell's ST-monad. Instead of explicit quantification however, we keep track of the free type variables inside the first argument of the typing judgements. Note in particular the typing rule for the runST-operator. In its assumption, we have a typing judgement of e of type ST X τ under the collection of free type variables Ξ, X (with this notation, we implicitly assume that $\mathbf{X} \notin \Xi$). To be able to conclude that runST {e} is of type τ under the collection of free type variables Ξ, we additionally require that τ does not contain X as a free type variable (imposed by $\Xi \vdash \tau$).

The remaining typing rules are inherited straightforwardly from $\lambda^{\vdash}$.

2.2 Defining the Intermediate Language $\lambda^{\models}$

In this section, we will define our intermediate language by means of a type-indexed logical relation that we define in Iris, a higher-order separation logic [Jung et al. 2018]. Before we do so, we give a very superficial Iris primer.

2.2.1 A Superficial Iris Primer. Iris is a higher-order separation logic geared towards reasoning about programs [Jung et al. 2018]. In the following grammar for Iris propositions, we list some of its constructs.

$$P = P * P \mid P \mathrel{-\!\!*} P \mid P \vee P \mid \triangleright P \mid \mu x.P \mid \Box P \mid {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}'} \mid \boxed{P}^{\mathcal{N}} \mid \ldots$$

Before going over these constructs, we quickly note that throughout this paper, we will shade out symbols that do not significantly contribute to an informal understanding. The reader who's unfamiliar with Iris can therefore just ignore them. For a deeper understanding, we refer the interested reader to Jung et al. [2018].

As with any separation logic, we have a separating conjunction $*$ (pronounced "star") and the separating implication $\mathrel{-\!\!*}$ (pronounced "magic wand"). The reader unfamiliar with separation logic may think of these for now as "regular" conjunction and implication. In section 5, we will use informal language to convey the required intuition where needed.

Moreover, we have the later modality $\triangleright$ (pronounced "later"), and the guarded fixpoint operator $\mu x.P$. The later modality represents in the logic an abstract form of step-indexing [Appel and McAllester 2001]. As such, a proposition $\triangleright P$ may be understood as stating that P holds *one step later*. The guarded fixpoint operator allows us to take *guarded* fixpoints. That is, for $\mu x.P$ to be well-defined every recursive occurrence of x needs to appear under (i.e., be guarded by) the later modality. With these we can, for instance, define the weakest precondition[1] for our pure language. Informally, the proposition wp e $\{\Phi\}$ encodes that if e evaluates to a value, then the postcondition (a predicate on values, Φ here) should hold for that value. Formally, we have the following.

[1]Throughout this paper, we will only use what, in Iris terminology, is known as the "stuck" version of the weakest precondition that does not imply safety.

$$
\begin{aligned}
\mathcal{V}_{int.}[\![1]\!](v, v') &\triangleq v = () * v' = () \\
\mathcal{V}_{int.}[\![B]\!](v, v') &\triangleq \exists b \in \{true, false\}.\ v = b * v' = b \\
\mathcal{V}_{int.}[\![Z]\!](v, v') &\triangleq \exists z \in \mathbb{Z}.\ v = z * v' = z \\
\mathcal{V}_{int.}[\![\tau_1 + \tau_2]\!](v, v') &\triangleq \bigvee_{i \in \{1,2\}} \exists w, w'.\ v = \mathsf{inj}_i\ w * v' = \mathsf{inj}_i\ w' * \mathcal{V}_{int.}[\![\tau_i]\!](w, w') \\
\mathcal{V}_{int.}[\![\tau_1 \times \tau_2]\!](v, v') &\triangleq \exists v_1, v_1', v_2, v_2'.\ v = (v_1, v_2) * v' = (v_1', v_2') \\
&\qquad * \mathcal{V}_{int.}[\![\tau_1]\!](v_1, v_1') * \mathcal{V}_{int.}[\![\tau_2]\!](v_2, v_2') \\
\mathcal{V}_{int.}[\![\tau_1 \to \tau_2]\!](v, v') &\triangleq \square\left(\forall w, w'.\ \mathcal{V}_{int.}[\![\tau_1]\!](w, w') \mathrel{-\!\!*} \mathsf{lift}\ \mathcal{V}_{int.}[\![\tau_2]\!]\ (v\ w, v'\ w')\right) \\
\mathcal{V}_{int.}[\![\mu X.\tau]\!](v, v') &\triangleq \exists w, w'.\ v = \mathsf{fold}\ w * v' = \mathsf{fold}\ w' * \triangleright\, \mathcal{V}_{int.}[\![\tau[\mu X.\tau/X]]\!](w, w')
\end{aligned}
$$

Fig. 2. Type-indexed logical relations on values for intermediate language

$$
\begin{aligned}
\mathsf{wp}\ e\ \{\Phi\} \triangleq\ & (e \textit{ is a value} \wedge {}^{\top}\!\Rrightarrow^{\top} \Phi(e)) \vee \\
& \left(e \textit{ is not a value} \mathrel{-\!\!*} {}^{\top}\!\Rrightarrow^{\emptyset} (\triangleright \forall e'.\ e \to e' \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\top} \mathsf{wp}\ e'\ \{\Phi\})\right)
\end{aligned}
$$

If e is a value, then it should satisfy Φ. If it is not, then under every step it can take to a new expression, say e', $\mathsf{wp}\ e'\ \{\Phi\}$ needs to hold. The apparent circularity is broken by our use of the guarded fixpoint here, which is indeed valid as the recursive occurrence of the weakest precondition appears under the later modality.

2.2.2 Presentation of the Logical Relations that Define the Semantically Typed Language. We first present a binary type-indexed logical relations model. Terms in $\lambda^{\vDash}$ will be those that are related to themselves in this relation. As is standard, we first define our logical relations on closed values, after which we will expand it to closed expressions, and finally to open expressions.

Logical Relations on Values. The value relations are defined in Figure 2. The relation is defined by structural recursion on the types, as well as by guarded recursion. Note that in all the cases but the one for $\mu X.\tau$, the types are structurally smaller in the recursive calls. In the case for $\mu X.\tau$, the recursive call appears under a later-modality, making Löb induction possible.

Two values are related at a base type, if they share the same value of that type. Values at a sum type are related if they are of the same injection, and the injected values are related. At the product type, values are related if they are both pairs, and their components are related. Two values are related at the arrow type if when applying them with any related values at the argument type, one obtains expressions that are related at the result type; we use a lift operator to lift the value predicate (relatedness at result type) to a predicate on expressions; we explain this operator in the next paragraph. Lastly, values are related at the recursive type if they are both folded, and their (unfolded) bodies are, *one step later*, related at the unfolded type.

Logical Relations on Expressions. We lift our value relation to a relation on closed expressions with a lift operator defined as follows.

$$
\begin{aligned}
&\mathsf{lift} : (\mathsf{Val} \to \mathsf{Val} \to \mathit{iProp}) \to (\mathsf{Expr} \to \mathsf{Expr} \to \mathit{iProp}) \\
&\mathsf{lift}\ \Phi\ (e, e') = \mathsf{wp}\ e\ \left\{v.\ \exists v'.\ e' \to^{*} v' * \Phi(v, v')\right\}
\end{aligned}
$$

Roughly, e and e' satisfy lift Φ, if when e evaluates to a value, say v, e' must do so as well, say to v', and we have furthermore that $\Phi(v, v')$.

For any type τ, we now define the expression relation at that type by $\mathcal{E}_{int.}[\![\tau]\!] = \mathsf{lift}\, \mathcal{V}_{int.}[\![\tau]\!]$.

Logical Relations on Open Expressions. To define the relation on open expressions, we first define a value relation on *contexts*.

$$\vec{\mathcal{V}}_{int.}[\![\varnothing]\!](\epsilon, \epsilon) \triangleq \mathsf{True}$$
$$\vec{\mathcal{V}}_{int.}[\![x : \tau; \Gamma]\!](v; \vec{v}, v'; \vec{v}') \triangleq \mathcal{V}_{int.}[\![\tau]\!](v, v') * \vec{\mathcal{V}}_{int.}[\![\Gamma]\!](v', \vec{v}')$$

In essence, two vectors of values are related at a context if each of their components are appropriately related.

Two open expressions are now related under a particular type and context, if after substitution by arbitrary related vectors of values (at that context), they are related as closed expressions (at the type).

$$\Gamma \vDash_{int.} e \le e' : \tau \triangleq \forall \vec{v}, \vec{v}'. \vec{\mathcal{V}}_{int.}[\![\Gamma]\!](\vec{v}, \vec{v}') \vdash \mathcal{E}_{int.}[\![\tau]\!](e[\vec{v}'/\vec{x}], e'[\vec{v}'/\vec{x}'])$$

Here we have used the symbol $\vdash$, to be interpreted as the provability judgement in Iris. Relatedness of open expressions is thus a meta-level statement.

Two contexts, say C and C, are related at $(\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$, if plugging them in with any two related terms, say $\Gamma \vDash_{int.} e \le e' : \tau$, we obtain two terms that are related at Γ' and τ', that is, we have $\Gamma' \vDash_{int.} C[e] \le C'[e'] : \tau'$. We denote this by $\vDash_{int.} C \le C' : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$.

Some Important Theorems. Out of our definition of lift (and adequacy of weakest preconditions)[2], we get the following adequacy theorem, stating that our logical relations model refinement.

Lemma 2.1 (Logical Relation Adequacy). *If* $\cdot \vDash_{int.} e \le e' : \tau$*, then if* e *halts to a value, so must* e'.

Its importance will become clear later on.

Given any context Γ and type τ, we say that a particular expression e is *semantically* typed if it is related to itself: that is, we have $\Gamma \vDash_{int.} e \le e : \tau$. The intermediate language, $\lambda^{\vDash}$ consists of all these *semantically* typed terms. We use $\Gamma \vDash_{int.} e : \tau$ as a shorthand for $\Gamma \vDash_{int.} e \le e : \tau$. Similarly, we write $\vDash_{int.} C : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$ as a shorthand for $\vDash_{int.} C \le C : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$.

The following theorem implies that $\lambda^{\vDash}$ contains all programs from $\lambda^{\vdash}$.

Theorem 2.2 (Fundamental Theorem Intermediate Language). *For any well* syntactically *typed expression (in* $\lambda^{\vdash}$*), say* $\Gamma \vdash e : \tau$*, we automatically have that* $\Gamma \vDash_{int.} e : \tau$.

Virtual Steps. The logical relation that we presented here suffers from a technical shortcoming that we hit upon in subsection 4.3. Here, we explain the issue and demonstrate how we address it.

At its core, the problem stems from the fact that our logical relation does not exploit the fact that the values in our programming language are "finite". As a consequence, our logical relation does not capture some of the pairs we would like it to.

Consider, for instance, the type $\mathsf{list}\, Z \triangleq \mu X.1+(Z \times X)$ and the following two identity functions on it: the simple identity function $\lambda x.x$, and the function $f \triangleq \mathsf{map}\ (\lambda x.x + 0)$ which traverses the list and adds 0 to all elements. The two functions are contextually equivalent, for a subtle reason: the values in our language are finite. Applying f to a list $v : \mathsf{list}\, Z$ will only terminate because v is finite. In contrast, the two functions are decidedly not contextually equivalent when considered as part of a language that allows for infinite lists (applying f to an infinite value diverges while applying $\lambda x.x$ trivially terminates).

[2]Our weakest precondition is just a particular instantiation of the very general one defined in the Iris programming logic, that comes with its own adequacy theorem.

Unfortunately, the definition of our logical relation is not able to take advantage of this intuition. For instance, our trivial function $\lambda x.x$ does not relate to f on the right-hand side.

$$\not\models \mathcal{V}_{int.}[\![\text{list Z}\to\text{list Z}]\!](\lambda x.x, f)$$

To see why this does not hold, let us unfold the definition for the function type.

$$\forall v, v'. \mathcal{V}_{int.}[\![\text{list Z}]\!](v, v') \not\models \mathcal{E}_{int.}[\![\text{list Z}]\!]((\lambda x.x)\ v, f\ v')$$

Remember now that we used guarded recursion to model relatedness between recursive types. As such, we need to consider all related pairs of values, v and v', even those that may only be "approximately related". More concretely, we must consider e.g. the following two lists, $[1, 2, 3, 4]$ and $[1, 2, 3, \text{true}]$ that are related at list Z "only up to 4 steps"; that is, unfolding the definitions of our value relation, we need to go under 4 $\triangleright$'s before we can finally inspect that something is amiss (i.e. true is not related to 4 at type Z as true is not even an integer).

Given such a pair related only up to a certain number of steps, we need to prove that applying $\lambda x.x$ and f to them gives us two *expressions* that are related "up to the same level of approximation". Note that our expression relation is defined in terms of the weakest precondition which is defined in terms of guarded recursion; as such two expressions can also be related only *approximately*. Roughly, two expressions e and e' are related at type list Z at a certain amount of steps when the following is satisfied: *if* e terminates to a value in less than that amount of steps, then e' terminates to a value as well and the two values are related (at list Z) at least up to the remaining amount of steps.

In our example, $(\lambda x.x)\ [1, 2, 3, 4]$ trivially evaluates to a value in only one step. Hence, we need to prove that $f\ [1, 2, 3, \text{true}]$ evaluates to a value as well. This is not true however as it will get stuck trying to evaluate $\text{true} + 0$.

The logical relations (and others like it in the literature) are thus far from complete. But as mentioned, we may find ourselves (as in subsection 4.3) to be exactly interested in proving the kind of equivalences as described above.

To address this problem, our approach is to proceed in two stages. We first define an alternative semantics for our $\lambda x.x$ function that takes additional, artificial or virtual steps; upon every application we traverse the function argument. Formally, we surround the function body with a new construct $\mapsto(\_)$, obtaining the expression $(\lambda x.\mapsto(x))\ [1, 2, 3, 4]$, that takes more than 3 steps, so that the problem above is no longer present. In a second stage, we will then prove that these additional virtual steps do not alter the termination behavior of programs, expressing essentially the fact that values are finite in our language.

To define the semantics with virtual steps, we enrich the grammar of expressions over which the relation is defined with a new "virtual step" primitive.

$$e ::= \ldots \mid \mapsto(e)$$

The evaluation contexts are further extended with the virtual step primitive, and the head steps on values are extended as follows.

$$\mapsto(w) \to_h w \quad \textit{if } w = (), b, z \qquad \mapsto(\text{inj}_1\ v) \to_h \text{inj}_1\ (\mapsto(v))$$
$$\mapsto(\text{inj}_2\ v) \to_h \text{inj}_2\ (\mapsto(v)) \qquad \mapsto((v_1, v_2)) \to_h (\mapsto(v_1), \mapsto(v_2))$$
$$\mapsto(\lambda x.e) \to_h \lambda y.((\lambda x.\mapsto(e))\ \mapsto(y)) \qquad \mapsto(\text{fold}\ v) \to_h \text{fold}\ (\mapsto(v))$$

The virtual step construct thus simply steps through a value leaving it unchanged save for the lambda case, where this "stepping through" is postponed until application, to its input and output.

As described above, we can now prove our equivalence by proving that 1) $\lambda x.\mapsto(x)$ relates to $\text{map}\ (\lambda x.x + 0)$, and 2) $\lambda x.x$ relates to $\lambda x.\mapsto(x)$. The nitty-gritty behind the approach is a bit technical

but not that important. Proving (1) becomes possible as the virtual step construct introduces the necessary approximation required to avoid the need to prove termination of an application to f with a value that only behaves partially as something of type list Z. We can prove (2) as we can abuse the fact that for *any* possibly untyped value v, the expression $\mapsto\!\!\to(v)$ is guaranteed to terminate (because essentially, v is finite).

Formally, we are now working with two different sets of grammars (and corresponding evaluation steps): the canonical one for $\lambda^{\vdash}$, and the one extended with the virtual step primitive corresponding to the one for $\lambda^{\models}$. Practically however, we treat $\lambda^{\vdash}$ in this paper as though it was defined by the same extended grammar while we leave the virtual step construct untyped.[3] This saves us a lot of notational overhead as we do not need to embed canonical terms into the extended grammar. It is also innocuous as the evaluation rules of the extended untyped language are still deterministic, so the syntactically well-typed fragment of the language (and its evaluation) remains unaffected. In the Coq formalization, we go through the trouble of explicitly defining both grammars, and our formal results use the canonical definition of $\lambda^{\vdash}$ from Section 2.1 (without virtual steps).

Notes. Apart from our addition of the virtual step construct, the definition of the logical relations presented here, and the theorems about them, is quite standard [Timany et al. 2017a].

What might seem more peculiar, is our definition of a semantic typing in terms of auto-relatedness of a binary relation (rather than satisfaction of a unary one). As we shall see in section 4, much of our proof relies on complex relational reasoning, for which a unary relation does not seem amenable.

3 OUR MAIN THEOREM AND OVERVIEW OF THE PROOF

Formal Preliminaries. Before stating full abstraction of the embedding from $\lambda^{\vdash}$ into $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$, we need to formally state some preliminaries. We start out by the notion of contextual equivalence. We state it for $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$; the definition for $\lambda^{\vdash}$ is analogous.

Definition 3.1 (Contextual Equivalence by Equitermination in $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$). Given two well-typed expressions in $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$ under the same context and type, say $\Xi \mid \Gamma \vdash \mathbf{e}_1 : \tau$ and $\Xi \mid \Gamma \vdash \mathbf{e}_2 : \tau$, then $\mathbf{e}_1$ and $\mathbf{e}_2$ are *contextually equivalent*, written $\Xi \mid \Gamma \vdash \mathbf{e}_1 \approx_{\mathrm{ctx}} \mathbf{e}_2 : \tau$, if for all well-typed contexts $\vdash \mathbf{C} : (\Xi \mid \Gamma; \tau) \Rightarrow (\cdot \mid \cdot\,; \mathbf{1})$, the following holds:

$$\mathbf{C}[\mathbf{e}_1] \Downarrow \textit{ iff } \mathbf{C}[\mathbf{e}_2] \Downarrow$$

For any $\mathbf{e}$, the notation $\mathbf{e} \Downarrow$ means that $\mathbf{e}$ terminates to a value (formally: $\exists \mathbf{v}, h_f.\ \langle \emptyset, \mathbf{e}\rangle \rightarrow^* \langle h_f, \mathbf{v}\rangle$).

Formal Statements of Main Theorems. Let us now formally state our two main results.

Theorem 3.2 (Reflection of Contextual Equivalence). *Given any two well-typed expressions, $\Gamma \vdash \mathbf{e}_1 : \tau$ and $\Gamma \vdash \mathbf{e}_2 : \tau$, if their embeddings (we denote $\lceil\lceil\_\rceil\rceil : \lambda^{\vdash} \rightarrow \boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$ the embedding from $\lambda^{\vdash}$ into $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$) are contextually equivalent, that is $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil\mathbf{e}_1\rceil\rceil \approx_{ctx} \lceil\lceil\mathbf{e}_2\rceil\rceil : \lceil\lceil\tau\rceil\rceil$, then the original terms are equivalent ($\Gamma \vdash \mathbf{e}_1 \approx_{ctx} \mathbf{e}_2 : \tau$).*

Out of the two theorems, this is the less interesting one. It states that if no context in $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$ can distinguish between $\lceil\lceil\mathbf{e}_1\rceil\rceil$ and $\lceil\lceil\mathbf{e}_2\rceil\rceil$, no context in $\lambda^{\vdash}$ can either. This is true, as every context in $\lambda^{\vdash}$ can be seen as a context in $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$ with the same semantics (by simply embedding it).

Theorem 3.3 (Preservation of Contextual Equivalence). *Given any two equivalent expressions, $\Gamma \vdash \mathbf{e}_1 \approx_{ctx} \mathbf{e}_2 : \tau$, then their embedding into $\boldsymbol{\lambda}^{\vdash}_{\mathrm{ST}}$ are still contextually equivalent; more specifically, we have that $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil\mathbf{e}_1\rceil\rceil \approx_{ctx} \mathbf{e}_2 : \lceil\lceil\tau\rceil\rceil$.*

[3] We do not add any new typing rules.

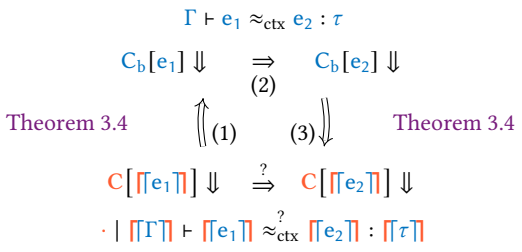

Fig. 3. Theorem 3.4 implies Theorem 3.3.

As discussed in the introduction, this is the most interesting theorem, and we will now outline its proof.

Outline Proof of Preservation of Equivalence. To prove Theorem 3.3, we show that contexts in $\lambda^{\vdash}_{ST}$ can be correctly emulated by contexts in $\lambda^{\vdash}$. This is formalized as follows:[4]

Theorem 3.4 (Correctly Emulating Stateful Contexts by Syntactically Typed Contexts). *Given an arbitrary context* C *in* $\lambda^{\vdash}_{ST}$*, such that it is well-typed as follows* $\vdash C : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot ; 1)$*, then there exists a well-typed context in* $\lambda^{\vdash}$*,* $\vdash C^{\vdash}_b : (\Gamma;\tau) \Rightarrow (\cdot ; 1)$ *such that for all* $\Gamma \vdash e : \tau$*, the following holds:*

$$C^{\vdash}_b[e] \Downarrow \textit{ iff } C[\lceil\lceil e \rceil\rceil] \Downarrow$$

To see why this theorem indeed implies preservation of equivalence, the reader can refer to Figure 3. By symmetry, it is enough to prove only the refinement that is depicted there (the implication with the question mark). Implications (1) and (3) follow from Theorem 3.4, while (2) follows from the given equivalence in $\lambda^{\vdash}$.

Our key insight is to not directly define this back-translation. Instead, we split up the problem by proving the following two theorems separately:

Theorem 3.5 (Correctly Emulating Stateful Contexts by Semantically Typed Contexts). *Given an arbitrary context* C *in* $\lambda^{\vdash}_{ST}$*, such that it is well-typed as follows* $\vdash C : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot ; 1)$*, there exists a* semantically *typed context in* $\lambda^{\vDash}$*,* $\vDash_{int.} C^{\vDash}_b : (\Gamma;\tau) \Rightarrow (\cdot ; 1)$ *such that for all* $\Gamma \vdash e : \tau$*, the following holds:*

$$C^{\vDash}_b[e] \Downarrow \textit{ iff } C[\lceil\lceil e \rceil\rceil] \Downarrow$$

Theorem 3.6 (Correctly Emulating Semantically Typed Contexts by Syntactically Typed Contexts). *Given an arbitrary context* C *in* $\lambda^{\vDash}$ *that it is semantically typed as follows* $\vDash_{int.} C : (\Gamma;\tau) \Rightarrow (\cdot ; 1)$*, there exists a* syntactically *typed context in* $\lambda^{\vdash}$*,* $\vdash C^{\vdash}_b : (\Gamma;\tau) \Rightarrow (\cdot ; 1)$ *such that for all* $\Gamma \vdash e : \tau$*, the following holds:*

$$C^{\vdash}_b[e] \Downarrow \textit{ iff } C[e] \Downarrow$$

We report on the proofs for these in section 5 and section 4 respectively.

In addition to the logical relation in subsection 2.2, we will introduce four more logical relations in the following two sections. These relations are used only in the subsections they are defined in, so there is no need to keep them all in mind at the same time.

[4] As we will remark in section 7, this theorem is actually stronger.

4 CORRECTLY BACK-TRANSLATING SEMANTICALLY TYPED CONTEXTS

We split up the challenge of proving Theorem 3.6 into three smaller problems, each to be tackled in the respective subsections of this section. Given an arbitrary context C in $\lambda^{\vDash}$ that it is semantically typed as follows $\vDash_{int.} C : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$, we will

(1) define $C^{\vdash}_{b}$, its syntactically typed emulation,
(2) prove that for all $\Gamma \vdash e : \tau$, if $C^{\vdash}_{b}[e] \Downarrow$ then $C[e] \Downarrow$, and conversely,
(3) if $C[e] \Downarrow$, then $C^{\vdash}_{b}[e] \Downarrow$.

4.1 Defining the Syntactically Typed Back-Translation

It is useful to note that the semantic typing in $\lambda^{\vDash}$ is compositional, but only so in one direction. That is, given for instance the terms e_1 and e_2, semantically typed at $\tau_1 \to \tau_2$ and τ_1 respectively, we can conclude that their application, $e_1\ e_2$, is semantically typed at τ_2. Conversely though, if $e\ e'$ is semantically typed at τ for instance, then in general, neither e nor e' need to be semantically typed.

When defining a back-translation on semantically typed terms therefore, we cannot do any structural induction on the "semantic typedness". Consequently, the back-translation that we will define can be more easily thought of as a back-translation on *untyped* (rather than semantically typed) contexts. Only in proving the correctness of the back-translation, shall we actually use the fact that our contexts are semantically typed.

We will now define our back-translation in two parts. The first of which will convert our context to a syntactically typed context by translating everything into a universal type. In the second phase, we will insert the necessary functions to obtain a context of the right type.

First Stage of Back-Translation. Terms are back-translated to a universal type, defined as follows.

$$\mathcal{U} = \mu X.(1 + B + Z + (X + X) + (X \times X) + (X \to X) + \mu Y.X)$$

The general idea is not new and has been used before [Devriese et al. 2016; Jacobs et al. 2021; New and Ahmed 2018; New et al. 2016; Patrignani et al. 2021].

To do this, we first define an inject and an extract function for each type constructor.

$$\begin{array}{ccc} \mathsf{inject}_A : A \to \mathcal{U} & \mathsf{inject}_\otimes : (\mathcal{U} \otimes \mathcal{U}) \to \mathcal{U} & \mathsf{inject}_\mu : (\mu Y.\mathcal{U}) \to \mathcal{U} \\ \mathsf{extract}_B : B \to 1 & \mathsf{extract}_\otimes : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U}) & \mathsf{extract}_\mu : \mathcal{U} \to (\mu Y.\mathcal{U}) \end{array}$$

Here, A ranges over our base types and $\otimes$ over $+$, $\times$, and $\to$. The injections put different types of values into the universe. They always succeed and consist of the right combination of inj_1's and inj_2's and a fold. The extractions try to extract a value from the universe, presupposing it is of a certain form; they diverge if they are applied to a value of the wrong form. They are a nesting of case's that return a diverging term in all the branches except the one that corresponds to the type constructor. With the aid of these functions, we define the translation by structural induction on terms in Figure 4. Note that the virtual step construct is just stripped away. The translation on contexts follows naturally.

Second Stage. Given our semantically typed context, $\vDash_{int.} C : (x_1 : \tau_1, \ldots, x_n : \tau_n; \tau) \Rightarrow (\cdot\,; 1)$, it follows that $\vdash \langle\!\langle C \rangle\!\rangle : (x_1 : \mathcal{U}, \ldots, x_n : \mathcal{U}; \mathcal{U}) \Rightarrow (\cdot\,; \mathcal{U})$. In the second stage of the back-translation, we will therefore insert the necessary wrappers such that we can finally arrive at a syntactically typed context of type $(x_1 = \tau_1, \ldots, x_n = \tau_n; \tau) \Rightarrow (\cdot\,; 1)$. For each type τ, we start out by defining the functions $\mathsf{embed}_\tau : \tau \to \mathcal{U}$ and $\mathsf{project}_\tau : \mathcal{U} \to \tau$. They are the generalization to the inject and extract-functions we saw earlier. They are defined by mutual induction on types; we first generalize over open types, say $\Xi \vdash \tau$, using a map $\Delta : \Xi \to \mathit{Var}$ (which we omit for closed types).

$$
\begin{aligned}
\langle\!\langle () \rangle\!\rangle &= \mathsf{inject}_1\ () & \langle\!\langle (\mathsf{e}_1,\mathsf{e}_2) \rangle\!\rangle &= \mathsf{inject}_\times\ (\langle\!\langle \mathsf{e}_1 \rangle\!\rangle,\langle\!\langle \mathsf{e}_2 \rangle\!\rangle) \\
\langle\!\langle \mathsf{e}_1; \mathsf{e}_2 \rangle\!\rangle &= \mathsf{extract}_1\ \langle\!\langle \mathsf{e}_1 \rangle\!\rangle; \langle\!\langle \mathsf{e}_2 \rangle\!\rangle & \langle\!\langle \pi_1\ \mathsf{e} \rangle\!\rangle &= \pi_1\ (\mathsf{extract}_\times\ \langle\!\langle \mathsf{e} \rangle\!\rangle) \\
\langle\!\langle \mathsf{x} \rangle\!\rangle &= \mathsf{x} & \langle\!\langle \mathsf{fold}\ \mathsf{e} \rangle\!\rangle &= \mathsf{inject}_\mu\ (\mathsf{fold}\ \langle\!\langle \mathsf{e} \rangle\!\rangle) \\
\langle\!\langle \lambda \mathsf{x}.\mathsf{e} \rangle\!\rangle &= \mathsf{inject}_\to\ (\lambda \mathsf{x}.\langle\!\langle \mathsf{e} \rangle\!\rangle) & \langle\!\langle \mathsf{unfold}\ \mathsf{e} \rangle\!\rangle &= \mathsf{unfold}\ (\mathsf{extract}_\mu\ \langle\!\langle \mathsf{e} \rangle\!\rangle) \\
\langle\!\langle \mathsf{e}_1\ \mathsf{e}_2 \rangle\!\rangle &= \mathsf{extract}_\to\ \langle\!\langle \mathsf{e}_1 \rangle\!\rangle\ \langle\!\langle \mathsf{e}_2 \rangle\!\rangle & \langle\!\langle \mapsto(\mathsf{e}) \rangle\!\rangle &= \langle\!\langle \mathsf{e} \rangle\!\rangle
\end{aligned}
$$

Fig. 4. Back-Translation of untyped terms into the universe (a selection of cases).

$$
\begin{aligned}
\mathsf{embed}_\mathsf{B} &= \mathsf{inject}_\mathsf{B} & \mathsf{project}_\mathsf{B} &= \mathsf{extract}_\mathsf{B} \\
\mathsf{embed}^\Delta_{\tau+\tau'} &= \left|\begin{array}{l}\mathsf{inject}_+ \circ \\ \quad \mathsf{map}_+(\mathsf{embed}^\Delta_\tau, \mathsf{embed}^\Delta_{\tau'})\end{array}\right. & \mathsf{project}^\Delta_{\tau+\tau'} &= \left|\begin{array}{l}\mathsf{map}_+(\mathsf{project}^\Delta_\tau, \mathsf{project}^\Delta_{\tau'}) \circ \\ \quad \mathsf{extract}_+\end{array}\right. \\
\mathsf{embed}^\Delta_{\tau\times\tau'} &= \left|\begin{array}{l}\mathsf{inject}_\times \circ \\ \quad \mathsf{map}_\times(\mathsf{embed}^\Delta_\tau, \mathsf{embed}^\Delta_{\tau'})\end{array}\right. & \mathsf{project}^\Delta_{\tau\times\tau'} &= \left|\begin{array}{l}\mathsf{map}_\times(\mathsf{project}^\Delta_\tau, \mathsf{project}^\Delta_{\tau'}) \circ \\ \quad \mathsf{extract}_\times\end{array}\right. \\
\mathsf{embed}^\Delta_{\tau\to\tau'} &= \left|\begin{array}{l}\lambda \mathsf{f}.\ \mathsf{extract}_\to \\ \quad (\mathsf{embed}^\Delta_{\tau'} \circ \mathsf{f} \circ \mathsf{project}^\Delta_\tau)\end{array}\right. & \mathsf{project}^\Delta_{\tau\to\tau'} &= \left|\begin{array}{l}\lambda \mathsf{f}. \\ \quad (\mathsf{project}^\Delta_{\tau'} \circ \mathsf{extract}_\to\ \mathsf{f} \circ \mathsf{embed}^\Delta_\tau)\end{array}\right. \\
\mathsf{embed}^\Delta_{\mu\mathsf{X}.\tau} &= \lambda \mathsf{x}.\pi_1\ ((\lambda \mathsf{y}.\ \mathsf{fix}\ \mathsf{gen}\ \mathsf{y})\ ())\ \mathsf{x} & \mathsf{project}^\Delta_{\mu\mathsf{X}.\tau} &= \lambda \mathsf{x}.\pi_2\ ((\lambda \mathsf{y}.\ \mathsf{fix}\ \mathsf{gen}\ \mathsf{y})\ ())\ \mathsf{x} \\
\mathsf{embed}^\Delta_\mathsf{X} &= \lambda \mathsf{x}.\pi_1\ (\Delta(\mathsf{X})\ ())\ \mathsf{x} & \mathsf{project}^\Delta_\mathsf{X} &= \lambda \mathsf{x}.\pi_2\ (\Delta(\mathsf{X})\ ())\ \mathsf{x}
\end{aligned}
$$

Here, $\circ$ is a shortcut for function composition. Given $\mathsf{f}_1 : \tau_1 \to \tau_1'$ and $\mathsf{f}_2 : \tau_2 \to \tau_2'$, we have $\mathsf{map}_+(\mathsf{f}_1, \mathsf{f}_2)$ and $\mathsf{map}_\times(\mathsf{f}_1, \mathsf{f}_2)$, the natural functions of type $(\tau_1 + \tau_2) \to (\tau_1' + \tau_2')$ and $(\tau_1 \times \tau_2) \to (\tau_1' \times \tau_2')$ respectively. The generator for recursive types, gen, is defined as follows.

$$
\lambda \mathsf{r} : (1 \to (\mu\mathsf{X}.\tau \to \mathcal{U}) \times (\mathcal{U} \to \mu\mathsf{X}.\tau)).\ \lambda\_.\left(\begin{array}{l}\left(\lambda x.\ \mathsf{inject}_\mu\ (\mathsf{fold}\ (\mathsf{embed}^{\Delta;\mathsf{X}\mapsto\mathsf{r}}_\tau\ (\mathsf{unfold}\ x)))\right), \\ \left(\lambda x.\ \mathsf{fold}\ (\mathsf{project}^{\Delta;\mathsf{X}\mapsto r}_\mu\ (\mathsf{unfold}\ (\mathsf{extract}_\mu\ x)))\right)\end{array}\right)
$$

The embed and $\mathsf{project}$ functions for a recursive type $\mu\mathsf{X}.\tau$ bring into scope the variable r which is used for the recursive occurrences. They share the same generator as a recursive call at X can occur at a contravariant position.

Adding these wrappers around our context, we arrive at the final back-translation on contexts. Suppose C is semantically typed of type $(\Gamma; \tau) \Rightarrow (\cdot; 1)$, then it is defined as follows.

$$
\mathscr{CP}\langle\!\langle \mathsf{C} \rangle\!\rangle = \begin{cases}\mathsf{project}_1\ (\langle\!\langle \mathsf{C} \rangle\!\rangle [\mathsf{embed}_\tau\ \cdot\ ]) & \Gamma = \cdot \\ \mathsf{let}\ \mathsf{f} = (\lambda \mathsf{x}_1 \ldots \mathsf{x}_\mathsf{n}.\ \cdot)\ \ \mathsf{in} \\ \mathsf{project}_1 (\langle\!\langle \mathsf{C} \rangle\!\rangle\ [\mathsf{embed}_\tau\ (\mathsf{f}\ \ (\mathsf{project}_{\tau_1} \mathsf{x}_1)\ \ldots\ (\mathsf{project}_{\tau_\mathsf{n}} \mathsf{x}_\mathsf{n}))]\ ) & \Gamma = \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_\mathsf{n} : \tau_\mathsf{n}\end{cases}
$$

4.2 Proving Correctness of Back-Translation; First Refinement

It is really difficult, if not impossible to *directly* prove the refinement in (2) as laid out in the introduction of this section. What one typically does instead is to define a binary relation, say $\mathcal{R}$, that models refinement. That is, if some e halts to value, and if it is related to some e', i.e. $\mathcal{R}(\mathsf{e}, \mathsf{e}')$, then e' must halt to a value as well.

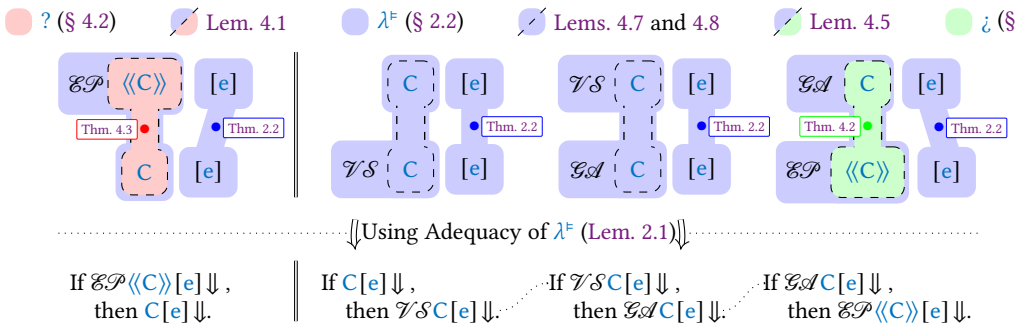

Fig. 5. Proving equitermination of $C[e]$ and $\mathscr{EP}\langle\langle C\rangle\rangle[e] \Downarrow$.

The relation defined in subsection 2.2 for the definition of our intermediate language actually satisfies this description so far by its adequacy result as stated in Lemma 2.1.

To actually be useful however, the relation has to contain the expressions whose refinement we are interested in; in our case these are $C_b^{\vdash}[e]$ and $C[e]$. To prove this, we first define a new binary relation that associates terms with their untyped translation into the universe. Afterwards, we present Lemma 4.1 that bridges the gap between this new relation and the one from subsection 2.2.

A high-level overview of the proof can be seen in the leftmost diagram of Figure 5. We have that $\langle\langle C\rangle\rangle$ is related to C in the new logical relation (to be denoted with subscript ?) by the fundamental theorem of that relation (its extension to contexts), Theorem 4.2. Given Lemma 4.1, we can then derive that adding the wrappers $\mathscr{EP}$ to our context $\langle\langle C\rangle\rangle$ will relate it to C in the relation that defines $\lambda^{\vdash}$. The syntactically typed term e is related to itself by Theorem 2.2. Using Lemma 2.1, we can then derive our refinement.

The new relation relates the syntactically well-typed translations into the universe with their original untyped forms. For each type constructor c, we first define a meta function $\mathsf{injected}_c^{\mathsf{Val}} : \mathsf{Val} \to \mathsf{Val}$, such that for all values v we have that $\mathsf{inject}_c\ v \to^* \mathsf{injected}_c^{\mathsf{Val}}(v)$.

$$
\begin{aligned}
&\mathcal{V}_? : \mathit{Val} \to \mathit{Val} \to \mathit{iProp} \\
&\mathcal{V}_?(v, v') = \left(v = \mathsf{injected}_1^{\mathsf{Val}}(())\ast v' = ()\right) \\
&\qquad \vee \left(\exists b \in \{\mathit{true}, \mathit{false}\}.\ v = \mathsf{injected}_{\mathsf{B}}^{\mathsf{Val}}(b)\ast v' = b\right) \\
&\qquad \vee \left(\exists z \in \mathbb{Z}.\ v = \mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(z)\ast v' = z\right) \\
&\qquad \vee \left(\exists w, w'.\ \bigvee_{i\in\{1,2\}} (v = \mathsf{injected}_{+}^{\mathsf{Val}}(\mathsf{inj}_i\ w)\ast v' = \mathsf{inj}_i\ w' \ast \triangleright \mathcal{V}_?(w, w'))\right) \\
&\qquad \vee \left(\exists v_1, v_1', v_2', v_2'.\ v = \mathsf{injected}_{\times}^{\mathsf{Val}}((v_1', v_2'))\ast v' = (v_1', v_2')\ast \triangleright \mathcal{V}_?(v_1, v_1')\ast \triangleright \mathcal{V}_?(v_2, v_2')\right) \\
&\qquad \vee \left(\exists e.\ v = \mathsf{injected}_{\to}^{\mathsf{Val}}(\lambda x.e)\ast \triangleright \Box(\forall w, w'.\ \mathcal{V}_?(w, w') \mathrel{-\!\!*} \mathsf{lift}\ \mathcal{V}_?\ (e[w/x], v'\ w'))\right) \\
&\qquad \vee \left(\exists w, w'.\ v = \mathsf{injected}_{\mu}^{\mathsf{Val}}(\mathsf{fold}\ w)\ast v' = \mathsf{fold}\ w' \ast \triangleright \mathcal{V}_?(w, w')\right)
\end{aligned}
$$

Note that this relation is not indexed over types, and that each of the recursive calls are guarded under a $\triangleright$. We are thus implicitly using the guarded fixpoint operator.

The relation on expressions follows by lifting the one on values. That is, we have $\mathcal{E}_?(e, e') = \mathsf{lift}\ \mathcal{V}_?\ (e, e')$.

Similar to how we extended our logical relation for $\lambda^{\vDash}$ to open expressions, we relate two expressions, say e and e' with free variables $x_1, \ldots, x_n$ as follows:

$$O_?^{x_1,\ldots,x_n}(e, e') = \forall \vec{v}, \vec{v}', |\vec{v}| = |\vec{v}'| = n. \ \ast_{0 \le i < n} \mathcal{V}_?(\vec{v}.i, \vec{v}'.i) \vdash \mathcal{E}_?(e[\vec{v}/\vec{x}], e'[\vec{v}'/\vec{x}])$$

The connection between, $\mathcal{E}_?$ and $\mathcal{E}_{int.}[\![\tau]\!]$, is provided by appropriately inserting embed_τ and $\mathsf{project}_\tau$.

Lemma 4.1 (Connective Lemma).

$$\frac{\mathcal{E}_?(e, e')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{project}_\tau\ e, e')} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](e, e')}{\mathcal{E}_?(\mathsf{embed}_\tau\ e, e')}$$

Lastly, we have the following fundamental theorem, which states that the logical relation actually satisfies what we intended it to.

Theorem 4.2 (Fundamental Theorem). *Given any untyped expression e with free variables $x_1, \ldots, x_n$, then we have $O_?^{x_1,\ldots,x_n}(\langle\!\langle e \rangle\!\rangle, e)$.*

4.3 Proving Correctness of Back-Translation; Second Refinement

We prove the reverse refinement (as laid out in the introduction of the section) by a chain of three parts as summarized by the rightmost three diagrams in Figure 5, utilizing two new intermediate contexts, $\mathscr{GAC}$ and $\mathscr{VSC}$. We need to do a bit more work compared to what was needed in subsection 4.2 because a completely symmetric approach does not hold up unfortunately. We explain where it falls short and correct for it accordingly, introducing the required machinery as we go along. In the end of this section, we explain how everything fits together to prove our refinement.

First off, we have a new logical relation that we denote with subscript ¿, that relates arbitrary untyped terms on the left-hand side with their syntactically typed translation into the universe. Its definition is symmetric to the one from subsection 4.2; as such, it has the following fundamental theorem.

Theorem 4.3 (Fundamental Theorem). *Given any untyped expression e with free variables $x_1, \ldots, x_n$, then we have $O_¿^{x_1,\ldots,x_n}(e, \langle\!\langle e \rangle\!\rangle)$.*

Note that this relation relates *possibly unsafe* untyped terms (on the left-hand side) to their syntactic counterparts in the universe. As such, our lift-operator uses an unsafe weakest precondition $\mathsf{wp}\ e\ \{\Phi\}$ that models only partial correctness; it says nothing when e evaluates to a stuck term. If we were to use a safe weakest precondition instead, we would not have been able to prove the fundamental theorem. This is because a safe weakest precondition implies safety of the program, i.e., that the program does not get stuck.

Analogous to Lemma 4.1, we might now try to prove the following:

Lemma 4.4 (Connective Lemma (False!)).

$$\frac{\mathcal{E}_¿(e, e')}{\mathcal{E}_{int.}[\![\tau]\!](e, \mathsf{project}_\tau\ e')} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](e, e')}{\mathcal{E}_¿(e, \mathsf{embed}_\tau e')}$$

The lemma does not hold up sadly. The one on the left is clearly false, as we can consider e.g. $\mathcal{E}_¿(6, \mathsf{injected}_Z^{\mathsf{Val}}(6))$, while the following is *not true*: $\mathcal{E}_{int.}[\![\mathsf{B}]\!](6, \mathsf{project}_\mathsf{B}\ (\mathsf{injected}_Z^{\mathsf{Val}}(6)))$, since the left term is a value and the right term will diverge.

We could try to fix this problem by strengthening our assumption, but unfortunately, even the converse lemma on the right fails to hold up. For instance, it is the case that $\mathcal{V}_{int.}[\![\mathsf{B} \to \mathsf{B}]\!](\lambda x.x, f)$

where $\mathsf{f} = \lambda\mathsf{x}.\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}$, as the two functions behave the same when applied to booleans. However, $\mathcal{V}_{\not\iota}(\lambda\mathsf{x}.\mathsf{x}, \mathsf{embed}_{\mathsf{B}\to\mathsf{B}}\ \mathsf{f})$ is not true, because applying these functions to non-boolean values, e.g. $\mathcal{V}_{\not\iota}(6, \mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6))$, yields 6 on the left, but diverges on the right when the function $\mathsf{embed}_{\mathsf{B}\to\mathsf{B}}\ \mathsf{f}$ attempts to extract a boolean from $\mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6)$.

The problem stems from the fact that in the untyped relation, all contracts with respect to types are off. We can use a function, like $\lambda\mathsf{x}.\mathsf{x}$ in our previous example, with any input we can think of, even though we might have thought of our identity function as only applicable to booleans.

To remedy the lemma, we will define two contravariant families of functions indexed by types, $\mathsf{guard}_\tau : \tau \to \tau$ and $\mathsf{assert}_\tau : \tau \to \tau$. If we *trust* an *expression* to be of type τ but if we do *not trust* its *surroundings*, then we can "guard" it by applying guard_τ. If we do *not* trust an *expression* to be of type τ, but we do *trust* its *surroundings*, then we can "assert" it by applying assert_τ to it.

For instance, $\mathsf{assert}_{\mathsf{B}}$ is defined by $\lambda\mathsf{x}.\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ \mathsf{false}$. So applying it to a certain value v say, acts as an identity function if *and only if* $\mathsf{v} : \mathsf{B}$. If we trust a certain function to be of type $\mathsf{B} \to \mathsf{Z}$ but do not trust that its environment respects this type, we can guard it with $\mathsf{guard}_{\mathsf{B}\to\mathsf{Z}}$ which is defined by $\lambda\mathsf{f}.\,\mathsf{guard}_{\mathsf{Z}} \circ \mathsf{f} \circ \mathsf{assert}_{\mathsf{B}}$. If the guarded function is fed a value v by an untrustworthy environment, then v will be asserted to be a boolean.

The guard functions on base types are just identity functions. On Z and 1, the assert's are respectively $\lambda\mathsf{x}.\mathsf{x} + 0$ and $\lambda\mathsf{x}.\mathsf{x};\ ()$. The other cases of guard/assert in the general definition on arbitrary types mirror the ones of embed/project respectively, after stripping away the inject's and extract's.

With the aid of these functions, we can now state a lemma that connects the two relations.

Lemma 4.5 (Connective Lemma).

$$\frac{\mathcal{E}_{\not\iota}(\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{assert}_\tau\ \mathsf{e}, \mathsf{project}_\tau\ \mathsf{e}')} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{\not\iota}(\mathsf{guard}_\tau\ \mathsf{e}, \mathsf{embed}_\tau\ \mathsf{e}')}$$

This allows us to fix the previous two examples. Consider again $\mathcal{E}_{\not\iota}(6, \mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6))$, then it is true that $\mathcal{E}_{int.}[\![\mathsf{B}]\!](\mathsf{assert}_{\mathsf{B}}\ 6, \mathsf{project}_{\mathsf{B}}\ (\mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6)))$ as $\mathsf{assert}_{\mathsf{B}}\ 6$ gets stuck in its evaluation. Similarly, consider $\mathcal{V}_{int.}[\![\mathsf{B} \to \mathsf{B}]\!](\lambda\mathsf{x}.\mathsf{x}, \mathsf{f})$ where f defined as before. Now it is true that $\mathcal{V}_{\not\iota}(\mathsf{guard}_{\mathsf{B}\to\mathsf{B}}\ \lambda\mathsf{x}.\mathsf{x}, \mathsf{embed}_{\mathsf{B}\to\mathsf{B}}\ \mathsf{f})$. The guard around our identity function will ensure that it can only be used with booleans, and applying it to 6 will evaluate to a stuck term.

Needing to add these guards and asserts to connect our two relations is exactly the reason we are proving the refinement from $\mathsf{C}[\mathsf{e}]$ to $\mathscr{PE}\langle\!\langle\mathsf{C}\rangle\!\rangle[\mathsf{e}]$ through an intermediate step $\mathscr{GA}\mathsf{C}[\mathsf{e}]$.

We define $\mathscr{GA}\ \mathsf{C}$, with C our semantically typed context of type $(\Gamma;\tau) \Rightarrow (\cdot;1)$, as follows.

$$\mathscr{GA}\mathsf{C} = \begin{cases} \mathsf{assert}_1\ (\mathsf{C}[\mathsf{guard}_\tau\ \cdot\ ]) & \Gamma = \cdot \\ \mathsf{let}\ \mathsf{f} = (\lambda\mathsf{x}_1 \ldots \mathsf{x}_\mathsf{n}.\,\cdot)\ \mathsf{in} \\ \mathsf{assert}_1(\mathsf{C}\left[\mathsf{guard}_\tau\ (\mathsf{f}\ \ (\mathsf{assert}_{\tau_1}\mathsf{x}_1)\ \ldots\ (\mathsf{assert}_{\tau_\mathsf{n}}\mathsf{x}_\mathsf{n}))\right]) & \Gamma = \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_\mathsf{n} : \tau_\mathsf{n} \end{cases}$$

Note that this is the same stratagem we used in the $\mathscr{EP}$-wrapper.

We still need some way to prove that adding the $\mathscr{GA}$ wrapper to our C does not modify its behavior. To do so, we can try to prove the following lemma.

Lemma 4.6 (Guards and Asserts as Non-operators (False!)).

$$\frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{guard}_\tau\ \mathsf{e}')} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{assert}_\tau\ \mathsf{e}')}$$

Unfortunately the above is not true, for the same reasons $\lambda\mathsf{x}.\mathsf{x}$ does not relate $\mathsf{map}\ (\lambda\mathsf{x}.\mathsf{x} + 0)$ as discussed in subsection 2.2.

To work around this issue, we split up this challenge using our virtual steps, by proving the following two lemmas.

Lemma 4.7 (Adding Virtual Steps & Guards/Asserts).

$$\frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mapsto(\mathsf{e}), \mathsf{guard}_\tau\ \mathsf{e}')} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mapsto(\mathsf{e}), \mathsf{assert}_\tau\ \mathsf{e}')}$$

Lemma 4.8 (Virtual Steps as Non-operators).

$$\frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mapsto(\mathsf{e}'))} \qquad \frac{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mathsf{e}')}{\mathcal{E}_{int.}[\![\tau]\!](\mathsf{e}, \mapsto(\mathsf{e}'))}$$

Analogously to the $\mathscr{GA}$-wrapper, we define $\mathscr{VS}$ by adding virtual steps around the edges.

Finally, we have all the ingredients to prove the desired refinement now. As mentioned earlier and as depicted in the three rightmost diagrams in Figure 5, we prove the refinement by composing three other refinements. All three are proven by logical relatedness in $\lambda^{\vDash}$ together and its adequacy result, Lemma 2.1.

For the first refinement, we know that C, being semantically well typed, is related to itself by definition. Moreover, we can prove, using Lemma 4.8, that adding the $\mathscr{VS}$ wrapper to the left-hand side does not change this fact. Lastly, e is related to itself by Theorem 2.2.

Similarly, we know that for the second refinement, C is related to itself. Likewise, we can prove, using Lemma 4.7, that adding the $\mathscr{VS}$ and $\mathscr{GA}$ wrappers, to the left and right-hand side respectively, leaves this unchanged.

Lastly, we know that C is related to $\langle\!\langle \mathsf{C} \rangle\!\rangle$ in the ¿ relation by Theorem 4.3. Lemma 4.5 now constitutes the core of the proof that adding the $\mathscr{GA}$ and $\mathscr{EP}$ wrappers will transfer this relatedness to relatedness in $\lambda^{\vDash}$.

5 CORRECTLY BACK-TRANSLATING STATEFUL CONTEXTS

Now that we are able to back-translate semantically typed contexts from $\lambda^{\vDash}$ into syntactically typed ones in $\lambda^{\vdash}$, let us see if we can back-translate $\lambda^{\vdash}_{\mathrm{ST}}$ contexts to $\lambda^{\vDash}$. We split up Theorem 3.5 into the following four smaller objectives. Consider an arbitrary $\vdash \mathsf{C} : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot; 1)$, we will

(1) define $\mathsf{C}^{\vDash}_{\mathsf{b}}$, its *semantically* typed emulation in $\lambda^{\vDash}$,
(2) prove that $\mathsf{C}^{\vDash}_{\mathsf{b}}$ is actually semantically typed; we have $\vDash_{int.} \mathsf{C}^{\vDash}_{\mathsf{b}} : (\Gamma; \tau) \Rightarrow (\cdot; 1)$,
(3) prove that for all $\Gamma \vdash \mathsf{e} : \tau$, if $\mathsf{C}[\lceil\lceil\mathsf{e}\rceil\rceil] \Downarrow$, then $\mathsf{C}^{\vDash}_{\mathsf{b}}[\mathsf{e}] \Downarrow$, and conversely
(4) prove that if $\mathsf{C}^{\vDash}[\mathsf{e}] \Downarrow$, then $\mathsf{C}[\lceil\lceil\mathsf{e}\rceil\rceil] \Downarrow$.

In subsection 5.1, subsection 5.2, and subsection 5.3, we will respectively go over the objectives (1), (2), and (3). Since the ideas for (4) are the same as for (3), we do not go over them here.

5.1 Definition of Back-Translation

As mentioned in the introduction, we will back-translate ST-computations to pure functions in $\lambda^{\vDash}$ using an explicit state-passing-style functions à la State monad. The local heap that will be passed around will not be syntactically typed though. Rather, we will encode a heap by a right-nested cartesian product with all the allocated values so far, together with a natural number that represents its size. Because these heaps grow over time, they cannot have a fixed syntactic type.

This representation is encoded by a meta-level function $\mathcal{E} : \mathit{List}\ \mathsf{Val} \to \mathsf{Val}$, taking a list of values to return a single value. For instance, suppose a (back-translated) state thread in which the values v_1, v_2 and v_3 have been allocated, then the heap of that thread will be represented by the following value:

$$\mathcal{E}([\mathsf{v}_1;\ \mathsf{v}_2;\ \mathsf{v}_3]) = ((\mathsf{v}_3, (\mathsf{v}_2, (\mathsf{v}_1, ()))), 3)$$

The zth location allocated in a state thread can be identified by the number z. As such, the heap of a back-translated state thread can be thought of as a list of values that grows to the right, and a location in a back-translated state thread will get mapped to the correct index in this list.

We now introduce the functions read, ref, and write that we will use to back-translate the respectively-named primitives in $\lambda^{\vdash}_{ST}$ by giving their specification in terms of $\mathcal{E}$.

$$\begin{aligned} \mathsf{read}\ \mathsf{z}\ \mathcal{E}(\vec{\mathsf{v}}) &\to^* (\mathcal{E}(\vec{\mathsf{v}}) \quad ,\vec{\mathsf{v}}.\mathsf{z}) && \text{if } 0 \le \mathsf{z} < |\vec{\mathsf{v}}| \\ \mathsf{ref}\ \mathsf{v}\ \mathcal{E}(\vec{\mathsf{v}}) &\to^* (\mathcal{E}(\vec{\mathsf{v}} +\!\!+ [\mathsf{v}]) \ ,|\vec{\mathsf{v}}|) \\ \mathsf{write}\ \mathsf{z}\ \mathsf{v}\ \mathcal{E}(\vec{\mathsf{v}}) &\to^* (\mathcal{E}(\vec{\mathsf{v}}[\mathsf{z} \mapsto \mathsf{v}]), ()) && \text{if } 0 \le \mathsf{z} < |\vec{\mathsf{v}}| \end{aligned}$$

The read takes a well-scoped index and an encoded heap and returns the same heap together with the value at that index. The function ref takes a value and an encoded list, returning the encoded heap of the old list with the new value appended to the right, together with its index. Note that indices permanently identify the correct location in the list, as the list can only grow rightward. Lastly, write takes a well-scoped index, a new value, and an encoded heap, returning the encoded heap updated with the new value at that index together with a unit value. The implementation for ref is quite easy, and the implementations for read and write use a fixpoint operator.

We further define the functions bind and return for the back-translation of their namesakes in $\lambda^{\vdash}_{ST}$, implemented as one would expect for a state monad. The compare-operator is back-translated to a simple function, $\lambda \mathsf{x}_1.\lambda \mathsf{x}_2.\mathsf{x}_1 = \mathsf{x}_2$ which compares the indices as integers.

The back-translation of $\lambda^{\vdash}_{ST}$ into $\lambda^{\vdash}$ is defined by structural induction on the terms, and we denote it by $\langle\!\langle \_ \rangle\!\rangle$. All other constructs in $\lambda^{\vdash}_{ST}$ are trivially back-translated. The back-translation on contexts follows naturally from the one on terms.

5.2 Semantic Well-Typedness

The need to Extend the Relation for $\lambda^{\vdash}$. Although our to be back-translated stateful context C does not have any ST types in its *boundary* (as it has type $(\cdot \mid \lceil\lceil \Gamma \rceil\rceil; \lceil\lceil \tau \rceil\rceil) \Rightarrow (\cdot \mid \cdot ; \mathbf{1})$), its typing derivation can contain them *internally*. To prove that the $\langle\!\langle \mathsf{C} \rangle\!\rangle$ is indeed related to itself in the logical relation for $\lambda^{\vdash}$, we will first refine that logical relation by indexing it over the richer types in $\lambda^{\vdash}_{ST}$.

More concretely, this refinement is represented by a new relation (to be denoted with subscript $\mathcal{X}$), that is closely linked to the old one. For the value relations for instance, we will have the following connection (grayed-out symbols can be ignored for now):

$$\forall \tau, \mathsf{v}, \mathsf{v}'. \mathcal{V}_{int.}[\![\tau]\!](\mathsf{v}, \mathsf{v}') \dashv\vdash \mathcal{V}_{\mathcal{X}}[\![\cdot \vdash \lceil\lceil \tau \rceil\rceil]\!]_{\triangle}(\mathsf{v}, \mathsf{v}')$$

In other words, the new relation coincides with the old one w.r.t. pure boundary types. However, the new relation *additionally* allows us to express relatedness of state computations and locations. A similar result holds for contexts. Given arbitrary $\Gamma, \tau, \Gamma', \tau', \mathsf{C}, \mathsf{C}'$, we have:

$$\vDash_{int.} \mathsf{C} \le \mathsf{C}' : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau') \ \textit{ iff } \ \vDash_{\mathcal{X}} \mathsf{C} \le \mathsf{C}' : (\lceil\lceil \Gamma \rceil\rceil; \lceil\lceil \tau \rceil\rceil) \Rightarrow (\lceil\lceil \Gamma' \rceil\rceil; \lceil\lceil \tau' \rceil\rceil)$$

In the next paragraph, we will construct this extended relation such that for any context in $\lambda^{\vdash}_{ST}$, say $\vdash \mathsf{C} : (\cdot \mid \lceil\lceil \Gamma \rceil\rceil; \lceil\lceil \tau \rceil\rceil) \Rightarrow (\cdot \mid \cdot ; \mathbf{1})$, we will be able to prove that $\vDash_{int.} \langle\!\langle \mathsf{C} \rangle\!\rangle \le \langle\!\langle \mathsf{C} \rangle\!\rangle : (\Gamma; \tau) \Rightarrow (\cdot ; \mathbf{1})$.

Defining the Extended Relation. As this new relation extends the one for $\lambda^{\vdash}$, the only thing of interest here is how we define it for ST and STRef types. Relatedness of two values at the ST-type, should capture the relation between back-translated state-computations. As such, state-passing functions should be related at ST if they emulate related state computations in $\lambda^{\vdash}_{ST}$. But what does it mean for state computations in $\lambda^{\vdash}_{ST}$ to be related?

One might answer that upon execution (in a blank slate), the two computations need to return related values. However, this is not enough: it should also be possible to execute the two computations in already existing heaps, at least on the condition that they are appropriately related. Additionally, ST computations cannot be expected to run correctly in the presence of ill-typed ST references, so we will additionally need to express that any STRef's for the same heap are appropriately typed with respect to the heap in which the computations are executed.

Given this intuition for ST computations, now let us think about what this means for back-translated state computations. Instead of an implicit heap that grows monotonically over time, we are now passing along explicit states (represented by lists of values and encoded by $\mathcal{E}$) and the *locations* in our state thread are now just *indices* into these explicit states. Thus, we will relate two state-passing functions at ST X τ, if when initialized with any two related states and the first one terminates, then (1) the second one terminates as well, (2) the two leave behind states that are still related, and (3) the two "return-values" (the values in the second component) are related at τ.

To formalize this intuition in our logical relations, we introduce some ghost state to track the contents of the states that are passed around. We define an Iris predicate $OwnStates_\gamma$, taking two *equal-sized* list of values, and a points-to predicate $\_ \mapsto_\gamma (\_, \_)$ that takes an index and two values (we're intentionally ignoring the subscript here for now). The two are intimately connected by a series of properties, for which we list a few here now (as before, readers unfamiliar with Iris may ignore parts in gray).

$$OwnStates_\gamma(\vec{v}, \vec{v}') \vdash \Rrightarrow OwnStates_\gamma(\vec{v} +\!\!+ [v], \vec{v}' +\!\!+ [v']) * |\vec{v}| \mapsto_\gamma (v, v') \tag{1}$$

$$OwnStates_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \vec{v}.z = v * \vec{v}'.z = v' \tag{2}$$

$$OwnStates_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \Rrightarrow OwnStates_\gamma(\vec{v}[z \mapsto w], \vec{v}'[z \mapsto w']) * z \mapsto_\gamma (w, w') \tag{3}$$

Equation 1 makes it clear that we can always expand our lists by appending a new pair of values to the right side, and in the process obtain a points-to at the correct index (the length of our previous lists) with this pair. Equation 2 states that a points-to must always agree with the lists in $OwnStates_\gamma$. Finally, we can write a new pair of values to a certain index of the list if we give up its points-to for a new points-to to the new values (Equation 3).

Let's now go over (a first iteration of) the value relation (still a bit naive) for STRef X τ.

$$\mathcal{V}_X[\![\text{STRef X } \tau]\!](v, v') \triangleq \exists z.\, v = z * v' = z \; * \boxed{\exists w, w'.\, z \mapsto_\gamma (w, w') * \mathcal{V}_X[\![\tau]\!](w, w')}$$

Two values are related at STRef X τ, if they are the same integer, say z, such that *invariably*, we have a points-to at that index for two values related at τ.

The rightmost factor in the separating conjunction here (the boxed formula) is called an *invariant*. Given any Iris proposition say P, the proposition $\boxed{P}$ is an *invariant* stating P holds at all times. In other words, at any point during the execution of the program the body of the invariant needs to be satisfied. Intuitively, the body should hold before and after any program step, once it is established. Hence, it can be accessed throughout the proof; we speak of "opening" and "closing" invariants. These refer, respectively, to the steps in proofs where we rely on the invariant holding and where we reestablish it. As per our intuitive explanation above, invariants can only be kept open during a single step of computation (according to operational semantics). [5] With these intuitions in mind, the invariant above captures that the exact values of w and w′ are subject to change: we may open the invariant, update the values using Equation 1 or Equation 3 for instance, and proceed to close it by providing different witnesses for the existential quantifier in the invariant.

[5] These intuitions are formally captured in the definition of the notion of weakest precondition (subsubsection 2.2.1) that uses the fancy update modality ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}'}$. We refer the interested reader to [Jung et al. 2018].

The (first iteration of) the value relation at $\mathsf{ST}\ \mathsf{X}\ \tau$ (still a bit naive) is as follows:

$$\mathcal{V}_{\mathcal{X}}[\![\mathsf{ST}\ \mathsf{X}\ \tau]\!](v, v') \triangleq \forall \vec{v}_i, \vec{v}'_i.\ \square\left(\mathit{OwnStates}_\gamma(\vec{v}_i, \vec{v}'_i) \mathrel{-\!\!*} \mathsf{wp}\ v\ \mathcal{E}(\vec{v}_i)\left\{(\mathcal{E}(\vec{v}_f), w).\ \exists \vec{v}'_f, w'.\right.\right.$$
$$\left.\left.(v'\ \mathcal{E}(\vec{v}'_i) \to^* (\mathcal{E}(\vec{v}'_f), w'))\ *\ \mathit{OwnStates}_\gamma(\vec{v}_f, \vec{v}'_f) * \mathcal{V}_{\mathcal{X}}[\![\tau]\!](w, w')\right\}\right)$$

Here, v and v' are related when for every two appropriate initial states, that is two states for which we have our ghost state $\mathit{OwnStates}_\gamma(\vec{v}_i, \vec{v}'_i)$, if we invoke v and v' with $\mathcal{E}(\vec{v}_i)$ and $\mathcal{E}(\vec{v}'_i)$ respectively, and if the first terminates to a value, then both of them do, and they return related states (for which we have $\mathit{OwnStates}_\gamma(\vec{v}_f, \vec{v}'_f)$) and return values related at τ.

There is one important thing we have not as yet mentioned. In general, we may have lots of different state threads, and the indices and values in the associated heaps should be tracked independently. Therefore, we use different instances of the ghost state for every thread. The faded out subscript γ that we saw in $\mathit{OwnStates}_\gamma$ and $\_ \mapsto_\gamma (\_, \_)$ was actually a name for a particular such instance.

Formally, our value relation is indexed by a type τ with free variables Ξ and a map $\Delta : \Xi \to \mathit{Names}$, that assigns names to our free variables. Given Δ, we use the name $\Delta(\mathsf{X})$ for our ghost state when we relate values at ST and STRef. The final definitions are as follows.[6]

$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathsf{STRef}\ \mathsf{X}\ \tau]\!]_\Delta(v, v') \triangleq \exists z.\ v = z * v' = z *$$
$$\boxed{\exists w, w'.\ z \mapsto_{\Delta(\mathsf{X})} (w, w') * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta(w, w')}^{\Delta(\mathsf{X}).z}$$
$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathsf{ST}\ \mathsf{X}\ \tau]\!]_\Delta(v, v') \triangleq \forall \vec{v}_i, \vec{v}'_i.\ \square\left(\mathit{OwnStates}_{\Delta(\mathsf{X})}(\vec{v}_i, \vec{v}'_i) \mathrel{-\!\!*} \mathsf{wp}\ v\ \mathcal{E}(\vec{v}_i)\left\{(\mathcal{E}(\vec{v}_f), w).\ \exists \vec{v}'_f, w'.\right.\right.$$
$$\left.\left.(v'\ \mathcal{E}(\vec{v}'_i) \to^* (\mathcal{E}(\vec{v}'_f), w'))\ *\ \mathit{OwnStates}_{\Delta(\mathsf{X})}(\vec{v}_f, \vec{v}'_f) * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta(w, w')\right\}\right)$$

The value relations on all the other types are defined as expected; the definition passes through Δ in recursive cases.

Given a naming scheme Δ, the relation on closed expressions is again defined using our lift-operator (subsection 2.2).

$$\mathcal{E}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta = \mathsf{lift}\ \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta$$

We're finally ready to define the relation on open expressions. When defining the relation at type τ with free variables Ξ, we now quantify over all naming schemes $\Delta : \Xi \to \mathit{Names}$. We have the following:

$$\Xi \mid \Gamma \vDash_{\mathcal{X}} e \le e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v}'.\ |\vec{v}| = |\vec{v}'| = |\Gamma|\ *\ \mathop{\Large *}_{0 \le i < |\Gamma|} \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \Gamma.i]\!]_\Delta(\vec{v}.i, \vec{v}'.i) \vdash$$
$$\mathcal{E}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta(e[\vec{x}/\vec{v}], e'[\vec{x}/\vec{v}'])$$

All possible naming schemes include of course the naming schemes that actually give us a different name for each type variable. On a more technical note, we do need to really quantify over *all* possible naming conventions, as it guarantees that we can get a fresh instance for the execution of each state thread.

This new relation does indeed refine the one from $\lambda^{\vDash}$. Stated generally, we have the following:

$$\forall \Gamma, \tau, e, e'.\ \Gamma \vDash_{\mathit{int}.} e \le e' : \tau \dashv\vdash \cdot \mid \lceil\lceil\Gamma\rceil\rceil \vDash_{\mathcal{X}} e \le e' : \lceil\lceil\tau\rceil\rceil$$

And lastly, we have the fundamental theorem, stating that the relation does indeed satisfy the purpose with which we defined it.

[6] You may have noticed also that our invariant has gained a superscript. This is just a name. Iris invariants are given names to preserve soundness; we refer the interested reader to Jung et al. [2018].

Theorem 5.1 (Fundamental Theorem). *Given a typed expression in $\lambda^{\vdash}_{\mathrm{ST}}$, say $\Xi \mid \Gamma \vdash \mathbf{e} : \tau$, we have the following:*

$$\Xi \mid \Gamma \vDash_{\mathcal{X}} \langle\!\langle \mathbf{e} \rangle\!\rangle \leq \langle\!\langle \mathbf{e} \rangle\!\rangle : \tau$$

Readers familiar with separation logic may notice that our usage of ghost state here is reminiscent to how one often uses it to reason about a physical heap in a stateful language. Indeed, we will not provide full details but the above definition of the value relation for ST computations is closely related to the definition of weakest preconditions for expressions in Iris logical relations for languages with primitive mutable state. We refer the interested reader to Jung et al. [2018].

5.3 Proving Correctness of Back-Translation; First Refinement

To prove the refinement from (3) as laid out in the introduction of this section, we will define another logical relation. Its definition shares the intuition with the relation from the previous section. As such, we will focus on the technicalities that are different.

On the left-hand side, we are now working in $\lambda^{\vdash}_{\mathrm{ST}}$, with actual *stateful* computations that we need to relate to their emulations as pure functions to the right-hand side. As such, our operator to lift our value relation to an expression relation will be different.

$$\begin{aligned} \mathsf{lift}_{\mathcal{R}} &: (\mathbf{Val} \to \mathsf{Val} \to \mathit{iProp}) \to (\mathbf{Expr} \to \mathsf{Expr} \to \mathit{iProp}) \\ \mathsf{lift}_{\mathcal{R}}\ \Phi\ (\mathbf{e}, \mathbf{e}') &= \mathsf{wp}\ \mathbf{e}\ \{\mathbf{v}.\ \exists \mathbf{v}'.\ \mathbf{e}' \to^* \mathbf{v}' * \Phi(\mathbf{v}, \mathbf{v}')\} \end{aligned}$$

Although defined completely analogously, its definition now uses the weakest precondition over the stateful language. As is standard, our weakest precondition will use a predicate, say $\mathit{OwnHeap}(\_)$, (taking as argument a heap, i.e. some $h : \mathbf{Loc} \rightharpoonup \mathbf{Val}$) to keep track of the physical heap. And connected with this predicate, we have a points-to predicate $\_ \mapsto \_$ taking a location and a value. The relation between $\mathit{OwnHeap}(\_)$ and $\_ \mapsto \_$ is described similarly to the ghost resources used in the previous section. Note that here, we do not index over a name as we're only using the one instance. As such we will refer to this points-to predicate as the global one.

Apart from the ghost resource used in our definition for the weakest precondition, we use the following predicates (all indexed over a name γ): The predicate $\mathit{OwnState}_\gamma : \mathit{List}\ \mathsf{Val} \to \mathit{iProp}$ taking a list of pure values, the predicate $\_ \mapsto_\gamma (\_) : \mathbb{N} \to \mathsf{Val} \to \mathit{iProp}$ taking an index and a pure value, $\mathit{OwnLocs}_\gamma : \mathit{List}\ \mathbf{Loc} \to \mathit{iProp}$ taking a list of locations, and finally $\_ \mapsto^{\square}_\gamma \_ : \mathbb{N} \to \mathbf{Loc} \to \mathit{iProp}$ taking an index and a location.

The predicates $\mathit{OwnState}_\gamma$ and $\_ \mapsto_\gamma \_$ behave in exactly the same way as the predicates used in the previous subsection (and our global predicates $\mathit{OwnHeap}(\_)$ and $\_ \mapsto \_$ for that matter). The latter two predicates, $\mathit{OwnLocs}_\gamma$ and $\_ \mapsto^{\square}_\gamma \_$, behave slightly differently. Below we list some of their properties.

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) \vdash \Rrightarrow \mathit{OwnLocs}_\gamma(\vec{\ell} +\!\!+ [\ell]) * |\vec{\ell}| \mapsto^{\square}_\gamma \ell \tag{4}$$

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) * z \mapsto^{\square}_\gamma \ell \vdash \vec{\ell}.z = \ell \tag{5}$$

$$\gamma \mapsto^{\square}_z \ell \vdash \gamma \mapsto^{\square}_z \ell * \gamma \mapsto^{\square}_z \ell \tag{6}$$

$$\gamma \mapsto^{\square}_z \ell * \gamma \mapsto^{\square}_z \ell' \vdash \ell = \ell' \tag{7}$$

Equation 4 and Equation 5 are similar. We have no lemma to rewrite a new location to a particular index though; instead the points-to predicate is freely duplicable (Equation 6), and as such its value can never change (Equation 7). Once we have allocated a new location in our list with Equation 4, the location at that index is immutable.

With these ghost states, we will now go over the value relation on references.

$$\mathcal{V}_{\mathcal{R}}[[\Xi \vdash \mathrm{STRef\ tX}\ \tau]]_\Delta(\mathrm{v},\mathrm{v}') \triangleq \exists \ell, z.\ \mathrm{v} = \ell * \mathrm{v}' = z * z \mapsto^{\square}_{\Delta(\mathrm{X}).1} \ell * \boxed{\exists \mathrm{w},\mathrm{w}'.\ \ell \mapsto \mathrm{w} * z \mapsto_{\Delta(\mathrm{X}).2} (\mathrm{w}') * \mathcal{V}_{\mathcal{R}}[[\Xi \vdash \tau]]_\Delta(\mathrm{w},\mathrm{w}')}^{\Delta(\mathrm{X}).z}$$

Note first that the relation is again indexed over by a naming scheme Δ. This Δ is now of type $\Xi \to (\mathit{Names} \times \mathit{Names})$, and hence it associates two names to each free type variable X, $\Delta(\mathrm{X}).1$ and $\Delta(\mathrm{X}).2$. Two values are located at a reference type, if the pure one is an integer, say z, and the stateful one a location, say ℓ for which we have a points-to to that index (in the instance named $\Delta(\mathrm{X}).1$). From the moment we allocate a new value in a state thread, bringing into scope a new location (on the left hand-side), we append a new value to the explicit list in its emulation. This way, the location should forever be related to the particular index of the appended value, which will be enforced by the points-to at $\Delta(\mathrm{X}).1$.

This is not all there is to it however. At any point, the value stored at the location and the particular value in the list that's being passed around in the emulation should be related at the type of the reference. This is encoded by the invariant. At any point, we have to have some appropriately related w and w' for which the location stores w (the global points-to $\ell \mapsto \mathrm{w}$ takes care of this), and the value at index z of the list that's being passed around is w', which will be enforced by the points-to at $\Delta(\mathrm{X}).2$.

To actually uphold these enforcements, we couple our ghost resources to each other and the execution of the thread and its emulation in the value relation of state computations.

$$\mathcal{V}_{\mathcal{R}}[[\Xi \vdash \mathrm{ST\ X}\ \tau]]_\Delta(\mathrm{v},\mathrm{v}') \triangleq \forall \vec{\ell_i}, \vec{\mathrm{v}}_i.\ |\vec{\ell_i}| = |\vec{\mathrm{v}}_i|.\ \square \Big(\mathit{OwnLocs}_{\Delta(\mathrm{X}).1}(\vec{\ell_i}) * \mathit{OwnState}_{\Delta(\mathrm{X}).2}(\vec{\mathrm{v}}_i) \mathrel{-\!\!*} \mathrm{wp}\ \mathrm{runST}\ \{\mathrm{v}\} \Big\{ \mathrm{w}.\ \exists \mathrm{w}', \vec{\ell_f}, \vec{\mathrm{v}}_f.\ |\vec{\ell_f}| = |\vec{\mathrm{v}}_f|.(\mathrm{v}'\ \mathcal{E}(\vec{\mathrm{v}}_i) \to^* (\mathcal{E}(\vec{\mathrm{v}}_f), \mathrm{w}')) * \mathit{OwnLocs}_{\Delta(\mathrm{X}).1}(\vec{\ell_f}) * \mathit{OwnState}_{\Delta(\mathrm{X}).2}(\vec{\mathrm{v}}_f) * \mathcal{V}_{\mathcal{R}}[[\Xi \vdash \tau]]_\Delta(\mathrm{w},\mathrm{w}') \Big\} \Big)$$

Similar to subsection 5.2, the relation is extended to open terms. We write $\Xi \mid \Gamma \vDash_{\mathcal{R}} \mathrm{e} \le \mathrm{e}' : \tau$ for e and e' related under Ξ, Γ, and τ. Likewise, the relation satisfies a fundamental theorem.

Theorem 5.2 (Fundamental Theorem). *Given a typed expression in $\lambda^{\vdash}_{\mathrm{ST}}$, say $\Xi \mid \Gamma \vdash \mathrm{e} : \tau$, we have the following:*

$$\Xi \mid \Gamma \vDash_{\mathcal{R}} \mathrm{e} \le \langle\!\langle \mathrm{e} \rangle\!\rangle : \tau$$

As importantly, we have the following adequacy lemma.

Lemma 5.3 (Logical Relation Adequacy). *If $\cdot \mid \cdot \vDash_{\mathcal{R}} \mathrm{e} \le \mathrm{e}' : \tau$, then if e halts to a value, so must e'.*

Finally, we have all the machinery at our disposal to prove our refinement. Take arbitrary e and C such that $\Gamma \vdash \mathrm{e} : \tau$ and $\vdash \mathrm{C} : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot ; 1)$. We need to prove that if $\mathrm{C}[\lceil\lceil \mathrm{e} \rceil\rceil]$ halts, then so does $\langle\!\langle \mathrm{C} \rangle\!\rangle[\mathrm{e}]$. Using Lemma 5.3, it suffices to prove that $\mathrm{C}[\lceil\lceil \mathrm{e} \rceil\rceil]$ is related to $\langle\!\langle \mathrm{C} \rangle\!\rangle[\mathrm{e}]$. The context C is related to $\langle\!\langle \mathrm{C} \rangle\!\rangle$ by (the generalization to contexts of) Theorem 5.2, so it remains to be proven that $\lceil\lceil \mathrm{e} \rceil\rceil$ is related to e. This follows by the same theorem however, as we have that $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil \mathrm{e} \rceil\rceil : \lceil\lceil\tau\rceil\rceil$ and $\langle\!\langle \lceil\lceil \mathrm{e} \rceil\rceil \rangle\!\rangle = \mathrm{e}$.

For the reverse refinement as described in the beginning of this section, a new logical relation is defined with which the refinement is proven in a similar way to what was done here.

6 DISCUSSIONS

Indirect Results from Full Abstraction. Even though our full abstraction result only says something *directly* about the terms in the pure language (their equivalence is preserved in the presence of ST),

it does entail non-obvious consequences for $\lambda^{\vdash}_{\mathrm{ST}}$ terms. To demonstrate this, let us consider again the commutativity example from Timany et al. [2017b],

$$\texttt{let } x = e_2 \texttt{ in } (e_1, x) \approx_{\mathrm{ctx}} (e_1, e_2) : \tau_1 \times \tau_2$$

which they prove for arbitrary (appropriately typed) *stateful* expressions e_1 and e_2. We will now argue that a very similar statement can now be made by

(1) assuming (quite uncontroversially we think) commutativity in the *pure* language, and
(2) using our full abstraction result from the *pure* into the stateful language.

To do this, consider the two *pure* programs, each taking two suspended computations as arguments:

$$e_1 \triangleq \lambda x_1 : 1 \to \tau_1.\, \lambda x_2 : 1 \to \tau_2.\, \texttt{let } y = x_2\ () \texttt{ in } (x_1\ (), y)$$
$$e_2 \triangleq \lambda x_1 : 1 \to \tau_1.\, \lambda x_2 : 1 \to \tau_2.\, (x_1\ (), x_2\ ())$$

The first expression e_1 first evaluates the second argument computation, then the first, returning both results as a pair. The second expression does the same but evaluates the two argument computations in order. Considering these programs as part of the *pure* language, their contextual equivalence is virtually synonymous with commutativity. Because of full abstraction, their contextual equivalence continues to hold when embedded into the stateful language. As the embedded terms can now take any suspended computation that can have *stateful* behavior, their equivalence depends on commutativity of *stateful* computations.

We believe that analogous arguments can be made for the other equivalences that are stated by Timany et al. [2017b]. Going through our full abstraction proof, we remark that we are also proving the preservation of contextual *refinement*, and as such we believe similar arguments are applicable for many of the refinements stated by Timany et al. [2017b].

We should add still that Timany et al. [2017b] proves commutativity for all terms, *even those that are typed at* ST*-types*. With our full abstraction result, the above argument only works for simple types unfortunately. If however, we had a full abstraction result from the recursive System F to its stateful counterpart with ST, we could recover even this by considering the equivalence between the following two terms.

$$e_1 \triangleq \Lambda X_1.\, \Lambda X_2.\, \lambda x_1 : 1 \to X_1.\, \lambda x_2 : 1 \to X_2.\, .\texttt{let } y = x_2\ () \texttt{ in } (x_1\ (), y)$$
$$e_2 \triangleq \Lambda X_1.\, \Lambda X_2.\, \lambda x_1 : 1 \to X_1.\, \lambda x_2 : 1 \to X_2.\, .(x_1\ (), x_2\ ())$$

7 RELATED WORK

Related to Stateful Encapsulation. Launchbury and Peyton Jones [1994] introduced an ST monad for the encapsulation of state in an otherwise pure programming language. In section 1, we discussed how our work is related to and inspired by this work.

Moggi and Sabry [2001] study the correctness of an ST monad by considering a range of programming languages, including a higher-order lambda calculus with higher kinds and a calculus with lazy evaluation. Compared to Moggi and Sabry [2001], the programming language we consider is simpler. However, the results that we establish are much stronger. In particular, Moggi and Sabry [2001] only prove type safety for the calculi that they consider, albeit that their notion of type safety does properly capture proper state encapsulation; programs crash if they access state threads that they should not.

Timany et al. [2017b] quote Moggi and Sabry [2001] saying "Indeed substantially more work is needed to establish soundness of equational reasoning with respect to our dynamic semantics (even for something as unsurprising as β-equivalence)", and improve on their work by stating and proving a set of equations (including β-equivalence) expected to hold in a pure cbv language.

Our results differ from Timany et al.'s in two ways. As explained in the introduction already, Timany et al. state and prove their equivalences w.r.t. a recursive cbv System F with an ST monad. As mentioned, our language, $\lambda^{\vdash}_{\mathsf{ST}}$ is based on theirs, but we omit System F's type abstraction and application, as we believe they are orthogonal.

Secondly, the notion of purity by Timany et al. is different from the one presented here. Timany et al. define a particular set of equivalences and explicitly prove these. Our notion of purity is more relational, i.e., we are saying intuitively that $\lambda^{\vdash}_{\mathsf{ST}}$ is *as pure as* $\lambda^{\vdash}$. Given *any two equivalent terms in* $\lambda^{\vdash}$, we simply get *for free* that they are still so in the presence of ST.

Note that our relational notion of purity is *not* implied by the results of Timany et al.. While they define a useful logical relation that models contextual equivalences in the presence of ST, there is no guarantee whatsoever that arbitrary pure equivalences can actually be proven with it. What's more, these proofs can be very hard (if they are possible at all). In fact, some of the equivalences presented by Timany et al. are proven only indirectly using a clever chain of contextual equivalences which are in turn proven in their logical relations — note that Timany et al. use two logical relations models for proving those equivalences where one of these logical relations models is only defined for the equivalence corresponding to the rec-hoisting equation. Moreover, even if we had a *complete* logical relation (which the one from Timany et al. most probably is not), this still would not guarantee that it contains any two pure terms equivalent in the absence of ST!

On the one hand, this means that our results are more general and express that ST does not increase expressiveness of the language. At the same time though, our result does not directly imply the equivalences proven by Timany et al.. Instead of proving for instance commutativity in the presence of ST, we only prove that *if it holds in the pure language*[7], then it is *preserved* in the presence of ST.

Related to Full Abstraction. The idea of back-translating into a universal type was first introduced by New et al. [2016] and Devriese et al. [2016] and later used again by Jacobs et al. [2021]; Patrignani et al. [2021]. Using logical relations to prove correctness of the back-translation is very common [Patrignani et al. 2019].

As mentioned in the introduction, using full abstraction to compare the expressive power over different languages has been done before [e.g., Parrow 2008; Patrignani et al. 2021]. Our use of a fully abstract embedding of a pure language to characterize purity is similar to how full abstraction has been used to characterize well-bracketed control flow by Skorstengaard et al. [2019].

Other criteria from the secure compilation community might also be suitable to model the preservation of purity. As mentioned before, Theorem 3.4 is actually stronger than Theorem 3.3, and corresponds to the strongest criteria in [Abate et al. 2019].

8 FUTURE WORK

Adding Polymorphism. We want to add *polymorphism*, proving full abstraction for the embedding of a recursive System F into its extension with the ST-monad. We think the strategy presented here is promising: We would first back-translate contexts in F_{ST} to pure semantically typed ones in F (something we think is a pretty forward extension to what we have done here already). Afterwards, we will back-translate arbitrary semantically typed contexts in F to syntactically typed ones.

The latter is more challenging. To do this, we need to define a universal type into which we can embed untyped code in System F. It is important though that the universal type we consider allows us to define the project/embed functions for any *polymorphic* type. As such, we might want

[7]Such an assumption is quite uncontroversial; the very reason for proving it in the presence of ST is that it holds in its absence! Proving it might still be hard though as it likely requires similar techniques as used by Timany et al..

to work with a family of universal types indexed by a finite set of free type variables, which may require higher-kinded recursive types.

Interestingly, if we only add polymorphism to the target language, by considering e.g. $\lambda^{\vdash}$ into $F_{\mu,\mathsf{ST}}$, we think we could simply back-translate polymorphic functions to thunked functions ($\langle\!\langle \Lambda.\mathbf{e} \rangle\!\rangle = \lambda\_.\langle\!\langle \mathbf{e} \rangle\!\rangle$) and type instantiation with application to unit ($\langle\!\langle \mathbf{e}[\tau] \rangle\!\rangle = \langle\!\langle \mathbf{e} \rangle\!\rangle\ ()$) as by Theorem 3.6 (contexts in $\lambda^{\vDash}$ can be back-translated to contexts in $\lambda^{\vdash}$), they need only be semantically typed.

Applicability to Other Work? We hypothesized already that Theorem 3.6 can be reused *without modification* in other full abstraction proofs with $\lambda^{\vdash}$ as a source language.

While novel results would be the most interesting of course, we can already explore the hypothesis by considering two existing full abstraction results from literature. New et al. [2016] prove full abstraction of closure conversion from a simply-typed lambda calculus with recursive types to a polymorphic language with recursive types and exceptions. By Theorem 3.6, the proof might be more easily refactored, as polymorphic functions can just be backtranslated to thunked functions.

Jacobs et al. [2021] prove full abstraction for embedding a simply-typed lambda calculus with recursive types into the cast calculus of its gradual counterpart. By Theorem 3.6, the proof might be refactored more easily, as e.g. a downward cast could just be implemented using an assert from subsection 4.3.

An Additional Formal Result? Aside from the actual statement, our *proof* of the full abstraction also seems to say a lot about the behavior of stateful computations. This is not surprising however; part our proof (emulating state computations by pure state-passing functions) corresponds exactly to Launchbury and Peyton Jones [1994]'s informal argument for why ST can be considered pure.

Instead of a full abstraction result, we want to formally state this intuition more directly. While we are not sure yet how to do so, we believe much of the machinery developed here will be relevant.

9 CONCLUSION

We have put forward a formal account for the preservation of purity when adding an ST monad to a pure call-by-value simply-typed lambda calculus $\lambda^{\vdash}$ with recursive types; the embedding from the pure language ($\lambda^{\vdash}$) into the stateful ($\lambda^{\vdash}_{\mathsf{ST}}$) one should be fully abstract. The statement has practical implications. Programmers need not worry about ST when reasoning about their pure code (at least w.r.t. equivalences), and the same holds for compiler optimizations. From a more theoretical standpoint, we can say ST does not provide additional expressive power over our pure language.

We have formally proven this result, and employed a novel proof technique in doing so. We defined our back-translation in two phases. In the first phase, we back-translated ST computations into an intermediate semantically typed language, $\lambda^{\vDash}$, defined with a binary logical relation in Iris. Doing so made it possible to emulate state computations by pure functions that pass around a state that's not syntactically typed. To prove that this back-translation was indeed semantically typed and correct, we made extensive use of Iris. We then separately proved that semantically typed contexts can, in turn, be correctly back-translated to syntactically typed ones. This we did with a translation into a universal type.

As we have seen, the semantically typed back-translation of ST provides additional insight into our proof. More generally, a semantically typed intermediate language might be useful to other researchers that study programming languages and compilers using some form of back-translation.

10 DATA AVAILABILITY STATEMENT

The results presented here are all fully formalized [Jacobs et al. 2022] in the Coq proof assistant on top of the Iris framework [Jung et al. 2018].

ACKNOWLEDGMENTS

This work was partially supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0054 and by Internal Funds KU Leuven grant C14/18/064.

REFERENCES

Martín Abadi. 1998. Protection in Programming-Language Translations: Mobile Object Systems. In *European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-49255-0_70

Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming*. Springer-Verlag.

Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 256–25615.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-Abstract Compilation by Approximate Back-Translation. In *Principles of Programming Languages (POPL '16)*. ACM, 164–177. https://doi.org/10.1145/2837614.2837618

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *History of Programming Languages*.

Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Artifact - Purity of an ST Monad. https://doi.org/10.5281/zenodo.6329773

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL, Article 7 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434288

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, 24–35. https://doi.org/10.1145/178243.178246

E. Moggi and Amr Sabry. 2001. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming* 11, 6 (2001), 591–627. https://doi.org/10.1017/S0956796801004154

Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP, Article 73 (July 2018), 30 pages. https://doi.org/10.1145/3236768

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *International Conference on Functional Programming*. ACM, 103–116. https://doi.org/10.1145/2951913.2951941

Joachim Parrow. 2008. Expressiveness of Process Algebras. *Electronic Notes in Theoretical Computer Science* 209 (April 2008), 173–186. https://doi.org/10.1016/j.entcs.2008.04.011

Marco Patrignani. 2020. Why should anyone use colours? or, syntax highlighting beyond code snippets. *arXiv preprint arXiv:2001.11334* (2020).

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Feb. 2019), 36 pages. https://doi.org/10.1145/3280984

Marco Patrignani, Eric Mark Martin, and Dominique Devriese. 2021. On the Semantic Expressiveness of Recursive Types. *Proc. ACM Program. Lang.* 5, POPL, Article 21 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434302

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 19:1–19:28. https://doi.org/10.1145/3290332

Amin Timany, Robbert Krebbers, and Lars Birkedal. 2017a. Logical relations in Iris. In *CoqPL, Date: 2017/01/21-2017/01/21, Location: Paris.*

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017b. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST. *Proc. ACM Program. Lang.* 2, POPL, Article 64 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158152

Philip Wadler. 1993. Monads for functional programming. In *Program Design Calculi*, Manfred Broy (Ed.). Springer Berlin Heidelberg, 233–264.