KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Preserving Language Abstractions

Gradual Typing and State Threads

Koen Jacobs

Supervisors:
Prof. dr. D. Devriese
Prof. dr. ir. A. Timany
(Aarhus University, Denmark)
Prof. dr. B. Jacobs

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2022

Preserving Language Abstractions

Gradual Typing and State Threads

Koen JACOBS

Examination committee:
Prof. dr. ir. P. Sas, chair
Prof. dr. D. Devriese, supervisor
Prof. dr. ir. A. Timany, supervisor
(Aarhus University, Denmark)
Prof. dr. B. Jacobs, supervisor
Prof. dr. ir. F. Piessens
Prof. dr. ir. T. Schrijvers
Dr. L. Stefanesco
(MPI-SWS, Germany)
Prof. dr. S. Balzer
(Carnegie Mellon University, U.S.)
Prof. dr. É. Tanter
(UChile, Chile)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

December 2022

© 2022 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Koen Jacobs, Celestijnenlaan 200A box 2402, B-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

I feel very fortunate for these past four years as a PhD student. Many people contributed to this fact, and in this preface, I'm taking the opportunity to explicitly thank them.

First, I want to thank my promotors. I want to thank Bart Jacobs in particular for allowing me such academic freedom, considering only whether the topics I was working on were of scientific value, regardless of whether they diverged more from his own. I want to thank Amin Timany in particular for playing a huge role in the development of my research skills —especially the beginning— by teaching me to think more formally and communicate more precisely. I am very grateful for all his crucial pointers, advice, critical questions, and —at a later stage— those of Dominique Devriese. Throughout all our collaborations, I never felt afraid to ask naive questions as both always were patient and approachable. Moreover, they were always quick to share their insights which often proved very valuable. Besides all three being really strong academically, I also had the pleasure to observe that they were equally amiable.

Going back a bit in time, I also want to thank Dominique and Andreas Nuyts for supervising my master thesis. In retrospect, they have significantly contributed to the likelihood of me starting a PhD in computer science.

Thanks to the additional members of my examination committee —Frank Piessens, Tom Schrijvers, Léo Stefanesco, Stephanie Balzer, and Éric Tanter— for their critical comments, interesting questions and suggestions. To all members of the examination committee, as well as the chair —Paul Sas—, a very big thank-you for all your work and having delivered it under such time restraints.

My gratitude also to the anonymous reviewers of our submissions for their constructive feedback. And thanks to the many people I met at the conferences and summer schools I attended for the interesting conversations we had.

Another thank-you to my supervisors for being so supportive of my short research visit to Éric Tanter at PLEIAD. Thanks to Éric and all his students for giving me such a warm welcome and the many insightful conversations, in particular with Éric, Damián Marquez, Matías Toro, and Kenji Maillard.

I also want to express my deepest gratitude to many of my colleagues in the computer science building more generally. Not only for the many interesting technical conversations, but also for the many outside-work exchanges, the ones in between, and the ones that just consisted in joking around :). I really enjoyed the company during the many lunches, coffee breaks, encounters on stairs and hallway, and I'll for sure miss it.

Additionally, I also want to say thanks to the many people providing me professional administrative support: Katrien Janssens, Annick Vandijck, Petra Asma, Anna Klewinghaus, and the many more people involved. Also, a big thanks to the people maintaining and cleaning the building, and the ones taking care of all the other logistics I am not even thinking about.

Those who contributed yet more indirectly (though very significantly) to me successfully obtaining this PhD —speaking tentatively here ;)—, were my friends, (grand) parents, siblings (in law), and nephews. I should probably say more so in daily life: I am incredibly thankful to them for so many things, and I am very grateful for having all of them in my life.

Finally, I note that the work in this thesis was partially supported by the Air Force Office of Scientific Research under award number FA9550-21-1-0054, by the Internal Funds KU Leuven grant C14/18/064, and by the Flemish Research Programme Cybersecurity.

Abstract

In this thesis, we are interested in the interplay between two conflicting desires of programming language designers. On the one hand, they want to extend their language with new features, making it e.g. more natural or efficient for programmers to express themselves. At the same time, they are often equally invested in making sure that the interesting characteristics of their language are not violated by the introduction of said features, as programmers (and the programs that they have written already) and the correctness of compiler optimizations might be relying on these.

In particular, we focus on two such scenarios where this tension is markedly present: The gradualization of statically-typed languages and the addition of state threads to the purely-functional programming language Haskell.

We argue why – unfortunately – the existing meta-theoretical properties in literature about these extensions leave unanswered whether they do preserve *all* the existing language abstractions. We answer this question more fully by considering a generic but strong property: full abstraction for the embedding from the initial language into its extension with the new features: that is, programs that are equivalent in the initial language remain so in the extended language. If this property is upheld, programmers don't need to worry about the possible interaction with code written using the new features; they can use the same kind of refactorings as they were used to, and we can be confident that existing libraries – whose correctness relies on certain abstraction in the language – remain correct when interacting with code using the new features.

For gradual typing, we show that this property functions as an interesting criterion for gradual language designers. We show that its failure sheds light on the fact that certain properties are not preserved and that it can weed out undesirable gradualizations. Moreover, we also demonstrate that the criterion is realistic, by reporting on a proof for the gradualization of a recursive simply-typed lambda calculus, mechanized in the Coq proof assistant using the Iris

program logic.

We also model the addition of state threads to Haskell in a simple setting by adding state threads to a call-by-value simply-typed lambda calculus with iso-recursive types. We report on a proof that the embedding into this extended language is fully abstract. Again, the proof is fully formalized in Coq using the Iris program logic. Moreover, it is of particular interest as it uses two novel insights: a back-translation into an intermediate semantically-typed language and the use of virtual steps to prove more refinements with a logical relation.

In sum, we take the existing idea of considering fully abstract embeddings to encode the intuition that the extended language preserves the initial language's abstractions, and by applying it to two novel contexts – gradualizing static types and adding ST to a purely functional language – we gain novel insight into these extensions, and intuition and proof strategies for fully abstract compilation more generally.

Beknopte samenvatting

In deze thesis zijn we geïnteresseerd in de wisselwerking tussen twee conflicterende verlangens van programmeertaal ontwikkelaars. Langs de ene kant willen ze hun taal vaak uitbreiden met nieuwe functionaliteiten om het bijvoorbeeld meer natuurlijk of efficiënt te maken om programma's te schrijven. Langs de andere kant zijn ze echter vaak even geïnvesteerd om te verzekeren dat de interessante eigenschappen van hun taal niet gebroken worden door het toevoegen van deze functionaliteiten, omdat programmeurs (en de correctheid van de programma's die ze al geschreven hebben) hiervan af kunnen hangen.

Specifiek, zullen we ons focussen op twee scenario's waar deze tegenstrijd zeer duidelijk is. De gradualisatie van statisch getypeerde programmeertalen en het toevoegen van state threads aan de puur-functionele programmeertaal Haskell.

We argumenteren waarom – spijtig genoeg – de eerder bestaande metatheoretische eigenschappen uit de literatuur over deze extensies de vraag openlaten of ze alle bestaande taal abstracties behouden. We beantwoorden deze vraag meer volledig door een generieke maar sterke eigenschap te beschouwen: full abstraction van het embedden van de initiële taal in haar uitbreiding met de nieuwe functionaliteiten: Dit wil zeggen, programma's die equivalent zijn in de initiële taal zijn nog steeds equivalent in de uitgebreide taal. Als aan deze eigenschap voldaan is, dan moeten programmeurs zich geen zorgen maken over de mogelijke interactie met code die geschreven is met de nieuwe functionaliteiten; Ze kunnen dezelfde refactorings gebruiken waaraan ze gewend waren en er zeker van zijn dat bestaande bibliotheken – waarvan hun correctheid afhangt van specifieke abstracties in de taal – nog steeds correct zijn wanneer ermee geïnterageerd wordt met code die de nieuwe functionaliteiten gebruikt.

Voor het gradueel typeren, tonen we aan dat deze eigenschap kan gebruikt worden als een interessant criterium voor graduele taal ontwikkelaars. We tonen aan dat wanneer er niet aan voldaan is, het aantoont dat sommige eigenschappen niet behouden zijn en dat het ongewenste gradualisaties kan uitsluiten. We

tonen alsook aan dat dit criterium realistisch is, door te rapporteren over een bewijs dat dit geldt voor de gradualisatie van een recursieve simpel getypeerde lambda calculus. Het bewijs is gemechaniseerd in de Coq bewijs assistent, gebruik makende van de Iris programma logica.

We modelleren alsook het toevoegen van state threads aan Haskell in een gesimplificeerde context door state threads toe te voegen aan een call-by-value simpel getypeerde lambda calculus met iso-recursieve types. We rapporteren over een bewijs dat het embedden in de uitgebreide taal fully abstract is. Dit bewijs is alsook geformaliseerd in Coq gebruik makende van de Iris programma logica. Het bewijs is ook van bijzonder belang omdat het gebruik maakt van twee nieuwe inzichten: Een back-translation met tussenstap in een semantisch getypeerde taal en het gebruik van virtuele stapjes om meerdere refinements te bewijzen met een logische relatie.

Kortom, we nemen het bestaande idee om het fully abstract embedden te beschouwen om de intuïtie te encoderen dat de uitgebreide taal de abstracties van de initiële taal behoudt, en door dit toe te passen op twee nieuwe scenario's – gradualiseren van statische types en het toevoegen van ST aan een puur-functionele taal – verkrijgen we nieuw inzicht in deze uitbreidingen, en intuïtie en bewijs strategieën voor fully abstract compilaties meer algemeen.

Contents

Abstract iii

Beknopte samenvatting v

Contents vii

1 Introduction 1
- 1.1 Setting the Scene . . . 2
  - 1.1.1 Existing Programming Language Abstractions versus Extensions . . . 2
  - 1.1.2 Some Background on Gradual Typing . . . 5
  - 1.1.3 Some Background on State Threads in Purely Functional Languages . . . 9
- 1.2 Explanation of Prerequisites . . . 10
  - 1.2.1 Contextual Equivalence . . . 10
  - 1.2.2 Fully Abstract Embedding . . . 12
- 1.3 Fully Abstract Embeddings when Adding Gradual Types & State Threads . . . 14
  - 1.3.1 Motivation . . . 14
  - 1.3.2 Proof Strategies . . . 16
  - 1.3.3 Overview of Thesis and Summary of Contributions . . . 18

2 Fully Abstract from Static to Gradual 19
- 2.1 Introduction . . . 20
- 2.2 Some Background . . . 24
  - 2.2.1 Gradualizing STLC_μ . . . 24
  - 2.2.2 Fully Abstract Compilation . . . 28
- 2.3 Gradual Languages Without an FAE . . . 29
  - 2.3.1 Gradualising a Parametric Language, Naively . . . 30
  - 2.3.2 Gradualizing a Parametric Language, Non-naively . . . 31
  - 2.3.3 Gradualizing an Information-flow Language with References 32

2.4 FAE for the GTLC$_\mu$ . . . 33
2.4.1 High-Level Overview . . . 33
2.4.2 Backtranslation . . . 35
2.4.3 Logical Relations and Correctness of the Backtranslation 44
2.5 Discussion . . . 53
2.5.1 Contextual Equivalence in GTLC$_\mu$ . . . 53
2.5.2 Partiality in the Source Language . . . 54
2.5.3 Recursive Types vs. Term Recursion? . . . 55
2.5.4 Robust Relational Hyperproperty Preservation for GTLC$_\mu$ 56
2.5.5 Other Gradual Languages . . . 57
2.5.6 The Pursuit of FAE . . . 57
2.6 Related Work . . . 59
2.7 Conclusion . . . 60

3 Purity of an ST Monad **61**
3.1 Introduction . . . 62
3.2 Preliminaries . . . 67
3.2.1 Definitions of $\lambda^{\vdash}$ and $\lambda^{\vdash}_{ST}$ . . . 67
3.2.2 Defining the Intermediate Language $\lambda^{\vDash}$ . . . 70
3.3 Our Main Theorem and Overview of the Proof . . . 76
3.4 Correctly Back-Translating Semantically Typed Contexts . . . 78
3.4.1 Defining the Syntactically Typed Back-Translation . . . 78
3.4.2 Proving Correctness of Back-Translation; First Refinement 81
3.4.3 Proving Correctness of Back-Translation; Second Refinement . . . 83
3.5 Correctly Back-Translating Stateful Contexts . . . 86
3.5.1 Definition of Back-Translation . . . 87
3.5.2 Semantic Well-Typedness . . . 88
3.5.3 Proving Correctness of Back-Translation; First Refinement 92
3.6 Discussions . . . 95
3.7 Related Work . . . 97
3.8 Future Work . . . 99
3.9 Conclusion . . . 100

4 Conclusions **103**

Bibliography **109**

List of Publications **117**

Chapter 1

Introduction

Programming language designers often face the conflict between the desire to extend their language with new features and the desire to maintain its existing properties and characteristics. The former allows programmers to express themselves more easily or efficiently. The latter is equally important, however, as programmers (and the correctness of the code they have written) and compilers rely on these properties.

With this in mind, we will study two practically and theoretically relevant language extensions.

1. The extension of statically typed languages by gradualizing their type system.
2. The extension of a purely-functional programming language with state threads.

Unfortunately, the literature prior to our work does not consider whether *all* the initial language abstractions are preserved upon these extensions. So we will set out to answer this question more completely by considering whether the *embedding into these extended languages is fully abstract*. That is (roughly), whether *all* initial equivalences continue to hold in the extended language.

Intuitively, this expresses that the abstractions (in the initial language, before extending it) are – to a very large extent – not broken upon extension. Practically, this means, e.g., that programmers can keep reasoning about their code like they were used to, not having to worry about the possible interaction with the

new features; that compiler optimizations are still valid; that existing code is guaranteed not to be broken when interacting with the new features, etc.

For gradual typing, we will demonstrate that this property is useful as a general criterion for gradual language designers, and we will show that it is realistic by reporting on a formal proof for the gradualization of a simply-typed lambda calculus with recursion. For state threads, we will model the extension of Haskell with state threads in a simplified setting by adding an ST monad to a call-by-value recursive simply-typed lambda calculus, after which we report on a formal proof that the embedding into the language extended with ST is fully abstract.

In this introduction, we give the necessary background to get a better understanding for what this all means, why it is important, and how it relates to previous work. We start out with some prerequisites. In §1.1.1, we go over a couple of examples to help understand why in general, when you are extending a language, you are very likely to break its abstractions. Afterwards, we give a short introduction about gradual typing (§1.1.2) and Haskell's ST monad (§1.1.3), emphasizing why we (in these scenarios) are heavily invested in preserving our language abstractions, citing existing literature to back this up. Last but not least, readers unfamiliar can familiarize themselves with contextual equivalence in §1.2.1. This is a necessary concept to understand fully abstract embeddings, finally explained in §1.2.2.

Having the required background out of the way, we'll motivate in more detail why we are considering full abstraction for the embeddings into the extensions with gradual types and state threads (§1.3.1). In §1.3.2, we'll talk a bit about how we can prove embeddings/compilers to be fully abstract. We'll introduce some existing proof strategies from literature and some other machinery that we'll make use of. Lastly, we'll summarize our contributions in §1.3.3.

1.1 Setting the Scene

1.1.1 Existing Programming Language Abstractions versus Extensions

We observe the following dichotomy in the study of programming languages.

- Programming languages often provide us with useful abstractions that can help us to reason about our code. A rich tradition of research consists

of formally defining and proving the guarantees that are provided by these abstractions.

- Programming languages are not "fixed". Researchers and implementers are constantly building upon existing languages, extending them in one way or another with a variety of motivations. They want, for example, to better accommodate some programming style, support some kind of domain-specific reasoning, make some existing features of the language more accessible...

Conflicting interests arise. Coming up with new features and adding them to a language may *seem* an unambiguously harmless endeavor. That is, until we start to wonder about their possible effect on the existing abstractions in our language. To get some more intuition for this, we go over some examples illustrating this friction.

Example 1.1.1 (Breaking parametricity with type inspection)**.** In a language like System F [46], we can write the following swap function, swapping the components in a pair of an arbitrary (polymorphic) product type.

$$swap : \forall X.\, \forall Y.\, X \times Y \to Y \times X$$

$$swap = \Lambda X.\, \Lambda Y. \lambda p : X \times Y.\, (\pi_2\ p, \pi_1\ p)$$

In all our excitement, we – as language designers – might now want to extend our expressive power over this kind of types. We might consider the possibility that for some of these instantiations, we want to have different implementations. We could be interested in a function that will – again – swap the components, but when both components are integers, it will simply negate them. Accordingly, we would add new primitives to our language that will allow us to write such functions.

$$swap' : \forall X.\, \forall Y.\, X \times Y \to Y \times X$$

$$swap' = \Lambda X.\, \Lambda Y. \lambda p : (X \times Y).$$

$$\texttt{if}\ X = \mathbb{Z} \wedge Y = \mathbb{Z}\ \texttt{then}\ (-(\pi_1\ p),\ -(\pi_2\ p))\ \texttt{else}\ (\pi_2\ p,\ \pi_1\ p)$$

Adding type inspection is a perfectly valid design decision; for any polymorphic type, we can express many more functions! But this advantage comes at a price; from a different perspective, it is itself a disadvantage!

Without the above type inspection, any well-typed function of $\forall X.\, \forall Y.\, X \times Y \to Y \times X$ is *guaranteed* to behave like the above swap-function. This is a

consequence of parametricity, formally defined by Reynolds [46] as a binary logical relation and later popularized by Wadler's paper [61] titled "Theorems for free!".

By adding type inspection (in the manner we did above), we sacrifice a large part of that reasoning power. Moreover, programs and compiler optimizations that relied upon these guarantees are no longer valid.

This sacrifice is even more apparent when we look at examples related to information hiding enforced using parametricity. In System F , one can encode "existential" types [45] to implement – for instance – a set of integers over an abstract interface, keeping its concrete internals hidden from its users. Concretely, the latter could be represented by the following type.

$$\exists X.\, X \times (\mathbb{Z} \to X \to X) \times (\mathbb{Z} \to X \to X) \times (\mathbb{Z} \to X \to \mathbb{B}) \times (X \to \mathbb{N})$$

In the different components of this product, we would encode the following (left to right): an empty set, functions to add and remove elements from a set, checking elementhood of an integer, and returning the size of a set.

Unfortunately though, the addition of type inspection as we suggested above, is a direct threat to our abstract interface as clients can simply do type inspection to inspect the internals of our implementation.

Example 1.1.2 (Breaking "purity" with global references)**.** Consider the canonical way of adding references to mutable values to a recursive simply-typed lambda calculus [45]. In the extended language, we can do a lot of new things. We can carry around shared memory across different computations, expressing things we couldn't do before.

There is a flip side though. Initially, e.g. a closure computing an integer (something of type $() \to \mathbb{Z}$), is guaranteed to act as a mathematical function modulo divergence; executing this closure will *always* give us the same output, independent of what code ran before or after it.

Obviously, we cannot guarantee this anymore with global memory into play, as we can easily write something like $\lambda x : ().\, r \leftarrow !r + 1;; !r$.

This trade-of can be important as programmers may expect functions to act "purely" and reason about their programs accordingly. Likewise, a compiler might make use of this property by computing the integer only once (the expression of type $() \to \mathbb{Z}$) and remembering it, something which is no longer valid in the presence of global references.

Example 1.1.3 (termination vs general recursion)**.** Take a bare simply-typed lambda calculus to which we add a general fixpoint operator; something that obviously provides us with great value in expressing programs.

Again, it is not for free. We have to pay for it by giving up termination. We cannot guarantee anymore that an expression $e : \mathbb{Z}$ will actually give us back an integer upon evaluation.

For most programming purposes, this is a price that is worth paying. At other times though, this is a no-go. There exist "programming" languages that function as proof assistants [63, 9]; they have rich type systems that encode propositions as types and proofs of a proposition as programs of the corresponding type. In such scenarios, we need every program (or proof) to necessarily terminate for our proof system to be sound hence one cannot just add general recursion.

The observations here should be no surprise: the less expressive power we have when writing programs of a given type, the stronger the guarantees we can assume for arbitrary programs of that type.

Often, the "initial" guarantees are hardly cared for though. They are considered a mere coincidence (if even considered at all), and the added expressiveness of the features (that break these guarantees) is prioritized. General recursion for instance, is often considered essential in a general-purpose programming language; indeed most programming languages have it, even though it comes "at the cost of" sacrificing termination. Moreover, most programming languages have global heap, (almost always) "at the cost of" the "pure" behavior of functions. In these cases, the aforementioned tension is simply not there.

At other times, we actually *do care* about the existing language guarantees, and we aspire to preserve them upon extension because programmers may simply rely on these principles, existing programs (whose correctness might crucially rely on the abstractions from the initial language) should not break in the extension, compiler optimizations on the initial language should remain valid etc.

In the next two sections (§1.1.2 and §1.1.3), we will introduce gradual typing and Haskell's ST monad, emphasizing why in these contexts, we want to preserve existing language abstractions.

1.1.2 Some Background on Gradual Typing

Motivation for Gradual Typing The motivation behind gradual typing starts with the recognition that both static and dynamic typing have their own merits [50].

With static type systems, one can *statically* enforce (a wide variety of) desirable properties on the programs of our types. In a typed lambda calculus, we know for instance that an expression of type $\mathbb{B}$ can evaluate to an actual boolean only, and furthermore, we are not able to accidentally treat it as a function say by applying it to an argument, thus successfully excluding any type errors.

With dynamically typed languages, the name of the game is enforcing properties *dynamically*, by checking them at runtime and throwing an error if they are not satisfied. In a dynamically-typed counterpart of a typed lambda calculus for instance, we also don't want to apply a boolean to an argument; but instead of statically enforcing that this can never happen, we just throw a type error at runtime if it does.

From the above perspective, static typing has the obvious advantage. But of course there is a flip side. In order to get the static guarantees from the static type system, our code has to – well – type-check! This can sometimes be a bit unwieldy, requiring too much effort from programmers who find themselves "fighting against the type-system". As such, a dynamically-typed language offers a different programming paradigms that's more flexible and easier to get started with.

Instead of having to choose either one of these paradigms, Siek and Taha [50] set forward a line of research on "gradually-typed languages" that encompass a static language, its dynamic counterpart, and everything in between. The authors motivate the gradual language as allowing "a programmer-controlled migration between dynamic and static typing". As such, we can start from (pre-existing) untyped code and gradually migrate it to be more statically-typed, gradually gaining the static guarantees for the parts we have already migrated. Related here, is the perspective by Garcia and Tanter [18], who note that gradual typing can help lower the barrier to more advanced typing disciplines. Lastly, Siek et al. [52] also note that "[a gradually typed language] puts the programmer in control of which regions of code are statically or dynamically typed". In other words, programmers can assess different priorities in different regions of code, and accordingly choose to program more statically or more dynamically.

What Does it Look Like? To get some idea of what a gradual language looks like and to be able to explain the properties in the next section, we briefly go over an example here: The gradualization of the simply-typed lambda calculus [52].

The first thing we do is to introduce some imprecision in our types. Formally, we add an unknown type to our grammar, $?$, that stands for a kind of wild card. We can for instance, talk about the type $\mathbb{Z} \times ?$ that consists of pairs whose first

component are integers and whose second component can be anything at all (statically, we don't know anything about it).

We can then generalize the existing typing rules to incorporate these imprecise types. To succinctly describe how this works, we quote Lehmann and Tanter [31]

> In essence, gradual typing is about plausible reasoning in presence of imprecise type information

Imprecision causes uncertainty which we take care of *optimistically*. We can apply a function of type $(? \times \mathbb{B}) \to \mathbb{Z}$ to an argument of type $(\mathbb{Z} \times ?)$ because inhabitants of $\mathbb{Z} \times ?$ are *plausibly* of type $? \times \mathbb{B}$; formally, we have a "consistency" relation between types (relating the two types).

This use of static optimism can turn out to be unfounded at runtime however. Indeed, at runtime, our values of type $\mathbb{Z} \times ?$ can turn out, e.g., to have an integer in the second component (instead of a boolean). To account for this, a cast is inferred and inserted, to be run at run-time. In general, well-typed terms in the gradual language are translated to a "cast calculus", where every use of "optimism" (or "consistency") in the typing derivation is translated to an appropriate cast.

Remark 1.1.1. Note that even in the presence of imprecision, the type system can still exclude lots of undesirable programs *statically*. We cannot apply the function of type $(? \times \mathbb{B}) \to \mathbb{Z}$ to an argument of type $? \times \mathbb{Z}$ for instance, because $? \times \mathbb{B}$ is not consistent with $? \times \mathbb{Z}$ as $\mathbb{B}$ is not consistent with $\mathbb{Z}$.

Existing Criteria for Gradual Languages When one designs a language, one hopes that it lives up to the intended use-cases one had in mind. As such, Siek et al. [52] state a couple of properties for the gradualization of the simply typed lambda calculus. They call these properties "refined criteria", arguing that they should be taken into account by gradual language designers in general.

Conservative extension The gradual language should completely contain the static language and the dynamic language. Moreover, the semantics of closed, fully precise (without making use of any imprecise types) programs in the gradual language should agree with their semantics in the static language. Likewise, embedding a closed dynamic program in the gradual language should result in a program with the same semantics.

Blame theorem For our purposes, it is sufficient to describe only a consequence of this theorem, intuitively summarized as follows: precisely-typed portions of code cannot be blamed for a runtime error.

Gradual Guarantee Making types more imprecise in any well-typed program should result in a program that is still well-typed. Moreover, if the initial program was free of type errors, then resulting program should be so as well, and if the initial one terminates to a value, the resulting one should terminate to the same one.

The conservative extension criterion just states that programmers can write fully static (as in fully precise, like in the statically-typed language) and fully dynamic programs, and that such programs should behave as expected; like they behave in the statically-typed or dynamic language respectively. The gradual guarantee provides some justification for the aforementioned migration path from dynamic code to more precisely-typed code. Making our types more precise should not have any influence on how terms are evaluated, except of course by returning runtime errors when these more precise types were not "correct"!

The blame theorem provides some justification for the desire that when we go through the trouble of making our annotations more precise, we should *gradually* get some more of the guarantees provided by the static typing discipline. Put differently, we shouldn't have to wait until our program is fully precise! Thus, as such, we get that fully precisely-typed portions of code are not to blame for runtime errors.

In general, types can give us much more than just type safety though.

Even in a simply typed case, we care about much more than that. Intuitively, we might expect for instance that composing a function $f : \mathbb{Z} \to \mathbb{B}$ with a function $g : \mathbb{B} \to \mathbb{Z}$ gives us a function on integers that can only reach two different integers (as we only have two different booleans). Although this property is not covered by the criteria we talked about, we still might find it very weird if it did not hold in the presence of gradual types (e.g. for functions $f = \lambda x : \mathbb{Z}.\, x :: ? :: \mathbb{B}$ and $g = \lambda x : \mathbb{B}.\, x :: ? :: \mathbb{Z}$, where we use $e :: \tau$ to denote the ascription of expression e to the type τ)![1]

We make the same observation in richer typing disciplines. Authors researching gradualizations of more expressive static types often supplement these criteria by introducing their own. Gradual language designers working on gradualizing polymorphic types for instance [33, 33, 4, 5, 59, 29], will often define some notion of parametricity. Gradualizations of security types [17, 58] will often come with some notion of non-interference.

[1] This is not as contrived as it might seem. Given some "lazy" and overly optimized evaluation of casts, this composition might even act as an identity function!

Unfortunately, no-one has of yet – prior to our work – proposed a general criterion to express that a gradual language preserves guarantees of a statically-typed language more generally.

1.1.3 Some Background on State Threads in Purely Functional Languages

28 years ago, Launchbury and Peyton Jones [30] added mutable state to the "purely functional" programming language called Haskell. Their decision was motivated as follows:

> Purely functional programming languages allow many algorithms to be expressed very concisely, but there are a few algorithms in which in-place updatable state seems to play a crucial role. For these algorithms, purely-functional languages, which lack updatable state, appear to be inherently inefficient (Ponder, McGeer & Ng [1988]).

Adding mutable state to the Haskell programming language was a very challenging issue, due to a number of reasons [30]. We will focus on one of them; Launchbury and Peyton Jones [30] wanted to make sure not to lose the "pure" aspect of the language (something which is – as we have seen already in Ex. 1.1.2 – not possible when naively adding state). In their introduction, they proudly state that (after adding "state threads"):

> ...all the usual techniques for reasoning about functional programs continue to work.

The authors come up with a nifty solution that we'll explain from a bird's eye here; a more detailed introduction can be found in §3.1. Launchbury and Peyton Jones [30] enforce stateful computations to be encapsulated in so-called state threads. We have a type `ST` s a whose values represent (unexecuted) stateful computations that – *when executed* – produce something with final value of type a (the first argument, s, is crucial, and we'll explain its purpose later on). Furthermore, we have a `runST` construct, that given such a state computation, executes it to give us back the result of that computation (something of type a). Here's the important part; the type system enforces that after executing a state thread, we cannot observe any of the references that were allocated in the process. In §3.1, we explain how the type system does this exactly, and it will become clear why ST is quantified over two types.

Going back to Ex. 1.1.2, we note that a function of type $() \to \mathbb{Z}$ can now allocate any number of references and do some arbitrary reading and writing to them. Important here, is the fact though that it cannot have used locations from different state threads. As such, we can expect it to still satisfy the property that it should always give back the same integer.

Ideally, we should be able to formalize our intuition that Haskell's state threads are "well-behaved". Fortunately, there exists some research already that contributes to this ideal.

Timany et al. [56] have recently backed up this intuition somewhat by modeling this extension in a call by value System F with an ST monad. They define a binary logical relation to prove a set of 9 rewriting principles (formally, a set of contextual equivalences/refinements, see §1.2.1) one can expect from a "pure" language like e.g. commutativity, stating that one can always rewrite $\texttt{let}\ x = e_2\ \texttt{in}\ (e_1, x)$ with (e_1, e_2) and vice versa (in other words, the evaluation of the two components should be *commutative*).

While the results by Timany et al. [56] encompass a couple of interesting equivalences in the presence of state threads, it unfortunately does not rule out that the possibility that some equivalences from the initial language are still broken in the presence of ST.

1.2 Explanation of Prerequisites

1.2.1 Contextual Equivalence

Before being able to introduce fully abstract embeddings, we need to know what *contextual equivalence* is. Readers already familiar with the latter are still recommended glancing over the examples though, as we'll use them later.

Defining this notion both formally and generically (in an arbitrary programming language) is a bit unwieldy (for the languages we will be working with, we'll give a formal account in Defs. 2.2.2, 2.2.3 and 3.3.1); rather, we give the following intuitive definition which satisfies for now.

Definition 1.2.1. In a given programming language, we say that *two expressions* (of the same interface) are *contextually equivalent* if we can always freely interchange the two as subexpressions in some larger final program (without changing its behavior).

The concept of contextual equivalence is well-known and goes back at least to Morris [36] who introduces it in a particular context, calling it "extensional equivalence".

Remark 1.2.1. Implicit in this definition is that everything is well-formed. In a simply typed language for instance, two equivalent expressions, say e and e', are always of the same interface, say $\Gamma \vdash _ : \tau$, and we known that plugging them into arbitrary surrounding code *that provides the accordingly-typed variables in* Γ *and expects something of* τ, we obtain two indiscernible programs. We write $\Gamma \vdash e \approx_{ctx} e' : \tau$ when e is contextually equivalent to e' under Γ and τ.

Let us see some examples of this, taking inspiration from our examples in §1.1.1.

Example 1.2.1. Continuing on what we said in Ex. 1.1.1, we can expect the following contextual equivalence in (terminating) System F.

$$f : \forall X.\, \forall Y.\, X \times Y \to Y \times X, x : \mathbb{Z}, y : \mathbb{Z} \vdash f\ [\mathbb{Z}]\ [\mathbb{Z}]\ (y, x) \approx_{ctx} (x, y) : \mathbb{Z} \times \mathbb{Z} \quad (1.1)$$

The surrounding code in our program can feed in any function f of the appropriate type, and any two integers x and y. As mentioned we know however that this function needs to act as a swap function.

Likewise, we can consider two equivalent implementations on our abstract interface to work with sets of integers. One, for instance, that uses lists of integers, say p, and another that works with *ordered* lists of integers (to e.g. make the membership operator more efficient), say p'. If both of our implementations do what we expect them to do, we can consider them contextually equivalent in System F.

$$\vdash p \approx_{ctx} p' : \exists X.\, X \times (\mathbb{Z} \to X \to X) \times (\mathbb{Z} \to X \to X) \times (\mathbb{Z} \to X \to \mathbb{B}) \times (X \to \mathbb{N}) \quad (1.2)$$

Intuitively, we can, e.g., put some stuff in an empty set (say 3,6,2), remove some stuff (say 7 and 6), and ask for its length; both implementations should agree upon the result (that is 2), independent from the fact that they are representing their data differently ($[3, 2]$ versus $[2, 3]$ before taking the length) as we cannot inspect these internals.

Example 1.2.2. Due to what we concluded in Ex. 1.1.2, we can expect the following contextual equivalence in some "pure" simply-typed lambda calculus.

$$c : () \to \mathbb{Z} \vdash c\ ();c\ ();c\ () \approx_{ctx} c\ () : \mathbb{Z} \quad (1.3)$$

Intuitively, the surrounding context can call this code with any closure c (possibly even multiple ones) calculating an integer. All of them should act as mathematical functions though (up to termination), as such, evaluating it three times, or just one time shouldn't matter.

Example 1.2.3. Lastly, continuing on 1.1.3, we can expect the following equivalence in a terminating language (without side effects).

$$f : () \to () \vdash f\ () \approx_{ctx} () : () \tag{1.4}$$

In sum, we can construct a lot of interesting contextual equivalences that directly reflect or rely on the intricate properties, guarantees or "reasoning principles" of a language.

Now that we know what contextual equivalence is, we can explain fully abstract embeddings.

1.2.2 Fully Abstract Embedding

Fully abstract embedding is a specific case of *fully abstract compilation*, a property from the secure compilation community, defined as follows [2, 1].

Definition 1.2.2. Given a *compiler* from a source to a target language, say $[\![_]\!] : \mathcal{L} \to \mathcal{L}$, we say that it is *fully abstract* if it *preserves* (Def. 1.2.4) and *reflects* (Def. 1.2.3) contextual equivalence.

Okay, so what does it mean for a compiler to preserve and reflect equivalences?

Definition 1.2.3. A compiler, say $[\![_]\!] : \mathcal{L} \to \mathcal{L}$, *reflects* contextual equivalence, if for all terms whose compilations are equivalent in the target language, we have that they are also equivalent in the source language.

That is, given arbitrary source terms (of the same interface), say e and e', we have the following:

$$\textit{If } [\![e]\!] \approx_{ctx} [\![e']\!], \textit{ then } e \approx_{ctx} e'. \tag{1.5}$$

Put differently: if in the source language, two terms are *not* contextually equivalent, then their compilations are not either. This property serves somewhat as a sanity check and as such, it is often regarded as a necessary condition for or a consequence of compiler correctness [42].

With regard to this thesis (where we are considering the special case of *embeddings* from initial languages into their extensions), this property is not that interesting as (under some natural assumptions) it is automatically satisfied. If in the initial language, two terms are not equivalent (that is, there exists some surrounding code that can distinguish the two), it is the case that there exists

some surrounding code in the extended language (more specifically, the exact "same" code) that can distinguish the two as well.

The other property is more interesting to us, so let us go over it now.

Definition 1.2.4. A compiler *preserves* contextual equivalence, if for all terms in the source language that are equivalent, their compilations are as well.

That is, given arbitrary source terms (of the same interface), say e and e', we have the following:

$$\textit{If } e \approx_{ctx} e', \textit{ then } [[e]] \approx_{ctx} [[e']]. \tag{1.6}$$

This means that for embeddings (compilers), if in the initial (source) language, no surrounding code can "distinguish" between two initial (source) programs, no surrounding code in the extended (target) language can either (on their compilations).

To get some intuition about this property, it is useful to present some extreme examples that do *not* satisfy it.

Example 1.2.4. The contextually equivalent terms in Eq. (1.1) are not equivalent anymore when considered in the extension of the language with type inspection (as described in Ex. 1.1.1). We can, for instance, consider the following surrounding code (or "context").

$$\texttt{let } f : \forall X.\, \forall Y.\, X \times Y \to Y \times X = \mathit{swap'} \texttt{ in}$$

$$\texttt{let } x = 5 \texttt{ in let } y = 6 \texttt{ in } [\cdot]$$

Here $[\cdot]$ represents the "hole" of this context. Plugging in $f\ [\mathbb{Z}]\ [\mathbb{Z}]\ (y, x)$ will give us a program that evaluates to $(-6, -5)$, while plugging in (x, y) we'll obtain a program that simply evaluates to $(5, 6)$.

Likewise, in the presence of type inspection, our two contextually equivalent implementations of our abstract interface for sets is also not preserved, as we can plug it in the following context:

$$\texttt{unpack } [\cdot] \texttt{ as } X; \big(\mathit{empty} : X,\ \mathit{add} : \mathbb{Z} \to X \to X,\ \mathit{remove} : \mathbb{Z} \to X \to X,$$

$$\mathit{elemof} : \mathbb{Z} \to X \to \mathbb{B},\ \mathit{size} : X \to \mathbb{N}\big) \texttt{ in}$$

$$\texttt{let } k = \mathit{add}\ 8\ (\mathit{add}\ 4\ \mathit{empty}) \texttt{ in}$$

$$\texttt{if } X = \mathit{List}\ \mathbb{N} \texttt{ then } \mathit{head}\ k \texttt{ else } 0$$

The contexts unpacks its hole; it checks out if the type used in our implementation is a list of numbers, and if it is, it will simply take the head of it (otherwise it will return 0). Plugging the implementation that uses lists (but not ordered lists) we will get a program that evaluates to 8. Plugging in the implementation that uses *ordered* lists, we will get a program evaluating to 4.

Example 1.2.5. The equivalence from Ex. 1.2.2 can also be broken, by canonically extending the language with mutable memory. Consider, for instance, the following surrounding code.

$$\texttt{let } r = \texttt{ref } 5 \texttt{ in}$$

$$\texttt{let } c = \lambda x : (). r \leftarrow !r + 1;; !r \texttt{ in } [\cdot]$$

Plugging in $c\ ();c\ ();c\ ()$ will give us a program that evaluates to 8, while plugging in $c\ ()$, the program will evaluate to 6.

Example 1.2.6. Lastly, the equivalence from Ex. 1.2.3 also no longer holds when we extend the language with general recursion. With ω a trivially diverging term, we can construct the following surrounding code.

$$\texttt{let } f = \lambda x : (). \omega \texttt{ in } [\cdot]$$

Indeed, plugging in the term $f\ ()$ will give us a diverging program, while plugging in $()$ we'll simply get a program that evaluates to unit.

At the end of §1.3.1, we'll give some intuition for what it means when an embedding *does* preserve all contextual equivalences.

1.3 Fully Abstract Embeddings when Adding Gradual Types & State Threads

1.3.1 Motivation

At the end of §1.1.2 and §1.1.3, we have seen that there exists quite some literature out there already stating and proving meta-theoretical properties about gradual languages and Haskell-like languages with ST. However, they leave unanswered an important question.

"Is this all there is to it?" The existing meta-theoretical properties in the pre-existing literature are a great start. But having these available, we are

still left with a question. Is this all there is to it? In other words, do these meta-theoretical properties (of the extended languages) completely cover the essence of the initial language that we hope to preserve? In both the scenarios we are considering, this is often very unclear; sometimes the answer is even a "no".

The Answer for Gradual Typing In §1.1.2, we saw already that the refined criteria are not enough as they only cover type safety. As such, gradual typing researchers define and prove their own additional guarantees about their gradual languages. But even with these additional properties, it is still not clear whether *all* the reasoning principles from the static typing discipline are preserved.

One clear example here are the gradualizations of polymorphic type systems. Most authors have the ambition to have their gradual language satisfy "parametricity" [33, 33, 4, 5, 59]. But all of them – with the notable exception of the partial gradualization by Labrada et al. [29] – define and satisfy only a *weaker* notion of "parametricity" than the one in System F; they lose some of the reasoning power that was available in the static language [11].

In a paper arguing that gradual languages should preserve meta-theoretical properties beyond type-safety, Garcia and Tanter [18] propose that the "soundness" of gradualized type systems is to be defined relative to some chosen "meaning of types", and that gradual language designers should be explicit about said meaning to know what we (as programmers) can (or cannot) expect from the types.

But can we really say that our gradual language is the gradualization of some statically typed language, if not all its reasoning principles are preserved? If this is not the case, programmers can rely *only* on the reasoning principles that follow by the meta-theoretical properties provided by the gradual language designers.

The Answer for State Threads We make a similar observation here. The collection of contextual equivalences (and refinements) proven by Timany et al. [56] (in a Haskell-like language with ST) are a great start. But still, it is very unclear whether they cover all the reasoning power from the initial language. Moreover, it can be very non-trivial to use that logical relation to prove new equivalences; in fact it is not even clear if they can even be proven at all using that logical relation.

Can we really say that adding state threads doesn't break pureness if we are not sure whether all the reasoning principles from the pure language are preserved upon adding state threads?

Our Contribution to this Question We'll take a *generic* and *ambitious* approach to *complement* the existing literature. Instead of defining more precise language-specific properties to be satisfied in the extensions though, we shift our focus to the *preservation* of properties guaranteed in the initial language, side-stepping difficult questions like what properties exactly the static (to-be-gradualized) types encompass, or what "purity" means exactly. With this strategy in mind, we will consider our embeddings (into the extended language) to be fully abstract. That is, we'll consider whether all the contextual equivalences that hold in the initial language are still upheld after extending said language.

Practically, this means that programmers can refactor and – to a very large extent – reason about their code in the initial language just like they were used to, without having to worry about the possible interactions with the extended language. Also, one can reuse the same compiler optimizations on subexpressions in the initial language, and be confident of their validity (assuming of course that they were valid in the first place, see Rmk. 1.3.2 also).

Remark 1.3.1. It is not a novel idea that a fully abstract translation can encode the intuition that the target language does not provide any more expressive power over the translated terms. As far as we know, it goes back to at least the early nineties [34, 47].

Remark 1.3.2. We are careful to point out, that proving a particular embedding to be fully abstract, we don't obtain any *absolute* knowledge about the extended language. That is, we only prove that *if* some equivalence holds in the initial language, it must continue to hold in the target language. As such, refactorings and compiler optimizations for instance are only valid if they were so in the initial language. A more elaborate discussion follows in §3.7.

Remark 1.3.3. Full abstraction guarantees the preservation of language abstractions to a very large extent but as noted by Abate et al. [3], it does not guarantee the preservation of *every* property we might care about. We note here that for the two language extensions we consider in this thesis, we are additionally proving a stronger property: "robust relational hyperproperty preservation", the strongest one laid out by Abate et al. [3].

In §1.3.3, we will cover our contributions in detail.

1.3.2 Proof Strategies

High-level Intuition How can we hope to prove full abstraction of an embedding? As mentioned in §1.2.2, preservation of contextual equivalence (Def. 1.2.4) is really the interesting part, so we'll focus on it here.

Let us remind ourselves that in order to prove preservation of equivalence, we need to prove the following: If in the initial language no surrounding code (or context) can distinguish between two terms, no surrounding code in the extended language can either. The way in which this is typically proven in literature is by arguing that we can "emulate" the contexts in the extended language somehow by contexts in the initial one [42]. One thus starts by defining candidate emulations or "back-translations" of the contexts (in the extended language) by contexts in the initial language, after which it is proven that these candidates actually do fulfill their intended purpose: that they correctly emulate their original contexts in some sense.

Defining Candidate Back-translations For both our two proofs, we will rely heavily on the idea of using a "universal type" to facilitate our back-translations. This is not novel; the idea has been used already (in several variations) in other fully abstract compilation proofs by New et al. [38] and Devriese et al. [10] and later by Patrignani et al. [43]. In Chapter 2, we'll use it to back-translate expressions of the unknown type. In Chapter 3, we'll also use it to translate arbitrary untyped expressions as an initial step in the first part of a two-phase back-translation (more on this later).

In Chapter 3, we'll also use the following intuition of Launchbury and Peyton Jones [30],

> ...a stateful computation is a *state transformer*, that is, a function from an initial state to a final state.

back-translating state computations first to untyped (yet still "semantically-typed") state transformers.

Proving "Correctness" of Candidate Back-translations Like many full abstraction proofs from literature [42], we will use binary (cross-language) logical relations to prove the "correctness" of our candidate back-translations.

We'll define our logical relations using the Iris programming logic [28]. This will allow us intuitive reasoning about cyclic features and modular reasoning about state – both the physical (e.g. the global heap in the language extended with ST) and the "logical" (e.g. the logical heap of state computations and state that is being passed around in state transformers) – and the complex relation between the two.

Although Iris hasn't previously been used to prove fully abstract compilation results, it has been used many times before to define binary relations for

relational reasoning, e.g. by Timany et al. [55], Frumin et al. [15], Timany et al. [56], and Timany and Birkedal [54] among many.

1.3.3 Overview of Thesis and Summary of Contributions

Here, we'll go over the remaining chapters in our thesis, summarizing our contributions along the way.

In Chapter 2, we consider full abstraction for the embedding from a static language into its gradualization as a useful criterion for gradual language designers. We demonstrate that it is useful; it can give us a new perspective on existing gradualizations and exclude undesirable ones. Moreover, we also show that it is a realistic criterion, by reporting on our proof for the gradualization of the simply-typed lambda calculus with iso-recursive types [52].

The proof is completely formalized in the Coq proof assistant [63] using the Iris library [28] allowing us to more easily deal with recursive and unknown types. It is – to our knowledge – the second full abstraction proof that is completely formalized (after [10]) and the first that uses a program logic.

Secondly, in chapter 3, we report on a full abstraction proof for the embedding of the simply typed lambda calculus with iso-recursive types into its extension with an ST-monad.

Again, the proof is formalized in Coq, making – as we shall see – extensive use of Iris. The proof itself also has two novel aspects.

1. We introduce a new proof technique; we first back-translate into an intermediate semantically-typed lambda calculus. In a separate theorem, we then show that semantically-typed backtranslations are sufficient. As we'll discuss, we think that the latter theorem can be reusable and that it provides us with additional insight about our proof.

2. We expose that the canonical way in which binary logical relations are defined for contextual refinement (in a simply-typed lambda calculus with recursive types) does not allow us to prove some naturally occurring equivalences, and we present a novel technique to work around this.

Lastly, in Chapter 4, we contemplate a bit more on what we have done and present some interesting future research directions.

Chapter 2

Fully Abstract from Static to Gradual

The contents of this chapter are taken from our publication with the same title [26], though it contains some additional notes and some changes were made to integrate it more nicely into this thesis. I – Koen Jacobs, the thesis student – am the first author of said paper. My advisors and co-authors Amin Timany and Dominique Devriese contributed ideas and guidance and contributed to the writing of the paper, but other work was performed by me.

What is a good gradual language? Siek et al. have previously proposed the refined criteria, a set of formal ideas that characterize a range of guarantees typically expected from a gradual language. While these go a long way, they are mostly focused on syntactic and type safety properties and fail to characterize how richer semantic properties and reasoning principles that hold in the static language, like non-interference or parametricity for instance, should be upheld in the gradualization.

In this chapter, we investigate and argue for a new criterion previously hinted at by Devriese et al. [11]: the embedding from the static to the gradual language should be fully abstract. Rather than preserving an arbitrarily chosen interpretation of source language types, this criterion requires that *all* source language equivalences are preserved. We demonstrate that the criterion weeds out erroneous gradualizations that nevertheless satisfy the refined criteria. At the same time, we demonstrate that the criterion is realistic by reporting on a mechanized proof that the property holds for a standard example: GTLC_μ, the natural gradualization of STLC_μ, the simply typed lambda-calculus with

equirecursive types. We argue thus that the criterion is useful for understanding, evaluating, and guiding the design of gradual languages, particularly those which are intended to preserve source language guarantees in a rich way.

2.1 Introduction

The Aim of Gradual Typing The aim of gradual typing is to enable a "programmer-controlled migration between dynamic and static typing" [50]. That is, given a static type system, the goal is to construct a gradual type system in which existing untyped codebases can be gradually migrated to follow the static typing discipline. To live up to this ambition, a gradual language should be well designed:

1. It should directly support existing untyped and typed programs and not break their well-formedness or well-typedness, or modify their semantics.
2. When the programmer adds type annotations to a part of the program, this should only modify the semantics when necessary: it is okay to throw type errors for wrong programs, but *accurate* type annotations should not change a program's semantics.
3. Finally, a programmer should not have to migrate an entire large codebase before enjoying the benefits of the type system. The type system's benefits (performance, type safety or others) should already apply *locally* in the fully typed parts of the code, no matter how dynamic the surrounding code may be.

If a gradual language provides these guarantees, it offers a viable migration path for large existing dynamically typed codebases. Programmers can flexibly spread the migration over time or even *economically* apply the static typing discipline where it is of most importance, without necessarily migrating the entire codebase.

As an aside, we briefly mention that the dynamic part of a gradual language need not be fully untyped. Given, for instance, a STLC-like language with security types as the static language, one can aim to gradualize only the security aspect of this typing discipline such that the dynamic language corresponds to the STLC [17].

Refined Criteria To formally capture these goals, Siek et al. [52] have put forward a set of refined criteria for a satisfactory gradual language. They

formally define when a gradual type is more precise than another and formalize the above properties for a specific gradual language (the GTLC, see below). They can be summarized in general as follows:

1. **Superset criteria** The gradual language must completely contain the static and dynamic language under consideration. Static/dynamic terms must be appropriately typed in the gradual language and their operational semantics must coincide with those in the static/dynamic language.
2. **The Gradual Guarantee** Weakening type annotations (e.g. by removing some of the annotations) of a *well-typed* term preserves its well-typedness and does not change its operational semantics. Note that starting from a more precise term and weakening its type annotations is a way of characterizing how *correct* type annotations can be added to the less precise resulting term.
3. **Blame theorem** The gradual language satisfies a blame theorem [62], which *implies* that fully-annotated pieces of code are never to blame for a cast error.[1]

While the two former criteria seem to satisfactorily capture the intuitive properties mentioned, the final criterion does so only partially.

The Refined Criteria are Not Enough The Blame theorem implies that the gradual language should preserve some guarantees of the static type system for fully annotated code. However, Garcia and Tanter [18] have recently pointed out that it does so only for those concerning type safety. Note that in this text, we use the words type safety to capture syntactic type safety properties, typically embodied by progress and preservation theorems in the style of Wright and Felleisen [64]. For some type systems this is sufficient, but type systems often provide additional guarantees.

Garcia and Tanter [18] propose that a well-designed gradual language should not just enforce type safety, but type soundness. Type soundness refers to a semantic interpretation of typing judgments $\Gamma \vDash t : \tau$, a property which depends on the type system at hand and can to some extent be chosen by the gradual language designer. For example, they explain how the meaning of STLC types can be taken to include termination or just type safety. For gradualizations of polymorphic types, they explain how type soundness can be

[1] A blame theorem is typically a lot stronger, but as we are not currently interested about the practical implementation of error documentation, this is the crux of it as a language criterion.

taken to include parametricity or not, and in fact, Devriese et al. [11, 12] have explained that existing gradual parametric languages only satisfy a standard form of parametricity that is strictly weaker than the one that Reynolds originally formulated for System F types [46]. For gradualizations of security type systems, Garcia and Tanter [18] explain how the security property of non-interference can be taken as part of the meaning of types.

Choosing the Meaning of Types? Essentially, what Garcia and Tanter [18] propose is that a gradual language should preserve the meaning of types, but leave that meaning open to interpretation. Letting gradual language designers select a meaning of types allows them to choose one they can actually enforce in the gradual language. In the context of gradual parametricity, for example, most people agree that the gradual language should enforce parametricity, an important reasoning principle which holds in static languages like System F. However, parametricity can be defined to mean very different things and it has taken several iterations to find a form of parametricity that gradual parametric languages can actually guarantee [33, 33, 4, 5, 59]. Because of the existence of a universal type, the parametricity offered by these calculi (if any) is defined by a type-world logical relation [11, 12] that is strictly weaker than traditional Reynolds-style parametricity that one may be accustomed to in System F [46]. In the case of security types, the reasoning principle of non-interference [58, 6] can also be defined in different ways (e.g. termination-sensitive or not) and it is not a priori clear which one is right.

As such, Garcia and Tanter's approach lets gradual language designers select a meaning of types that they can enforce and thus offers flexibility. However, this raises the question if it is really enough to enforce only a carefully selected meaning of types, which is strictly weaker than the meaning of types in the static language. For example, can we really claim to have gradualized System F if we only enforce part of the reasoning principles that hold in System F? For programmers, this would mean that embedding existing static programs in the gradual language requires one to ascertain that their correctness, safety or security only relies on those reasoning principles that are preserved in the gradual language. It also means that in the static language, only those optimizations and refactorings should be applied for which the validity follows just from the reasoning principles that are preserved in the gradual language. Restricting the meaning of types to an enforceable subset like this may be unavoidable for some static languages, but in this chapter, we explore a more ambitious approach.

Preserving Reasoning More Generally? What if we were building a gradual language that aims to preserve arbitrary reasoning about static programs?

Devriese et al. [11, 12] have recently proposed a criterion that takes this perspective, at least for reasoning about program equivalences: the fully abstract embedding (FAE) property. The idea is to consider the embedding function from the static language into the gradual language as a translation function and consider whether it satisfies full abstraction, a property proposed by Abadi [1, 2] in the context of secure compilation [42]. This criterion implies that the gradual language should preserve contextual equivalences in the static language. Not a suitably selected set of contextual equivalences, but simply all of them.

If two static terms always behave the same in the static language (i.e. no surrounding static code can distinguish the two), then they should always behave the same when embedded in the gradual language (i.e. *gradual* code should not distinguish the two either). In short, equivalent terms in the static language should be equivalent (as fully annotated terms) in the gradual language.

Another way of interpreting this criterion is that contexts in the gradual language should not have more distinguishing power over static terms than static contexts do and that abstractions and reasoning principles from the static language should remain valid in the gradual language. Practically, it means for example that refactorings and compiler optimizations which are semantics-preserving in the static language are also sound in the gradual extension.

What is in this Chapter? After providing some background in §2.2, we demonstrate the value of the FAE criterion in §2.3 by showing how it successfully detects unsatisfactory gradualizations. In other words, we study three gradual languages where the FAE fails.

Next, in §2.4, we show that the FAE is also realistic, by proving the property for a simple but representative example; the GTLC_μ. Taking inspiration from similar proofs in the context of secure compilation [10, 38, 13], we prove that the embedding of STLC_μ into GTLC_μ is fully abstract. This proof has been mechanically verified using Coq and Iris [28], and can be checked out at `https://github.com/scaup/fae-gtlc-mu`[2].

In §2.5, we elaborate on some of the choices made in earlier sections and lay out some interesting future work. We discuss related work in §2.6 and conclude in §2.7.

Contributions Our central contribution is the study of FAE as a general formal criterion for satisfactory gradual languages, supplementing Siek et al.'s refined

[2]A VM-image preinstalled with the required libraries and the right version of Coq can be downloaded at `https://doi.org/10.5281/zenodo.4071674`

criteria. It captures the intuition that a gradual language should preserve the static language's benefits also when it comes to reasoning. Specifically, we demonstrate the following

- We explain a number of interesting gradual languages where FAE fails, i.e. where the criterion adequately identifies unsatisfactory gradualizations.
- We show that the criterion is realistic, by proving it for the GTLC$_\mu$, with respect to STLC$_\mu$. This proof is mechanically verified using Coq and builds on Iris machinery like guarded recursion for convenient reasoning. This proof is only the second[3] mechanized full abstraction proof in the literature (the first using Iris) and (perhaps surprisingly) one of the first mechanized results about a gradual language.

2.2 Some Background

Before we continue, let us quickly refresh some material from earlier work, specifically the Gradually Typed Lambda Calculus (GTLC) with recursive types (§2.2.1) and fully abstract translations (§2.2.2).

2.2.1 Gradualizing STLC$_\mu$

First, we introduce GTLC$_\mu$: the gradualization of the STLC$_\mu$. We follow the standard literature in doing so [52], remarking where we slightly deviate.

The Statics Gradualizing the STLC$_\mu$ — the simply-typed lambda calculus with sum, product, and (iso-)recursive types — we want to obtain a language in which we can freely navigate the landscape between fully annotated code and fully dynamic code. To this end, we extend the grammar of our types as follows.

$$\tau ::= B \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid X \mid \mu X.\tau \mid \ ?$$

[3]The first mechanized proof is reported on by Devriese et al. [13].

Aside from *some* base type[4], arrow types, sum types, product types, and recursive types, we now have the additional *unknown* type, $?$ and of course all the types containing it.[5]

Semantically, a value of type $?$ can be of *any* static type. Accordingly, we can make the type-annotations in our code as precise or as vague as we desire, navigating toward the static/dynamic end of the spectrum respectively.

In the presence of potentially imprecisely annotated code, the job of type checking in the gradual language is twofold. While it should allow for untyped code (possibly failing upon execution), it should do its best to reject code for which it can *statically* be deduced that it is up to no good. Consequently, type-checked code should retain the safety guarantee as much as is statically possible.

For instance, the term $(\lambda x : \mathbb{N} \times \mathbb{B}. \neg(\pi_2\ x))\ 6$ should not type-check, as we statically see that an integer is not of type $\mathbb{N} \times \mathbb{B}$. Type-checking should only allow the application of a function to an argument if the advertised input type of the function is consistent with the actual type of the argument, i.e. if they are *conceivably* compatible. The type $\mathbb{N} \times \mathbb{B}$ is obviously not consistent with $\mathbb{N}$, nor, for instance, with $? + ?$. It is however consistent with $\mathbb{N} \times \mathbb{B}$, and more interestingly, with $\mathbb{N} \times ?$, $? \times \mathbb{B}$, $? \times ?$, and $?$.

Formally, we have the following typing rule for application.

$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e' : \tau_1' \qquad \tau_1 \sim \tau_1'}{\Gamma \vdash e\ e' : \tau_2}$$

Consistency between types, $\tau \sim \tau'$, is defined in Fig. 2.1. Note that we assume that two types are consistent if they are after possible α-conversion.

The Dynamics The dynamics of the gradual language are defined by translating its terms to a different language, the *cast calculus*. This calculus contains an explicit cast construct, $e : \tau \Rightarrow \tau'$, to cast an expression, e, from one type τ to another (consistent) type τ'. Through the translation of a well-typed gradual term, the implicit applications of consistency are replaced by explicit casts. Translating for instance, $(\lambda x : \mathbb{N}.\, x)\ ((\lambda x : ?.\, x)\ \mathit{True})$, we obtain the

[4]To keep the formalities in §2.4 from being annoyingly verbose, we assume only one base type here (in the Coq development, this is the unit type). When giving examples, we will sometimes assume multiple concrete base types like $\mathbb{N}$ or $\mathbb{B}$ for instance, as it is entirely moral to do so.

[5]We will always consider *closed* types (i.e. types that do not contain any free variables); in STLC$_\mu$, type variables are only meaningful to the extent that they are bound to define a recursive type.

$$? \sim \tau \qquad \tau \sim ? \qquad B \sim B \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \to \tau_2 \sim \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 + \tau_2 \sim \tau_1' + \tau_2'}$$

$$\frac{\tau_1 \sim \tau_1' \qquad \tau_2 \sim \tau_2'}{\tau_1 \times \tau_2 \sim \tau_1' \times \tau_2'} \qquad \frac{\tau \sim \tau'}{\mu X.\, \tau \sim \mu X.\, \tau'} \qquad X \sim X$$

Figure 2.1: The consistency relation between types.

following:

$$(\lambda x : \mathbb{N}.\, x)\ (((\lambda x : ?.\, x)\ (\mathit{True} : \mathbb{B} \Rightarrow ?)) : ? \Rightarrow \mathbb{N})$$

We summarize the cast calculus in Fig. 2.2. Unsurprisingly, the dynamics of the cast calculus mostly coincide with STLC$_\mu$; looking at the expressions, we have constants k; variables x; lambda-abstractions $\lambda x : \tau.\, e$ and applications $e\ e$; pairs (e, e) and projections $\pi_{1,2}\ e$ as constructor/destructors of product types; injections $\texttt{inj}_{1,2}\ e$ and case matches $\texttt{case}\ e\ \texttt{of}\ (e \mid e)$ as constructors/destructor of sum types; and $\texttt{fold}$ and $\texttt{unfold}$ as constructor/destructor of recursive types. The only thing of interest here is the cast construct and the evaluation rules concerning it.

We define a collection of ground types, G, through which we define the dynamics of the casts (to be explained later). Aside from B and $? \to ?$, we also include $? + ?$ and $? \times ?$, following Siek [51]. Lastly, we also add $\mu X.?$ whose values are exactly those that we can unfold at least one time.

Casting a value from a ground type to the unknown forms a value $v : G \Rightarrow ?$, corresponding to the idea that such a cast will always go through, as we are merely losing type information. The same is true for casting a value from one function type to another, $v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2$, as such a cast is suspended (*Factor-App*).

Then, there are a lot of casts which are just decomposed into other casts. When trying to cast down a value of the unknown type to a (non-ground) type τ, we factor out the cast through the corresponding ground type (*Factor-Down*). Conversely, we have (*Factor-Up*). A cast between function types is reduced — upon application of the function to an argument — to the *contravariant* cast on the argument, and a *covariant* one on the result (*Factor-App*). Following Siek [51], a cast between sum types is delegated to the appropriate cast of the two components (*Factor-Sum*), and a cast between product types is delegated to the appropriate casts on each component (*Factor-Prod*). Lastly, a cast of a value between two recursive types recursively goes through its body. Casts between (iso)-recursive types behave thus similar to casts between product and

$$G ::= B \mid ? \to ? \mid ? + ? \mid ? \times ? \mid \mu X.\, ?$$

$$e ::= k \mid x \mid \lambda x : \tau.\, e \mid e\ e \mid (e, e) \mid \pi_1\ e \mid \pi_2\ e \mid \mathtt{inj}_1\ e \mid \mathtt{inj}_2\ e \mid \mathtt{case}\ e\ \mathtt{of}\ (e \mid e) \mid$$
$$\mathtt{fold}\ e \mid \mathtt{unfold}\ e \mid e : \tau \Rightarrow \tau \mid \mathtt{CastError}$$

$$v ::= k \mid \lambda x : \tau.\, e \mid \mathtt{fold}\ v \mid (v, v) \mid \mathtt{inj}_{1,2}\ v \mid v : G \Rightarrow ? \mid v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'$$

$$K ::= [\cdot] \mid K\ e \mid v\ K \mid \pi_1\ K \mid \pi_2\ K \mid \mathtt{inj}_1\ K \mid \mathtt{inj}_2\ K \mid (K, e) \mid (v, K) \mid$$
$$\mathtt{case}\ K\ \mathtt{of}\ (e \mid e) \mid \mathtt{fold}\ K \mid \mathtt{unfold}\ K \mid K : \tau \Rightarrow \tau'$$

$$\mathcal{S} : \mathit{Type} \rightharpoonup \mathit{Type}$$

$$\mathcal{S}(\tau_1 \times \tau_2) \triangleq ? \times ?$$

$$\mathcal{S}(\tau_1 \to \tau_2) \triangleq ? \to ?$$

$$\mathcal{S}(\tau_1 + \tau_2) \triangleq ? + ?$$

$$\mathcal{S}(\mu.\, \tau) \triangleq \mu X.\, ?$$

...

$$v : B \Rightarrow B \to_{\mathsf{h}} v \qquad \textit{Base-Id}$$

$$v : ? \Rightarrow ? \to_{\mathsf{h}} v \qquad \textit{Unknown-Id}$$

$$v : \tau \Rightarrow ? \to_{\mathsf{h}} v : \tau \Rightarrow G \Rightarrow ? \quad \text{if } \mathcal{S}(\tau) = G \neq \tau \qquad \textit{Factor-Up}$$

$$v : ? \Rightarrow \tau \to_{\mathsf{h}} v : ? \Rightarrow G \Rightarrow \tau \quad \text{if } \mathcal{S}(\tau) = G \neq \tau \qquad \textit{Factor-Down}$$

$$v : G \Rightarrow ? \Rightarrow G \to_{\mathsf{h}} v \qquad \textit{Ground-Id}$$

$$v : G_1 \Rightarrow ? \Rightarrow G_2 \to_{\mathsf{h}} \mathtt{CastError} \quad \text{if } G_1 \neq G_2 \qquad \textit{Ground-Fail}$$

$$(v : \tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2')\ w \to_{\mathsf{h}} (v\ (w : \tau_1' \Rightarrow \tau_1)) : \tau_2 \Rightarrow \tau_2' \qquad \textit{Factor-App}$$

$$(\mathtt{inj}_i\ v) : \tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2' \to_{\mathsf{h}} \mathtt{inj}_i\ (v_i : \tau_i \Rightarrow \tau_i') \quad \text{for } i = 1, 2 \qquad \textit{Factor-Sum}$$

$$(v_1, v_2) : \tau_1 \times \tau_2 \Rightarrow \tau_1' \times \tau_2' \to_{\mathsf{h}} (v_1 : \tau_1 \Rightarrow \tau_1',\ v_2 : \tau_2 \Rightarrow \tau_2') \qquad \textit{Factor-Prod}$$

$$(\mathtt{fold}\ v) : \mu X.\, \tau \Rightarrow \mu Y.\, \tau' \to_{\mathsf{h}} \mathtt{fold}\ (v : \tau[X \mapsto \mu X.\, \tau] \Rightarrow \tau'[Y \mapsto \mu Y.\, \tau']) \qquad \textit{Unfold-Rec}$$

$$K[e] \to K[e'] \quad \text{if } e \to_{\mathsf{h}} e' \qquad \textit{Congruence}$$

$$K[\mathtt{CastError}] \to \mathtt{CastError} \quad \text{if } K \neq [\cdot] \qquad \textit{Report-CastError}$$

... $\qquad \dfrac{\Gamma \vdash e : \tau \qquad \tau \sim \tau'}{\Gamma \vdash (e : \tau \Rightarrow \tau') : \tau'} \qquad \Gamma \vdash \mathtt{CastError} : \tau$

Figure 2.2: Cast calculus of GTLC$_\mu$

sum types; they are all evaluated eagerly. This is in contrast to casts between function types, which are suspended until application. The latter is unavoidable however; *strictly* evaluating a cast between two function types requires checking if the final cast is respected for *any* well-typed input.

In the remaining evaluation rules, casts actually fail or succeed. In *Base-Id*, *Unknown-Id* and *Ground-Id*, a cast trivially goes through. It must fail in *Ground-Fail* however, as one tries to cast a value from one ground type to a different one.

In contrast to Siek et al. [52], we do not implement blame, as we do not need it for our purposes (see §2.5). Instead, a failing cast — following Siek and Taha [50] — evaluates to the stuck term `CastError` (*Ground-Fail*, *Report-CastError*).

2.2.2 Fully Abstract Compilation

As we have seen in §1.2.2, fully abstract compilation [1, 2, 42] is generally defined as follows.[6]

Definition 2.2.1 (Fully abstract compilation)**.** Given a compiler, say $[\![_]\!] : \mathcal{L}_s \to \mathcal{L}_t$, and equivalence relations in the source and target language, say $\simeq$ and $\simeq$, we have fully abstract compilation if the compiler both preserves and reflects said equivalences. That is, if the following holds.

$$\forall e_1, e_2, e_1 \simeq e_2 \text{ if and only if } [\![e_1]\!] \simeq [\![e_2]\!]$$

The equivalence relations at stake here relate terms that behave similarly under any possible interaction. Take, for instance the well-typed terms $\cdot \vdash 3 + 3 : \mathbb{N}$ and $\cdot \vdash \pi_2\ (\mathsf{True}, 6) : \mathbb{N}$ in the STLC. These two terms are equivalent, as no well-typed surrounding program can ever distinguish the two. This is formalized as *contextual equivalence*.

Definition 2.2.2 (Contextual equivalence in STLC_μ)**.** Two terms, e_1 and e_2, are contextually equivalent at a context Γ and type τ, written $\Gamma \vdash e_1 \simeq_{ctx} e_2$, iff $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$ and for every context, $C : (\Gamma; \tau) \Rightarrow (\cdot\,; \mathbb{1})$, we have that the following holds.[7]

$$C[e_1] \Downarrow \text{ if and only if } C[e_2] \Downarrow$$

[6] We adhere to the same convention that we introduced in Chapter 1: expressions, types, and relations with respect to static/source languages will be formatted in blue; those related with target/gradual languages will be formatted in red.

[7] Here, and throughout the rest of this chapter, we denote $e \Downarrow$ to mean, $\exists v.\, e \to^* v$.

A surrounding program is formalized by a context; a programs with a hole. Any context C can be plugged in with a term e such that the program $C[e]$ is obtained. A context C has type $(\Gamma;\tau) \Rightarrow (\Gamma';\tau')$ if when plugged in with a well-typed term $\Gamma \vdash e : \tau$, we obtain the well-typed term $\Gamma' \vdash C[e] : \tau'$.

Notice that the definition is stronger than it appears. For example, if e_1 and e_2 are of boolean type, contextual equivalence between the two implies that if one of them reduces to a boolean value, the other one must reduce to the *same* value. This can be shown using the following context in which $\Omega_{\mathbb{1}}$ a diverging term of type $\mathbb{1}$.

$$\mathit{if}\ [_]\ \mathit{then}\ \mathrm{unit}\ \mathit{else}\ \Omega_{\mathbb{1}}$$

To formalize contextual equivalence in GTLC_μ, we chose an analogous definition by equi-termination.

Definition 2.2.3 (Contextual equivalence in Gradual Language)**.** Two terms, e_1 and e_2, in GTLC_μ are equivalent, written $e_1 \simeq_{ctx} e_2$, iff $\Gamma \vdash e_1, e_2 : \tau$ and for all gradual contexts $C : (\Gamma;\tau) \Rightarrow (\cdot\,;\mathbb{1})$, we have $C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$.

Notice that this definition does not distinguish between divergence and runtime errors. In §2.5.1, we motivate this choice and discuss the alternatives in detail.

2.3 Gradual Languages Without an FAE

In this section, we touch upon three gradualizations from the literature that fail the FAE criterion. Two of these are the result of a straightforward but naive application of the AGT framework [16]. AGT (Abstracting Gradual Typing) is a general framework that uses abstract interpretation of types to gradualize an arbitrary static language. It requires only a static language, a syntax for the gradual types, and a Galois connection between sets of static types (ordered by the subset relation) and gradual types (ordered by the precision relation). This connection defines the set of static types that represents a gradual type and the most precise gradual type that represents a given set of static types. Knowledge of AGT is not a prerequisite for this section however.

The AGT framework ensures that any resulting gradual language satisfies — by construction — the refined criteria. Unfortunately, the derived dynamics of the gradual language are only designed to preserve type safety. As a result, the gradualization often fails to preserve the guarantees of the static typing discipline that go beyond mere type safety [18].

So far, problems like these have only been described in terms of the failure of properties that are specific to the static language at hand. In this section, we demonstrate that they are detected through failure of the general FAE criterion.

2.3.1 Gradualising a Parametric Language, Naively

Toro et al. [59] define their static language SF, which is essentially a representation of System F that makes it more amenable to applying AGT-recipe. Because of parametricity, we have, for instance, the following contextual equivalence in this language:

$$\lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0 \simeq_{ctx} \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5$$

Indeed, from parametricity [46] we know that instantiating a function f of type $\forall X.\, X \to \mathbb{Z}$ with the type $\mathbb{Z}$ (or any type for that matter) and applying it to a value, will always return the same integer. Intuitively, this is because the language does not allow f to detect which type it is being applied to and therefore, it must behave the same for any type.

If the FAE criterion holds for a gradualization of SF, the same equivalence must hold after embedding:

$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0\rceil\rceil \simeq^{?}_{ctx} \lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5\rceil\rceil$$

When Toro et al. [59] first applied AGT to SF naively, they used a standard representation of evidence (a concept introduced in AGT with which the dynamics of the gradualization is derived) as just a pair of gradual types. However, the resulting gradualization broke parametricity and the above equivalence. In their calculus, they can construct a function of type $\forall X.\, X \to \mathbb{Z}$ that acts as the successor function when instantiated with $\mathbb{Z}$, thereby demonstrating the lack of parametricity [59, §7.1]:

$$\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x$$

This function can be used to discriminate the two terms we saw before:

$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 0\rceil\rceil\ (\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x) \to^{*} 1$$

$$\lceil\lceil \lambda f : \forall X.\, X \to \mathbb{Z}.\, f[\mathbb{Z}]\ 5\rceil\rceil\ (\Lambda X.\, \lambda x : X.\, (\lambda y : ?.\, (\lambda z : ?.\, z + 1)\ y)\ x) \to^{*} 6$$

In other words, although the FAE criterion is not in any way tied to System F, the criterion accurately detects the failure of parametricity in this naive gradualization.[8]

2.3.2 Gradualizing a Parametric Language, Non-naively

In the last paragraph, we have examined an obviously naive gradualization of a parametric language. However, even less naive gradual languages are not without problems. Devriese et al. [11, 12] have pointed out that even gradual polymorphic languages that come with a proof of parametricity [59, 5, 39] break certain equivalences that hold in System F.

The problem is demonstrated using the type Univ:

$$\text{Univ} = \exists Y.\, \forall X.\, (X \to Y) \times (Y \to X)$$

In a sense, this type expresses the existence of a universal type: a single type Y which every other type X can be embedded into and extracted from. Interestingly, Devriese et al. [12] demonstrate that standard Reynolds parametricity implies degeneracy of this type: instantiating X with an arbitrary type, embedding a value of X into Y and then extracting it again, must necessarily diverge.

This property can be used to show contextual equivalence of the following two terms [12]:

$$e_1 = \lambda x : \text{Univ}.\, \text{unpack } x \text{ as } Y, x' \text{ in}$$
$$\quad \text{let } x'' : (\mathbb{1} \to Y) \times (Y \to \mathbb{1}) = x'\ [\mathbb{1}] \text{ in } (\pi_2\ x'')\ ((\pi_1\ x'')\ \ \text{unit})$$
$$e_2 = \lambda x : \text{Univ}.\, \Omega_{\mathbb{1}} \quad \textit{(where } \Omega_{\mathbb{1}} \textit{ a diverging term of type } \mathbb{1}\textit{)}$$

Of course, gradual parametric languages contain the gradual type ? and indeed, the equivalence of $\lceil\lceil e_1 \rceil\rceil$ and $\lceil\lceil e_2 \rceil\rceil$ no longer holds because of this. It is not that hard to construct a context that uses ? to construct a non-degenerate value of type $\lceil\lceil \text{Univ} \rceil\rceil$ and use it to break the equivalence above:

$$[\cdot]\ \ (\text{pack } ?, \Lambda X.\, \langle \lambda x : X.\, x : X \Rightarrow ?, \lambda x : ?.\, x : ? \Rightarrow X \rangle \text{ as Univ})$$

What this means is that the parametricity offered by gradual parametric calculi is a weaker form of parametricity than the one that holds in System F. Because

[8] FAE also fails here because of partiality (divergence/errors) present in the gradual language, but this is orthogonal to the issue here.

of this difference, not all equivalences that hold in System F are preserved in these languages. The FAE criterion picks up this difference in the kind of parametricity that is offered by the language. The fact that not all of System F's equivalences continue to hold in a gradual parametric calculus may be important for users to be aware of. As such, we think one should be careful to claim that these calculi are gradualizations of System F, but instead regard them as gradualizations of a variant of System F with a universal type.

2.3.3 Gradualizing an Information-flow Language with References

Toro et al. [58] set out to gradualize the security aspect in the typing discipline of a non-trivial information-flow language with references. They first define their static language $\mathrm{SSL}_{\mathrm{ref}}$, a statically-typed language with references in which types and terms are indexed by a secrecy label in a lattice. A value of type τ_ℓ corresponds to a value of type τ that is indexed by a particular secrecy level ℓ. The static typing discipline then ensures that in a well-typed program, computations of some value at a certain secrecy level are never influenced by data coming from a higher secrecy level. In other words, confidential data at a level ℓ can never leak — directly or indirectly — to a level $\ell' \preccurlyeq \ell$. This general property is referred to as "non-interference".

We have (given $L \preccurlyeq H$) the following contextual equivalence in this language.

$$\lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L . \lambda^H b : \mathbb{B}_H . (f\ b) \simeq_{ctx} \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L . \lambda^H b : \mathbb{B}_H . (f\ \neg b)$$

Indeed, by non-interference, a function of type $\mathbb{B}_H \xrightarrow{H} \mathbb{B}_L$ cannot actually depend upon its argument. Therefore, we can safely negate its argument for instance.

Like before, in §2.3.1, naively applying AGT (with evidence represented by pairs of labels) gives us a language in which this is not the case anymore. Toro et al. [58] show that this language cannot hope to satisfy non-interference by giving the following example:

$$\mathtt{true}_? :: \mathbb{B}_H :: \mathbb{B}_? :: \mathbb{B}_L \rightarrow^* \mathtt{true}_L$$

From this counterexample, it is again not hard to construct a gradual context C demonstrating that the aforementioned equivalence is not preserved:

$$C \triangleq [\cdot]\ g\ t \text{ where}$$

$$g \triangleq \lambda^H b : \mathbb{B}_H . b :: \mathbb{B}_? :: \mathbb{B}_L$$

$$t \triangleq \mathtt{true}_? :: \mathbb{B}_H$$

Indeed, we now have the following:

$$\lceil\lceil \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L . \lambda^H b : \mathbb{B}_H . (f\ b) \rceil\rceil\ g\ t \to^* \mathtt{true}_L$$

$$\lceil\lceil \lambda^H f : \mathbb{B}_H \xrightarrow{H} \mathbb{B}_L . \lambda^H b : \mathbb{B}_H . (f\ \neg b) \rceil\rceil\ g\ t \to^* \mathtt{false}_L$$

2.4 FAE for the GTLC_μ

Now that we have demonstrated that the FAE criterion can successfully detect gradual languages that are unsatisfactory in the sense that they do not preserve static equivalences in the gradualization, the question remains whether the criterion is not too ambitious. Can we really hope for gradual languages that preserve arbitrary static equivalences? In this section, we show that an FAE may not be too much to hope for, by proving the property in at least one simple but representative setting. We report on our computer-verified proof[9] that it is indeed satisfied by the GTLC_μ for source language STLC_μ, laid out in §2.2.1.

In §2.4.1, we present a high-level overview of the proof, introducing the concept of a *backtranslation* and its role in the proof. We present our backtranslation in §2.4.2, and we prove its validity in §2.4.3.

2.4.1 High-Level Overview

Proving Reflection of Contextual Equivalences Proving the reflection of contextual equivalences is the less interesting part of this proof, as it directly follows from the superset criterion. Below, we provide a quick proof.

Given two arbitrary static terms, say e_1 and e_2, whose embeddings are equivalent in the gradual language, we need to prove that they are equivalent in the static language. Thus given an arbitrary static context, say $\mathcal{C}$, we need to prove that $\mathcal{C}[e_1] \Downarrow$ implies $\mathcal{C}[e_2] \Downarrow$ (the reverse direction is symmetric). As depicted in Fig. 2.3, this follows directly from the superset criterion (1,3) and the given contextual equivalence in the gradual language, instantiated with $\lceil\lceil \mathcal{C} \rceil\rceil$ (2).

In §2.4.3, we briefly mention how the same can be proven without appeal to the superset criterion.

Proving Preservation of Contextual Equivalences Intuitively, we need to prove somehow that gradual contexts do not possess more distinguishing power

[9] It can be checked out at `https://github.com/scaup/fae-gtlc-mu`.

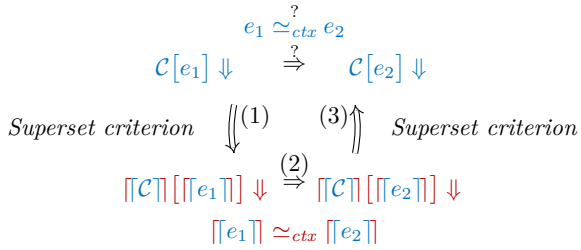


Figure 2.3: Reflection of equivalences: overview proof. Structure of diagram taken from Devriese et al. [10].

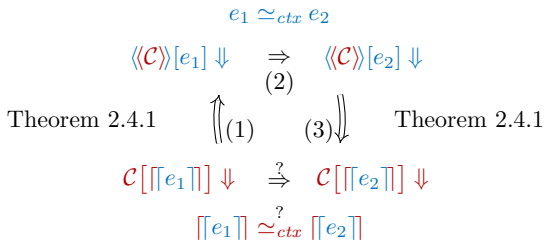


Figure 2.4: Preservation of equivalences: overview proof. Structure of diagram taken from Devriese et al. [10].

over fully annotated terms than static contexts do. One way to do this is to *emulate* the behavior of a *gradual context* by that of a *static context*. Formally, we would have an emulation function (following the literature on fully abstract compilation proofs [42], we also refer to this function as a "backtranslation"), $\langle\!\langle _ \rangle\!\rangle : \mathcal{C} \mapsto \langle\!\langle \mathcal{C} \rangle\!\rangle$, which maps gradual contexts to their appropriate emulations in the static language. With this in mind, we have the following proof sketch.

Given two equivalent static terms, say $e_1 \simeq_{ctx} e_2$, we need to prove that $\lceil\lceil e_1\rceil\rceil$ and $\lceil\lceil e_2\rceil\rceil$ are equivalent in the gradual language. That is, given an arbitrary gradual context, say $\mathcal{C}$, we need to prove that $\mathcal{C}[\lceil\lceil e_1\rceil\rceil] \Downarrow$ implies $\mathcal{C}[\lceil\lceil e_2\rceil\rceil] \Downarrow$ (the reverse direction is again symmetric). As depicted in Fig. 2.4, this follows if we can construct a valid emulation function (1,3), together with the given contextual equivalence in the static language, instantiated with $\langle\!\langle\mathcal{C}\rangle\!\rangle$ (2).

What it means for our emulation function to be valid, is easily extracted from the requirements of the proof; we have the following specification.

Theorem 2.4.1 (Specification for the backtranslation)**.** *Given arbitrary well-typed static term, $\Gamma \vdash e : \tau$, and arbitrary target context, $C : (\lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot\,; \mathbb{1})$,*

$$\langle\!\langle k \rangle\!\rangle \triangleq k \qquad \langle\!\langle (e_1, e_2) \rangle\!\rangle \triangleq (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle)$$

$$\langle\!\langle x \rangle\!\rangle \triangleq x \qquad \langle\!\langle \mathtt{inj}_1\ e \rangle\!\rangle \triangleq \mathtt{inj}_1\ \langle\!\langle e \rangle\!\rangle$$

$$\langle\!\langle \lambda x : \tau.\, e \rangle\!\rangle \triangleq \lambda x : \langle\!\langle \tau \rangle\!\rangle.\, \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{inj}_2\ e \rangle\!\rangle \triangleq \mathtt{inj}_2\ \langle\!\langle e \rangle\!\rangle$$

$$\langle\!\langle e_1\ e_2 \rangle\!\rangle \triangleq \langle\!\langle e_1 \rangle\!\rangle\ \langle\!\langle e_2 \rangle\!\rangle \qquad \langle\!\langle \mathtt{case}\ e\ \mathtt{of}\ (e_1 \mid e_2) \rangle\!\rangle \triangleq \mathtt{case}\ \langle\!\langle e \rangle\!\rangle\ \mathtt{of}\ (\langle\!\langle e_1 \rangle\!\rangle \mid \langle\!\langle e_2 \rangle\!\rangle)$$

$$\langle\!\langle \pi_1\ e \rangle\!\rangle \triangleq \pi_1\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{fold}\ e \rangle\!\rangle \triangleq \mathtt{fold}\ \langle\!\langle e \rangle\!\rangle$$

$$\langle\!\langle \pi_2\ e \rangle\!\rangle \triangleq \pi_2\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathtt{unfold}\ e \rangle\!\rangle \triangleq \mathtt{unfold}\ \langle\!\langle e \rangle\!\rangle$$

Figure 2.5: The uninteresting parts of the backtranslation on expressions.

we must have that

$$\langle\!\langle C \rangle\!\rangle : (\Gamma;\ \tau) \Rightarrow (\cdot\ ;\mathbb{1}) \quad \textit{and} \quad \langle\!\langle C \rangle\!\rangle[e] \Downarrow \quad \textit{iff} \quad C\left[\lceil\lceil e \rceil\rceil\right] \Downarrow.$$

Exactly defining our backtranslation and proving that it satisfies Theorem 2.4.1 will be the subject of §2.4.2 and §2.4.3 respectively.

2.4.2 Backtranslation

As seen in §2.2.1, the dynamics of GTLC_μ are defined by a translation to the cast calculus, in which non-trivial consistencies are interspersed with explicit casts. We define our backtranslation directly on this cast calculus; this turns out to be much easier, despite the fact that the cast calculus allows for much more contexts. Throughout this section, we define the backtranslation only on *terms*, but it naturally extends to contexts.

The Easy Part Structurally, the cast calculus is just the static language extended with the cast construct. Much of the backtranslation is therefore quite uninteresting, as can be seen in Fig. 2.5.

Naturally, these parts of the backtranslating preserve well-typedness; a fully annotated term (i.e. a term with no casts) of type τ is backtranslated to a term of type $\langle\!\langle \tau \rangle\!\rangle$, where the uninteresting parts of the backtranslation on types can be seen in Fig. 2.6. We quickly note also that embedding terms from the static language into the gradual one and backtranslating them forms a retraction pair; we have that $\langle\!\langle \lceil\lceil _ \rceil\rceil \rangle\!\rangle$ is the identity function.

$$\langle\!\langle B \rangle\!\rangle \triangleq B$$

$$\langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle$$

$$\langle\!\langle \tau_1 + \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle$$

$$\langle\!\langle \tau_1 \times \tau_2 \rangle\!\rangle \triangleq \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle$$

$$\langle\!\langle X \rangle\!\rangle \triangleq X$$

$$\langle\!\langle \mu X.\, \tau \rangle\!\rangle \triangleq \mu X.\, \langle\!\langle \tau \rangle\!\rangle$$

Figure 2.6: The uninteresting parts of the backtranslation on types.

The Universe Of course, things get more interesting when we look at non-static gradual expressions. To begin with, we require a static type that can accommodate expressions of, for instance, the unknown type for which — semantically — expressions can have *any* static type. Taking inspiration from New et al. [38] and Devriese et al. [10], we will backtranslate expressions of the unknown type to expressions of the following static type.

$$\langle\!\langle ? \rangle\!\rangle \triangleq \mu X.\, (B + (X \to X) + (X + X) + (X \times X) + X) \tag{2.1}$$

This type is often referred to as a universal type of the STLC_μ, as we can embed every static type in it. Throughout this section, we denote it by $\mathscr{U}$. The above completes the backtranslation on types given in Fig. 2.6.

For convenience, we will use a variant-like syntax with labels (even though technically, we only have binary sum types):

$$\langle (B : B); (\to\!: (\mathscr{U} \to \mathscr{U})); (+ : (\mathscr{U} + \mathscr{U})); (\times : (\mathscr{U} \times \mathscr{U})); (\mu : \mathscr{U}) \rangle$$

This syntax allows us to use constructors $\mathtt{inj}_B$, $\mathtt{inj}_\to$, $\mathtt{inj}_\times$, and $\mathtt{inj}_\mu$, as well as the following generalized case match.

$$\mathtt{case}\ e\ \mathtt{of}\ (\lambda x : B.\, e_1) \mid (\lambda x : \mathscr{U} + \mathscr{U}.\, e_2) \mid$$
$$(\lambda x : \mathscr{U} \times \mathscr{U}.\, e_3) \mid (\lambda x : \mathscr{U} \to \mathscr{U}.\, e_4) \mid (\lambda x : \mathscr{U}.\, e_5)$$

Moreover, we shall further abbreviate this if in all but one of the branches we have the same function body (that is independent of the input argument). The expression below for instance, shall be abbreviated to

$$\texttt{case}\ e\ \texttt{of}\ (\lambda x : \mathscr{U} + \mathscr{U}\,.\,e'') \mid (\lambda _ : _\,.\,e').$$

$$\texttt{case}\ e\ \texttt{of}\ (\lambda _ : B\,.\,e') \mid (\lambda x : \mathscr{U} + \mathscr{U}\,.\,e'') \mid$$

$$(\lambda _ : \mathscr{U} \times \mathscr{U}\,.\,e') \mid (\lambda _ : \mathscr{U} \to \mathscr{U}\,.\,e') \mid (\lambda _ : \mathscr{U}\,.\,e')$$

Backtranslating Casts to Functions The only non-static construct in our gradual language is the cast construct. To complete the backtranslation on terms from Fig. 2.5, it remains to define the backtranslation of that construct. We will backtranslate a cast of a term e from τ to τ' to the application of a function, $\mathscr{F}_{\tau \Rightarrow \tau'}$, to be defined later:

$$\langle\!\langle e : \tau \Rightarrow \tau' \rangle\!\rangle \triangleq \mathscr{F}_{\tau \Rightarrow \tau'}\ \langle\!\langle e \rangle\!\rangle \tag{2.2}$$

To ensure well-typedness of our backtranslation, this function $\mathscr{F}_{\tau \Rightarrow \tau'}$ must be of type $\langle\!\langle \tau \rangle\!\rangle \to \langle\!\langle \tau' \rangle\!\rangle$.

Some Simple Examples Let us look at some simple examples.

Consider some value of base type, $v : B$. Given $\langle\!\langle v \rangle\!\rangle : B$, we backtranslate $v : B \Rightarrow ?$ by injecting $\langle\!\langle v \rangle\!\rangle$ into the universe:

$$\langle\!\langle v : B \Rightarrow ? \rangle\!\rangle = (\lambda u : B.\ \texttt{fold}\ (\texttt{inj}_B\ u))\ \langle\!\langle v \rangle\!\rangle$$

Note that the application of this function always succeeds; it will always terminate to a value of type $\mathscr{U}$, reflecting the fact that $v : B \Rightarrow ?$ is a value of type $?$.

Conversely though, casting *down* a value of the unknown type, say $v : ?$, to base type B should only go through if the unknown value was "of the right form", i.e. $v = (b : B \Rightarrow ?)$. Accordingly, we backtranslate $v : ? \Rightarrow B$ as follows:

$$\langle\!\langle v : ? \Rightarrow B \rangle\!\rangle = (\lambda u : \mathscr{U}\,.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : B\,.\,x) \mid (\lambda x : _\,.\ \Omega_B))\ \langle\!\langle v \rangle\!\rangle$$

If v was of the form $b : B \Rightarrow ?$, i.e. the evaluation of $\langle\!\langle v \rangle\!\rangle$ of the form $\texttt{fold}\ (\texttt{inj}_B\ b)$, then the application will evaluate to b. If not, it will evaluate to a diverging term Ω_B of type B, implemented using recursive types.

In general, we have for any ground type G a function $\mathit{inject}_G : \langle\!\langle G \rangle\!\rangle \to \mathscr{U}$ and a function $\mathit{extract}_G : \mathscr{U} \to \langle\!\langle G \rangle\!\rangle$, defined in Fig. 2.7. They all follow — more or less — the same pattern. For the case of the ground type $\mu X.\,?$, we do have a small caveat as we have to put an extra $\texttt{unfold}$ /$\texttt{fold}$ into place.

$$\mathit{inject}_B \triangleq \lambda u : B.\ \texttt{fold}\ (\texttt{inj}_B\ u) : B \to \mathscr{U}$$

$$\mathit{inject}_{?+?} \triangleq \lambda u : \mathscr{U} + \mathscr{U}.\ \texttt{fold}\ (\texttt{inj}_+\ u) : \mathscr{U} + \mathscr{U} \to \mathscr{U}$$

$$\mathit{inject}_{?\times ?} \triangleq \lambda u : \mathscr{U} \times \mathscr{U}.\ \texttt{fold}\ (\texttt{inj}_\times\ u) : \mathscr{U} \times \mathscr{U} \to \mathscr{U}$$

$$\mathit{inject}_{?\to ?} \triangleq \lambda u : \mathscr{U} \to \mathscr{U}.\ \texttt{fold}\ (\texttt{inj}_\to\ u) : (\mathscr{U} \to \mathscr{U}) \to \mathscr{U}$$

$$\mathit{inject}_{\mu X.?} \triangleq \lambda u : \mu X.\mathscr{U}.\ \texttt{fold}\ (\texttt{inj}_\mu\ (\texttt{unfold}\ u)) : \mu X.\mathscr{U} \to \mathscr{U}$$

$$\mathit{extract}_B \triangleq \lambda u : \mathscr{U}.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : B.\, x) \mid (\lambda x : _.\ \Omega_B) : \mathscr{U} \to B$$

$$\mathit{extract}_{?+?} \triangleq \lambda u : \mathscr{U}.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : \mathscr{U} + \mathscr{U}.\, x) \mid (\lambda x : _.\ \Omega_{\mathscr{U}+\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} + \mathscr{U})$$

$$\mathit{extract}_{?\times ?} \triangleq \lambda u : \mathscr{U}.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : \mathscr{U} \times \mathscr{U}.\, x) \mid (\lambda x : _.\ \Omega_{\mathscr{U}\times\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} \times \mathscr{U})$$

$$\mathit{extract}_{?\to ?} \triangleq \lambda u : \mathscr{U}.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : \mathscr{U} \to \mathscr{U}.\, x) \mid (\lambda x : _.\ \Omega_{\mathscr{U}\to\mathscr{U}}) : \mathscr{U} \to (\mathscr{U} \to \mathscr{U})$$

$$\mathit{extract}_{\mu X.?} \triangleq \lambda u : \mathscr{U}.\ \texttt{case}\ (\texttt{unfold}\ u)\ \texttt{of}\ (\lambda x : \mathscr{U}.\texttt{fold}\ x) \mid (\lambda x : _.\ \Omega_{\mu X.\mathscr{U}}) : \mathscr{U} \to \mu X.\mathscr{U}$$

Figure 2.7: Defining the injection and extraction functions for every ground type

$$B \rightsquigarrow B \qquad ? \rightsquigarrow ? \qquad ? \rightsquigarrow G \qquad G \rightsquigarrow ?$$

$$\frac{\tau \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad G \rightsquigarrow ?}{\tau \rightsquigarrow ?} \qquad \frac{? \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad G \rightsquigarrow \tau}{? \rightsquigarrow \tau}$$

$$\frac{\tau_1' \rightsquigarrow \tau_1 \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'} \qquad \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{\tau_1 + \tau_2 \rightsquigarrow \tau_1' + \tau_2'}$$

$$\frac{\tau[X \mapsto \mu X.\, \tau] \rightsquigarrow \tau'[Y \mapsto \mu Y.\, \tau']}{\mu X.\, \tau \rightsquigarrow \mu Y.\, \tau'}$$

Figure 2.8: Decomposition of casts in terms of each other.

We remark that the extract and inject functions satisfy the specification that arises from the evaluation rule *Ground-Id*, as composing $\mathit{extract}_G$ with inject_G will behave as the identity function.

Composing $\mathit{extract}_{G_1}$ with inject_{G_2} where $G_1 \neq G_2$, will behave as $\lambda x : \langle\!\langle G_2 \rangle\!\rangle . \Omega_{\langle\!\langle G_1 \rangle\!\rangle}$, in accordance to the *Ground-Fail* evaluation rule. Indeed, we are backtranslating a cast error to a diverging term.

$$\langle\!\langle \texttt{CastError} : \tau \rangle\!\rangle \triangleq \Omega_{\langle\!\langle \tau \rangle\!\rangle}$$

Casts that Decompose Towards a general definition of the backtranslation for the casts, we want to mimic the cast calculus as faithful as possible, as doing so will make it easier to prove that it satisfies Theorem 2.4.1. More precisely, we want our backtranslation to reflect the manner in which casts are decomposed in terms of each other. We summarize this structure (the decompositions of casts) in a binary relation on types, depicted in Fig. 2.8; *derivations* of this relation reflect the way in which casts are decomposed in terms of each other.

As a first step, we now start defining the backtranslation by induction on this relation, after which we will tweak the relation slightly to accommodate the backtranslation of casts between recursive types.

The atomic casts are backtranslated easily enough. We backtranslate $G \Rightarrow ?$ to inject_G and $? \Rightarrow G$ to $\mathit{extract}_G$ from the previous section, reflecting both *Ground-Id* and *Ground-Fail*. The casts $B \Rightarrow B$ and $? \Rightarrow ?$ are trivially backtranslated to $\lambda x : B.\ x$ and $\lambda x : \mathscr{U}.\ x$ respectively, mirroring *Base-Id* and *Unknown-Id*.

We will consider the compositional casts now. Given $f_1 : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_1' \rangle\!\rangle$, $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1 \Rightarrow \tau_1'$, $\tau_2 \Rightarrow \tau_2'$ respectively, we can backtranslate $\tau_1 + \tau_2 \Rightarrow \tau_1' + \tau_2'$, mirroring *Factor-Sum*:

$$\lambda s : \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle .\, \texttt{case}\ s\ \texttt{of}\ \lambda x_1 : \langle\!\langle \tau_1 \rangle\!\rangle .\, \texttt{inj}_1\ (f_1\ x_1)\ \mid\ \lambda x_2 : \langle\!\langle \tau_2 \rangle\!\rangle .\, \texttt{inj}_2\ (f_2\ x_2)$$

Similarly, given $f_1 : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_1' \rangle\!\rangle$, $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1 \Rightarrow \tau_1'$, $\tau_2 \Rightarrow \tau_2'$ respectively, we backtranslate $\tau_1 \times \tau_2 \Rightarrow \tau_1' \times \tau_2'$, mirroring *Factor-Prod*.

$$\lambda p : \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle . \, (f_1 \ (\pi_1 \ p), \ f_2 \ (\pi_2 \ p))$$

Given $f_1 : \langle\!\langle \tau_1' \rangle\!\rangle \to \langle\!\langle \tau_1 \rangle\!\rangle$ and $f_2 : \langle\!\langle \tau_2 \rangle\!\rangle \to \langle\!\langle \tau_2' \rangle\!\rangle$, backtranslations for $\tau_1' \Rightarrow \tau_1$ and $\tau_2 \Rightarrow \tau_2'$ respectively, we mirror the backtranslation of $\tau_1 \to \tau_2 \Rightarrow \tau_1' \to \tau_2'$ according to *Factor-App*.

$$\lambda f : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle . \ \lambda x : \langle\!\langle \tau_1' \rangle\!\rangle . \ f_2 \ (f \ (f_1 \ x))$$

This reflects the suspended nature of the cast as f_1 will only be applied on the argument when it is actually supplied, after which f_2 is applied on the result (after applying f).

Unfortunately, the relation as is does not allow for a valid definition of the backtranslation; delegating a cast between two recursive types, say $\mu X. \tau \Rightarrow \mu Y. \tau'$ to $\tau[X \mapsto \mu X. \tau] \Rightarrow \tau'.[Y \mapsto \mu Y. \tau']$, would clearly make the definition cyclic. Naively backtranslating, for instance, the cast $\mu X. ? \times X \Rightarrow \mu X. \mathbb{N} \times X$, we would obtain the following infinite definition:

$$\lambda x : \mu X. (\mathscr{U} \times X) \, . \texttt{fold}$$

$$\big((\lambda p : \mathscr{U} \times \mu X. (\mathscr{U} \times X) \, . (\mathit{extract}_{\mathbb{N}} \ (\pi_1 \ p) \, , \ f \ (\pi_2 \ p))) \ (\texttt{unfold} \ x) \big)$$

$$\textit{where} \quad f \triangleq \lambda x : \mu X. (\mathscr{U} \times X). \ldots$$

To solve this, we simply implement a fixpoint operator with recursive types that we then use to expose a recursive call, i, that we make use of accordingly:

$$\lambda y : \mu X. (\mathscr{U} \times X). \overrightarrow{fix} \big(\lambda i : \mu X. (\mathscr{U} \times X) \to \mu Y. (\mathbb{N} \times Y). \big(\lambda x : \mu X. (\mathscr{U} \times X). \big)$$

$$\texttt{fold} \ ((\lambda p : \mathscr{U} \times \mu X. (\mathscr{U} \times X). (\mathit{extract}_{\mathbb{N}} \ (\pi_1 \ p), \ i \ (\pi_2 \ p))) \ (\texttt{unfold} \ x)))) \ y$$

The General Definition To accommodate our last example, we slightly tweak the relation in Fig. 2.8, obtaining the one depicted in Fig. 2.9. It is by induction on this relation that we establish the general definition of the backtranslation on casts. In it, every judgment is now accompanied by a partial function mapping static variables to a pair of gradual types. In the backtranslation, this partial function will encode the variables in scope that correspond to a recursive call. A variable, say f, for which $F(f) = \mu X. \tau \rightsquigarrow \mu Y. \tau'$, corresponds to a recursive call to a function of type $\langle\!\langle \mu X. \tau \rangle\!\rangle \to \langle\!\langle \mu Y. \tau' \rangle\!\rangle$.

$$\textsc{Atomic-Base}\quad F \vdash_s B \rightsquigarrow B$$

$$\textsc{Atomic-Unknown}\quad F \vdash_s ? \rightsquigarrow ?$$

$$\textsc{Atomic-Unknown-G}\quad F \vdash_s ? \rightsquigarrow G$$

$$\textsc{Atomic-G-Unknown}\quad F \vdash_s G \rightsquigarrow ?$$

$$\textsc{FactorUp-G}\quad \frac{F \vdash_s \tau \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad F \vdash_s G \rightsquigarrow ?}{F \vdash_s \tau \rightsquigarrow ?}$$

$$\textsc{FactorDown-G}\quad \frac{F \vdash_s ? \rightsquigarrow G \qquad \mathcal{S}(\tau) = G \neq \tau \qquad F \vdash_s G \rightsquigarrow \tau}{F \vdash_s ? \rightsquigarrow \tau}$$

$$\textsc{ThroughArrow}\quad \frac{F \vdash_s \tau_1' \rightsquigarrow \tau_1 \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 \to \tau_2 \rightsquigarrow \tau_1' \to \tau_2'}$$

$$\textsc{ThroughSum}\quad \frac{F \vdash_s \tau_1 \rightsquigarrow \tau_1' \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 + \tau_2 \rightsquigarrow \tau_1' + \tau_2'}$$

$$\textsc{ThroughProd}\quad \frac{F \vdash_s \tau_1 \rightsquigarrow \tau_1' \qquad F \vdash_s \tau_2 \rightsquigarrow \tau_2'}{F \vdash_s \tau_1 \times \tau_2 \rightsquigarrow \tau_1' \times \tau_2'}$$

$$\textsc{ExposeRecursiveCall}\quad \frac{(\mu X.\tau \rightsquigarrow \mu Y.\tau') \notin \mathrm{Im}(F) \qquad F(f)\ \textit{undefined} \qquad F[(f \mapsto \mu X.\tau \rightsquigarrow \mu Y.\tau')] \vdash_s \tau[X \mapsto \mu X.\tau] \rightsquigarrow \tau'[Y \mapsto \mu Y.\tau']}{F \vdash_s \mu X.\tau \rightsquigarrow \mu Y.\tau'}$$

$$\textsc{Atomic-UseRecursion}\quad \frac{F(f) = \mu X.\tau \rightsquigarrow \mu Y.\tau'}{F \vdash_s \mu X.\tau \rightsquigarrow \mu Y.\tau'}$$

Figure 2.9: Alternative consistency relation for definition of the backtranslation.

Backtranslating a judgment between two recursive types, say $F \vdash_s \mu X.\tau \rightsquigarrow \mu Y.\tau'$, to a static function (to be denoted by $\mathscr{F}^F_{\mu X.\tau \rightsquigarrow \mu Y.\tau'}$), we follow one of the following two paths.

A) If we "previously" exposed a recursive call for this already, i.e. we have $F(f) = \mu X.\tau \rightsquigarrow \mu Y.\tau'$, then the backtranslation is exactly that variable f.

B) If we have no exposed call for this, i.e. $(\mu X.\tau \rightsquigarrow \mu Y.\tau') \notin \mathrm{Im}(F)$, then we unfold the bodies and expose an additional recursive call. That is, we now consider the backtranslation of the judgment $F[(f \mapsto \mu X.\tau \rightsquigarrow \mu Y.\tau')] \vdash_s \tau[X \mapsto \mu X.\tau] \rightsquigarrow \tau[Y \mapsto \mu Y.\tau']$, where we have extended F to $F[(f \mapsto \mu X.\tau \rightsquigarrow \mu Y.\tau')]$ for some fresh f.

The complete backtranslation by induction on the relation in Fig. 2.9 is now depicted in Fig. 2.10; the function $\mathscr{F}_{\tau \Rightarrow \tau'}$ in Eq. (2.2) is now defined as $\mathscr{F}^{\emptyset}_{\tau \rightsquigarrow \tau'}$.

$$\mathscr{F}^{F}_{B\rightsquigarrow B} \triangleq \lambda x : \langle\!\langle B \rangle\!\rangle.\ x$$

$$\mathscr{F}^{F}_{?\rightsquigarrow ?} \triangleq \lambda x : \langle\!\langle ? \rangle\!\rangle.\ x$$

$$\mathscr{F}^{F}_{G\rightsquigarrow ?} \triangleq \mathit{inject}_G$$

$$\mathscr{F}^{F}_{?\rightsquigarrow G} \triangleq \mathit{extract}_G$$

$$\mathscr{F}^{F}_{\tau\rightsquigarrow ?} \triangleq \lambda x : \langle\!\langle \tau \rangle\!\rangle.\ \mathscr{F}^{F}_{G\rightsquigarrow ?}\ (\mathscr{F}^{F}_{\tau\rightsquigarrow G}\ x) \qquad \text{if } \mathcal{S}(\tau) = G \neq \tau$$

$$\mathscr{F}^{F}_{?\rightsquigarrow \tau} \triangleq \lambda x : \langle\!\langle ? \rangle\!\rangle.\ \mathscr{F}^{F}_{G\rightsquigarrow \tau}\ (\mathscr{F}^{F}_{?\rightsquigarrow G}\ x) \qquad \text{if } \mathcal{S}(\tau) = G \neq \tau$$

$$\mathscr{F}^{F}_{\tau_1\to\tau_2\rightsquigarrow\tau_1'\to\tau_2'} \triangleq \lambda f : \langle\!\langle \tau_1 \rangle\!\rangle \to \langle\!\langle \tau_2 \rangle\!\rangle.\ \lambda x : \langle\!\langle \tau_1' \rangle\!\rangle.\ \mathscr{F}^{F}_{\tau_2\rightsquigarrow\tau_2'}\ (f\ (\mathscr{F}^{F}_{\tau_1'\rightsquigarrow\tau_1}\ x))$$

$$\mathscr{F}^{F}_{\tau_1+\tau_2\rightsquigarrow\tau_1'+\tau_2'} \triangleq \lambda s : \langle\!\langle \tau_1 \rangle\!\rangle + \langle\!\langle \tau_2 \rangle\!\rangle.\ \mathtt{case}\ s\ \mathtt{of}\ \lambda x_1 : \langle\!\langle \tau_1 \rangle\!\rangle.\ \mathtt{inj}_1\ (\mathscr{F}^{F}_{\tau_1\rightsquigarrow\tau_1'}\ x_1)\ \mid$$
$$\lambda x_2 : \langle\!\langle \tau_2 \rangle\!\rangle.\ \mathtt{inj}_2\ (\mathscr{F}^{F}_{\tau_2\rightsquigarrow\tau_2'}\ x_2)$$

$$\mathscr{F}^{F}_{\tau_1\times\tau_2\rightsquigarrow\tau_1'\times\tau_2'} \triangleq \lambda p : \langle\!\langle \tau_1 \rangle\!\rangle \times \langle\!\langle \tau_2 \rangle\!\rangle.\ (\mathscr{F}^{F}_{\tau_1\rightsquigarrow\tau_1'}(\pi_1\ p),\ \mathscr{F}^{F}_{\tau_2\rightsquigarrow\tau_2'}(\pi_2\ p))$$

$$\mathscr{F}^{F}_{\mu X.\tau\rightsquigarrow\mu Y.\tau'} \triangleq \begin{cases} f & \mathit{if}\ F(f) = \mu X.\tau \rightsquigarrow \mu Y.\tau' \\ \lambda x : \mu X.\langle\!\langle\tau\rangle\!\rangle.\ \overrightarrow{fix}(\lambda f : \mu X.\langle\!\langle\tau\rangle\!\rangle \to \mu Y.\langle\!\langle\tau'\rangle\!\rangle. & \\ \quad \lambda r : \mu X.\langle\!\langle\tau\rangle\!\rangle.\ \mathtt{fold}\ (\mathscr{F}^{F[(f\mapsto \mu X.\tau\rightsquigarrow\mu Y.\tau')]}_{\tau[X\mapsto\mu X.\tau]\rightsquigarrow\tau'[Y\mapsto\mu Y.\tau']}\ (\mathtt{unfold}\ r)))\ x & \dagger \end{cases}$$

$$\dagger : \mathit{if}\ (\mu X.\tau \rightsquigarrow \mu Y.\tau') \notin \mathrm{Im}(F)$$

Figure 2.10: Defining the function in the backtranslation of casts

Is this Well-defined? Our backtranslation needs to be defined between *any* two consistent types. Therefore, the validity of this definition depends on the assumption that for each two consistent types, $\tau \sim \tau'$, we can derive $\emptyset \vdash_s \tau \rightsquigarrow \tau'$. Fortunately, this is indeed the case, as we have formally proven it in our Coq development. What follows is an intuitive explanation for why this is so.

Given two arbitrary (closed) consistent types, $\tau \sim \tau'$, one can imagine the following procedure to attempt building up a derivation tree for $\emptyset \vdash_s \tau \rightsquigarrow \tau'$: If $\emptyset \vdash_s \tau \rightsquigarrow \tau'$ is a base case, then great, we are done. Else, we just apply (backward reasoning) the applicable inference-rule (it is easily checked that there is always a *unique* rule that is applicable) and instantiate the procedure among all newly created subbranches.

Of course, this procedure needs to terminate! That is, we need to obtain a finite tree. To guarantee this, something must therefore consistently decrease — in a well-founded ordering — in the application of every inference rule.

This does appear the case for *most* of the constructors in the relation: One of the types in the pair of an argument (a judgment above the line) seems to — eventually — always be strictly smaller (structurally) than one of the types in the pair of the conclusion, while the remaining type in the pair of the argument is structurally smaller or equal than that of the remaining type in the conclusion.[10]

Considering ThroughSum for instance, we have $\tau_1 < \tau_1 + \tau_2$ and $\tau_1' < \tau_1' + \tau_2'$ for the first argument and $\tau_2 < \tau_1 + \tau_2$ and $\tau_2' < \tau_1' + \tau_2'$ for the second. A similar results applies to ThroughProd and ThroughArrow even though the latter is contravariant in one of its arguments. Composing FactorDown-$(? + ?)$ (taking τ to be $\tau_1 + \tau_2$) with ThroughSum, we get $? \leq ?$ and $\tau_1 < \tau_1 + \tau_2$ for the first argument, and $? \leq ?$ and $\tau_2 < \tau_1 + \tau_2$ for the second. Similar results apply to the composition of FactorDown-$(? \times ?)$ with ThroughProd and the composition between FactorDown-$(? \to ?)$ with ThroughArrow.

Unfortunately, the same is not true for ExposeRecursiveCall and possibly FactorUp-$(\mu X.\, ?)$ and FactorDown-$(\mu X.\, ?)$. In ExposeRecursiveCall for instance, the types *in the conclusion* are typically structurally smaller than those in the arguments!

Fortunately, this is not a problem however, as it is guaranteed that ExposeRecursiveCall will only be applicable a *finite* amount of times. Indeed, there is only a finite amount of recursive calls to be exposed before having exerted all possibilities, after which the base-case counterpart (Atomic-UseRecursion) is bound to be applicable.

In the previous example, $\mu X.\, ? \times X \sim \mu X.\, \mathbb{N} \times X$, we apply ExposeRecursiveCall one time, after which Atomic-UseRecursion becomes applicable. Generally though, some branches will require multiple applications of ExposeRecursiveCall before Atomic-UseRecursion becomes applicable. As an example, one can consider a derivation tree for $\cdot \vdash_s \mu X.\, X + ? + X \to ? + X + X \rightsquigarrow \mu X.\, X + X + ? \to X + ? + X$.

The reader who is interested in the formal side of this argument, is welcome to check out our Coq code.

[10] We will write $<$ for structurally smaller than and $\leq$ for structurally smaller or equal than.

2.4.3 Logical Relations and Correctness of the Backtranslation

Of course, the backtranslation is only as useful as it is correct; in order to complete the proof in §2.4.1, we need to prove that Theorem 2.4.1 holds. To do so, we set up two logical relations models, as is often done in fully abstract compilation proofs [42].

We define our logical relations in the Iris program logic [28]. Iris features an abstract form of step-indexing, a technique that is usually employed in defining logical relations models for programming languages featuring recursive types [7]. Using Iris allows us to reason about programs at a high level of abstraction, especially for the use of step indexes. In addition to recursive types, we also use step-indexing for modeling the unknown type in our logical relations. The use of a program logic for expressing logical relations for programming languages featuring recursive types goes back to Dreyer et al. [14].

In §2.4.3 we present a short Iris primer. Afterwards, in §2.4.3, we specify the properties that our logical relations should satisfy in order to be able to prove the correctness of our backtranslation (see Theorem 2.4.1). We define our logical relations models in §2.4.3. In §2.4.3, we demonstrate how our logical relations models satisfy the properties presented in §2.4.3.

To get the intuition behind the essence of our logical relations, one does not need to possess any knowledge of Iris. Accordingly, hurried readers who feel comfortable to brush aside the technicalities may skip this primer and go to the paragraph titled "The Logical Relation on Values", focusing only on the value relations presented in Fig. 2.11 (ignoring the modalities and reading $*$ as $\wedge$) and trusting that the relations are naturally extended to (open) expressions as described informally in the other paragraphs afterwards.

An Iris Primer

Iris [28] is a modal, separation logic geared toward reasoning about the correctness of programs. While the framework is vast, here, we restrict ourselves to the small subset necessary to understand our logical relations.

Below, we present the grammar of the relevant parts of *iProp*, the universe of Iris propositions.

$$P ::= P * P \mid P \mathrel{-\!\!*} P \mid \Box\, P \mid \triangleright P \mid x \mid \mu x.P \mid \boxed{P} \mid \Rrightarrow P \mid \mathsf{wp}\; e\, \{\Phi\} \mid \ldots$$

Iris is a *separation* logic, hence we have separating conjunction, $P * Q$ (pronounced "P star Q"), and a magic wand operator, $P \mathrel{-\!\!*} Q$ (pronounced "P wand Q"), analogous respectively to ordinary conjunction and ordinary implication. In general, Iris propositions are *ephemeral*; we can only use them once (similar to linear/affine types). That is, once we use an Iris proposition to prove something, we lose it; we have "consumed" it. For instance, if in trying to prove $P * Q \vdash R * S$ we use P to prove R, then only Q remains in the proof effort of S. Here, $\vdash$ is the entailment relation on Iris propositions.

Besides ephemeral propositions, we have *persistent* ones; propositions that can be freely duplicated. That is, if P is persistent, then we have $P \vdash P * P$. The $\Box$-modality, $\Box\, P$ (pronounced "persistently P"), asserts that P holds *persistently*. In particular, $\Box\, P$ is persistent and hence duplicable, and we have $\Box\, P \vdash P$. As we will discuss, we use the persistently modality to express and enforce that our logical relations on values are persistent.

Iris features a notion of abstract step-indexing. Given an Iris proposition P, $\triangleright P$ (pronounced "later P") asserts that P holds one step later. The abstract step-indexing in Iris allows for the construction of so-called guarded recursive propositions and predicates. The proposition $\mu x.P$ is well-defined if all occurrences of x in P are guarded, i.e., they appear under the later modality. Intuitively, guarded recursive definitions are the unique fixpoint characterized by the equation: $\mu x.P \equiv P[\mu x.P/x]$.

Iris features invariants and resources to facilitate reasoning about programs. An Iris invariant $\boxed{P}$ asserts that P holds invariantly, i.e., at all times. The update modality, $\Rrightarrow$, allows us to update resources (allocation, modification, and deallocation) and to access invariants. The proposition $\Rrightarrow P$ states that P holds *after an appropriate update of the underlying resources and/or accessing invariants.*[11] Iris invariants are persistent and can hence be freely duplicated. The following rule allows us to create invariants:

$$\triangleright P \vdash \Rrightarrow \boxed{P} \qquad \text{(invariant allocation)}$$

As a program logic, Iris is equipped with a weakest precondition calculus to reason about programs. The weakest precondition of a program e with respect to a postcondition $\Phi : \mathit{Val} \to \mathit{iProp}$, written $\mathsf{wp}\ e\ \{\Phi\}$, holds if whenever e reduces to a value v, the postcondition Φ holds for v, i.e., $\Phi(v)$ holds.[12] In some places we write $\mathsf{wp}\ e\ \{v.\ \Phi(v)\}$ instead of $\mathsf{wp}\ e\ \{\Phi\}$. Iris weakest preconditions

[11] In Iris, invariants have names, and the constructs interacting with them (e.g., the to-be-discussed weakest preconditions and the update modality) are annotated with masks to track how these invariants are accessed. This is because accessing the same invariant twice in a nested fashion is generally unsound. We elide these details here which are not crucial for following the arguments presented in this chapter; see our Coq development for more details.

are defined in terms of the more low-level Iris-constructs in such a way that they tie the aforementioned abstract step indexes to the evaluation steps taken by the program.

Crucially, update modalities can be eliminated when proving that weakest preconditions hold:

$$\frac{P \vdash \mathsf{wp}\, e\, \{\Phi\}}{\Rrightarrow P \vdash \mathsf{wp}\, e\, \{\Phi\}} \qquad \frac{P \vdash \mathsf{wp}\, e\, \{\Phi\}}{P \vdash \mathsf{wp}\, e\, \{v.\ \Rrightarrow \Phi(v)\}} \qquad \text{(WP and update)}$$

Specifying the Logical Relation Models

In order to prove Theorem 2.4.1, we shall set up two logical relations models, $\Gamma \vDash e \preccurlyeq e' : \tau$ and $\Gamma \vDash e \succcurlyeq e' : \tau$, both indexed by *gradual* types, relating static terms to gradual terms. We shall carefully construct these relations so as to have them satisfy Lemmas 2.4.2 to 2.4.5.

Lemma 2.4.2. *Given an arbitrary well-typed static term e for which $\Gamma \vdash e : \tau$ holds and an arbitrary context C for which $C : (\lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot\,; \mathbb{1})$ holds, we have*

$$\vdash \cdot \vDash \langle\!\langle C \rangle\!\rangle\, [e] \preccurlyeq C\, [\![\lceil\lceil e\rceil\rceil]\!] : \mathbb{1}$$

Lemma 2.4.3 (Adequacy of $\preccurlyeq$-logical relations)**.**

$$\textit{If} \vdash \cdot \vDash e \preccurlyeq e' : \tau, \textit{ then } e \Downarrow \textit{ implies } e' \Downarrow$$

Lemma 2.4.4. *Given an arbitrary static term e for which $\Gamma \vdash e : \tau$ holds and an arbitrary context C for which $C : (\lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot\,; \mathbb{1})$ holds, we have*

$$\vdash \cdot \vDash \langle\!\langle C \rangle\!\rangle\, [e] \succcurlyeq C\, [\![\lceil\lceil e\rceil\rceil]\!] : \mathbb{1}$$

Lemma 2.4.5 (Adequacy of $\succcurlyeq$-logical relations)**.**

$$\textit{If} \vdash \cdot \vDash e \succcurlyeq e' : \tau, \textit{ then } e' \Downarrow \textit{ implies } e \Downarrow$$

Given Lemmas 2.4.2 to 2.4.5, Theorem 2.4.1 follows easily. The challenge is of course to define the logical relation models (§2.4.3) and prove that they do indeed satisfy said lemmas(§2.4.3).

[12] This actually corresponds to the Iris proposition $\mathsf{wp}_{\mathsf{Stuck}}\, e\, \{\Phi\}$, the *stuck* version of the weakest precondition that does not prove safety.

Defining the Logical Relation Models

We define our logical relations models following the standard pattern. We first define the logical relations on closed values and expressions. Afterwards, we canonically extend the relations on closed expressions to the general relations on open terms.

Throughout this section, we focus exclusively on the $\preccurlyeq$-relations; the $\succcurlyeq$-relations are defined analogously.

The Logical Relations on Values We begin by defining the logical relations on closed values, $\mathcal{V}_{\preccurlyeq}[\![\tau]\!] : \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$, in Fig. 2.11. Such relations are

$$\mathcal{V}_{\preccurlyeq} \triangleq \mu\Psi.\ \mathbf{V}_{\preccurlyeq}(\Psi)$$

$$\mathbf{V}_{\preccurlyeq}[\![B]\!](v, v') \triangleq \exists b : B.\ v = b * v' = \lceil\!\lceil b \rceil\!\rceil$$

$$\mathbf{V}_{\preccurlyeq}[\![\tau_1 + \tau_2]\!](v, v') \triangleq \bigvee_{i \in \{1,2\}} \exists v_i, v_i'.\ v = \mathtt{inj}_i\ v_i * v' = \mathtt{inj}_i\ v_i' * \mathbf{V}_{\preccurlyeq}[\![\tau_i]\!](v_i, v_i')$$

$$\mathbf{V}_{\preccurlyeq}[\![\tau_1 \times \tau_2]\!](v, v') \triangleq \exists v_1, v_2, v_1', v_2'.\ v = (v_1, v_2) * v' = (v_1', v_2') * \mathbf{V}_{\preccurlyeq}[\![\tau_1]\!](v_1, v_1') * \mathbf{V}_{\preccurlyeq}[\![\tau_2]\!](v_2, v_2')$$

$$\mathbf{V}_{\preccurlyeq}[\![\tau_1 \to \tau_2]\!](v, v') \triangleq \Box(\forall w, w'.\ \mathbf{V}_{\preccurlyeq}[\![\tau_1]\!](w, w') \mathrel{-\!\!*} \mathcal{L}_{\preccurlyeq}(\mathbf{V}_{\preccurlyeq}[\![\tau_2]\!])(v\ w, v'\ w')) \qquad (\mathcal{L}_{\preccurlyeq} \text{ to be defined})$$

$$\mathbf{V}_{\preccurlyeq}[\![\mu X.\tau]\!](v, v') \triangleq \Box(\exists w, w'.\ v = \mathtt{fold}\ w * v' = \mathtt{fold}\ w' * \triangleright \Psi[\![\tau[X \mapsto \mu X.\tau]]\!](w, w'))$$

$$\begin{aligned}\mathbf{V}_{\preccurlyeq}[\![?]\!](v, v') \triangleq \Box(&\exists w, w'.\ (v = \mathtt{fold}\ (\mathtt{inj}_B\ w) * v' = w' : B \Rightarrow ? * \triangleright \Psi[\![B]\!](w, w'))\\ &\vee (v = \mathtt{fold}\ (\mathtt{inj}_+\ w) * v' = w' : ? + ? \Rightarrow ? * \triangleright \Psi[\![? + ?]\!](w, w'))\\ &\vee (v = \mathtt{fold}\ (\mathtt{inj}_\times\ w) * v' = w' : ? \times ? \Rightarrow ? * \triangleright \Psi[\![? \times ?]\!](w, w'))\\ &\vee (v = \mathtt{fold}\ (\mathtt{inj}_\to\ w) * v' = w' : ? \to ? \Rightarrow ? * \triangleright \Psi[\![? \to ?]\!](w, w'))\\ &\vee (v = \mathtt{fold}\ (\mathtt{inj}_\mu\ w) * v' = (\mathtt{fold}\ w') : \mu X.? \Rightarrow ? * \triangleright \Psi[\![?]\!](w, w')))\end{aligned}$$

Figure 2.11: The logical relations on values

usually defined by induction on the structure of the type, i.e., τ. However, we cannot do this for our type system due to the presence of recursive types, $\mu X.\tau$, and the unknown type, $?$. Instead, we define our relation as a guarded

recursive predicate, $\mathcal{V}_{\preccurlyeq} \triangleq \mu\Psi.\ \mathbf{V}_{\preccurlyeq}(\Psi)$. The higher-order predicate $\mathbf{V}_{\preccurlyeq}$ of type $(\mathit{Type} \to \mathit{Val} \to \mathit{Val} \to \mathit{iProp}) \to \mathit{Type} \to \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$ is defined by induction on types. For cases where the type does not become structurally smaller, i.e., in recursive types and the unknown type, we use guarded recursion; note that all such occurrences are guarded by a later modality. Readers familiar with the usual encoding of logical relations in program logics featuring abstract step-indexing may note that our approach differs slightly from the usual practice, e.g., by Timany et al. [56], where only the value relation for the recursive type is defined using guarded recursive predicates.

Notice also that we use the persistently modality three times to ensure that the relations are persistent in order for them to be duplicable. This is necessary as values can be used multiple times, e.g., when passed as a function argument.

The goal of this relation is twofold, as can be seen from Lemma 2.4.2. It must relate static values (on the left-hand side) to their embeddings (on the right-hand side), as well as gradual values (on the right-hand side) to their appropriate static values in backtranslation. The latter is due to having the backtranslation of the context on the left-hand side. With this in mind, the value relation at the base type, which intuitively requires that the two values are related if they are "the same", can also be formally expressed as follows ($\dashv\vdash$ denotes the logical equivalence relation on Iris propositions):

$$\mathbf{V}_{\preccurlyeq}[\![B]\!](v, v') \dashv\vdash \exists b : B.\ v = \langle\!\langle b \rangle\!\rangle * v' = b$$

Values at a sum type are related if they are constructed using the same injection and the injected values are related at the appropriate type. Value at a product type, $\tau_1 \times \tau_2$, are related if they are both pairs in which the first components are related at τ_1 and the second components at τ_2.

Intuitively, two functions, say v and v', are related at $\tau_1 \to \tau_2$, if for any arbitrary pair of values w and w' related at τ_1, we have that the two expressions $v\ w$ and $v'\ w'$ are related at τ_2. We define the relation on closed expressions by lifting the relation on closed values. Given any relation on values, the higher-order predicate $\mathcal{L}_{\preccurlyeq}$ of type $(\mathit{Val} \to \mathit{Val} \to \mathit{iProp}) \to \mathit{Expr} \to \mathit{Expr} \to \mathit{iProp}$ lifts it to a relation on expressions. We define the predicate $\mathcal{L}_{\preccurlyeq}$ later on.

Values at a recursive type, $\mu X.\tau$, are related if they are both folded and the original (unfolded) values are, one step later, related at type $\tau[X \mapsto \mu X.\tau]$. Two values, say v and v', are related at the unknown type if for some ground type G (excluding $\mu X.?$ here), we have some w and w' related (one step later) at G for which v is the result of applying inject_G to w and v' is equal to $w' : G \Rightarrow ?$. They are also related if, for some w and w' related (one step later) at $?$, we have that v is the result of applying $\mathit{inject}_{\mu X.?}$ to $\texttt{fold}\ w$ and v' is equal to $\texttt{fold}\ w' : \mu X.? \Rightarrow ?$.

Lifting Value Relations to Expression Relations Intuitively, what we expect of the expression relation is that if two closed expressions e and e' are related (in the $\preccurlyeq$-relation) at a type τ, then it must follow that $e \to^* v$ implies $e' \to^* v'$ for some v' such that $\mathbf{V}_{\preccurlyeq}[\![\tau]\!](v, v')$. We define this expression relation following the same idea that is usually employed for defining binary logical relations models in Iris, e.g., in Frumin et al. [15], Timany et al. [55, 56], and Timany and Birkedal [54]. That is, we use weakest preconditions for the left-hand side, and a custom-defined ghost resource and an appropriate invariant to keep track of the execution on right-hand side. Intuitively, in this approach, a binary logic is emulated as a unary logic for the left-hand side term, while the right-hand side term is treated as a specification for how the left term should behave.

Given a relation on values, Ψ, we define the lifting of that relation to closed expressions as follows.

$$\mathcal{L}_{\preccurlyeq}(\Psi)(e, e') \triangleq \left| \begin{array}{l} \forall e_i, K.\ \mathit{Initially}(e_i) * \mathit{Currently}_2(K[e']) \mathrel{-\!\!*} \\ \qquad \mathsf{wp}\ e\ \{v.\ \exists v'.\ \mathit{Currently}_2(K[v']) * \Psi(v, v')\} \end{array} \right.$$

Here, we have two ghost resources, $\mathit{Currently}_1$ and $\mathit{Currently}_2$, which are defined such that their expressions are required to always be the same:

$$\mathit{Currently}_1(e_1) * \mathit{Currently}_2(e_2) \vdash e_1 = e_2$$

Consequently, we can only update the expression if we update *both* propositions.

$$\mathit{Currently}_1(e_1) * \mathit{Currently}_2(e_1) \vdash \Rrightarrow \mathit{Currently}_1(e_2) * \mathit{Currently}_2(e_2) \qquad \text{(Currently update)}$$

We can allocate resources to establish $\mathit{Currently}_1(e')$ and $\mathit{Currently}_2(e')$:

$$\vdash \Rrightarrow \mathit{Currently}_1(e') * \mathit{Currently}_2(e') \qquad \text{(Currently allocation)}$$

The proposition $\mathit{Initially}(e_i)$ is an invariant, defined as follows.

$$\mathit{Initially}(e_i) \triangleq \boxed{\exists z. \mathit{Currently}_1(z) * e_i \to^* z}$$

The $\mathit{Currently}_1$ resource is therefore always owned by this invariant. Keep in mind though that this invariant is persistent and needs to be preserved during the entire execution, even when it is not returned in the postcondition. Moreover, maintaining this invariant poses another restriction on how we can update $\mathit{Currently}_1$ and $\mathit{Currently}_2$: As seen in Eq. (Currently update), we can only update these ghost-resources by accessing the invariant (in order to obtain $\mathit{Currently}_1$). Note however that because of the invariant $\mathit{Initially}(e_i)$, we can only update $\mathit{Currently}_1$ and $\mathit{Currently}_2$ to an expression e' *if* e' *is such that* $e_i \to^* e'$.

Importantly, we also want to reason modularly about the relatedness of two *sub*-expressions. This idea is captured in the following rule:

$$\frac{\mathcal{L}_{\preccurlyeq}(\Phi)(e, e') \qquad \forall v, v'.\ \Phi(v, v') \implies \mathcal{L}_{\preccurlyeq}(\Psi)(K[v], K'[v'])}{\mathcal{L}_{\preccurlyeq}(\Psi)(K[e], K'[e'])}$$

This rule follows straightforwardly from the definition of the expression relation above together with the bind rule for Iris weakest preconditions:

$$\frac{\mathsf{wp}\ e\ \{\Phi\} \qquad \forall v.\ \Phi(v) \implies \mathsf{wp}\ K[v]\ \{\Psi\}}{\mathsf{wp}\ K[e]\ \{\Psi\}}$$

Having defined $\mathcal{L}_{\preccurlyeq}$, we have completed the relation on values, and moreover the relation on closed expressions, $\mathcal{E}_{\preccurlyeq}[\![\tau]\!] : \mathit{Expr} \to \mathit{Expr} \to \mathit{iProp}$, follows easily.

$$\mathcal{E}_{\preccurlyeq}[\![\tau]\!] = \mathcal{L}_{\preccurlyeq}(\mathcal{V}_{\preccurlyeq}[\![\tau]\!])$$

The Relations on Open Expressions Following the standard practice, we now extend our logical relations (to open terms) based on our relations on closed expressions and values. Open terms are related if they are related as closed terms under any possible substitution of related values at the appropriate types. To formalize this, we first extend our value relations for gradual types to relations on *vectors* of values for a gradual *typing contexts*, $\mathcal{G}_{\preccurlyeq}$:

$$\begin{aligned} \mathcal{G}_{\preccurlyeq}[\![\cdot]\!](\varepsilon, \varepsilon) &\triangleq \mathsf{True} \\ \mathcal{G}_{\preccurlyeq}[\![x : \tau, \Gamma]\!]((v; \vec{w}), (v'; \vec{w}')) &\triangleq \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v') * \mathcal{G}_{\preccurlyeq}[\![\Gamma]\!](\vec{w}, \vec{w}') \end{aligned} \tag{2.3}$$

Here, ε is the empty vector. We now define our logical relations:

$$\Gamma \vDash e \preccurlyeq e' : \tau \triangleq \forall \vec{v}, \vec{v}'. \mathcal{G}_{\preccurlyeq}[\![\Gamma]\!](\vec{v}, \vec{v}') \mathrel{-\!\!*} \mathcal{E}_{\preccurlyeq}[\![\tau]\!](e[\vec{x} \mapsto \vec{v}], e'[\vec{x} \mapsto \vec{v}']) \tag{2.4}$$

The Logical Relations Satisfy the Specifications

Here, we give proof sketches for why Lemmas 2.4.2 and 2.4.3 hold; given the analogously-defined $\succcurlyeq$-relations, the arguments for Lemmas 2.4.4 and 2.4.5 are similar.

Proof Sketch for Lemma 2.4.3 Let e and e' be two closed expressions for which we can derive in Iris that they are related at type τ, i.e., the following holds:

$$\vdash \cdot \vDash e \preccurlyeq e' : \tau$$

We show that $e \Downarrow$ implies $e' \Downarrow$. By definition $\cdot \vDash e \preccurlyeq e' : \tau$ is equivalent to $\mathcal{E}_{\preccurlyeq}[\![\tau]\!](e, e')$, and hence to the following:

$$\forall e_i, K.\ \mathit{Initially}(e_i) * \mathit{Currently}_2(K[e']) \mathrel{-\!\!*}$$

$$\mathsf{wp}\ e\ \{v.\ \exists v'.\mathit{Currently}_2(K[v']) * \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v')\}$$

We use Eq. (Currently allocation) to create the pair of propositions, $\mathit{Currently}_1(e')$ and $\mathit{Currently}_2(e')$, and use the former together with Eq. (invariant allocation) to establish $\mathit{Initially}(e')$; note that $e' \to^* e'$ holds trivially.[13] Now by picking e_i to be e' and K to be the empty evaluation context, $[\cdot]$, we obtain the following:

$$\mathsf{wp}\ e\ \{v.\ \exists v'.\mathit{Currently}_2(v') * \mathcal{V}_{\preccurlyeq}[\![\tau]\!](v, v')\}$$

Using Eq. (WP and update) and the invariant $\mathit{Initially}(e')$ we can conclude:

$$\mathsf{wp}\ e\ \{v.\ \exists v'.\ e' \to^* v'\}$$

At this point, we can apply the adequacy theorem of Iris's weakest preconditions. It states that for any meta-level (i.e. outside Iris) postcondition[14] φ, if $\mathsf{wp}\ e\ \{\varphi\}$ holds, then $\varphi(v)$ holds at the meta level for any value v that e reduces to. Here $\varphi(v)$ is taken to be $\exists v'.\ e' \to^* v'$.

Proof Sketch for Lemma 2.4.2 Lemma 2.4.2 follows from the compatibility lemmas in Fig. 2.12 and the much more interesting one given below, which plays a central role in our full abstraction argument:

Lemma 2.4.6 (Compatibility lemma for casts)**.**

$$\frac{\Gamma \vDash e \preccurlyeq e' : \tau_i \qquad \tau_i \sim \tau_f}{\Gamma \vDash (\mathscr{F}^{\emptyset}_{\tau_i \rightsquigarrow \tau_f} e) \preccurlyeq (e' : \tau_i \Rightarrow \tau_f) : \tau_f}$$

Remember that the backtranslation of casts mirrors the structure of the cast calculus quite faithfully as we defined it by induction on the relation in Fig. 2.9. Moreover, we prove this lemma by induction on said relation. To do so though, we generalize it to $\mathscr{F}^{F}_{\tau_i \rightsquigarrow \tau_f}$ for an arbitrary derivation $F \vdash_s \tau_i \rightsquigarrow \tau_f$. Of course, $\mathscr{F}^{F}_{\tau_i \rightsquigarrow \tau_f}$ contains some free variables ($\mathrm{dom}(F)$), so we will want to substitute these with (arbitrary) "appropriate" values. We first formalize this notion of "appropriate" values: a partial function, $\alpha : \mathit{Var} \rightharpoonup \mathit{Val}$, serves as an appropriate

[13] Note that here we are using Eq. (WP and update) implicitly.
[14] For our Coq formalization, this means a Coq predicate over values.

$$\frac{(x,\tau)\in\Gamma}{\Gamma\vDash x\preccurlyeq x:\tau}\qquad\frac{\Gamma\vDash e_1\preccurlyeq e_1':\tau_1\qquad\Gamma\vDash e_2\preccurlyeq e_2':\tau_2}{\Gamma\vDash(e_1,e_2)\preccurlyeq(e_1',e_2'):\tau_1\times\tau_2}$$

$$\frac{\Gamma\vDash e\preccurlyeq e':\tau_1\times\tau_2\qquad i\in\{1,2\}}{\Gamma\vDash\pi_i\ e\preccurlyeq\pi_i\ e':\tau_i}\qquad\frac{\Gamma\vdash b:B}{\Gamma\vDash b\preccurlyeq b:B}$$

$$\frac{\Gamma\vDash e\preccurlyeq e':\tau_i\qquad i\in\{1,2\}}{\Gamma\vDash \mathtt{inj}_i\ \ e\preccurlyeq \mathtt{inj}_i\ \ e':\tau_1+\tau_2}$$

$$\frac{\Gamma\vDash e\preccurlyeq e':\tau_1+\tau_2\qquad\Gamma\vDash e_1\preccurlyeq e_1':\tau_1\to\tau\qquad\Gamma\vDash e_2\preccurlyeq e_2':\tau_2\to\tau}{\Gamma\vDash \mathtt{case}\ e\ \mathtt{of}\ (e_1\mid e_2)\preccurlyeq \mathtt{case}\ e'\ \mathtt{of}\ (e_1'\mid e_2'):\tau}$$

$$\frac{\Gamma\vDash e\preccurlyeq e':\tau_1\to\tau_2\qquad\Gamma\vDash e_1\preccurlyeq e_1':\tau_1}{\Gamma\vDash e\ e_1\preccurlyeq e'\ e_1':\tau_2}$$

$$\frac{x:\tau_1,\Gamma\vDash e\preccurlyeq e':\tau_2}{\Gamma\vDash\lambda x:\langle\!\langle\tau_2\rangle\!\rangle.\ e\preccurlyeq\lambda x:\tau_2.\ e':\tau_1\to\tau_2}\qquad\frac{\Gamma\vDash e\preccurlyeq e':\tau[X\mapsto\mu X.\tau]}{\Gamma\vDash \mathtt{fold}\ e\preccurlyeq \mathtt{fold}\ e':\mu X.\tau}$$

$$\frac{\Gamma\vDash e\preccurlyeq e':\mu X.\tau}{\Gamma\vDash \mathtt{unfold}\ e\preccurlyeq \mathtt{unfold}\ e':\tau[X\mapsto\mu X.\tau]}$$

Figure 2.12: Unexciting compatibility lemmas

substituting function in $\mathscr{F}^F_{\tau_i\rightsquigarrow\tau_f}$ if the following condition, defined as $\mathcal{H}[\![F]\!]_{\preccurlyeq}(\alpha)$ holds:

$$\ast_{f\mapsto(\tau_1\rightsquigarrow\tau_2)\in F}\ \square(\forall v,v'.\mathcal{V}_{\preccurlyeq}[\![\tau_1]\!](v,v')\mathrel{-\!\!*}\mathcal{E}_{\preccurlyeq}[\![\tau_2]\!](\alpha(f)\ v,v':\tau_1\Rightarrow\tau_2)) \quad (2.5)$$

That is, for each $F(f)=(\tau_1\rightsquigarrow\tau_2)$ we must have $\alpha(f):\langle\!\langle\tau_1\rangle\!\rangle\to\langle\!\langle\tau_2\rangle\!\rangle$ defined such that it serves as a successful emulation of a cast from τ_1 to τ_2. Given this definition, we obtain the following generalization of Lemma 2.4.6, where $\mathscr{F}^F_{\tau_i\rightsquigarrow\tau_f}[F\mapsto\alpha]$ denotes the substitution of the free variables in $\mathscr{F}^F_{\tau_i\rightsquigarrow\tau_f}$ with the values defined by α.

Lemma 2.4.7 (Compatibility lemma for casts – Generalized)**.**

$$\frac{\Gamma\vDash e\preccurlyeq e':\tau_i\qquad F\vdash_s\tau_i\rightsquigarrow\tau_f}{\forall\alpha.\mathcal{H}[\![F]\!]_{\preccurlyeq}(\alpha)\mathrel{-\!\!*}\Gamma\vDash\mathscr{F}^F_{\tau_i\rightsquigarrow\tau_f}[F\mapsto\alpha]\ e\preccurlyeq e':\tau_i\Rightarrow\tau_f:\tau_f}$$

This lemma states that the backtranslation $\mathscr{F}^F_{\tau_i\rightsquigarrow\tau_f}$ must relate to a cast $\tau_i\Rightarrow\tau_f$ after arbitrary substitution with a function that satisfies $\mathcal{H}[\![F]\!]_{\preccurlyeq}$.

This definition resembles the extension of our relation from closed to open expressions, i.e., Eq. (2.4), in which Eq. (2.3) is analogous to Eq. (2.5). The proof of this generalized lemma is by induction on the alternative consistency relation (Fig. 2.9). For more details see our Coq formalization.

As an aside, we briefly remark that the compatibility lemmas in Fig. 2.12 (not including Lemma 2.4.7), together with the adequacy of logical relations, imply the superset criterion and hence the proof of reflection sketched in Fig. 2.3.

2.5 Discussion

2.5.1 Contextual Equivalence in GTLC_μ

Equivalence by Equi-termination In this chapter, we have stated the full abstraction of the embedding from STLC_μ to GTLC_μ with respect to standard notions of contextual equivalence by equi-termination in both STLC_μ and GTLC_μ.

As mentioned in §2.2.2, such an equivalence in GTLC_μ does not distinguish between divergence and dynamic type errors. More concretely, for contextually equivalent e and e' ($\Gamma \vdash e \simeq_{ctx} e' : \tau$), there can still be a gradual context, say $C : (\Gamma; \tau) \Rightarrow (\cdot\,; \mathbb{1})$, for which $C[e]$ diverges and $C[e']$ gives a runtime error. For instance, the two terms, $\lceil\lceil \lambda f : \mathbb{B} \to \mathbb{B}.\pi_2(f\ \mathtt{true}, \Omega)\rceil\rceil$ and $\lceil\lceil \lambda f : \mathbb{B} \to \mathbb{B}.\Omega\rceil\rceil$, are still equivalent even though they could be instantiated in a context $[\cdot]\ f_{\mathtt{CastError}}$ where $f_{\mathtt{CastError}}$ is defined such that upon application it will always return a cast error.

Equating divergence with runtime errors allows us to get away with a subtle difference in expressiveness between gradual and static contexts: the former have the ability to cause dynamic errors while the latter do not. This is appropriate however, if we consider cast errors as a debugging mechanism of the language, which is not observable in production settings.

Equivalence that Distinguishes Cast Errors and Divergence We might be interested in this slight difference in expressiveness; formally, we have the following definition.

Definition 2.5.1 (Contextual equivalence distinguishing cast errors and divergence)**.** Two terms e_1, e_2 in GTLC_μ are equivalent iff $\Gamma \vdash e_1, e_2 : \tau$ and for all gradual contexts $C : (\Gamma; \tau) \Rightarrow (\cdot\,; \mathbb{1})$, we have both ($C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$) and ($C[e_1] \Uparrow$ iff $C[e_2] \Uparrow$).

With respect to this definition, full abstraction of the embedding fails as the two terms from the previous paragraph are not equivalent anymore.

As an interesting future work one could try to address this by designing the gradual language to prevent static terms from being exposed to `CastError`s. Alternatively, one might introduce an equivalent form of dynamic errors in the static language. This might seem like a way to "define the problem away", but this is not the case: it simply makes it clear to source language developers that they should take the possibility of dynamic errors into account when reasoning about their code. We have not proven this, but we believe our existing backtranslation can easily be adjusted to prove full abstraction with respect to Definition 2.5.1 by backtranslating dynamic errors to the equivalent error in the static language.

Equivalence Accounting for Blame Going further, one can wonder how blame would effect contextual equivalence and full abstraction of the embedding. However, defining contextual equivalence that is aware of blame seems quite challenging. When translating a pair of similarly-behaving terms to the cast calculus, we would somehow need to smartly assign blame-labels to both these terms so as to recover equivalence for non-trivial cases.

Whether or not a meaningful definition of blame-aware equivalence exists, how it would look like, and how it would affect full abstraction are all challenging questions that we leave as future work.

2.5.2 Partiality in the Source Language

It is also worth observing that our proof crucially relies on the existence of recursive types in the source language. The definition of the universal type $\mathscr{U}$, the backtranslation between recursive types, and the backtranslation of cast errors to diverging Ω terms rely on this. But why have we not worked with a total simply typed lambda calculus STLC, and does GTLC also satisfy the FAE criterion with respect to this?

To answer the latter question first, consider that contextual equivalence is strictly stronger in the total STLC than in STLC_μ. For example, the term $\lambda f : \mathbb{B} \to \mathbb{B}.\, \pi_2\ (f\ \mathit{true}, 43)$ is contextually equivalent to $\lambda f : \mathbb{B} \to \mathbb{B}.\, 43$, essentially because invoking f and ignoring its result is equivalent to not invoking it if we are sure that f will terminate. In GTLC, we can apply both functions to a function that diverges or fails upon application, breaking the two terms'

contextual equivalence.[15] Hence, the GTLC (as a gradualization of the total STLC) does not satisfy the FAE criterion.

Is this a counterargument to the FAE criterion? No! It merely highlights an important property of the GTLC, namely that it enforces a specific semantics of static STLC types and that this semantics is not the one from STLC but the one from a version of STLC *which has some source of partiality* (like the recursive types in STLC_μ). More concretely, the GTLC does not make any effort to enforce termination, which is part of the semantics of function types in STLC. Because of this, GTLC can only be considered as a gradualization of a partial variant of STLC (like STLC_μ), not of the total STLC. Ultimately though, the question concerning which static language the GTLC is a gradualization of is not that clear-cut, allowing for different interpretations. From an historical perspective, the static language would be a total STLC. Practically speaking though – from the perspective of the programmer –, it is useful to think of a recursive variant whose equivalences *are* preserved in the gradual language.

In other words, the failure of GTLC with respect to STLC should not be considered as a weakness of the FAE criterion or a clever choice of example by us. Instead, this failure of the criterion accurately demonstrates that GTLC does not enforce the semantics of types from STLC (but does so for the semantics of types in e.g. STLC_μ). This immediately suggests another interesting question that we will not touch in detail: can we gradualize the STLC? In other words, is it possible to construct a gradual language that enforces the interpretation of types from STLC, i.e., the totality of functions?

It is likely that the latter is probably too much to ask for. Such a gradualization would have to be total as well (assuming gradual terms to easily interface with static types). Moreover, Maillard et al. [32] have shown already that even if we were to only tame non-divergence in a gradual counterpart (ignoring error terms as another source of partiality) to a total STLC (that conservatively extends it), we have to sacrifice graduality.[16]

2.5.3 Recursive Types vs. Term Recursion?

So it is clear that GTLC should be considered the gradualization of a partial STLC. But still, that does not make it clear why we have used STLC_μ. "Why

[15] Consider, for instance, the gradual term $e = \lambda x : \mathbb{B}.\, (\lambda y : \mathbb{B}.\, y)\ ((\lambda y : ?.\, y)\ ((\lambda y : \mathbb{N}.\, y)\ ((\lambda y : ?.\, y)\ x)))$. The term $\lceil\lceil (\lambda f : \mathbb{B} \to \mathbb{B}.\, 43) \rceil\rceil\ e$ evaluates to 43, while $\lceil\lceil (\lambda f : \mathbb{B} \to \mathbb{B}.\, \pi_2\ (f\ \text{true}, 43)) \rceil\rceil\ e$ fails.

[16] More precisely, they state a fire-triangle theorem: gradualizing a total STLC (or CIC, the calculus of inductive concstructions [44]), it can only satisfy two out of the following three properties: non-divergence, being a conservative extension to STLC (or CIC), and graduality.

bother with recursive *types*," the reader may wonder, "if adding recursion to STLC with just a *fix* operator will do the trick?" In other words, we may ask ourselves whether or not $\mathrm{GTLC}_{\mathrm{fix}}$, the natural gradualization of $\mathrm{STLC}_{\mathrm{fix}}$ (a version of the STLC without recursive types but with a builtin term-level fixpoint operator *fix*) satisfies the FAE criterion. If it does, it is immediately clear that it will need a separate argument. As we have pointed out above, our proof of FAE does crucially rely on recursive types for constructing the universal type $\mathscr{U}$, so it clearly does not extend to $\mathrm{STLC}_{\mathrm{fix}}$.

Even so, we do believe that the embedding from $\mathrm{STLC}_{\mathrm{fix}}$ into $\mathrm{GTLC}_{\mathrm{fix}}$ is fully abstract. We are aware of ongoing work to prove a fully abstract embedding from $\mathrm{STLC}_{\mathrm{fix}}$ into $\mathrm{STLC}_{\mu,\mathrm{fix}}$ using the approximate backtranslation technique by Devriese et al. [10, 13]. Because full abstraction is preserved upon composition, this result could then be composed with ours to prove that the FAE criterion holds between $\mathrm{GTLC}_{\mu,\mathrm{fix}}$ and $\mathrm{STLC}_{\mathrm{fix}}$. Moreover, as the operational semantics of $\mathrm{GTLC}_{\mathrm{fix}}$ is trivially contained in $\mathrm{GTLC}_{\mu,\mathrm{fix}}$, every context in $\mathrm{GTLC}_{\mathrm{fix}}$ can be seen as a context in $\mathrm{GTLC}_{\mu,\mathrm{fix}}$; we can conclude therefore that the embedding from $\mathrm{STLC}_{\mathrm{fix}}$ into $\mathrm{GTLC}_{\mathrm{fix}}$ preserves contextual equivalences.

In other words, we believe we should be able to recover the result about $\mathrm{STLC}_{\mathrm{fix}}$ by composing the current result with a separately interesting one from other work, whose proof requires a different proof technique (approximate backtranslation [10]). By making the current result only cover STLC_{μ}, we have been able to separate concerns and obtain separately interesting but composable results.

Trying to tackle a full abstraction proof from $\mathrm{STLC}_{\mathrm{fix}}$ into $\mathrm{GTLC}_{\mathrm{fix}}$ directly would require a family of approximate universal types to back-translate into, in essence redoing much of the same work that has been done already by Patrignani et al. [43].

2.5.4 Robust Relational Hyperproperty Preservation for GTLC_{μ}

It is interesting to note that in our FAE-proof for GTLC_{μ}, our backtranslation on contexts is actually a bit overspecified; for any gradual context, we define a backtranslation that correctly emulates it, *independent of the term that is plugged into it.* Therefore, we have proven something stronger than is absolutely necessary. In principle, it would be enough to construct a backtranslation of a context such that it only works for two given terms under consideration in Fig. 2.4. This is what happens, for instance, in backtranslations that are based on trace semantics, e.g. Patrignani et al. [41].

In fact, the existence of an arbitrary backtranslation is actually documented as

a proposed stronger formal criterion in the field of secure compilation. It is the strongest robust-property-preservation property from a broad range introduced by Abate et al. [3]: *Robust Relational Hyperproperty Preservation*. Abate et al. [3] give an alternative "property-free" characterization which (in this setting) exactly corresponds to Theorem 2.4.1.

Interestingly, this suggests that GTLC does not just preserve equivalences from STLC_μ, but also unary (safety) and k-ary properties. Unfortunately, the framework of Abate et al. [3] can only be applied to languages with traces modeling the interaction of a program with the outside world. In this context, such traces would either be artificial or trivial so we stick to the current binary setting. However, we plan to extend the FAE criterion to include preservation of unary and k-ary properties in follow-up work.

2.5.5 Other Gradual Languages

We believe our proposal raises interesting questions about existing gradual languages. We have already talked a bit about gradual parametricity and gradual security types, but what about other gradualizations? Many other gradual languages enforce static type systems with properties that go beyond type safety, for example ownership types [49], effect systems [8], session types [22] etc.

It would be interesting to investigate in more detail whether these gradualizations enforce source language type-based reasoning and if so, to what extent. Do they satisfy the FAE criterion with respect to their source language? If they do not, are there possibilities to enforce more of the source types semantics, and if so, how? In some cases, the dynamic language also offers useful abstractions, and it might also be interesting to consider preservation of the dynamic language's properties, i.e. fully abstract embedding for the dynamic language.

2.5.6 The Pursuit of FAE

When gradualizing a static language, we should strive to preserve its reasoning principles as much as possible. To this end, adherence to the FAE criterion provides a fixed, unambiguous, and general goal. As discussed, FAE implies that two terms in the static language are equivalent in the static language if, and only if they are equivalent in the gradual language. This means that all static-based reasoning principles involving contextual equivalences would remain sound when applied to the fully static parts in a dynamic codebase, regardless of what the surrounding untyped code may do. In other words, not only would

static refactorings remain valid when performed in a dynamic codebase, the static abstractions (expressed as contextual equivalences) would remain valid in the presence of dynamic code (e.g., the untyped code cannot peer through abstract data types).

Another way to look at FAE is to see what the implications are of FAE failing. In such a situation, we have none of the aforementioned advantages, e.g. refactoring the code to use an alternative implementation of an abstract data type is not necessarily safe. Moreover, if FAE fails for a particular gradualization, then some static equivalences must necessarily be broken in the presence of untyped code! Therefore, failure of FAE should be carefully investigated. Such an investigation can have one of the following two outcomes. It might actually suggest a better design that satisfies FAE. For gradualizing the total STLC for instance, we have pointed out such a possibility in §2.5.2. Alternatively, the investigation might also make clear however that enforcing FAE is not really feasible at a reasonable cost. In this case, pointing this out makes it (formally) clear that some reasoning is lost in the gradualization process; an important message for programmers. One can furthermore try to contextualize this failure, as we will illustrate below for GTLC as a gradualization of the total STLC.[17]

Investigating Failure of FAE for GTLC **w.r.t. the Total** STLC We can investigate which weaker properties (regarding the preservation of reasoning principles) are still satisfied. For instance, we might restrict ourselves to a meaningful subset of equivalences and prove that at the least, they are still preserved upon embedding. In the case of GTLC as a gradualization of total STLC, we might, for instance, consider only those equivalences that also hold in STLC_μ and prove that at least they are preserved upon embedding.

Alternatively, we might shift towards a different static language, one whose equivalences are preserved more naturally upon embedding. We have shifted from GTLC as a gradualization of total STLC to GTLC_μ as a gradualization of STLC_μ. As remarked upon in §2.5.3, other partial variants like STLC_{fix} are also possible. We might also theorize that adding a non-recoverable error term to the total STLC (thereby making it partial) will also give us a gradualization (à la GTLC) that satisfies FAE.

We might wonder what can or cannot happen to the static equivalences that are broken upon embedding. More concretely, we may wonder whether given any two equivalent static terms, we could still prove that their embeddings satisfy some other, much weaker (but not meaningless) relation. For instance, given two equivalent static terms in the STLC, we can hypothesize that the pair of

[17] Remember that in §2.5.2, we showed that the GTLC with respect to the total STLC does not satisfy FAE.

embeddings in the GTLC still satisfies the relation $\mathcal{R}$ defined below.

$$\mathcal{R}_{(\Gamma,\tau)}(e, e') \quad \textit{iff} \quad \nexists C : (\Gamma;\tau) \Rightarrow (\cdot\,;\mathbb{B}).\ (C[e] \rightarrow^* \texttt{true}) \wedge (C[e'] \rightarrow^* \texttt{false})$$

To be clear, we do not think of this constraint as one whose adherence constitutes a desirable language; rather, we think of it as a bare minimum.

2.6 Related Work

The FAE criterion has already been put forward by Devriese et al. [11], although the authors only demonstrate that it does not hold for gradual parametric calculi, as discussed in §2.3.2.

It is not feasible to provide a comprehensive overview of gradual typing here, so we will focus on research about correctness properties for gradual languages. However, it is worth discussing the two lineages of gradual typing that Greenberg [19] identifies: the dynamic-first approach (which starts from an untyped language and adds a type system to it) versus the static-first approach (which starts from a static type system and makes it safely interoperable with untyped code). It is fair to say that our approach aligns more naturally with the latter approach, as we are motivated by preservation of reasoning principles in a pre-existing static type system. Nevertheless, this does not mean that FAE is incompatible with the dynamic-first approach. For example, dynamic idioms like occurrence typing [57] are not necessarily contradictory with FAE; as long as the language under consideration offers useful abstractions, a gradual language can be expected to enforce them.

Correctness Properties for Gradual Type Systems We have already discussed the refined criteria of Siek et al. [52] for gradual typing in the introduction and we will not repeat this here. The proposal of Garcia and Tanter [18], stating that gradual languages should not just preserve type safety but type soundness, has also been discussed there.

New et al. [39] have proposed *graduality*, a semantic interpretation of the gradual guarantee [52]. While adherence to graduality could enforce new constraints on the gradual language, it is clear that in general, it does not imply FAE. Indeed, the gradualization of a total STLC does satisfy graduality [37] but its embedding is not fully abstract as explained in §2.5.2.

Greenman et al. [20] have proposed a Complete Monitoring property for gradual type systems. They define when a value is owned by a component and then require that the gradual language checks contracts whenever a value crosses a

component boundary. The property is intended to aid in debugging dynamic type errors by requiring correct blame assignment and to highlight the benefits of more precise but more costly strategies for tracking blame. As such, the property captures a different requirement than ours and as such, should be considered largely orthogonal.

Fully Abstract Compilation and Backtranslations The criterion of fully abstract compilation is a well-known concept in the field of secure compilation. In that setting, fully abstract compilation captures the requirement that a secure compiler should enforce source language abstractions. The concept itself and the many examples in the literature have been surveyed by Patrignani et al. [42]. Most closely related to our work are the results by Devriese et al. [10] and New et al. [38], who also use a universal type in the construction of a backtranslation.

In addition to compiler security, fully abstract compilers (translations) have also been used to study the expressiveness of program calculi [40]. In a certain sense, our criterion requires that the gradual language is equally expressive than the static language.

Logical Relations in Iris The program logic Iris [28] which we have used to prove the correctness of our backtranslation, is a versatile program logic used for different applications. Several projects have used Iris for modeling logical relations which inspired the construction of ours [15, 55, 56, 54].

2.7 Conclusion

In the introduction, we argued that the refined criteria of Siek et al. [52] are not a sufficient condition for a good gradual language. This has been argued before by Garcia and Tanter [18] and it is by now not a very controversial statement in the gradual typing community. Rather than requiring the preservation of a source-language-specific and designer-selected meaning of types, this chapter explores a generic and ambitious alternative, previously suggested by Devriese et al. [11]: the fully abstract embedding (FAE) of the static language into the gradual one. This chapter offers a first exploration; we have shown that the criterion is both useful (identifies problems in unsatisfactory gradualizations) and realistic (attainable and provable for real gradualizations). We believe that future gradual language designers should think about the criterion to evaluate the consequences of their design choices.

Chapter 3

Purity of an ST Monad

Full Abstraction by Semantically Typed Back-Translation

The contents of this chapter are taken from our publication with the same title [24], though it contains some additional notes and some changes were made to integrate it more nicely into this thesis. I – Koen Jacobs, the thesis student – am the first author of said paper. My advisors and co-authors Amin Timany and Dominique Devriese contributed ideas and guidance and contributed to the writing of the paper, but other work was performed by me.

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 chapter, 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.

3.1 Introduction

The ST monad Purity is one of the defining characteristics of programming languages like Haskell [21]. 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 [30] have proposed the `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):

$$\texttt{return} :: \forall s, a.\, a \to \texttt{ST}\ s\ a$$

$$(\gg\!=) :: \forall s, a, b.\, \texttt{ST}\ s\ a \to (a \to \texttt{ST}\ s\ b) \to \texttt{ST}\ s\ b$$

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

$$\texttt{readSTRef} :: \forall s, a.\, \texttt{STRef}\ s\ a \to \texttt{ST}\ s\ a$$

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

Computations in the `ST` monad have type $\texttt{ST}\ s\ a$ 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 $a \rightarrow$ `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

$$\texttt{do}\ x \leftarrow cmp1$$

$$cmp2$$

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:

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

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 [30] 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. [56] 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 chapter, 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 [2, 1], 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. 43, 40]. 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 [30] 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 [60]. 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 this chapter: 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 [38, 10, 43] (and Chapter 2). 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 chapter, 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 §3.6.

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 [23] in the Coq proof assistant on top of the Iris framework [28].

Overview chapter The remainder of the chapter is structured as follows. In §3.2, 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 §3.3, 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 §3.4, the second in §3.5. In the sections that follow, we have some discussions, go over related and future work, and conclude this chapter.

3.2 Preliminaries

3.2.1 Definitions of $\lambda^{\vdash}$ and $\lambda^{\vdash}_{ST}$

Here, we present $\lambda^{\vdash}$ and $\lambda^{\vdash}_{ST}$. Adhering to our color convention, we use a blue font for everything related to $\lambda^{\vdash}$ and a *red* one for its extension with `ST`: $\lambda^{\vdash}_{ST}$. 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, as treated already in Chapter 2. The latter is the former extended with the same `ST` monad that is defined by Timany et al. [56]. In essence, $\lambda^{\vdash}_{ST}$ 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 $\lambda^{\vdash}_{ST}$ in Figure 3.1. 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 $\lambda^{\vdash}_{ST}$, 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, to write programs in which we want to check equality between two locations, a compare primitive.

Note that all these additional constructors, except *runST* and *compare*, can form values. As such, *ST* computations can be thought of as scripts whose

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

Figure 3.1: Grammars of $\lambda^{\vdash}$ and $\lambda^{\vdash}_{ST}$. Common parts are left in black.

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 $\mathsf{e} \to \mathsf{e}'$. Evaluation steps in $\lambda^{\vdash}_{ST}$ are stateful however, requiring as an additional argument a heap, $h : \mathit{Loc} \rightharpoonup \mathit{Val}$, 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).

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

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

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

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

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.

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

The effectful head steps are defined below.

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

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

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

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 $return\ \ell$.

Lastly, we will introduce the typing rules. The ones for $\lambda^{\vdash}$ are of the form $\Gamma \vdash \mathtt{e} : \tau$ where we assume every type to be closed. We omit them here as well, as they are entirely standard [45]. Typing rules in $\lambda^{\vdash}_{ST}$ are of the form $\Xi \mid \Gamma \vdash e : \tau$, 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.

$$\frac{\Xi \mid \Gamma \vdash e : \mathit{STRef}\ X\ \tau \qquad \Xi \mid \Gamma \vdash e' : \mathit{STRef}\ X\ \tau}{\Xi \mid \Gamma \vdash \mathit{compare}\ e\ e' : B}$$

$$\frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash X}{\Xi \mid \Gamma \vdash \mathit{ref}\ e : \mathit{ST}\ X\ (\mathit{STRef}\ X\ \tau)} \qquad \frac{\Xi \mid \Gamma \vdash e : \mathit{STRef}\ X\ \tau}{\Xi \mid \Gamma \vdash\ !e : \mathit{ST}\ X\ \tau}$$

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

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

$$\frac{\Xi, X \mid \Gamma \vdash e : \mathit{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathit{runST}\ \{e\} : \tau}$$

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 $\mathit{ST}\ X\ \tau$ under the collection of free type variables Ξ, X (with this notation, we implicitly assume that $X \notin \Xi$). To be able to conclude that $\mathit{runST}\ \{e\}$ 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 $\lambda^{\vdash}$.

3.2.2 Defining the Intermediate Language $\lambda^{\vDash}$

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 [28] that we introduced in §2.4.3.

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.

$$\mathcal{V}_{int.}[\![1]\!](\mathsf{v},\mathsf{v}') \triangleq \mathsf{v} = () * \mathsf{v}' = ()$$

$$\mathcal{V}_{int.}[\![\mathsf{B}]\!](\mathsf{v},\mathsf{v}') \triangleq \exists b \in \{true, false\}.\, \mathsf{v} = b * \mathsf{v}' = b$$

$$\mathcal{V}_{int.}[\![\mathsf{Z}]\!](\mathsf{v},\mathsf{v}') \triangleq \exists z \in \mathbb{Z}.\, \mathsf{v} = z * \mathsf{v}' = z$$

$$\mathcal{V}_{int.}[\![\tau_1+\tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq \bigvee_{i\in\{1,2\}} \exists \mathsf{w},\mathsf{w}'.\, \mathsf{v} = \mathsf{inj}_i\ \mathsf{w} * \mathsf{v}' = \mathsf{inj}_i\ \mathsf{w}' * \mathcal{V}_{int.}[\![\tau_i]\!](\mathsf{w},\mathsf{w}')$$

$$\mathcal{V}_{int.}[\![\tau_1\times\tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq \exists \mathsf{v}_1,\mathsf{v}_1',\mathsf{v}_2,\mathsf{v}_2'.\, \mathsf{v} = (\mathsf{v}_1,\mathsf{v}_2) * \mathsf{v}' = (\mathsf{v}_1',\mathsf{v}_2') * \mathcal{V}_{int.}[\![\tau_1]\!](\mathsf{v}_1,\mathsf{v}_1') * \mathcal{V}_{int.}[\![\tau_2]\!](\mathsf{v}_2,\mathsf{v}_2')$$

$$\mathcal{V}_{int.}[\![\tau_1\to\tau_2]\!](\mathsf{v},\mathsf{v}') \triangleq \Box\left(\forall \mathsf{w},\mathsf{w}'.\, \mathcal{V}_{int.}[\![\tau_1]\!](\mathsf{w},\mathsf{w}') \mathrel{-\!\!*} \mathsf{lift}\ \mathcal{V}_{int.}[\![\tau_2]\!]\ (\mathsf{v}\ \mathsf{w},\mathsf{v}'\ \mathsf{w}')\right)$$

$$\mathcal{V}_{int.}[\![\mu\mathsf{X}.\tau]\!](\mathsf{v},\mathsf{v}') \triangleq \exists \mathsf{w},\mathsf{w}'.\, \mathsf{v} = \mathsf{fold}\ \mathsf{w} * \mathsf{v}' = \mathsf{fold}\ \mathsf{w}' * \triangleright\, \mathcal{V}_{int.}[\![\tau[\mu\mathsf{X}.\tau/\mathsf{X}]]\!](\mathsf{w},\mathsf{w}')$$

Figure 3.2: Type-indexed logical relations on values for intermediate language

Logical Relations on Values The value relations are defined in Figure 3.2. 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 $\mu\mathsf{X}.\tau$, the types are structurally smaller in the recursive calls. In the case for $\mu\mathsf{X}.\tau$, 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.

$$\mathsf{lift} : (\mathsf{Val} \to \mathsf{Val} \to \mathit{iProp}) \to (\mathsf{Expr} \to \mathsf{Expr} \to \mathit{iProp})$$

$$\mathsf{lift}\ \Phi\ (\mathsf{e}, \mathsf{e}') = \mathsf{wp}\ \mathsf{e}\ \{\mathsf{v}.\ \exists \mathsf{v}'.\ \mathsf{e}' \to^* \mathsf{v}' * \Phi(\mathsf{v}, \mathsf{v}')\}$$

Roughly, e and e' satisfy $\mathsf{lift}\ \Phi$, if when e evaluates to a value (we use the stuck version of the weakest precondition), say v, e' must do so as well, say to v', and we have furthermore that $\Phi(\mathsf{v}, \mathsf{v}')$.

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*.

$$\vec{\mathcal{V}}_{int.}[\![\varnothing]\!](\epsilon, \epsilon) \triangleq \mathsf{True}$$

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

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).

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

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 $\Gamma \vDash_{int.} \mathsf{e} \leq \mathsf{e}' : \tau$, we obtain two terms that are related at Γ' and τ', that is, we have $\Gamma' \vDash_{int.} \mathsf{C}[\mathsf{e}] \leq \mathsf{C}'[\mathsf{e}'] : \tau'$. We denote this by $\vDash_{int.} \mathsf{C} \leq \mathsf{C}' : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$.

Some Important Theorems Out of our definition of lift (and adequacy of weakest preconditions)[1], we get the following adequacy theorem, stating that our logical relations model refinement.

[1] 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.

Lemma 3.2.1 (Logical Relation Adequacy)**.** *If* $\cdot \vDash_{int.} \mathsf{e} \leq \mathsf{e}' : \tau$*, 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 $\Gamma \vDash_{int.} \mathsf{e} \leq \mathsf{e} : \tau$. The intermediate language, $\lambda^{\vDash}$ consists of all these *semantically* typed terms. We use $\Gamma \vDash_{int.} \mathsf{e} : \tau$ as a shorthand for $\Gamma \vDash_{int.} \mathsf{e} \leq \mathsf{e} : \tau$. Similarly, we write $\vDash_{int.} \mathsf{C} : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$ as a shorthand for $\vDash_{int.} \mathsf{C} \leq \mathsf{C} : (\Gamma; \tau) \Rightarrow (\Gamma'; \tau')$.

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

Theorem 3.2.2 (Fundamental Theorem Intermediate Language)**.** *For any well* syntactically *typed expression (in* $\lambda^{\vdash}$*), say* $\Gamma \vdash \mathsf{e} : \tau$*, we automatically have that* $\Gamma \vDash_{int.} \mathsf{e} : \tau$*.*

Virtual Steps The logical relation that we presented here suffers from a technical shortcoming that we hit upon in §3.4.3. 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 $\mathsf{list\ Z} \triangleq \mu\mathsf{X}.1+(\mathsf{Z} \times \mathsf{X})$ and the following two identity functions on it: the simple identity function $\lambda\mathsf{x}.\mathsf{x}$, and the function $\mathsf{f} \triangleq \mathsf{map}\ (\lambda\mathsf{x}.\mathsf{x} + 0)$ 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 $\mathsf{v} : \mathsf{list\ Z}$ 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 $\lambda\mathsf{x}.\mathsf{x}$ trivially terminates).

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

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

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

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

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 $[1, 2, 3, \mathsf{true}]$ that are related at $\mathsf{list}\ Z$ "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" (or unfolds), 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 $\mathsf{list}\ Z$ at a certain amount of unfolds 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 $\mathsf{list}\ Z$) at least up to the remaining amount of unfolds.

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 $f\ [1, 2, 3, \mathsf{true}]$ evaluates to a value as well. This is not true however as it will get stuck trying to evaluate $\mathsf{true} + 0$.

The logical relations (and others like it in the literature) are thus far from complete. But as mentioned, we may find ourselves (as in §3.4.3) 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 $\vdash\!\!\rightarrow(_)$, obtaining the expression $(\lambda x.\vdash\!\!\rightarrow(x))\ [1, 2, 3, 4]$, 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.

$$e ::= \ldots \mid \vdash\!\!\rightarrow(e)$$

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

$$\vdash\!\!\!\rightarrow(\mathsf{w}) \to_h \mathsf{w} \quad \textit{if} \quad \mathsf{w} = (), \mathsf{b}, \mathsf{z} \qquad \vdash\!\!\!\rightarrow(\mathsf{inj}_1\ \mathsf{v}) \to_h \mathsf{inj}_1\ (\vdash\!\!\!\rightarrow(\mathsf{v}))$$

$$\vdash\!\!\!\rightarrow(\mathsf{inj}_2\ \mathsf{v}) \to_h \mathsf{inj}_2\ (\vdash\!\!\!\rightarrow(\mathsf{v})) \qquad \vdash\!\!\!\rightarrow((\mathsf{v}_1, \mathsf{v}_2)) \to_h (\vdash\!\!\!\rightarrow(\mathsf{v}_1), \vdash\!\!\!\rightarrow(\mathsf{v}_2))$$

$$\vdash\!\!\!\rightarrow(\lambda \mathsf{x}.\mathsf{e}) \to_h \lambda \mathsf{y}.((\lambda \mathsf{x}.\vdash\!\!\!\rightarrow(\mathsf{e}))\ \vdash\!\!\!\rightarrow(\mathsf{y})) \qquad \vdash\!\!\!\rightarrow(\mathsf{fold}\ \mathsf{v}) \to_h \mathsf{fold}\ (\vdash\!\!\!\rightarrow(\mathsf{v}))$$

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) $\lambda \mathsf{x}.\vdash\!\!\!\rightarrow(\mathsf{x})$ relates to $\mathsf{map}\ (\lambda \mathsf{x}.\mathsf{x} + 0)$, and 2) $\lambda \mathsf{x}.\mathsf{x}$ relates to $\lambda \mathsf{x}.\vdash\!\!\!\rightarrow(\mathsf{x})$. 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 $\mathsf{list}\ \mathsf{Z}$. We can prove (2) as we can abuse the fact that for *any* possibly untyped value v, the expression $\vdash\!\!\!\rightarrow(\mathsf{v})$ 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 $\lambda^{\vDash}$. Practically however, we treat $\lambda^{\vdash}$ in this chapter as though it was defined by the same extended grammar while we leave the virtual step construct untyped.[2] 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 §3.2.1 (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 [55].

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 §3.4, much of our proof relies on complex relational reasoning, for which a unary relation does not seem amenable.

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

3.3 Our Main Theorem and Overview of the Proof

Formal Preliminaries Here, we state contextual equivalence for $\lambda^{\vdash}_{ST}$; the definition for $\lambda^{\vdash}$ is defined similarly as in Def. 2.2.2.

Definition 3.3.1 (Contextual Equivalence by Equitermination in $\lambda^{\vdash}_{ST}$)**.** Given two well-typed expressions in $\lambda^{\vdash}_{ST}$ under the same context and type, say $\Xi \mid \Gamma \vdash e_1 : \tau$ and $\Xi \mid \Gamma \vdash e_2 : \tau$, then e_1 and e_2 are *contextually equivalent*, written $\Xi \mid \Gamma \vdash e_1 \approx_{\text{ctx}} e_2 : \tau$, if for all well-typed contexts $\vdash C : (\Xi \mid \Gamma; \tau) \Rightarrow (\cdot \mid \cdot \,; 1)$, the following holds:

$$C[e_1] \Downarrow \quad \textit{iff} \quad C[e_2] \Downarrow$$

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

Formal Statements of Main Theorems Similarly to Chapter 2, we have our two main results as follows.

Theorem 3.3.1 (Reflection of Contextual Equivalence)**.** *Given any two well-typed expressions, $\Gamma \vdash \mathsf{e}_1 : \tau$ and $\Gamma \vdash \mathsf{e}_2 : \tau$, if their embeddings (we denote $\lceil\lceil _ \rceil\rceil : \lambda^{\vdash} \rightarrow \lambda^{\vdash}_{ST}$ the embedding from $\lambda^{\vdash}$ into $\lambda^{\vdash}_{ST}$) are contextually equivalent, that is $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil \mathsf{e}_1 \rceil\rceil \approx_{ctx} \lceil\lceil \mathsf{e}_2 \rceil\rceil : \lceil\lceil\tau\rceil\rceil$, then the original terms are equivalent ($\Gamma \vdash \mathsf{e}_1 \approx_{ctx} \mathsf{e}_2 : \tau$).*

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

Theorem 3.3.2 (Preservation of Contextual Equivalence)**.** *Given any two equivalent expressions, $\Gamma \vdash \mathsf{e}_1 \approx_{ctx} \mathsf{e}_2 : \tau$, then their embedding into $\lambda^{\vdash}_{ST}$ are still contextually equivalent; more specifically, we have that $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil \mathsf{e}_1 \rceil\rceil \approx_{ctx} \lceil\lceil \mathsf{e}_2 \rceil\rceil : \lceil\lceil\tau\rceil\rceil$.*

As was the case in Chapter 2, this is the most interesting theorem, and we will now outline its proof.

Outline Proof of Preservation of Equivalence The proof outline is similar as the one from Chapter 2. To prove Theorem 3.3.2, we show that contexts in $\lambda^{\vdash}_{ST}$ can be correctly emulated by contexts in $\lambda^{\vdash}$. This is formalized as follows:[3]

[3] As we will remark in §3.7, this theorem is actually stronger.

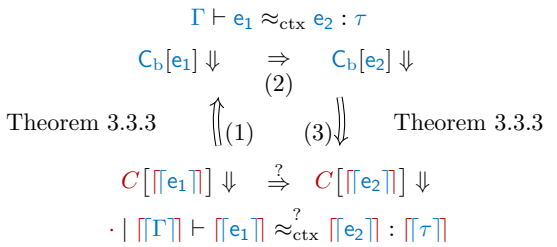


Figure 3.3: Theorem 3.3.3 implies Theorem 3.3.2.

Theorem 3.3.3 (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:*

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

To see why this theorem indeed implies preservation of equivalence, the reader can refer to Figure 3.3. 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 3.3.3, 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 3.3.4 (Correctly Emulating Stateful Contexts by Semantically Typed Contexts)**.** *Given an arbitrary context* C *in* $\lambda^{\vdash}_{ST}$ *that 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:*

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

Theorem 3.3.5 (Correctly Emulating Semantically Typed Contexts by Syntactically Typed Contexts)**.** *Given an arbitrary context* C *in* $\lambda^{\vDash}$ *that 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:*

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

We report on the proofs for these in §3.5 and §3.4 respectively.

In addition to the logical relation in §3.2.2, 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.

3.4 Correctly Back-Translating Semantically Typed Contexts

We split up the challenge of proving Theorem 3.3.5 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 $\vDash_{int.} \mathsf{C} : (\Gamma; \tau) \Rightarrow (\cdot\,; 1)$, we will

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

3.4.1 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, $\mathsf{e}_1\ \mathsf{e}_2$, is semantically typed at τ_2. Conversely though, if $\mathsf{e}\ \mathsf{e}'$ 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.

$$\langle\!\langle () \rangle\!\rangle = \mathsf{inject}_1\ () \qquad \langle\!\langle (\mathsf{e}_1,\mathsf{e}_2) \rangle\!\rangle = \mathsf{inject}_\times\ (\langle\!\langle \mathsf{e}_1 \rangle\!\rangle,\langle\!\langle \mathsf{e}_2 \rangle\!\rangle)$$

$$\langle\!\langle \mathsf{e}_1;\ \mathsf{e}_2 \rangle\!\rangle = \mathsf{extract}_1\ \langle\!\langle \mathsf{e}_1 \rangle\!\rangle;\ \langle\!\langle \mathsf{e}_2 \rangle\!\rangle \qquad \langle\!\langle \pi_1\ \mathsf{e} \rangle\!\rangle = \pi_1\ (\mathsf{extract}_\times\ \langle\!\langle \mathsf{e} \rangle\!\rangle)$$

$$\langle\!\langle \mathsf{x} \rangle\!\rangle = \mathsf{x} \qquad \langle\!\langle \mathsf{fold}\ \mathsf{e} \rangle\!\rangle = \mathsf{inject}_\mu\ (\mathsf{fold}\ \langle\!\langle \mathsf{e} \rangle\!\rangle)$$

$$\langle\!\langle \lambda \mathsf{x}.\mathsf{e} \rangle\!\rangle = \mathsf{inject}_\to\ (\lambda \mathsf{x}.\langle\!\langle \mathsf{e} \rangle\!\rangle) \qquad \langle\!\langle \mathsf{unfold}\ \mathsf{e} \rangle\!\rangle = \mathsf{unfold}\ (\mathsf{extract}_\mu\ \langle\!\langle \mathsf{e} \rangle\!\rangle)$$

$$\langle\!\langle \mathsf{e}_1\ \mathsf{e}_2 \rangle\!\rangle = \mathsf{extract}_\to\ \langle\!\langle \mathsf{e}_1 \rangle\!\rangle\ \langle\!\langle \mathsf{e}_2 \rangle\!\rangle \qquad \langle\!\langle \vdash\!\!\to(\mathsf{e}) \rangle\!\rangle = \langle\!\langle \mathsf{e} \rangle\!\rangle$$

Figure 3.4: Back-Translation of untyped terms into the universe (a selection of cases).

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

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

The general idea is not new and has been used before [38, 37, 10, 43] (and Chapter 2).

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

$$\mathsf{inject}_\mathsf{A} : \mathsf{A} \to \mathcal{U} \qquad \mathsf{inject}_\otimes : (\mathcal{U} \otimes \mathcal{U}) \to \mathcal{U} \qquad \mathsf{inject}_\mu : (\mu \mathsf{Y}.\mathcal{U}) \to \mathcal{U}$$

$$\mathsf{extract}_\mathsf{A} : \mathcal{U} \to \mathsf{A} \qquad \mathsf{extract}_\otimes : \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U}) \qquad \mathsf{extract}_\mu : \mathcal{U} \to (\mu \mathsf{Y}.\mathcal{U})$$

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 3.4. Note that the virtual step construct is just stripped away. The translation on contexts follows naturally.

Second Stage Given our semantically typed context, $\vDash_{int.} \mathsf{C} : (\mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_\mathsf{n} : \tau_\mathsf{n}; \tau) \Rightarrow (\ \cdot\ ; 1)$, it follows that $\vdash \langle\!\langle \mathsf{C} \rangle\!\rangle : (\mathsf{x}_1 : \mathcal{U}, \ldots, \mathsf{x}_\mathsf{n} : \mathcal{U}; \mathcal{U}) \Rightarrow (\ \cdot\ ; \mathcal{U})$. 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

$(\mathsf{x}_1 = \tau_1, \ldots, \mathsf{x}_n = \tau_n; \tau) \Rightarrow (\,\cdot\,; 1)$. 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).

$$\mathsf{embed}_{\mathsf{B}}^{\cdot} = \mathsf{inject}_{\mathsf{B}} \qquad \mathsf{project}_{\mathsf{B}}^{\cdot} = \mathsf{extract}_{\mathsf{B}}$$

$$\mathsf{embed}^{\Delta}_{\tau+\tau'} = \left|\begin{array}{l}\mathsf{inject}_+ \circ \\ \quad \mathsf{map}_+(\mathsf{embed}^{\Delta}_{\tau}, \mathsf{embed}^{\Delta}_{\tau'})\end{array}\right. \qquad \mathsf{project}^{\Delta}_{\tau+\tau'} = \left|\begin{array}{l}\mathsf{map}_+(\mathsf{project}^{\Delta}_{\tau}, \mathsf{project}^{\Delta}_{\tau'}) \circ \\ \quad \mathsf{extract}_+\end{array}\right.$$

$$\mathsf{embed}^{\Delta}_{\tau\times\tau'} = \left|\begin{array}{l}\mathsf{inject}_\times \circ \\ \quad \mathsf{map}_\times(\mathsf{embed}^{\Delta}_{\tau}, \mathsf{embed}^{\Delta}_{\tau'})\end{array}\right. \qquad \mathsf{project}^{\Delta}_{\tau\times\tau'} = \left|\begin{array}{l}\mathsf{map}_\times(\mathsf{project}^{\Delta}_{\tau}, \mathsf{project}^{\Delta}_{\tau'}) \circ \\ \quad \mathsf{extract}_\times\end{array}\right.$$

$$\mathsf{embed}^{\Delta}_{\tau\to\tau'} = \left|\begin{array}{l}\lambda \mathsf{f}.\, \mathsf{extract}_\to \\ \quad (\mathsf{embed}^{\Delta}_{\tau'} \circ \mathsf{f} \circ \mathsf{project}^{\Delta}_{\tau})\end{array}\right. \qquad \mathsf{project}^{\Delta}_{\tau\to\tau'} = \left|\begin{array}{l}\lambda \mathsf{f}. \\ \quad (\mathsf{project}^{\Delta}_{\tau'} \circ \mathsf{extract}_\to\ \mathsf{f} \circ \mathsf{embed}^{\Delta}_{\tau})\end{array}\right.$$

$$\mathsf{embed}^{\Delta}_{\mu\mathsf{X}.\tau} = \lambda \mathsf{x}.\pi_1\ ((\lambda \mathsf{y}.\ \mathsf{fix\ gen\ y})\ ())\ \mathsf{x} \qquad \mathsf{project}^{\Delta}_{\mu\mathsf{X}.\tau} = \lambda \mathsf{x}.\pi_2\ ((\lambda \mathsf{y}.\ \mathsf{fix\ gen\ y})\ ())\ \mathsf{x}$$

$$\mathsf{embed}^{\Delta}_{\mathsf{X}} = \lambda \mathsf{x}.\pi_1\ (\Delta(\mathsf{X})\ ())\ \mathsf{x} \qquad \mathsf{project}^{\Delta}_{\mathsf{X}} = \lambda \mathsf{x}.\pi_2\ (\Delta(\mathsf{X})\ ())\ \mathsf{x}$$

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.

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

The embed and 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.

$$\mathscr{EP}\langle\!\langle \mathsf{C} \rangle\!\rangle = \begin{cases} \mathsf{project}_1\ (\langle\!\langle \mathsf{C} \rangle\!\rangle[\mathsf{embed}_\tau\ \cdot\ ]) & \textit{if}\ \ \Gamma = \ \cdot \\ \mathsf{let}\ \mathsf{f} = (\lambda \mathsf{x}_1 \ldots \mathsf{x}_n . \cdot)\ \ \mathtt{in} \\ \mathsf{project}_1\big(\langle\!\langle \mathsf{C} \rangle\!\rangle\ [\mathsf{embed}_\tau\ (\mathsf{f}\ \ (\mathsf{project}_{\tau_1} \mathsf{x}_1) \ldots (\mathsf{project}_{\tau_n} \mathsf{x}_n))]\big) & \end{cases} \quad \dagger$$

$$\dagger\text{: } \textit{if}\ \Gamma = \mathsf{x}_1 : \tau_1, \ldots, \mathsf{x}_n : \tau_n$$

3.4.2 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.

The relation defined in §3.2.2 for the definition of our intermediate language actually satisfies this description so far by its adequacy result as stated in Lemma 3.2.1.

To actually be useful however, the relation has to contain the expressions whose refinement we are interested in; in our case these are $\mathsf{C}^{\vdash}_{\mathsf{b}}[\mathsf{e}]$ and $\mathsf{C}[\mathsf{e}]$. To prove this, we first define a new binary relation that associates terms with their untyped translation into the universe. Afterwards, we present Lemma 3.4.1 that bridges the gap between this new relation and the one from §3.2.2.

A high-level overview of the proof can be seen in the leftmost diagram of Figure 3.5. We have that $\langle\!\langle \mathsf{C} \rangle\!\rangle$ 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 3.4.2. Given Lemma 3.4.1, we can then derive that adding the wrappers $\mathscr{EP}$ to our context $\langle\!\langle \mathsf{C} \rangle\!\rangle$ will relate it to C in the relation that defines $\lambda^{\vDash}$. The syntactically typed term e is related to itself by Theorem 3.2.2. Using Lemma 3.2.1, 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 $\mathsf{injected}^{\mathsf{Val}}_{\mathsf{c}} : \mathsf{Val} \to \mathsf{Val}$, such that for all values v we have

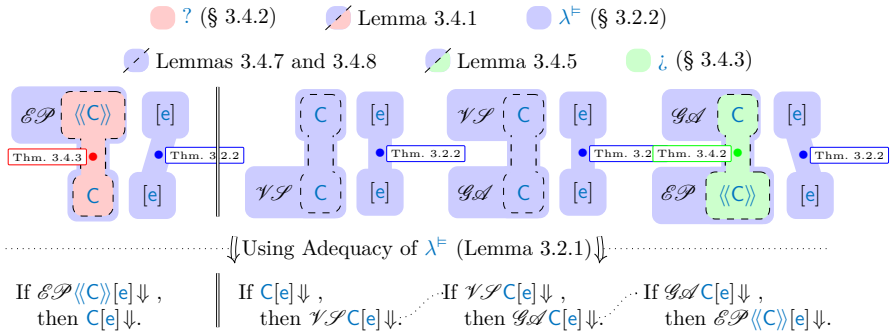


Figure 3.5: Proving equitermination of C[e] and $\mathscr{EP}\langle\!\langle C \rangle\!\rangle[e] \Downarrow$.

that $\mathsf{inject_c\ v} \to^* \mathsf{injected_c^{Val}(v)}$.

$$\mathcal{V}_? : \mathit{Val} \to \mathit{Val} \to \mathit{iProp}$$

$$\begin{aligned}
\mathcal{V}_?(\mathsf{v},\mathsf{v}') = {} & \big(\mathsf{v} = \mathsf{injected_1^{Val}}(()) * \mathsf{v}' = ()\big) \\
& \vee \big(\exists b \in \{\mathit{true}, \mathit{false}\}.\ \mathsf{v} = \mathsf{injected_B^{Val}}(\mathsf{b}) * \mathsf{v}' = \mathsf{b}\big) \\
& \vee \big(\exists z \in \mathbb{Z}.\ \mathsf{v} = \mathsf{injected_Z^{Val}}(\mathsf{z}) * \mathsf{v}' = \mathsf{z}\big) \\
& \vee \Big(\exists \mathsf{w},\mathsf{w}'.\ \bigvee_{i\in\{1,2\}} \big(\mathsf{v} = \mathsf{injected_+^{Val}}(\mathsf{inj_i\ w}) * \mathsf{v}' = \mathsf{inj_i\ w}' * \triangleright\, \mathcal{V}_?(\mathsf{w},\mathsf{w}')\big)\Big) \\
& \vee \big(\exists \mathsf{v}_1,\mathsf{v}_1',\mathsf{v}_2',\mathsf{v}_2'.\ \mathsf{v} = \mathsf{injected_\times^{Val}}((\mathsf{v}_1',\mathsf{v}_2')) * \mathsf{v}' = (\mathsf{v}_1',\mathsf{v}_2') * \\
& \qquad \triangleright\, \mathcal{V}_?(\mathsf{v}_1,\mathsf{v}_1') * \triangleright\, \mathcal{V}_?(\mathsf{v}_2,\mathsf{v}_2')\big) \\
& \vee \big(\exists \mathsf{e}.\ \mathsf{v} = \mathsf{injected_\to^{Val}}(\lambda \mathsf{x}.\mathsf{e}) * \\
& \qquad \triangleright\, \square(\forall \mathsf{w},\mathsf{w}'.\ \mathcal{V}_?(\mathsf{w},\mathsf{w}') \mathrel{-\!\!*} \mathsf{lift}\ \mathcal{V}_?\ (\mathsf{e}[\mathsf{w}/\mathsf{x}], \mathsf{v}'\ \mathsf{w}'))\big) \\
& \vee \big(\exists \mathsf{w},\mathsf{w}'.\ \mathsf{v} = \mathsf{injected_\mu^{Val}}(\mathsf{fold\ w}) * \mathsf{v}' = \mathsf{fold\ w}' * \triangleright\, \mathcal{V}_?(\mathsf{w},\mathsf{w}')\big)
\end{aligned}$$

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 $\mathcal{E}_?(\mathsf{e},\mathsf{e}') = \mathsf{lift}\ \mathcal{V}_?\ (\mathsf{e},\mathsf{e}')$.

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 $\mathsf{x}_1, \ldots, \mathsf{x}_n$ as follows:

$$\mathcal{O}_?^{\mathsf{x}_1,\ldots,\mathsf{x}_n}(\mathsf{e},\mathsf{e}') = \forall \vec{\mathsf{v}}, \vec{\mathsf{v}}', |\vec{\mathsf{v}}| = |\vec{\mathsf{v}}'| = n.\ \ast_{0\le i<n} \mathcal{V}_?(\vec{\mathsf{v}}.i, \vec{\mathsf{v}}'.i) \vdash \mathcal{E}_?(\mathsf{e}[\vec{\mathsf{v}}/\vec{\mathsf{x}}], \mathsf{e}'[\vec{\mathsf{v}}'/\vec{\mathsf{x}}])$$

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

Lemma 3.4.1 (Connective Lemma)**.**

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

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

Theorem 3.4.2 (Fundamental Theorem)**.** *Given any untyped expression* e *with free variables* $\mathsf{x}_1, \ldots, \mathsf{x}_n$*, then we have* $\mathcal{O}_?^{\mathsf{x}_1,\ldots,\mathsf{x}_n}(\langle\!\langle \mathsf{e} \rangle\!\rangle, \mathsf{e})$.

3.4.3 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 3.5, utilizing two new intermediate contexts, $\mathscr{GA}\mathsf{C}$ and $\mathscr{VS}\mathsf{C}$. We need to do a bit more work compared to what was needed in §3.4.2 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 §3.4.2; as such, it has the following fundamental theorem.

Theorem 3.4.3 (Fundamental Theorem)**.** *Given any untyped expression* e *with free variables* $\mathsf{x}_1, \ldots, \mathsf{x}_n$*, then we have* $\mathcal{O}_¿^{\mathsf{x}_1,\ldots,\mathsf{x}_n}(\mathsf{e}, \langle\!\langle \mathsf{e} \rangle\!\rangle)$.

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 $\mathsf{wp}\ \mathsf{e}\ \{\Phi\}$ 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 3.4.1, we might now try to prove the following:

Lemma 3.4.4 (Connective Lemma (False!))**.**

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

The lemma does not hold up sadly. The one on the left is clearly false, as we can consider e.g. $\mathcal{E}_{\not\iota}(6, \mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6))$, while the following is *not true*: $\mathcal{E}_{int.}[\![\mathsf{B}]\!](6, \mathsf{project}_{\mathsf{B}}\ (\mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6)))$, 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 $\mathcal{V}_{int.}[\![\mathsf{B} \to \mathsf{B}]\!](\lambda \mathsf{x}.\mathsf{x}, \mathsf{f})$ 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 $\mathsf{guard}/\mathsf{assert}$

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 3.4.5 (Connective Lemma)**.**

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

This allows us to fix the previous two examples. Consider again $\mathcal{E}_{¿}(6, \mathsf{injected}_{\mathsf{Z}}^{\mathsf{Val}}(6))$, 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 $\mathcal{V}_{¿}(\mathsf{guard}_{\mathsf{B}\to\mathsf{B}}\ \lambda\mathsf{x}.\mathsf{x}, \mathsf{embed}_{\mathsf{B}\to\mathsf{B}}\ \mathsf{f})$. 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 C[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.

$$\mathscr{GA}\,\mathsf{C} = \begin{cases} \mathsf{assert}_1\ (\mathsf{C}[\mathsf{guard}_{\tau}\ \cdot\ ]) & \textit{if } \Gamma = \ \cdot \\ \mathtt{let}\ \mathsf{f} = (\lambda \mathsf{x}_1 \ldots \mathsf{x}_n.\,\cdot)\ \mathtt{in} \\ \mathsf{assert}_1\big(\mathsf{C}\,[\mathsf{guard}_{\tau}\ (\mathsf{f}\ \ (\mathsf{assert}_{\tau_1}\mathsf{x}_1)\ \ldots\ (\mathsf{assert}_{\tau_n}\mathsf{x}_n))]\big) & \textit{if } \Gamma = \mathsf{x}_1:\tau_1,\ldots,\mathsf{x}_n:\tau_n \end{cases}$$

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 3.4.6 (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 §3.2.2.

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

Lemma 3.4.7 (Adding Virtual Steps & Guards/Asserts)**.**

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

Lemma 3.4.8 (Virtual Steps as Non-operators)**.**

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

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 3.5, 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 3.2.1.

For the first refinement, we know that C, being semantically well typed, is related to itself by definition. Moreover, we can prove, using Lemma 3.4.8, that adding the $\mathscr{VS}$ wrapper to the left-hand side does not change this fact. Lastly, e is related to itself by Theorem 3.2.2.

Similarly, we know that for the second refinement, C is related to itself. Likewise, we can prove, using Lemma 3.4.7, 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 3.4.3. Lemma 3.4.5 now constitutes the core of the proof that adding the $\mathscr{GA}$ and $\mathscr{EP}$ wrappers will transfer this relatedness to relatedness in $\lambda^{\vDash}$.

3.5 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 $\lambda^{\vdash}_{ST}$ contexts to $\lambda^{\vDash}$. We split up Theorem 3.3.4 into the following four smaller objectives. Consider an arbitrary $\vdash C : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\ \cdot \mid \cdot\ ; 1)$, we will

1. define $\mathsf{C}^{\vDash}_{\mathrm{b}}$, its *semantically* typed emulation in $\lambda^{\vDash}$,
2. prove that $\mathsf{C}^{\vDash}_{\mathrm{b}}$ is actually semantically typed; we have $\vDash_{int.} \mathsf{C}^{\vDash}_{\mathrm{b}} : (\Gamma; \tau) \Rightarrow (\cdot\ ; 1)$,

3. prove that for all $\Gamma \vdash \mathsf{e} : \tau$, if $C[\lceil\lceil \mathsf{e} \rceil\rceil] \Downarrow$, then $\mathsf{C}^{\vDash}_{\mathsf{b}}[\mathsf{e}] \Downarrow$, and conversely
4. prove that if $\mathsf{C}^{\vDash}[\mathsf{e}] \Downarrow$, then $C[\lceil\lceil \mathsf{e} \rceil\rceil] \Downarrow$.

In §3.5.1, §3.5.2, and §3.5.3, 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.

3.5.1 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 $\mathcal{E}$.

$$\mathsf{read}\ \mathsf{z}\ \mathcal{E}(\vec{\mathsf{v}}) \to^* (\mathcal{E}(\vec{\mathsf{v}}) \quad ,\vec{\mathsf{v}}.\mathsf{z}) \quad \textit{if } 0 \leq \mathsf{z} < |\vec{\mathsf{v}}|$$

$$\mathsf{ref}\ \mathsf{v}\ \mathcal{E}(\vec{\mathsf{v}}) \to^* (\mathcal{E}(\vec{\mathsf{v}} +\!\!+ [\mathsf{v}]), |\vec{\mathsf{v}}|)$$

$$\mathsf{write}\ \mathsf{z}\ \mathsf{v}\ \mathcal{E}(\vec{\mathsf{v}}) \to^* (\mathcal{E}(\vec{\mathsf{v}}[\mathsf{z} \mapsto \mathsf{v}]), ()) \quad \textit{if } 0 \leq \mathsf{z} < |\vec{\mathsf{v}}|$$

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.

3.5.2 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 $(\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot\,; 1)$), its typing derivation can contain them *internally*. To prove that the $\langle\!\langle C \rangle\!\rangle$ 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:

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

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

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 $ST\ X\ \tau$, 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).

$$OwnStates_\gamma(\vec{v}, \vec{v}') \vdash \Rrightarrow OwnStates_\gamma(\vec{v} +\!\!+ [v], \vec{v}' +\!\!+ [v']) * |\vec{v}| \mapsto_\gamma (v, v') \tag{3.1}$$

$$OwnStates_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \vec{v}.z = v * \vec{v}'.z = v' \tag{3.2}$$

$$OwnStates_\gamma(\vec{v}, \vec{v}') * z \mapsto_\gamma (v, v') \vdash \Rrightarrow OwnStates_\gamma(\vec{v}[z \mapsto w], \vec{v}'[z \mapsto w']) * z \mapsto_\gamma (w, w') \tag{3.3}$$

Equation 3.1 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 3.2 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 3.3).

Let's now go over (a first iteration of) the value relation (still a bit naive) for $\mathit{STRef}\ X\ \tau$.

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

Two values are related at $\mathit{STRef}\ X\ \tau$, 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). [4] 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 3.1 or Equation 3.3 for instance, and proceed to close it by providing different witnesses for the existential quantifier in the invariant.

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

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

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

[4] These intuitions are formally captured in the definition of the notion of weakest precondition that uses the fancy update modality, as explained in §2.4.3.

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 $\Delta(X)$ for our ghost state when we relate values at ST and STRef. The final definitions are as follows.[5]

$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathit{STRef}\ X\ \tau]\!]_\Delta(\mathsf{v}, \mathsf{v}') \triangleq \exists z.\, \mathsf{v} = z * \mathsf{v}' = z * \boxed{\exists \mathsf{w}, \mathsf{w}'.\, z \mapsto_{\Delta(X)} (\mathsf{w}, \mathsf{w}') * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta(\mathsf{w}, \mathsf{w}')}^{\Delta(X).z}$$

$$\mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \mathit{ST}\ X\ \tau]\!]_\Delta(\mathsf{v}, \mathsf{v}') \triangleq \forall \vec{\mathsf{v}}_i, \vec{\mathsf{v}}'_i.\ \square \Big(\mathit{OwnStates}_{\Delta(X)}(\vec{\mathsf{v}}_i, \vec{\mathsf{v}}'_i) \twoheadrightarrow \mathsf{wp}\, \mathsf{v}\ \mathcal{E}(\vec{\mathsf{v}}_i) \Big\{ (\mathcal{E}(\vec{\mathsf{v}}_f), \mathsf{w}).$$
$$\exists \vec{\mathsf{v}}'_f, \mathsf{w}'.\, (\mathsf{v}'\ \mathcal{E}(\vec{\mathsf{v}}'_i) \to^* (\mathcal{E}(\vec{\mathsf{v}}'_f), \mathsf{w}'))\ * \mathit{OwnStates}_{\Delta(X)}(\vec{\mathsf{v}}_f, \vec{\mathsf{v}}'_f) * \mathcal{V}_{\mathcal{X}}[\![\Xi \vdash \tau]\!]_\Delta(\mathsf{w}, \mathsf{w}') \Big\} \Big)$$

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 (§3.2.2).

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

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:

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

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:

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

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

[5] 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. [28].

Theorem 3.5.1 (Fundamental Theorem)**.** *Given a typed expression in* $\lambda^{\vdash}_{ST}$*, say* $\Xi \mid \Gamma \vdash e : \tau$*, we have the following:*

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

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. [28].

3.5.3 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 $\lambda^{\vdash}_{ST}$, 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.

$$\mathsf{lift}_{\mathcal{R}} : (Val \to \mathsf{Val} \to iProp) \to (Expr \to \mathsf{Expr} \to iProp)$$

$$\mathsf{lift}_{\mathcal{R}}\ \Phi\ (e, \mathsf{e}') = \mathsf{wp}\ e\ \{v.\ \exists \mathsf{v}'.\ \mathsf{e}' \to^{*} \mathsf{v}' * \Phi(v, \mathsf{v}')\}$$

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 $OwnHeap(_)$, (taking as argument a heap, i.e. some $h : Loc \rightharpoonup Val$) 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 $OwnHeap(_)$ 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 $OwnState_{\gamma} : List\ \mathsf{Val} \to iProp$ 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, $\mathit{OwnLocs}_\gamma : \mathit{List}\ \mathit{Loc} \to \mathit{iProp}$ taking a list of locations, and finally $_ \mapsto^{\square}_\gamma _ : \mathbb{N} \to \mathit{Loc} \to \mathit{iProp}$ 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.

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) \vdash \Rrightarrow \mathit{OwnLocs}_\gamma(\vec{\ell} +\!\!+ [\ell]) * |\vec{\ell}| \mapsto^{\square}_\gamma \ell \tag{3.4}$$

$$\mathit{OwnLocs}_\gamma(\vec{\ell}) * z \mapsto^{\square}_\gamma \ell \vdash \vec{\ell}.z = \ell \tag{3.5}$$

$$\gamma \mapsto^{\square}_z \ell \vdash \gamma \mapsto^{\square}_z \ell * \gamma \mapsto^{\square}_z \ell \tag{3.6}$$

$$\gamma \mapsto^{\square}_z \ell * \gamma \mapsto^{\square}_z \ell' \vdash \ell = \ell' \tag{3.7}$$

Equation 3.4 and Equation 3.5 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 3.6), and as such its value can never change (Equation 3.7). Once we have allocated a new location in our list with Equation 3.4, the location at that index is immutable.

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

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

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, $\Delta(X).1$ and $\Delta(X).2$. 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 $\Delta(X).1$). 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 $\Delta(X).1$.

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 $\ell \mapsto w$ 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 $\Delta(X).2$.

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.

$$\mathcal{V}_{\mathcal{R}}[\![\Xi \vdash ST\ X\ \tau]\!]_\Delta(v, \mathsf{v}') \triangleq$$

$$\forall \vec{\ell_i}, \vec{\mathsf{v}_i}.\ |\vec{\ell_i}| = |\vec{\mathsf{v}_i}|.\ \square \Big(\mathit{OwnLocs}_{\Delta(X).1}(\vec{\ell_i}) * \mathit{OwnState}_{\Delta(X).2}(\vec{\mathsf{v}_i}) \mathrel{-\!\!*}$$

$$\mathsf{wp}\ \mathit{runST}\ \{v\} \Big\{ w.\ \exists \mathsf{w}', \vec{\ell_f}, \vec{\mathsf{v}_f}.\ |\vec{\ell_f}| = |\vec{\mathsf{v}_f}|.(\mathsf{v}'\ \mathcal{E}(\vec{\mathsf{v}_i}) \to^* (\mathcal{E}(\vec{\mathsf{v}_f}), \mathsf{w}')) *$$

$$\mathit{OwnLocs}_{\Delta(X).1}(\vec{\ell_f}) * \mathit{OwnState}_{\Delta(X).2}(\vec{\mathsf{v}_f}) * \mathcal{V}_{\mathcal{R}}[\![\Xi \vdash \tau]\!]_\Delta(w, \mathsf{w}') \Big\} \Big)$$

Similar to §3.5.2, the relation is extended to open terms. We write $\Xi \mid \Gamma \vDash_{\mathcal{R}} e \leq \mathsf{e}' : \tau$ for e and e' related under Ξ, Γ, and τ. Likewise, the relation satisfies a fundamental theorem.

Theorem 3.5.2 (Fundamental Theorem)**.** *Given a typed expression in $\lambda^{\vdash}_{ST}$, say $\Xi \mid \Gamma \vdash e : \tau$, we have the following:*

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

As importantly, we have the following adequacy lemma.

Lemma 3.5.3 (Logical Relation Adequacy)**.** *If $\cdot \mid \cdot \vDash_{\mathcal{R}} e \leq \mathsf{e}' : \tau$, 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 $\Gamma \vdash \mathsf{e} : \tau$ and $\vdash C : (\cdot \mid \lceil\lceil\Gamma\rceil\rceil; \lceil\lceil\tau\rceil\rceil) \Rightarrow (\cdot \mid \cdot\ ; 1)$. We need to prove that if $C[\lceil\lceil \mathsf{e} \rceil\rceil]$ halts, then so does $\langle\!\langle C \rangle\!\rangle[\mathsf{e}]$. Using Lemma 3.5.3, it suffices to prove that $C[\lceil\lceil \mathsf{e} \rceil\rceil]$ is related to $\langle\!\langle C \rangle\!\rangle[\mathsf{e}]$. The context C is related to $\langle\!\langle C \rangle\!\rangle$ by (the generalization to contexts of) Theorem 3.5.2, so it remains to be proven that $\lceil\lceil \mathsf{e} \rceil\rceil$ is related to e. This follows by the same theorem however, as we have that $\cdot \mid \lceil\lceil\Gamma\rceil\rceil \vdash \lceil\lceil \mathsf{e} \rceil\rceil : \lceil\lceil\tau\rceil\rceil$ and $\langle\!\langle \lceil\lceil \mathsf{e} \rceil\rceil \rangle\!\rangle = \mathsf{e}$.

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.

3.6 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 $\lambda^{\vdash}_{ST}$ terms. To demonstrate this, let us consider again the commutativity example from Timany et al. [56],

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

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. [56]. 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. [56].

We should add still that Timany et al. [56] proves commutativity for all terms, *even those that are typed at* $\texttt{ST}$*-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.

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

$$e_2 \triangleq \Lambda X_1.\, \Lambda X_2.\, \lambda x_1 : 1 \to X_1.\, \lambda x_2 : 1 \to X_2.\, .(x_1\ (), x_2\ ())$$

Do we *need* a two step back-translation? Some readers might still wonder "do you really *need* to define a back-translation in two phases, using an intermediate semantically-typed language?". *Technically*, you probably do not. The only point we want to make though is that our approach is surprisingly convenient; in contrast, a more "direct" approach – at least in the ways we can think of – might actually become very convoluted, making it near to impossible in practice.

To illustrate this point, let us flesh out how we think such a back-translation would look like. As we don't see any alternative, we are assuming that we are still back-translating expressions of state thread computations to some kind of state monad type, i.e. some type $H \to H \times \tau$, where H grows in size as we allocate more references.

It seems most natural to go for some kind of list, as we don't really know up front how many locations we will be allocated. But list of what type? Well, we want to be able to allocate references of any type. So we could try to go for a list of universes, i.e. $H \triangleq \mathit{list}\, \mathcal{U}$. Note however that now, upon reading and writing to our heap, we will have to respectively project out of and embed into the universe. Moreover, our back-translation will have to be type-directed, to insert to the right $\mathsf{project}_\tau$ and embed_τ. Even if this seems quite complex already, it might still be manageable. But note that we haven't talked about higher-order references, nor the fact that we can also allocate state computations themselves! That is, we should be able to append stuff of $H \to H \times \tau$ into $\mathit{list}\, \mathcal{U}$; we thus need to wrap $H \to H \times \tau$ into $\mathcal{U}$ and upon reading unwrap it again. Note also that even if one is able to write down a not-obviously-wrong back-translation (these wrappers will get complicated very easily), one is still left to prove its correctness. The latter would involve lots of the same techniques that we have used but in a much more convoluted setting.

Alternatively, we could try to define a family of approximate back-translations, supporting only a finite number of allocations, or a finite number reference types; we would then try proving that we can find a back-translation which is "precise" enough for some particular run. Aside from the problems above, this will bring with it its own challenges, and we foresee such an approach very difficult as well.

Closer to Haskell We have chosen a simplified language to which we added ST. This allowed us to isolate the problem, but as such, there is some distance between our formalization and the addition of ST to the Haskell programming language.

Adding type-polymorphism is a first difference whose challenges we discuss in §3.8. Other differences include the fact that we are working in a call-by-value language, as opposed to the call-by-need one and that we don't have e.g. the infinite lists that are made possible by it. While we do think that said features are orthogonal to the problem statement, it won't necessarily be easy to incorporate them into our formalization as merely formalizing a call-by-need evaluation strategy is non-trivial.

3.7 Related Work

Related to Stateful Encapsulation Launchbury and Peyton Jones [30] introduced an `ST` monad for the encapsulation of state in an otherwise pure programming language. In §3.1, we discussed how our work is related to and inspired by this work.

Moggi and Sabry [35] 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 [35], the programming language we consider is simpler. However, the results that we establish are much stronger. In particular, Moggi and Sabry [35] 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. [56] quote Moggi and Sabry [35] 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, $\lambda^{\vdash}_{ST}$ 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 $\lambda^{\vdash}_{ST}$ 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*[6], 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. [38] and Devriese et al. [10] and later used again by Patrignani et al. [43] (and in Chapter 2). Using logical relations to prove correctness of the back-translation is very common [42].

As mentioned in the introduction, using full abstraction to compare the expressive power over different languages has been done before [e.g., 40, 43]. 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. [53].

Other criteria from the secure compilation community might also be suitable to model the preservation of purity. As mentioned before, Theorem 3.3.3 is actually stronger than Theorem 3.3.2, and corresponds to the strongest criteria in [3].

[6]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.

Purity Sabry [48] motivate the need to formally define "purity" to know *what* one needs to prove when one adds mutable state to a purely functional programming language. They analyze different properties, including the notion we used here: the preservation of the contextual equivalences when embedding a known pure language into some extension.

They make the case however that the preservation of contextual equivalence (upon an embedding from an intuitively-"pure" language into some target) is not a good property to characterize "purity" of the target language. They argue this with a counter example: the embedding of PCF into PCF_{por} – two languages they assume to be pure – is *not* fully abstract.

We grant the point they are making here; language abstractions can be broken in ways that intuitively don't seem to involve any violation of "purity".

In the end though, we don't think the consideration of full abstraction of the embedding into the ST-extended language is erroneous. Suppose that this embedding were not fully abstract; then it would still violate existing reasoning principles (even though they might not be violated necessarily because of "purity").

Moreover, maybe purity can be seen as more of a relational account? Extending PCF into PCF_{por} or adding a fixpoint operator to a total STLC, gives us embeddings that are not fully abstract; but maybe there is something to be said for the view that both target languages are still pure, but not *as pure* as the respective source languages?

3.8 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 $\text{F}_{\texttt{ST}}$ 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 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 $F_{\mu,ST}$, we think we could simply back-translate polymorphic functions to thunked functions ($\langle\!\langle \Lambda.e \rangle\!\rangle = \lambda_.\langle\!\langle e \rangle\!\rangle$) and type instantiation with application to unit ($\langle\!\langle e[\tau] \rangle\!\rangle = \langle\!\langle e \rangle\!\rangle\ ()$) as by Theorem 3.3.5 (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 3.3.5 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. [38] 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 3.3.5, the proof might be more easily refactored, as polymorphic functions can just be backtranslated to thunked functions.

In Chapter 2, we have proven full abstraction for embedding a simply-typed lambda calculus with recursive types into the cast calculus of its gradual counterpart. By Theorem 3.3.5, this proof might be refactored more easily, as e.g. a downward cast could just be implemented using an assert from §3.4.3.

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 [30]'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.

3.9 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 ($\lambda^{\vdash}_{ST}$) 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, $\lambda^{\models}$, 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.

Chapter 4

Conclusions

In this thesis, we considered two language extensions: gradualizing static typing disciplines and adding ST to a Haskell-like language. We explained why upon developing these extensions, one should hope to preserve the initial language guarantees. We demonstrated however that unfortunately the preexisting literature answers this question only partially.

To answer this question more fully, we have set out to ask whether the embedding into these extensions is fully abstract. That is, whether contextually equivalent terms in the initial language are still equivalent in the presence of contexts in the extended language; a generic but strong property as in a given language, contextual equivalences capture many of its reasoning principles. Practically, it means for instance that programmers can reason about their code in the initial language just like they were used to – without having to worry about the new features in the extension – and that compiler optimizations on initial code fragments can simply be reused.

For gradual typing, we studied this criterion and demonstrated its usefulness. We showed that *failure* of the criterion can exclude undesirable gradualizations. We argued that it can also offer a moment of contemplation, raising whether different design decisions can hope to recover the property and if not, that perhaps a compromise is justifiable and that failure of the criterion makes it formally explicit that some reasoning of the static language is lost. Lastly, we proved it to be realistic by reporting on a proof for the gradualization of the simply-typed lambda calculus with iso-recursive types, formalized in the Coq proof assistant using the Iris program logic.

Regarding the addition of the ST monad to Haskell, we modelled this problem

in a simplified context: adding an ST monad to a call-by-value simply-typed lambda calculus with iso-recursive types. We reported on a mechanized proof for the property in the Coq proof assistant using the Iris program logic. The proof exhibits two novel features that might interest researchers studying related topics. The first of which is a two-part back-translation form the target language into an intermediate semantically-typed language, and from the latter to the syntactically-typed source language. The second of which is the introduction of virtual-steps, allowing us to prove certain contextual refinements that are not logically related.

More generally, we have contributed to the understanding of fully abstract embeddings as a way to encode the intuition that the initial language properties are not lost upon extension. Moreover, even though full abstraction of an embedding is a very strong property (not many embeddings satisfy it), we have presented additional evidence that it is realistic and – more importantly – that there exist *interesting* language extensions that do nevertheless satisfy it: the gradualization of STLC_μ that gradualizes only its static typing system and its extension with mutable state via an `ST` interface. As such, these results provide additional testimony to the fact that full abstraction of an embedding is not necessarily "too restrictive" for language designers that wish to extend their language while maintaining its abstractions.

Lastly, we have also shown that using a suitable program logic, mechanized proofs of fully abstract compilation are realistic.

To conclude, we mention a couple of interesting directions to look further into.

Other properties One of the impressions we are left with is that we have developed a lot of machinery out of which we have "only" proven full abstraction and the stronger Theorems 2.4.1 and 3.3.3 – named "robust relational hyperproperty preservation" by Abate et al. [3] – of our embeddings. Our intuition tells us however that – for both extensions we considered – there are stronger and more intuitive properties that we can prove with (modifications to) the technical machinery we developed. In the next couple of paragraphs, we will sketch out a couple. For each of these, we think it would be interesting to flesh them out more fully, prove them for the extensions considered in this thesis, check their relation to each other and the existing properties from literature, check which other compilers from literature may satisfy these, and consider whether they are interesting as generic properties and if so, when they should be considered (with respect to which design goals).

For instance, we think that – for both extensions – we could prove the following.

Conjecture 4.0.1. For every term in the extended language typed at a source

interface, say $\lceil\lceil\Gamma\rceil\rceil \vdash e : \lceil\lceil\tau\rceil\rceil$, there exists a source term typed at that interface, say $\Gamma \vdash e'$, whose embedding is contextually equivalent, i.e. $\lceil\lceil\Gamma\rceil\rceil \vdash e \approx_{ctx} \lceil\lceil e'\rceil\rceil : \lceil\lceil\tau\rceil\rceil$.

We foresee that the machinery for our back-translations would allow us to define these source terms, and that an equivalent property would hold for contexts: target contexts at source interfaces are equivalent to their back-translated source contexts (where we define equivalent contexts as contexts that equi-behave after plugging them in with the same input). We think the implications of this conjecture are interesting, and that they should be explored in detail to consider whether this is also a noteworthy proposition to consider in general (for other embeddings or compilations).

The above properties are very strong. Alternatively, one might want to generalize the notion of contextual equivalence to reason about target terms more directly. We start out by defining the notion of – say – a "pattern", in essence, it is a regular context (with any arbitrary outer interface) but with n numbered holes each at their own interface. One can then consider the equivalence of two patterns that have the same outer interface, the same amount of holes, and the same interfaces at each of said holes; we define them to be equivalent if after arbitrary substitution of a correspondingly well-typed n-tuple of terms in the holes, we obtain two terms (typed at the outer interface of the patterns) that are contextually equivalent. One can then consider the compilation or embedding of such patterns, and ask whether equivalence between patterns are preserved; note that in the target language, such compiled patterns are now substituted by arbitrary *target* terms. Preservation of equivalence between patterns (or likewise, refinement of patterns) would allow certain reasoning to be more convenient. The reader may remember how we derived preservation of commutativity (at source interfaces) in §3.6, by constructing a contextual equivalence with closures; alternatively, we might derive this more directly, by considering the following pattern, presumably equivalent in the pure language, and obtain its preservation in the presence of ST:

$$\texttt{let}\ x = [\ \cdot^2]\ \texttt{in}\ ([\ \cdot^1], x) \approx ([\ \cdot^1], [\ \cdot^2])$$

Lastly, we observe also that neither full abstraction nor the properties defined above directly talk about target *interfaces*. In some scenarios it might make sense to reason more directly about these though; restricting their behavior in some particular sense. As such, we might try to generalize patterns by allowing meta-variables in the types of their interfaces that are to be substituted by arbitrary types (after which substitution of correspondingly well-typed expressions happen). Preservation of such generalized patterns then entails

arbitrary substitution by target types. Proving such a result will undoubtedly be more difficult however, and it is not even clear when this would be a desirable property. Note also that closely related to this are fully abstract translations with a parametric source language like System F, as such translations already compile quantified interfaces that – depending on how they are translated – could be instantiated with arbitrary types in the target language.

Relation between Syntactic Typing and Semantic Typing Another interesting direction to look into is the relation between a syntactically typed language and its semantically typed counterparts. Many typed languages contain unsafe features that in essence, bypass the type system. The documentation of these features contain warnings and ask programmers to use them only with much care.

Often, there are no formal proposals for what exactly the acceptable use of such features are, with the exception we know of being Rust [27]. Rather, it is loosely defined by the collective intuition of the users of that language, and it may even depend on the situation. Let us consider now a formalization that takes the form of a semantically-typed counterpart to the language.

Depending on the purpose of these features, this semantically-typed language will of course look differently. If the use of such features is merely to express oneself more *easily* or *efficiently*, while otherwise maintaining all language abstractions from the syntactically typed language, then, we might wonder whether we can prove full abstraction (and other properties) into this semantically typed extension.

In a simplified context, we might already try to investigate the possibilities. We might consider interesting semantically-typed counterparts to different source languages, e.g. $\mathrm{STLC}_{\mu}^{Ref}$ (STLC_{μ} with higher-order mutable state), or some variant of System F, and see whether we can obtain full abstraction (and stronger properties) results into these counterparts. Aside from the direct implications (programmers can prove their untyped code to semantically typed and be assured that they are not violating any language abstractions), such results could also prove helpful to obtain full abstraction results of translations from the syntactically typed language into any other language; a semantically-typed back-translation suffices.

Possible use cases of the latter include gradualizations of our source language; expressions in a cast calculus could be back-translated to untyped pairs out of which the first component is merely representing our evidence. Moreover, we can consider a language extension with existential types that witness any *semantic* type (type predicate allowed by the logical relation). Packing constructions

could simply be back-translated by merely thunking them.

Elaborating on Virtual Steps Thirdly, we think it is useful to elaborate on the use of virtual-steps introduced in §3.2.2. We might want to get rid of the virtual step primitive first, and replace it by an appropriately generalized version of the wp predicate in Iris. Adequacy of this generalized version would depend on the fact that for a terminating program, each intermediate step will be of finite depth. This would solve the problem we encountered much more elegantly, and it would significantly lower its adaptation cost.

Also, we think that the same approach can be generalized to prove refinements that depend on the finiteness of locations that are allocated at any time. As such, we might want to further generalize the wp such that its adequacy will additionally depend on the fact that for a terminating program, the amount of allocated locations in a heap (during each step of evaluation) is bound. Moreover, we think that this might be very helpful to prove for instance full abstraction of $\mathrm{STLC}_{\mu}^{Ref}$ embedded into an interesting semantically-typed counterpart.

Semantic Typing, Gradual Typing and System F Finally, a last direction we want to investigate is the relation between a gradual extension and a semantically-typed extension in the specific context of some variant of System F as a source language. This relation can be witnessed by the following observation: given any untyped expression that semantically satisfies some interface, we could try to encode it in the gradual language by embedding every subexpression to the unknown type and wrapping the resulting term in ascriptions to make it of the right type.

As such, we believe that the problem of back-translating a semantically-typed counterpart of some variant of System F is closely related to developing a gradualization that satisfies the fully abstract embedding criterion. Moreover, it might be helpful to study both problems in parallel. One could start out for instance by taking inspiration on the cast calculus proposed by Labrada et al. [29], as their notion of plausible sealing might need to be reflected in a back-translation.

Bibliography

[1] M. Abadi. “Protection in Programming-Language Translations”. In: *Secure Internet Programming*. Springer-Verlag, 1999. ISBN: 3-540-66130-1. DOI: 10.1007/BFb0055109.

[2] M. Abadi. “Protection in Programming-Language Translations: Mobile Object Systems”. en. In: *European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, July 1998, pp. 291–291. ISBN: 978-3-540-65460-5 978-3-540-49255-9. DOI: 10.1007/3-540-49255-0_70.

[3] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault. “Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation”. In: *Computer Security Foundations Symposium*. June 2019. DOI: 10.1109/CSF.2019.00025.

[4] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. “Blame for All”. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’11. Austin, Texas, USA: Association for Computing Machinery, 2011, pp. 201–214. ISBN: 9781450304900. DOI: 10.1145/1926385.1926409. URL: https://doi.org/10.1145/1926385.1926409.

[5] A. Ahmed, D. Jamner, J. G. Siek, and P. Wadler. “Theorems for Free for Free: Parametricity, with and without Types”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Aug. 2017), 39:1–39:28. DOI: 10.1145/3110283.

[6] A. A. de Amorim, M. Fredrikson, and L. Jia. “Reconciling Noninterference and Gradual Typing”. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS ’20. Saarbrücken, Germany: Association for Computing Machinery, 2020, pp. 116–129. ISBN: 9781450371049. DOI: 10.1145/3373718.3394778. URL: https://doi.org/10.1145/3373718.3394778.

[7] A. W. Appel and D. McAllester. “An Indexed Model of Recursive Types for Foundational Proof-Carrying Code”. In: *ACM Trans. Program. Lang. Syst.* 23.5 (Sept. 2001), pp. 657–683. ISSN: 0164-0925. DOI: 10.1145/504709.504712. URL: https://doi.org/10.1145/504709.504712.

[8] F. Bañados Schwerter, R. Garcia, and É. Tanter. “A Theory of Gradual Effect Systems”. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’14. Association for Computing Machinery, Aug. 2014, pp. 283–295. ISBN: 978-1-4503-2873-9. DOI: 10.1145/2628136.2628149.

[9] T. Coquand and G. Huet. “The calculus of constructions”. PhD thesis. INRIA, 1986.

[10] D. Devriese, M. Patrignani, and F. Piessens. “Fully-abstract compilation by approximate back-translation”. In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 164–177. DOI: 10.1145/2837614.2837618.

[11] D. Devriese, M. Patrignani, and F. Piessens. “Parametricity versus the universal type”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 38. DOI: 10.1145/3158126.

[12] D. Devriese, M. Patrignani, and F. Piessens. “Two Parametricities versus Three Universal Types”. Submitted to the Journal of the ACM. 2020. URL: http://soft.vub.ac.be/~dodevrie/poly-seal-no-j-201910.pdf.

[13] D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. “Modular, Fully-Abstract Compilation by Approximate Back-Translation”. In: *Logical Methods in Computer Science* 13.4 lmcs:4011 (Oct. 2017). DOI: 10.23638/LMCS-13(4:2)2017. arXiv: 1703.09988 [cs.PL].

[14] D. Dreyer, A. Ahmed, and L. Birkedal. “Logical Step-Indexed Logical Relations”. In: *2009 24th Annual IEEE Symposium on Logic In Computer Science*. 2009, pp. 71–80. DOI: 10.1109/LICS.2009.34.

[15] D. Frumin, R. Krebbers, and L. Birkedal. “ReLoC: A mechanised relational logic for fine-grained concurrency”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018, pp. 442–451. DOI: 10.1145/3209108.3209174.

[16] R. Garcia, A. M. Clark, and É. Tanter. “Abstracting gradual typing”. In: *ACM SIGPLAN Notices*. Vol. 51. 1. ACM. 2016, pp. 429–442. DOI: 10.1145/2914770.2837670.

[17] R. Garcia and É. Tanter. “Deriving a Simple Gradual Security Language”. In: *arXiv preprint arXiv:1511.01399* (2015).

[18] R. Garcia and É. Tanter. “Gradual Typing as if Types Mattered”. Workshop on Gradual Typing. 2020.

[19] M. Greenberg. "The Dynamic Practice and Static Theory of Gradual Typing". In: *Summit on Advances in Programming Languages (SNAPL)*. Ed. by B. S. Lerner, R. Bodík, and S. Krishnamurthi. Vol. 136. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 6:1–6:20. ISBN: 978-3-95977-113-9. DOI: 10.4230/LIPIcs.SNAPL.2019.6.

[20] B. Greenman, M. Felleisen, and C. Dimoulas. "Complete Monitors for Gradual Types". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), 122:1–122:29. DOI: 10.1145/3360548.

[21] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. "A History of Haskell: Being Lazy with Class". In: *History of Programming Languages*. 2007.

[22] A. Igarashi, P. Thiemann, Y. Tsuda, V. T. Vasconcelos, and P. Wadler. "Gradual Session Types". en. In: *Journal of Functional Programming* 29 (2019). ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796819000169.

[23] K. Jacobs, D. Devriese, and A. Timany. *Artifact - Purity of an ST Monad*. 2022. DOI: 10.5281/zenodo.6329773.

[24] K. Jacobs, D. Devriese, and A. Timany. "Purity of an ST Monad: Full Abstraction by Semantically Typed Back-Translation". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527326. URL: https://doi.org/10.1145/3527326.

[25] K. Jacobs, A. Nuyts, and D. Devriese. "How to Do Proofs: Practically Proving Properties about Effectful Programs' Results (Functional Pearl)". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 1–13. ISBN: 9781450368155. DOI: 10.1145/3331554.3342603. URL: https://doi.org/10.1145/3331554.3342603.

[26] K. Jacobs, A. Timany, and D. Devriese. "Fully Abstract from Static to Gradual". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434288. URL: https://doi.org/10.1145/3434288.

[27] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.

[28] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018).

[29] E. Labrada, M. Toro, É. Tanter, and D. Devriese. “Plausible Sealing for Gradual Parametricity”. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527314. URL: https://doi.org/10.1145/3527314.

[30] J. Launchbury and S. L. Peyton Jones. “Lazy Functional State Threads”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI ’94. ACM, 1994, pp. 24–35. ISBN: 089791662X. DOI: 10.1145/178243.178246.

[31] N. Lehmann and É. Tanter. “Gradual Refinement Types”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, 2017, pp. 775–788. ISBN: 9781450346603. DOI: 10.1145/3009837.3009856. URL: https://doi.org/10.1145/3009837.3009856.

[32] K. Maillard, M. Lennon-Bertrand, N. Tabareau, and É. Tanter. “A Reasonably Gradual Type Theory”. In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547655. URL: https://doi.org/10.1145/3547655.

[33] J. Matthews and A. Ahmed. “Parametric Polymorphism through Run-Time Sealing or, Theorems for Low, Low Prices!” In: vol. 4960. LNCS. 2008, pp. 16–31. DOI: 10.1007/978-3-540-78739-6_2.

[34] J. C. Mitchell. “On abstraction and the expressive power of programming languages”. In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1991, pp. 290–310.

[35] E. Moggi and A. Sabry. “Monadic Encapsulation of Effects: A Revised Approach (Extended Version)”. In: *Journal of Functional Programming* 11.6 (Nov. 2001), pp. 591–627. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796801004154.

[36] J. J. H. Morris. “Lambda-calculus models of programming languages.” PhD thesis. Massachusetts Institute of Technology, 1969.

[37] M. S. New and A. Ahmed. “Graduality from Embedding-Projection Pairs”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236768. URL: https://doi.org/10.1145/3236768.

[38] M. S. New, W. J. Bowman, and A. Ahmed. “Fully Abstract Compilation via Universal Embedding”. In: *International Conference on Functional Programming*. ACM, 2016, pp. 103–116. DOI: 10.1145/2951913.2951941.

[39] M. S. New, D. Jamner, and A. Ahmed. “Graduality and Parametricity: Together Again for the First Time”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), 46:1–46:32. DOI: 10.1145/3371114.

[40] J. Parrow. "Expressiveness of Process Algebras". en. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory (LIX 2006) 209 (Apr. 2008), pp. 173–186. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.011.

[41] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. "Secure Compilation to Protected Module Architectures". In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015), 6:1–6:50. ISSN: 0164-0925. DOI: 10.1145/2699503.

[42] M. Patrignani, A. Ahmed, and D. Clarke. "Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work". In: *ACM Comput. Surv.* 51.6 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3280984. URL: https://doi.org/10.1145/3280984.

[43] M. Patrignani, E. M. Martin, and D. Devriese. "On the Semantic Expressiveness of Recursive Types". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434302.

[44] C. Paulin-Mohring. *Introduction to the calculus of inductive constructions*. 2015.

[45] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[46] J. C. Reynolds. "Types, Abstraction, and Parametric Polymorphism". In: *Information Processing*. North Holland, 1983, pp. 513–523.

[47] J. G. Riecke. "Fully abstract translations between functional languages". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1991, pp. 245–254.

[48] A. Sabry. "What is a purely functional language?" In: *Journal of Functional Programming* 8.1 (1998), pp. 1–22.

[49] I. Sergey and D. Clarke. "Gradual Ownership Types". en. In: *Programming Languages and Systems*. Springer, Berlin, Heidelberg, Mar. 2012, pp. 579–599. DOI: 10.1007/978-3-642-28869-2_29.

[50] J. G. Siek and W. Taha. "Gradual typing for functional languages". In: *Scheme and Functional Programming Workshop*. Vol. 6. 2006, pp. 81–92.

[51] J. G. Siek. *GitHub - jsiek/gradual-typing-in-agda: Formalizations of Gradually Typed Languages in Agda*. https://github.com/jsiek/gradual-typing-in-agda. (Accessed on 10/18/2019). 2019.

[52] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. "Refined Criteria for Gradual Typing". In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 274–293. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.274. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5031.

[53] L. Skorstengaard, D. Devriese, and L. Birkedal. "StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 19:1–19:28. ISSN: 2475-1421. DOI: 10.1145/3290332.

[54] A. Timany and L. Birkedal. "Mechanized Relational Verification of Concurrent Programs with Continuations". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341709. URL: https://doi.org/10.1145/3341709.

[55] A. Timany, R. Krebbers, and L. Birkedal. "Logical relations in Iris". In: *CoqPL, Date: 2017/01/21-2017/01/21, Location: Paris*. 2017.

[56] A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. Birkedal. "A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of RunST". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158152. URL: https://doi.org/10.1145/3158152.

[57] S. Tobin-Hochstadt and M. Felleisen. "The Design and Implementation of Typed Scheme". In: *ACM SIGPLAN Notices* 43.1 (Jan. 2008), pp. 395–406. ISSN: 0362-1340. DOI: 10.1145/1328897.1328486.

[58] M. Toro, R. Garcia, and É. Tanter. "Type-Driven Gradual Security with References". In: *ACM Trans. Program. Lang. Syst.* 40.4 (Dec. 2018), 16:1–16:55. ISSN: 0164-0925. DOI: 10.1145/3229061.

[59] M. Toro, E. Labrada, and É. Tanter. "Gradual Parametricity, Revisited". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 17:1–17:30. ISSN: 2475-1421. DOI: 10.1145/3290330.

[60] P. Wadler. "Monads for functional programming". In: *Program Design Calculi*. Ed. by M. Broy. Springer Berlin Heidelberg, 1993, pp. 233–264. ISBN: 978-3-662-02880-3.

[61] P. Wadler. "Theorems for free!" In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 347–359.

[62] P. Wadler and R. B. Findler. "Well-typed programs can't be blamed". In: *European Symposium on Programming*. Springer. 2009, pp. 1–16.

[63] *What is Coq? | The Coq Proof Assistant*. [Online; accessed 11. Oct. 2022]. Sept. 2022. URL: https://coq.inria.fr/about-coq.

[64] A. K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness". en. In: *Information and Computation* 115.1 (Nov. 1994), pp. 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093.

List of Publications

K. Jacobs, A. Nuyts, and D. Devriese. "How to Do Proofs: Practically Proving Properties about Effectful Programs' Results (Functional Pearl)". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*. TyDe 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 1–13. ISBN: 9781450368155. DOI: 10.1145/3331554.3342603. URL: https://doi.org/10.1145/3331554.3342603

K. Jacobs, A. Timany, and D. Devriese. "Fully Abstract from Static to Gradual". In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434288. URL: https://doi.org/10.1145/3434288

K. Jacobs, D. Devriese, and A. Timany. "Purity of an ST Monad: Full Abstraction by Semantically Typed Back-Translation". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022). DOI: 10.1145/3527326. URL: https://doi.org/10.1145/3527326



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
https://www.distrinet.cs.kuleuven.be