Radboud University Nijmegen

Faculty of Science

Verified Translation of Guarded Programs

Thesis MSc Computing Science

Author:
David Läwen
s1105105

Supervisor and first assessor:
dr. Robbert Krebbers

Co-supervisor:
Bálint Kocsis

Second assessor:
prof. dr. Herman Geuvers

15th September 2025

Abstract

Systems based on the Curry-Howard correspondence are rendered unsound by diverging terms; as such, self-referential definitions are typically restricted to ensure termination. For coinductive types, this involves checking *productivity*, i.e., that any finite prefix of the data can be computed in finite time. Syntactic conditions (such as Rocq's productivity checker) can be used, but are known to reject valid productive definitions.

A more flexible approach due to Nakano [2000] is to enforce productivity through a *guarded* typing discipline. We consider a guarded extension of the simply-typed lambda calculus, based on the work of Clouston et al. [2016], which features the so-called 'later' type former $\blacktriangleright$. The system admits unrestricted self-references (including negative ones), provided that a definition of type τ refers to itself only 'later', at type $\blacktriangleright\tau$.

Executing these guarded programs efficiently is an open problem; at the same time, the $\blacktriangleright$ type former and its associated term formers appear essential only to type checking, suggesting that we can erase them while retaining the strong guarantees they provide.

We present a program translation from the guarded language to untyped λ-calculus, for which we verify preservation of program behaviour, including productivity. To show that our source language enforces productivity, we give a mechanised proof of strong normalisation, an existing result from the literature. Both results are proved using a logical relation that interprets types in a step-indexed logic, and are machine-checked using the Rocq Prover and Iris framework.

Contents

Chapter 1

Introduction

As a programming language feature, self-referential definitions pose an obvious source of non-termination. In systems based on the Curry-Howard correspondence, diverging terms are associated with proofs of falsity; as such, self-referential definitions typically underlie certain restrictions to ensure termination. When operating on inductive types, *termination checking* typically involves a syntactic criterion, demanding that recursion (via self-references) occurs strictly on subcomponents of the input data. Such a *structurally recursive* definition terminates, as the finite input decreases in size with each step.

When dealing with *coinductive* types, data may be infinitely large, and the relevant notion becomes that of *productivity*, which dually requires that the output *increases* in size with each step. To illustrate, let us consider the coinductive type of streams over natural numbers, with a single constructor '::' (cons), and the following two stream instances:

$$\begin{aligned} \mathsf{toggle} &\triangleq 1 :: 0 :: \mathsf{toggle} \\ \mathsf{interleave}\ (x :: xs)\ ys &\triangleq x :: \mathsf{interleave}\ ys\ xs \end{aligned}$$

Here, toggle describes the sequence $1, 0, 1, 0, 1, 0, \ldots$ ad infinitum, and interleave generates the sequence featuring elements from two given streams in alternation. Both toggle and interleave are productive, as they generate at least one element of the output stream prior to recursing. More specifically, both definitions feature a self-reference on the right only under the constructor '::'.

Syntactic productivity checks. The latter characterisation gives rise to a syntactic criterion similar to that for inductive types: syntactic productivity checking [Coquand, 1993; Giménez, 1994], as used e.g. by the Rocq Prover, demands that when generating instances of coinductive types, recursive calls must be nested under a constructor call, i.e. '::' in the case of streams. While the above two definitions satisfy this property, syntactic checks are known to reject many valid (i.e., productive) definitions in the presence of higher-order functions, even leading to work on 'coding around' productivity checking [Danielsson, 2010].

An example for a productive definition that gets rejected is the so-called *regular paperfolding sequence*. It describes the left and right folds (encoded as 1 and 0, respectively) that arise when repeatedly folding a piece of paper in the same direction, starting with

1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, . . . , and can be defined using toggle and interleave:

$$\mathsf{paperfolds} \triangleq \mathsf{interleave}\ \mathsf{toggle}\ \mathsf{paperfolds}$$

While the above definition is productive, the following version, which appears to be well-formed, does not satisfy productivity:

$$\mathsf{paperfolds}' \triangleq \mathsf{interleave}\ \mathsf{paperfolds}'\ \mathsf{toggle}$$

To see why, we can try to obtain the first element of both paperfolds and paperfolds′ by unfolding their definitions until we obtain an expression of the form $h :: t$. For paperfolds, replacing toggle with $1 :: 0 ::$ toggle lets us unfold the definition of interleave, giving us 1 as the first element of the stream. In the case of paperfolds′, the first argument to interleave is not toggle, but paperfolds′ itself. Unfolding the definition of paperfolds′ merely yields the same pattern we started with, and never an occurrence of cons (::). Hence, we cannot compute the head of paperfolds′, or any finite prefix for that matter. Syntactic productivity checking fails to identify the difference between the two definitions, causing both versions to be rejected.

Productivity via types. A more flexible approach, originally due to Nakano [2000], is to enforce productivity through the typing discipline, rather than syntactically. A new type modality is introduced to distinguish at the type level between data available now, and data which can only be accessed later. Subsequent work has employed guarded type systems both as logics [Appel et al., 2007; Birkedal and Møgelberg, 2013] and operationally [Clouston et al., 2016; Abel and Vezzosi, 2014; Severi, 2019].

To illustrate, let us consider the work of Clouston et al. [2016], who present the *guarded lambda calculus*, or gλ-calculus. Their system extends the simply-typed lambda calculus with guarded recursive types using Nakano's modality, which they refer to as 'later'—following Appel et al. [2007]—and denote with ▶.

In recursive types, all self-references must be *guarded* by an occurrence of ▶. Returning to our example of streams, the type of guarded streams over some type τ is given by the guarded recursive equation

$$\mathsf{Str}^{\mathsf{g}}\ \tau \triangleq \tau \times \blacktriangleright \mathsf{Str}^{\mathsf{g}}\ \tau.$$

The two components of the product type correspond to the head and tail which make up a stream instance, meaning that $h :: t$ is encoded as a pair, and a stream as an infinite right-deep nesting of pairs. Crucially, $\mathsf{Str}^{\mathsf{g}}\ \tau$ appears under a ▶ on the right-hand side, guarding the self-reference. While the head of a stream has type τ and is thus available now, the tail has type $\blacktriangleright \mathsf{Str}^{\mathsf{g}}\ \tau$, meaning it can only be accessed after one time step.

The type of toggle is $\mathsf{Str}^{\mathsf{g}}\ \mathbf{N}$, i.e. a stream over natural numbers, and interleave has type $\mathsf{Str}^{\mathsf{g}}\ \tau \to \blacktriangleright \mathsf{Str}^{\mathsf{g}}\ \tau \to \mathsf{Str}^{\mathsf{g}}\ \tau$; here, the second stream is only required 'later', as its elements occur exclusively in the tail of the output. The first input's type cannot be wrapped with ▶ though, as it provides the head of the output.

In term definitions, self-references must occur at a later type. As such, paperfolds is typeable, since the self-reference occurs in the second argument of interleave and thus at type $\blacktriangleright \mathsf{Str}^{\mathsf{g}}\ \mathbf{N}$, while paperfolds′ is rejected.

Productive programming. In contrast with earlier work, the gλ-calculus of Clouston et al. describes the reduction of terms with an operational semantics. Their system can thus be used not only as a logic, but also as a programming language, in the sense that programs can be evaluated. But while Clouston et al. consider the reduction of programs formally, an open question is how gλ-programs might be run efficiently in practice.

At the same time, gλ's later type former $\blacktriangleright$ and associated term-level connectives appear essential only to the guarded typing discipline; operationally, their role lies merely in delaying reduction of parts of the program, an effect that can easily be achieved without the need for exotic term formers.

These observations lead us to ask whether we can erase the guarded term formers from gλ-programs, thus translating them to the familiar setting of λ-calculus while preserving the semantics of programs, including the productivity guarantees of gλ.

Overview of our contributions. Our work addresses the above points as follows:

(1) We present a program translation from a subset of the gλ-calculus targeting the untyped, call-by-value λ-calculus.

(2) We verify that the translation to untyped λ-calculus preserves the semantics of source programs, and thus also their productivity, using a *logical relations* argument, which is fully mechanised in the Rocq Prover using the Iris framework.

(3) To show that our source language satisfies productivity, we give a mechanised proof of strong normalisation, an existing result from the literature. Here, we improve upon prior work in our abstract treatment of gλ's later type former $\blacktriangleright$.

As such, we demonstrate that the guarded constructs of gλ can indeed be erased while maintaining productivity, as **(2)** proves a bi-implication between termination of the source and of the target, with matching result values (for base types).

The subset of gλ that we consider is that satisfying *causality*, where outputs of functions cannot depend on data deeper within the input. This restriction, which naturally arises from the system with $\blacktriangleright$, is lifted by Clouston et al. via another type modality, called 'constant' and written $\blacksquare$. The constant type former is beyond the scope of our work.

We choose untyped call-by-value λ-calculus as our target for **(1)** as it closely resembles existing languages and intermediate representations used in the (verified) compilation of functional code, such as OCaml's `Lambda`, via the Malfunction compilation target wrapper [Dolan, 2016], or (untyped) CakeML [Kumar et al., 2014].

Our proof for **(3)** uses a unary logical relation, which additionally serves as an intermediate step towards the binary logical relation we use for **(2)**. In both logical relations proofs, we construct semantic models via the operational semantics of our source (and target) language by interpreting the types of gλ to predicates in a *step-indexed* higher-order logic. In doing so, we closely follow the 'logical approach' (as identified by Timany et al. [2024]): we use semantic models built over an operational semantics in combination with step-indexing [Appel and McAllester, 2001] to model recursive types, as well as *weakest preconditions* [Dijkstra, 1975], which aid mechanised reasoning about program executions in a proof assistant.

For machine-checked developments following the 'logical approach', the separation logic framework *Iris* [Krebbers et al., 2017; Jung et al., 2018b] for the Rocq Prover has emerged as the de facto standard during the past decade. Our mechanisation uses the *Iris Proof Mode* [Krebbers et al., 2017] and the logic *siProp* defined in the Iris framework, allowing us to work in an embedded step-indexed object logic while enjoying context visualisation and proof tactics closely resembling those available in native Rocq. However, we use neither the Iris logic, nor do we employ *separation logic* (as typically seen in the logical approach), as we reason only about *pure* languages with no notion of resources.

The Rocq sources of our mechanisation are available online [Läwen, 2025].

Abstract step-indexing. The modality of Nakano has also proven fruitful in a distinct line of work as an abstraction mechanism for *step-indexing*. Step-indexing is a tool for stratifying circular, non-well-founded constructions. Objects, such as types or propositions, are successively approximated by a sequence indexed by non-negative integers. The i-th element of such a sequence typically expresses an approximation of the object that is valid for i steps of computation, hence the name *step-index*. Well-foundedness can then be ensured by restricting self-references to strictly smaller indices.

Pioneered by Appel and McAllester [2001] to model recursive types, subsequent work has shown that step-indexing is highly versatile, and can be used to build models and *logical relations* over operational semantics to handle many advanced language features not (easily) accounted for in denotational models, such as higher-order state [Ahmed et al., 2002; Ahmed, 2004], control effects [Dreyer et al., 2012], concurrency [Svendsen and Birkedal, 2014], and substructural types [Jung et al., 2018a; Dang et al., 2020].

While powerful, the technique of step-indexing leads to tedious and error-prone index arithmetic when reasoning directly in the model (see e.g. Dreyer et al. [2011]). Fortunately, Appel et al. [2007] found a suitable abstraction for step-indexed reasoning in the modality of Nakano, which they introduce as the modal operator $\triangleright$ (also called 'later'), the counterpart of the type former $\blacktriangleright$ under the Curry-Howard correspondence. The later modality $\triangleright$ internalises step-indexing in the logic, where it can be used to express propositions whose truth depends on the underlying step-index: intuitively, the proposition $\triangleright P$ holds at index $n + 1$ if P holds at n.

The later modality of Appel et al. has subsequently become the abstraction of choice for step-indexing in many program logics. In particular, the later modality $\triangleright$ is used in Iris, and the *siProp* logic, allowing us to reason about gλ's type former $\blacktriangleright$ abstractly. It is thus the later modality that lets us avoid explicitly indexed constructions, as seen in existing strong normalisation proofs for gλ.

Strong normalisation of gλ. Our result in **(3)** is not novel as such, since Clouston et al. [2016] give a pen-and-paper proof of strong normalisation for the gλ-calculus, using a denotational semantics in the *topos of trees*. They show adequacy of the denotational semantics using a logical relation between denotations and syntactic terms, where strong normalisation falls out as a corollary. Abel and Vezzosi [2014] previously mechanised a proof of strong normalisation for a similar guarded language in Agda, but we do contribute the first mechanised proof for the gλ-calculus of Clouston et al., albeit without

the 'constant' modality ■ and associated term formers.

The proofs of Clouston et al. and Abel and Vezzosi use logical relations over a denotational semantics, where explicit indexing is present both in the semantics and in the logical relation itself. We interpret types based on the operational semantics, and the logic of step-indexed propositions *siProp* affords us abstract step-indexing via the later modality ▷, to which we interpret the ► type former of our object language.

Outline. The strong normalisation proof for our guarded source language introduces the key constructions that we re-use in the binary logical relation for verifying the translation to our untyped target language. Accordingly, the remainder of the present work is structured as follows:

- In §2, we introduce the syntax, operational semantics, and type system of our source language, a subset of Clouston et al.'s gλ-calculus.
- We prove strong normalisation of our source language in §3. First, we introduce our base logic of step-indexed propositions (*siProp*), and derive a program logic for reasoning about terms of our source language. We then show strong normalisation with a logical relations argument, interpreting types as predicates in *siProp*.
- Our untyped target language and the translation of terms to it are presented in §4.
- The correctness proof for the program translation follows in §5, where we again use *siProp* as our base logic, but derive a new program logic for reasoning about terms of the source and target alongside each other. We now use a binary logical relation between source and target terms, indexed by the types of the source language. From the logical relation, we can conclude a form of behavioural equivalence, allowing us to prove semantic preservation for our program translation.
- In §6, we cover aspects of our formal development, that is, the mechanisation of our results in the Rocq prover using the Iris framework.
- We discuss related work in §7. There, we compare our work to similar step-indexed logical relations, and the proofs of strong normalisation given by Clouston et al. and Abel and Vezzosi. We also place our verified translation in the broader context of compiler verification efforts, in particular those using logical relations, and touch on approaches for recovering non-causal, productive functions.
- Finally, we conclude in §8 with possible improvements to our work, and outlook on how our verified translation might be extended to write and run guarded programs.

Chapter 2

The Guarded Lambda Calculus

Before we discuss compilation of guarded programs and the corresponding compiler correctness proof, we will first introduce the language under consideration, a subset of the gλ-calculus [Clouston et al., 2016]. In §2.1, we point out some key features of the gλ-calculus, relating them back to the examples we saw in §1. We then proceed with a formal description of the term syntax, operational semantics, and type system in §§2.2 and 2.3.

2.1 Key Features

In §1, we saw guarded streams over a type τ with the following guarded self-referential definition:

$$\mathsf{Str}^{\mathsf{g}}\ \tau \triangleq \tau \times \blacktriangleright \mathsf{Str}^{\mathsf{g}}\ \tau.$$

In the gλ-calculus, self-referential types are expressed by *iso-recursive* types, written as $\mu\alpha.\tau$, and the equation form above thus becomes

$$\mu\alpha.\ \tau \times \blacktriangleright\alpha.$$

Here, we immediately see the two main ways in which gλ extends STLC: recursive types, along with the later type modality $\blacktriangleright$. As in the equation form, the self-reference (here in the bound occurrence of α) appears under a $\blacktriangleright$, i.e. is *guarded*. The system enforces guardedness through a well-formedness restriction on types, specifically on the recursive type former μ. The later type modality is introduced by the novel term former `next`, which takes an expression of type τ to type $\blacktriangleright\tau$. Crucially, there is no general elimination form or function that takes $\blacktriangleright\tau$ to τ.

We are nearly ready to present some stream instances from the introduction in the syntax of the gλ-calculus, but in order to recursively define elements of $\mathsf{Str}^{\mathsf{g}}\ \mathbf{N}$, we need a fixed-point combinator term. As originally observed by Nakano [2000], such a guarded fixed-point combinator fix should have type $(\blacktriangleright\tau \to \tau) \to \tau$, where $\mathsf{fix}\ f$ can be reduced to $f\ (\texttt{next}\ (\mathsf{fix}\ f))$. Compared to the standard reduction of $\mathsf{fix}\ f$ to $f\ (\mathsf{fix}\ f)$, the occurrence of `next` expresses that the subsequent application of fix may only occur one time step later. While we define such a fixed-point combinator in §2.3, we will for now simply postulate the existence of a term $\mathsf{fix} : (\blacktriangleright\tau \to \tau) \to \tau$ which reduces as specified.

The stream consisting only of zeros can then be defined by

$$\mathsf{zeros} \triangleq \mathsf{fix}\ (\lambda s : \blacktriangleright \mathsf{Str^g}\ \mathbf{N}.\ 0 :: s) \quad : \quad \mathsf{Str^g}\ \mathbf{N}.$$

Note that here the cons operator (::) has type $\mathbf{N} \to \blacktriangleright \mathsf{Str^g}\ \mathbf{N} \to \mathsf{Str^g}\ \mathbf{N}$, and thus takes the 'later' stream instance s to a stream instance 'now' by prepending 0 to the front, which corresponds to the syntactic guardedness criterion of s appearing directly below a constructor. Importantly, a clearly unproductive definition such as loop $\triangleq$ loop is precluded by the type system, as it would correspond to the term fix $(\lambda s.\ s)$, while the fixed-point combinator expects a function of type $\blacktriangleright \tau \to \tau$. We can now define the function toggle from §1 by

$$\mathsf{toggle} \triangleq \mathsf{fix}\ (\lambda s : \blacktriangleright \mathsf{Str^g}\ \mathbf{N}.\ 1 :: (\mathtt{next}\ (0 :: s))) \quad : \quad \mathsf{Str^g}\ \mathbf{N}.$$

In this definition, the inner cons yields an instance of $\mathsf{Str^g}\ \mathbf{N}$, and must therefore be wrapped with next, again expressing the fact that the stream tail is only available after one time step.

Let us now look at the definition of interleave in the type system of gλ. Aside from next, we use another novel operator $\circledast : \blacktriangleright(\tau_1 \to \tau_2) \to \blacktriangleright\tau_1 \to \blacktriangleright\tau_2$, which allows function application under the later type former: We will present the $\circledast$ operator formally in §2.2 and §2.3, but for now, it lets us define the interleaving of two streams by

$$\mathsf{interleave}' \triangleq \mathsf{fix}\ (\lambda g\ s\ t.\ \mathsf{hd}\ s :: (g \circledast (\mathtt{next}\ t) \circledast \mathsf{tl}\ s)) \quad : \quad \mathsf{Str^g}\ \tau \to \mathsf{Str^g}\ \tau \to \mathsf{Str^g}\ \tau.$$

While correct, this definition's type does not record the fact that the second input of interleave is only accessed after one time step, motivating the more general definition

$$\mathsf{interleave} \triangleq \mathsf{fix}\ (\lambda g\ s\ t.\ \mathsf{hd}\ s :: (g \circledast t \circledast \mathtt{next}\ (\mathsf{tl}\ s))) \quad : \quad \mathsf{Str^g}\ \tau \to \blacktriangleright \mathsf{Str^g}\ \tau \to \mathsf{Str^g}\ \tau,$$

whose type captures this additional information about the second argument. In fact, we need precisely this more general type when defining the paperfolds function from the introduction:

$$\mathsf{paperfolds} \triangleq \mathsf{fix}\ (\lambda s : \blacktriangleright \mathsf{Str^g}\ \mathbf{N}.\ \mathsf{interleave}\ \mathsf{toggle}\ s) \quad : \quad \mathsf{Str^g}\ \mathbf{N}$$

Given that the recursion variable s has type $\blacktriangleright \mathsf{Str^g}\ \mathbf{N}$, the above definition would not type check with interleave′, which expects a second argument of type $\mathsf{Str^g}\ \mathbf{N}$. More importantly, the problematic paperfolds′ definition from §1, with the arguments to interleave switched, is rejected by the type system, since the type of s does not match the first argument of interleave. In this way, the type system of the gλ-calculus can accept recursive definitions as productive even when they are not syntactically guarded, and can distinguish them from syntactically similar definitions which do not satisfy productivity. Note, however, that we can only express causal functions without the $\blacksquare$ modality (see §7.4).

As illustrated above, the addition of $\blacktriangleright$ along with the guardedness restriction ensures that functions on μ-types are *productive*, i.e., that any finite prefix of a possibly infinite output can be computed in finite time.

2.2 Syntax and Operational Semantics

Given below is the syntax of open gλ-terms.

$$
\begin{array}{rcll}
\mathit{GExp} \ni e & ::= & x & \textit{Variables} \\
& | & \mathtt{lit}\ n & \textit{Natural numbers} \\
& | & \langle\rangle \mid \langle e, e\rangle \mid \mathtt{proj}_1\ e \mid \mathtt{proj}_2\ e & \textit{Products} \\
& | & \mathtt{fail}\ e \mid \mathtt{inj}_1\ e \mid \mathtt{inj}_2\ e & \textit{Sums} \\
& | & \mathtt{case}\ e\ \mathtt{of}\ x.e\,;x.e & \\
& | & \lambda x.e \mid e\ e & \textit{Functions} \\
& | & \mathtt{fold}\ e \mid \mathtt{unfold}\ e & \textit{Recursion operations} \\
& | & \mathtt{next}\ e \mid e_1 \circledast e_2 & \textit{'Later' operations}
\end{array}
$$

Most of the connectives are fairly standard for a functional language: The usual λ-calculus is extended with natural numbers $\mathtt{lit}\ n$, pairs $\langle e, e\rangle$ with projections $\mathtt{proj}_i\ e$, unit $\langle\rangle$, case analysis $\mathtt{case}\ e\ \mathtt{of}\ \ldots$ on injections $\mathtt{inj}_i\ e$, and the exfalso term former $\mathtt{fail}$, as well as iso-recursion via $\mathtt{fold}$ and $\mathtt{unfold}$. In addition, we have the two 'later' operations $\mathtt{next}$ and $\circledast$, which allow the definition of *causal* guarded recursive functions.

The semantics of the gλ-calculus is call-by-name, and subterms are evaluated left-to-right. The grammar of gλ-values and that of evaluation contexts $K[-]$ is as follows:

$$
\begin{array}{rcll}
\mathit{GVal} \ni v & ::= & \mathtt{lit}\ n \mid \langle\rangle \mid \langle e, e\rangle \mid \mathtt{inj}_1\ e \mid \mathtt{inj}_2\ e & \textit{Values} \\
& | & \lambda x.e \mid \mathtt{fold}\ e \mid \mathtt{next}\ e & \\
\\
\mathit{GCtx} \ni K & ::= & \cdot \mid \mathtt{proj}_1\ K \mid \mathtt{proj}_2\ K & \textit{Evaluation contexts} \\
& | & \mathtt{case}\ K\ \mathtt{of}\ x.e\,;x.e \mid K\ e & \\
& | & \mathtt{unfold}\ K \mid K \circledast e \mid v \circledast K &
\end{array}
$$

Note that pairs and injections, as well as $\mathtt{fold}$ and $\mathtt{next}$, are value forms, that is, they constitute values even when their subterms do not. They are also not evaluated under, and are thus excluded in the grammar of evaluation contexts, while evaluation does occur under projections, $\mathtt{unfold}$, and at the scrutinee position of pattern match, among others.

As one would expect for call-by-name evaluation, only the left subterm of applications is reduced, while the argument is not. The first rule for *base reduction* $\mapsto_{\mathrm{b}}$ below states how applications are β-reduced once the left-hand side is evaluated to a λ-abstraction.

$$
\begin{array}{rcll}
(\lambda x.\ e_1)\ e_2 & \mapsto_{\mathrm{b}} & e_1[e_2/x] & \\
\mathtt{proj}_i\ \langle e_1, e_2\rangle & \mapsto_{\mathrm{b}} & e_i & i \in \{1,2\} \\
\mathtt{case}\ (\mathtt{inj}_i\ e)\ \mathtt{of}\ x_1.e_1\,;x_2.e_2 & \mapsto_{\mathrm{b}} & e_i[e/x_i] & i \in \{1,2\} \\
\mathtt{unfold}\ (\mathtt{fold}\ e) & \mapsto_{\mathrm{b}} & e & \\
\mathtt{next}\ e_1 \circledast \mathtt{next}\ e_2 & \mapsto_{\mathrm{b}} & \mathtt{next}\ (e_1\ e_2) &
\end{array}
$$

Indeed, all the first four rules for $\mapsto_{\mathrm{b}}$ are cases of β-reduction, which allow the removal of immediately adjacent introduction and elimination forms. The applicative operator $\circledast$ is neither an introduction nor an elimination form, but is needed to facilitate application under $\mathtt{next}$. Furthermore, $\circledast$ differs from application $e_1\ e_2$ in that right subterm (the argument) may be reduced. While seemingly going against the call-by-name reduction

strategy, both subterms of $\circledast$ must have form $\texttt{next}\ e$ before the final rule can be applied, taking $\circledast$ to regular application under a next.

The call-by-name reduction relation is denoted by $\mapsto$, and is defined by closing $\mapsto_{\mathrm{b}}$ over contexts K. Reduction steps have the form $K[e_1] \mapsto K[e_2]$, where $K \in \mathit{GCtx}$ is an evaluation context and $e_1 \mapsto_{\mathrm{b}} e_2$ is a base step, both as defined above. We use $\mapsto^*$ to denote the reflexive-transitive closure of $\mapsto$.

Two key results regarding $\mapsto$ are determinism and the context rule given below.

Lemma 2.2.1 (Determinism of $\mapsto$)**.** *The reduction relation $\mapsto$ on gλ-terms is deterministic: Given $e, e', e'' \in \mathit{GExp}$, if e steps by both $e \mapsto e'$ and $e \mapsto e''$, we have that $e' = e''$.*

Lemma 2.2.2 (Stepping under contexts)**.** *For any evaluation context K and $e, e' \in \mathit{GExp}$, if $e \mapsto e'$, then also $K[e] \mapsto K[e']$.*

2.3 Type System

We already touched on the key features of the type system for gλ, in particular that it includes the 'later' type former $\blacktriangleright$ as well as iso-recursive types $\mu\alpha.\tau$. The full syntax of pre-types is given by:

$$
\begin{array}{rcll}
\mathit{GType} \ni \tau & ::= & \alpha & \textit{Type variables} \\
& \mid & \mathbf{N} & \textit{Natural numbers} \\
& \mid & \mathbf{1} \mid \tau \times \tau & \textit{Products} \\
& \mid & \mathbf{0} \mid \tau + \tau & \textit{Sums} \\
& \mid & \tau \to \tau & \textit{Functions} \\
& \mid & \mu\alpha.\tau & \textit{Recursive types} \\
& \mid & \blacktriangleright\tau & \textit{Later}
\end{array}
$$

We take the unary type formers $\blacktriangleright$ and $\mu\alpha.-$ to bind more closely than binary type formers, e.g. $\blacktriangleright\tau_1 \times \tau_2$ denotes $(\blacktriangleright\tau_1) \times \tau_2$, and μ-bindings extend as far as possible to the right, meaning that e.g. $\mu\alpha.\ \tau_1 \to \tau_2$ denotes $\mu\alpha.(\tau_1 \to \tau_2)$.

The above grammar does not yet enforce the guardedness restriction on μ-recursive types introduced in §1 though. Well-formedness of types is enforced by the *type formation* judgement $\Delta \vdash \tau$, stating that type τ is well-formed in the context Δ, a finite set of type variables $\alpha \in \mathit{TVar}$. The rules of the type formation judgement are given in Figure 2.1.

Most of the cases are as expected: The base types $\mathbf{N}$, $\mathbf{1}$ and $\mathbf{0}$ are trivially well-formed in any context, well-formedness for the unary $\blacktriangleright$ and binary type formers $\times$, $+$ and $\to$ is defined pointwise, and rule WF-TVar checks that type variables are bound. WF-Rec is where guardedness is enforced. Aside from extending the context Δ with α, we have the side condition $\mathrm{guarded}_\alpha\ \tau$, referring to the judgement defined in Figure 2.1.

Clouston et al. [2016] leave the definition of $\mathrm{guarded}_{(-)}$ implicit, and state informally that in the syntax tree of a type, the μ-bound variable α must always occur under at least one $\blacktriangleright$. This intuition is what the remaining rules in Figure 2.1 express. Guardedness in α holds trivially for the base types and any β distinct from α, and is again defined pointwise for $\times$, $+$ and $\to$, and similarly for recursive types. Rule G-Later is an axiom: in a type of form $\blacktriangleright\tau$, any occurrence of α is guarded by the $\blacktriangleright$.

$$\textsc{WF-TVar}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha} \qquad \textsc{WF-Rec}\ \frac{\Delta, \alpha \vdash \tau \qquad \mathsf{guarded}_\alpha\ \tau}{\Delta \vdash \mu\alpha.\tau} \qquad \textsc{WF-Later}\ \frac{\Delta \vdash \tau}{\Delta \vdash \blacktriangleright\tau}$$

$$\textsc{WF-Base}\ \frac{\tau_{\mathsf{base}} \in \{\mathbf{N}, \mathbf{1}, \mathbf{0}\}}{\Delta \vdash \tau_{\mathsf{base}}} \qquad \textsc{WF-BinTy}\ \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \odot \in \{\times, +, \to\}}{\Delta \vdash \tau_1 \odot \tau_2}$$

$$\textsc{G-TVar}\ \frac{\alpha \neq \beta}{\mathsf{guarded}_\alpha\ \beta} \qquad \textsc{G-Rec}\ \frac{\mathsf{guarded}_\alpha\ \tau}{\mathsf{guarded}_\alpha\ (\mu\beta.\tau)} \qquad \textsc{G-Later}\ \frac{}{\mathsf{guarded}_\alpha\ (\blacktriangleright\tau)}$$

$$\textsc{G-Base}\ \frac{\tau_{\mathsf{base}} \in \{\mathbf{N}, \mathbf{1}, \mathbf{0}\}}{\mathsf{guarded}_\alpha\ \tau_{\mathsf{base}}} \qquad \textsc{G-BinTy}\ \frac{\mathsf{guarded}_\alpha\ \tau_1 \qquad \mathsf{guarded}_\alpha\ \tau_2 \qquad \odot \in \{\times, +, \to\}}{\mathsf{guarded}_\alpha\ (\tau_1 \odot \tau_2)}$$

Figure 2.1: Type formation rules and type variable guardedness judgement

The typing judgement is defined inductively in Figure 2.2, and is of the form $\Gamma \vdash e : \tau$. Here, the *typing context* Γ is a finite map associating term variables $x \in \mathit{GVar}$ with types $\tau \in \mathit{GType}$, the mappings of which we write as $x : \tau$. We follow the convention of Barendregt [1984], assuming term variables to be distinct and working up to α-conversion.

Most of the typing rules are standard: rules T-Var, T-Abs and T-App are exactly as in the simply-typed lambda calculus, and we have the usual (unrestricted) rules for sums and products. By the Curry-Howard correspondence, rule T-Fail expresses the exfalso principle, as it allows an inhabitant of arbitrary type τ to be derived from an inhabitant of $\mathbf{0}$. T-Fold and T-Unfold are the standard folding and unfolding rules for iso-recursive types. They express how the term-level constructs `fold` and `unfold` manipulate the μ-recursive type of their subterm, either unfolding it by one step, or folding back a recursive unfolding of the type. Finally, T-Next and T-Star express the applicative functor structure of $\blacktriangleright$, that is, the unit and applicative law, respectively.

Concluding the streams example. Having formally defined our system under consideration, we return to our example from §2.1. The type of guarded streams over some type τ, given by $\mathsf{Str}^{\mathsf{g}}\ \tau \triangleq \mu\alpha.\ \tau \times \blacktriangleright\alpha$, is well-formed by WF-Rec and the fact that $\mathsf{guarded}_\alpha\ (\tau \times \blacktriangleright\alpha)$, since α does not occur free in τ.

The cons (::), head and tail functions on streams are essentially pairing and projection, but we require (un)fold operations for the recursive type:

$$\begin{array}{lcll} (::) & \triangleq & \lambda x.\ \lambda s.\ \mathtt{fold}\ \langle x, s\rangle & : \quad \tau \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}\ \tau \to \mathsf{Str}^{\mathsf{g}}\ \tau \\ \mathsf{hd} & \triangleq & \lambda s.\ \mathtt{proj}_1\ (\mathtt{unfold}\ s) & : \quad \mathsf{Str}^{\mathsf{g}}\ \tau \to \tau \\ \mathsf{tl} & \triangleq & \lambda s.\ \mathtt{proj}_2\ (\mathtt{unfold}\ s) & : \quad \mathsf{Str}^{\mathsf{g}}\ \tau \to \blacktriangleright\mathsf{Str}^{\mathsf{g}}\ \tau \end{array}$$

$$\textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\textsc{T-Lit}\ \frac{}{\Gamma \vdash \mathtt{lit}\ n : \mathbf{N}} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \textsc{T-Pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\textsc{T-Proj}\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \mathtt{proj}_i\ e : \tau_i} \qquad \textsc{T-Fail}\ \frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathtt{fail}\ e : \tau} \qquad \textsc{T-Inj}\ \frac{\Gamma \vdash e : \tau_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathtt{inj}_i\ e : \tau_1 + \tau_2}$$

$$\textsc{T-Case}\ \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ x_1.e_1\,; x_2.e_2 : \tau}$$

$$\textsc{T-Fold}\ \frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{fold}\ e : \mu\alpha.\tau} \qquad \textsc{T-Unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathtt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

$$\textsc{T-Next}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{next}\ e : \blacktriangleright\tau} \qquad \textsc{T-Star}\ \frac{\Gamma \vdash e_1 : \blacktriangleright(\tau_1 \to \tau_2) \qquad \Gamma \vdash e_2 : \blacktriangleright\tau_1}{\Gamma \vdash e_1 \circledast e_2 : \blacktriangleright\tau_2}$$

Figure 2.2: Typing rules of gλ

The other missing definition was that of the guarded fixpoint combinator. While Clouston et al. present a version of Turing's combinator, we follow Nakano and modify the (untyped) Y combinator due to Curry:

$$Y \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

The Y combinator becomes typeable in our system by replacing application with $\circledast$ and inserting `next`, `fold` and `unfold` as appropriate.

$$\begin{array}{rcll} \mathsf{Rec}_\tau & \triangleq & \mu\alpha.\, \blacktriangleright(\alpha \to \tau) & \\ L & \triangleq & \lambda f : \blacktriangleright\tau \to \tau.\, \lambda x : \mathsf{Rec}_\tau.\, f\ (\mathtt{unfold}\ x \circledast \mathtt{next}\ x) & : \quad (\blacktriangleright\tau \to \tau) \to \mathsf{Rec}_\tau \to \tau \\ \mathsf{fix} & \triangleq & \lambda f : \blacktriangleright\tau \to \tau.\, (L\ f)\ (\mathtt{fold}\ (\mathtt{next}\ (L\ f))) & : \quad (\blacktriangleright\tau \to \tau) \to \tau \end{array}$$

The expression $\mathsf{fix}\ f$ then reduces as follows:

$$\begin{array}{ll} & f\ (\mathtt{unfold}\ (\mathtt{fold}\ (\mathtt{next}\ (L\ f))) \circledast \mathtt{next}\ (\mathtt{fold}\ (\mathtt{next}\ (L\ f)))) \\ \mapsto & f\ (\mathtt{next}\ (L\ f) \circledast \mathtt{next}\ (\mathtt{fold}\ (\mathtt{next}\ (L\ f)))) \\ \mapsto & f\ (\mathtt{next}\ ((L\ f)\ (\mathtt{fold}\ (\mathtt{next}\ (L\ f))))). \end{array}$$

The last line matches $f\ (\mathtt{next}\ (\mathsf{fix}\ f))$, that is, our definition for fix behaves as posited at the start of the chapter.

Chapter 3

Normalisation of the gλ-Calculus

A guarded type system, as employed by the gλ-calculus, serves first and foremost to guarantee productivity, and thereby ensure that programs normalise. As such, strong normalisation (SN) is one of the key properties of interest for the calculus, a proof of which is given by Clouston et al. [2016]. They prove SN via a denotational semantics in the *topos of trees*. The denotations of types and the relation used in the proof feature explicit indexing, leading to index arithmetic and bookkeeping that obscure the proof somewhat. Abel and Vezzosi [2014] present an SN proof—mechanised in Agda—for a similar calculus using a *saturated sets* semantics, again with explicit indexing. A detailed comparison with both proofs is given in §7.1.

In this chapter, we present an alternative proof that avoids explicit indexing entirely. We use a unary *logical relation*, defined via the operational semantics of gλ in the step-indexed logic *siProp*. Overall, the SN proof we present is motivated by the following:

- It establishes the prerequisite for our semantics-preserving translation in §4 and §5, i.e., that the causal subset of gλ is indeed terminating, and thus productive.
- We improve on existing SN proofs for similar calculi in our abstract treatment of the object-level later $\blacktriangleright$ and the absence of explicit indices. While our proof does use step-indexed constructions and reasoning, all indexing is hidden away by the later modality $\triangleright$ of the *siProp* logic.
- We explain the steps and constructions involved in a logical relations argument, and formally define the step-indexed logic *siProp*, all of which we return to in §5.
- The SN proof further lays the groundwork for the binary logical relation in §5, which closely follows the unary logical relation that we employ in showing SN.

We first give a high-level overview of our approach in §3.1, including a proof outline of our logical relations argument and highlighting some key points of interest. We then proceed to specify the target of our logical relation in §3.2, the step-indexed logic *siProp*, for which we give the grammar, a semantic model, and syntactic proof rules. For reasoning about gλ-programs, we derive a program logic from our base logic *siProp* in §3.3. In §3.4, we then define our logical relation and semantic typing judgement, which we finally use to show our strong normalisation result for well-typed gλ-terms in §3.5.

3.1 A Logical Approach to Strong Normalisation

The first use of a *logical relation* is generally attributed to Tait [1967] in showing strong normalisation for the simply-typed lambda calculus. However, in contrast to Tait, we follow the more recent practice of building our logical relation over an *operational* semantics, rather than a denotational one. Such logical relations require auxiliary techniques to handle recursive types [Birkedal and Harper, 1999]; motivating our use of *step-indexing* to model recursive types, as pioneered in the Foundational Proof-Carrying Code project [Appel and McAllester, 2001; Ahmed et al., 2002; Ahmed, 2004].

The marriage of semantic models based on operational semantics with step-indexing has proven fruitful in handling many advanced language features, especially when further combined with *higher-order concurrent separation logic* [Svendsen et al., 2013; Svendsen and Birkedal, 2014]—though we forego the latter in the present work—and the use of *program logics* to provide further abstraction [Dreyer et al., 2011; Turon et al., 2013] and aid mechanisation. The culmination of the above lines of works leads to what Timany et al. [2024] dub the 'logical approach'. Though their treatment is geared mainly towards type soundness and representation independence, we draw upon the general proof outline they present, which is equally applicable to the properties we wish to prove.

Proof method outline. At the core of the logical approach lies the semantic interpretation of types, or *logical relation*, denoted by $[\![-]\!]$, which maps values to propositions in the program logic, and is defined in terms of the operational semantics of the language. We lift this so-called *value interpretation* to both expressions and typing contexts, allowing us to define the *semantic typing judgement* $\Gamma \vDash e : \tau$, which expresses that the expression e is *semantically* well-typed with type τ in the context Γ.

Once the semantic typing judgement is in place, the proof itself comes in two parts:

- **Adequacy.** The so-called *adequacy* theorem states that the property of interest for expressions e (e.g. that e is safe to execute, or in our case, that e strongly normalises) holds for all semantically well-typed closed terms, that is, if $\emptyset \vDash e : \tau$. Most often, the definition of the semantic typing judgement is chosen to imply the property of interest (almost) immediately, making this proof near-trivial.

- **Fundamental Property.** In a second step, we wish to show that syntactic well-typedness implies semantic well-typedness, that is, $\Gamma \vdash e : \tau$ implies $\Gamma \vDash e : \tau$, which is commonly referred to as the *Fundamental Lemma* (or *Fundamental Property*). The Fundamental Lemma is established by proving a semantic version of each syntactic typing rule of the language, meaning a derivation matching the syntactic rule where each $\vdash$ is replaced by a $\vDash$. In the case of the gλ-calculus, we must show, for instance, a semantic version of the T-Pair rule, stated as follows:

$$\frac{\Gamma \vDash e_1 : \tau_1 \qquad \Gamma \vDash e_2 : \tau_2}{\Gamma \vDash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

  Proving the semantic typing rules is typically the most involved step, but once they are in place, the fundamental theorem follows immediately by induction on the syntactic typing judgement.

Chaining together the above two properties with *modus ponens*, we can show the property of interest: by the Fundamental Property, all syntactically well-typed programs are also semantically well-typed, and thus satisfy the desired property (in our case, strong normalisation) by the adequacy theorem.

Defining features of our logical relation. The following sections share many commonalities with sections 4 and 5 of Timany et al. [2024], and we refer readers unfamiliar with logical relations proofs to their work for additional background and explanation. We still introduce all relevant concepts as we require them, but focus more so on aspects in which our logical relation and proof differ from their treatment, and the standard format of a logical relations proof.

Indeed, there are a number of key aspects in which we depart from the treatment of Timany et al., modifying the general approach to suit our object language and property of interest. A detailed comparison with the example of Timany et al. can be found in §7.2.1, the main points from which we briefly list for the familiar reader:

- **Pure language and step-indexing.** Interpreting the pure gλ-calculus semantically does not involve reasoning about state. We choose the step-indexed logic of *siProp* as the target for our logical relation, as it is sufficiently expressive, and simpler than the Iris logic. The use of step-indexing is necessary to reason about data being available 'later' in the object language, a defining feature of the gλ-calculus.

- **Call-by-name semantics.** The operational semantics of the gλ-calculus are call-by-name, meaning that β-reduction is performed prior to reducing the argument, and we do not reduce under pairs, injections, `fold` and `next`. As such, we must account for the fact that the subterms in the aforementioned cases are expressions, not values, as seen more commonly in logical relations proofs.

- **Guarded recursive types.** Perhaps most interesting is the interpretation of recursive types $\mu\alpha.\tau$. They are interpreted to a fixpoint in *siProp* that must be guarded at the logic level by the later modality $\triangleright$. Guardedness of the logic-level fixpoint is typically achieved by inserting a $\triangleright$ in the interpretation of recursive types $\mu\alpha.\tau$, but showing SN for our language demands a different approach: we instead lift the guardedness restriction of the gλ-calculus to the logic level. We use the fact that τ is guarded in α according to the type formation judgement (§2.3), along with interpreting the object-level type former $\blacktriangleright$ to the logic-level later modality $\triangleright$.

Before we can present our logical relation and prove the adequacy theorem and Fundamental Property, we must introduce the logic *siProp* of step-indexed propositions into which we interpret gλ-types (§3.2), along with the program logic with which we reason about gλ-terms and their reduction (§3.3). We then proceed to define our logical relation (§3.4), for which we then show adequacy and the Fundamental Property, which together imply strong normalisation of gλ (§3.5).

3.2 Base Logic

For reasoning about the pure gλ-calculus, we work in the *siProp* logic of step-indexed propositions. The *siProp* logic is provided in the Iris framework as an embedded logic in Rocq, and is essentially the fragment of the Iris logic without resources. We further benefit from the Iris Proof Mode [Krebbers et al., 2017, 2018], which provides meta-programming tactics analogous to those of native Rocq for *siProp*. These offer a much greater level of abstraction than achievable when working in the model of *siProp*, or even with custom proof rules. We refer to §6 for further details on the formalisation.

Given below is the syntax of types and step-indexed propositions of the *siProp* logic.[1]

$$\tau, \sigma ::= 1 \mid \mathsf{GVal} \mid \mathsf{GExp} \mid \mathsf{siProp} \mid \tau \times \sigma \mid \tau \to \sigma \mid \dots$$

$$t, u, P, Q ::= x \mid \lambda\, x : \tau.\, t \mid t(u) \mid () \mid (t, u) \mid \pi_i\, t \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall\, x : \tau.\, P \mid \exists\, x : \tau.\, P \mid t =_\tau u \mid \triangleright P \mid \mu\, x : \tau.\, t \mid \dots$$

The higher-order logic *siProp* features a sort of types and a sort of terms. Its grammar subsumes that of the simply-typed lambda calculus, with a number of additional types, most notably that of step-indexed propositions siProp (we use sans-serif font for types), and the types GVal and GExp for the syntactic categories of gλ, corresponding to *GVal* and *GExp* in the meta-theory, which we embed into *siProp*. We omit the typing rules of *siProp*, as they are largely standard and follow from the use of the meta-variables: We use τ and σ for types, P and Q range over step-indexed propositions of type siProp, and Φ and Ψ are used for predicates, that is, functions to siProp. Aside from the standard connectives of propositional logic, the inhabitants of siProp include:

- higher-order and impredicative quantifiers $\forall\, x : \tau.\, P$ and $\exists\, x : \tau.\, P$, ranging over *siProp* types τ; along with
- the logic-level later modality $\triangleright$ and guarded fixpoint combinator μ, which we use to interpret the gλ-calculus's type formers $\blacktriangleright$ and $\mu\alpha.-$.

Note that we overload part of the syntax from the gλ-calculus in *siProp*, most notably λ-abstraction, the μ binder, and the type formers $\times$, $+$ and $\to$.

3.2.1 Model of Step-Indexed Propositions

The siProp type expresses step-indexed propositions, which hold for some (non-negative) number of time steps, or indefinitely. A model of step-indexed propositions is given by the set *SIProp*, comprising downwards-closed subsets of the natural numbers:

$$\mathit{SIProp} \triangleq \{X \subseteq \mathbb{N} \mid \forall\, n \geq m.\, n \in X \Rightarrow m \in X\}$$

The restriction to downwards-closed subsets $X \subseteq \mathbb{N}$ means that any statement that holds for n steps must also hold for $m \leq n$ steps (*SIProp* thus corresponds to the set $\mathbb{N} \cup \{\omega\}$).

[1] The *siProp* logic, as defined in the Iris framework, also features connectives `next` and $\blacktriangleright$. We use neither in our work, hence we omit them to avoid confusion with the syntax of gλ. We further omit the 0 type and sum types $\tau + \sigma$, whose term formers are standard.

Elements of *SIProp* give the set of indices at which a statement is valid, where $\emptyset$ expresses that a statement is false, while $\mathbb{N}$ represents a statement that always holds.

SIProp comprises a so-called *ordered family of equivalences* (OFE), which consists of a set along with an algebraic structure that equips the set with a form of step-indexing. We use OFEs, originally proposed by Di Gianantonio and Miculan [2002], to assign a semantic model to the *siProp* logic. In order to define our semantic interpretation of *siProp*, we must formally introduce OFEs and some related notions.

OFEs and non-expansive functions. OFEs build on the notion of n-equivalence. Elements x, y of the OFE's carrier set are said to be n-equivalent (written $x \overset{n}{=} y$) if they are equivalent for n steps, as captured formally by the following definition.

Definition 3.2.1 (Ordered Family of Equivalences (OFE))**.** An *ordered family of equivalences* (OFE) is a tuple $(T, (\overset{n}{=} \subseteq T \times T)_{n\in\mathbb{N}})$, such that

(1) $(\overset{n}{=})$ is an equivalence relation for all $n \in \mathbb{N}$;

(2) $n \geq m \Rightarrow (\overset{n}{=}) \subseteq (\overset{m}{=})$, for all $n, m \in \mathbb{N}$; and

(3) $x = y$ iff $\forall n.\ x \overset{n}{=} y$, for all x and y.

Property (2) states that for larger values of n, the relation $\overset{n}{=}$ becomes increasingly refined, while (3) expresses that in the limit, $\overset{n}{=}$ agrees with plain equality (=) on the carrier.

For $X, Y \in$ *SIProp*, the family of equivalence relations $(\overset{n}{=})_{n\in\mathbb{N}}$ is defined by

$$X \overset{n}{=} Y \triangleq \forall\, m \leq n.\ m \in X \Leftrightarrow m \in Y.$$

We can lift any set X to the step-indexed setting as the *discrete* OFE ΔX, which imposes upon X the degenerate n-equivalence that ignores n, i.e. defines $\overset{n}{=}$ as equality in X. The product and sum of two OFEs, with $\overset{n}{=}$ defined pointwise, are again OFEs.

For n-equivalence to remain meaningful under function application, the function space between OFEs is restricted to functions that are *non-expansive*. Intuitively, non-expansiveness of $f : T \to U$ means that elements which cannot be distinguished within n steps (in T) remain indistinguishable within n steps (in U) under application of f.

Definition 3.2.2 (Non-expansive)**.** A function $f : T \to U$ is *non-expansive* if for all $n \in \mathbb{N}$ and $x, y \in X$, we have that $x \overset{n}{=} y$ implies $f(x) \overset{n}{=} f(y)$. We then write $f : X \overset{\text{ne}}{\longrightarrow} Y$.

Accordingly, the semantic model of *siProp* identifies function types $\tau \to \sigma$ in *siProp* with the non-expansive function space $[\![\tau]\!] \overset{\text{ne}}{\longrightarrow} [\![\sigma]\!]$, which forms an OFE. Crucially, non-expansive functions allow rewriting of step-indexed equality under their application (as the definition states). While non-expansive functions cannot make equal inputs distinct, they may make distinct inputs equal: this gives rise to the notion of *contractiveness*.

Definition 3.2.3 (Contractive)**.** A function $f : X \to Y$ is *contractive* if for all $n \in \mathbb{N}$ and $x, y \in X$, we have that $\forall m < n.\ x \overset{m}{=} y$ implies $f(x) \overset{n}{=} f(y)$.

Equivalently, f is contractive if $f(x) \overset{0}{=} f(y)$ and $x \overset{n}{=} y \Rightarrow f(x) \overset{n+1}{=} f(y)$.

$$\begin{aligned}
[\![1]\!] &\triangleq \Delta\{()\} & [\![\mathit{siProp}]\!] &\triangleq \mathit{SIProp}\\
[\![\mathsf{GVal}]\!] &\triangleq \Delta \mathit{GVal} & [\![\tau_1 \times \tau_2]\!] &\triangleq [\![\tau_1]\!] \times [\![\tau_2]\!]\\
[\![\mathsf{GExp}]\!] &\triangleq \Delta \mathit{GExp} & [\![\tau \to \sigma]\!] &\triangleq [\![\tau]\!] \xrightarrow{\mathsf{ne}} [\![\sigma]\!]
\end{aligned}$$

$$\begin{aligned}
[\![x : \tau]\!]_\rho &\triangleq \rho(x) & [\![(t,u) : \tau \times \sigma]\!]_\rho &\triangleq ([\![t : \tau]\!]_\rho, [\![u : \sigma]\!]_\rho)\\
[\![\lambda\, x : \tau.\, t : \tau \to \sigma]\!]_\rho &\triangleq \lambda\, a.\, [\![t : \sigma]\!]_{\rho[x:=a]} & [\![\pi_i(t) : \tau_i]\!]_\rho &\triangleq \pi_i([\![t : \tau_1 \times \tau_2]\!]_\rho)\\
[\![t(u) : \sigma]\!]_\rho &\triangleq [\![t : \tau \to \sigma]\!]_\rho([\![u : \tau]\!]_\rho) & [\![() : 1]\!]_\rho &\triangleq ()
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{False} : \mathsf{siProp}]\!]_\rho &\triangleq \emptyset\\
[\![\mathsf{True} : \mathsf{siProp}]\!]_\rho &\triangleq \mathbb{N}\\
[\![t =_\tau u : \mathsf{siProp}]\!]_\rho &\triangleq \left\{n \mid [\![t : \tau]\!]_\rho \overset{n}{=} [\![u : \tau]\!]_\rho\right\}\\
[\![P \wedge Q : \mathsf{siProp}]\!]_\rho &\triangleq [\![P : \mathsf{siProp}]\!]_\rho \cap [\![Q : \mathsf{siProp}]\!]_\rho\\
[\![P \vee Q : \mathsf{siProp}]\!]_\rho &\triangleq [\![P : \mathsf{siProp}]\!]_\rho \cup [\![Q : \mathsf{siProp}]\!]_\rho\\
[\![P \Rightarrow Q : \mathsf{siProp}]\!]_\rho &\triangleq \left\{n \mid \forall\, m \le n.\, m \in [\![P : \mathsf{siProp}]\!]_\rho \Rightarrow m \in [\![Q : \mathsf{siProp}]\!]_\rho\right\}\\
[\![\forall\, x : \tau.\, P : \mathsf{siProp}]\!]_\rho &\triangleq \textstyle\bigcap_{v \in [\![\tau]\!]} [\![P : \mathsf{siProp}]\!]_{\rho[x:=v]}\\
[\![\exists\, x : \tau.\, P : \mathsf{siProp}]\!]_\rho &\triangleq \textstyle\bigcup_{v \in [\![\tau]\!]} [\![P : \mathsf{siProp}]\!]_{\rho[x:=v]}\\
[\![\mu\, x : \tau.\, t : \mathsf{siProp}]\!]_\rho &\triangleq \mathit{fix}_{[\![\tau]\!]}(\lambda u.\, [\![t : \mathsf{siProp}]\!]_{\rho[x:=u]})\\
[\![\triangleright\, P : \mathsf{siProp}]\!]_\rho &\triangleq \left\{n \mid n = 0 \vee n - 1 \in [\![P : \mathsf{siProp}]\!]_\rho\right\}
\end{aligned}$$

Figure 3.1: Semantic interpretation of the base logic *siProp*

Complete OFEs. An OFE's step-indexed equality $\overset{n}{=}$ becomes more refined with each step; this motivates the notion of a *chain* as a sequence $(c_n)_{n\in\mathbb{N}}$ of elements where the n-th element is n-equal to all subsequent elements:

$$\forall n, m.\, n \le m \Rightarrow c_n \overset{n}{=} c_m$$

A chain has a *limit* if there is some c_∞ to which all c_n are n-equal, that is, $c_n \overset{n}{=} c_\infty$ for all n, leading to a notion of (chain-)*completeness* for OFEs.

Definition 3.2.4 (Complete OFE (COFE))**.** An OFE T is *complete* (a *COFE*) if every chain $(c_n)_{n\in\mathbb{N}}$ in T has a limit.

All OFEs discussed in the present section are complete. COFEs form a category **COFE**, where the objects are COFEs, and the morphisms between them are non-expansive functions. **COFE** is *Cartesian closed*, thus allowing the interpretation of the simply-typed lambda calculus, and similarly the type formers of *siProp*.

COFEs further allow the definition of objects as the limit of a chain. In particular, we can recursively define propositions as the unique fixpoint of a function over propositions,

which is internalised in the *siProp* logic by the *guarded fixpoint* combinator $\mu\, x : \tau.\ t$. It allows the construction of recursive predicates without restriction on the variance of self-references; instead, occurrences of x in t must be *guarded*, this time by appearing under the later modality $\triangleright$. The existence of unique guarded fixpoints is a consequence of the following theorem.

Theorem 3.2.1 (Banach's fixpoint)**.** *Given an inhabited COFE T and a contractive function $f : T \to T$, a unique fixpoint $\mathit{fix}_T\ f$ of f exists, satisfying $f\ (\mathit{fix}_T\ f) = \mathit{fix}_T\ f$.*

The OFE *SIProp* is complete (i.e. a COFE), in which the interpretation of $\triangleright$ is contractive, while all other logical connectives are non-expansive. When composing a contractive and a non-expansive function, contractiveness is preserved; the *syntactic* guardedness condition of x appearing under a $\triangleright$ is thus sufficient to ensure contractiveness.

Much like in gλ, the fixpoint combinator μ must be restricted beyond well-typedness: $\mu\, x : \tau.\ t$ is well-formed only if τ (and thus its interpretation) is inhabited, and if x is guarded by $\triangleright$ in t.

Semantic interpretation. Figure 3.1 describes the semantic interpretation $[\![-]\!]$ of types to COFEs, and that of terms $[\![- : \tau]\!]_\rho$ with type τ under the variable assignment ρ, mapping into $[\![\tau]\!]$. In the interpretation of types, we inject our (meta-level) sets *GExp* and *GVal* of gλ-terms and values into the step-indexed setting as discrete OFEs (using Δ).

All OFEs used to interpret *siProp*'s types are complete, permitting the use of $\mathit{fix}_{[\![\tau]\!]}$ to interpret the μ combinator. In the following, we use μ only to construct predicates, that is, functions to siProp, whose type is trivially inhabited by terms like $\lambda x.\ \mathsf{True}$.

3.2.2 Syntactic Entailment Relation

We denote the entailment relation of the semantic model by $\vDash$, which is defined by

$$X \vDash Y \triangleq \forall\, n.\ n \in X \Rightarrow n \in Y,$$

that is, by subset inclusion $X \subseteq Y$. Entailment between propositions $P, Q : \mathsf{siProp}$, can thus be proven via the model by $[\![P : \mathsf{siProp}]\!] \vDash [\![Q : \mathsf{siProp}]\!]$ (in *SIProp*). In practice, however, we do not wish to unfold the semantic model in our proofs, motivating the use of a syntactic entailment relation with a set of proof rules.

We write $\vdash$ for syntactic entailment on *siProp*, defined inductively by the rules in Figure 3.2, along with the rules for the usual connectives of higher-order intuitionistic logic, which are standard and omitted. We write $P \dashv\vdash Q$ for $P \vdash Q$ and $Q \vdash P$. For $\mathsf{True} \vdash P$ we use the shorthand $\vdash P$. We can then prove the syntactic entailment relation $\vdash$ sound with respect to semantic entailment in *SIProp*.

Theorem 3.2.2 (Soundness of the Semantic Model)**.** *If propositions are related by syntactic entailment $\vdash$, their interpretations are related by semantic entailment in the model:*

$$(P \vdash Q) \Rightarrow [\![P]\!] \vDash [\![Q]\!]$$

The theorem states that any judgement derivable using our proof rules is also valid in the semantic model, allowing us to work only with the proof rules in the following.

Rules for guarded recursion and the later modality:

μ-Unfold
$\vdash (\mu\, x.\, t) = t[\mu\, x.\, t/x]$

$\triangleright$-Mono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

$\triangleright$-Intro
$P \vdash \triangleright P$

$\triangleright$-And
$\triangleright(P \land Q) \dashv\vdash \triangleright P \land \triangleright Q$

$\triangleright$-Or
$\triangleright(P \lor Q) \dashv\vdash \triangleright P \lor \triangleright Q$

$\triangleright$-Impl
$\triangleright(P \Rightarrow Q) \dashv\vdash (\triangleright P \Rightarrow \triangleright Q)$

$\triangleright$-Forall
$\triangleright(\forall\, x : \tau.\, P) \dashv\vdash \forall\, x : \tau.\, \triangleright P$

$\triangleright$-Exists
$$\frac{\mathsf{inhabited}(\tau)}{\triangleright(\exists\, x : \tau.\, P) \dashv\vdash \exists\, x : \tau.\, \triangleright P}$$

Figure 3.2: Proof rules for the *siProp* logic

Adequacy of the logic. In the mechanised setting, we work in a shallow embedding of the step-indexed logic, meaning we can reason about meta-level (Rocq) terms and types from within the logic. Including GVal and GExp as types in *siProp* is mainly illustrative; the object logic actually embeds all meta-types as discrete OFEs, but we do not make this formal in the present text. In the following, we take the liberty of using our meta-theoretic sets and judgements, such as *GVal*, *GExp*, *GType*, or $\Gamma \vdash e : \tau$, as if they were part of *siProp*, intermingling the syntax of *siProp* with our meta-notations.

Importantly, we wish to use *siProp* to prove results outside the object logic in our meta-theory. Doing so requires an adequacy result for *siProp*, stating that propositions we can prove via the rules for $\vdash$ are indeed meaningful at the meta-level.

Theorem 3.2.3 (Adequacy of *siProp*)**.** *Given a proposition* φ : siProp *that does not contain the later modality* $\triangleright$*, it follows from* $\mathsf{True} \vdash \varphi$ *that* φ *holds at the meta-level.*

3.3 Deriving the Program Logic

The *siProp* logic of the previous section does not supply any connectives for reasoning about programs and their evaluation. In the following, we define a program logic—in particular, the *weakest precondition* (WP) connective—in terms of the base logic to reason about the reduction of gλ-programs. We subsequently present a number of derived proof rules that allow us to reason about the weakest precondition connective.

Total weakest precondition. For some postcondition Φ : GVal $\to$ siProp, our total weakest precondition wp e [Φ] states that e reduces to some value v satisfying the predicate Φ. The total WP does not explicitly require safety for all possible reductions, but our WP does imply safety due to the language we are considering: since gλ's step relation $\mapsto$ is deterministic, if e steps to a value v, then e always terminates. In other words, e is strongly normalising, which is exactly our property of interest.

Rules for total weakest precondition of $g\lambda$:

$$\text{TWP-Val}\quad \Phi(v) \vdash \mathsf{wp}\ v\ [\Phi] \qquad \text{TWP-Step}\quad \frac{\mathsf{wp}\ e_2\ [\Phi] \qquad e_1 \mapsto e_2}{\mathsf{wp}\ e_1\ [\Phi]}$$

$$\text{TWP-Bind}\quad \frac{\mathsf{wp}\ e\ \left[v.\ \mathsf{wp}\ K[v]\ [\Phi]\right]}{\mathsf{wp}\ K[e]\ [\Phi]} \qquad \text{TWP-Impl}\quad \frac{\mathsf{wp}\ e\ [\Phi] \qquad \forall\, v.\ \Phi(v) \Rrightarrow \Psi(v)}{\mathsf{wp}\ e\ [\Psi]}$$

Figure 3.3: Notable rules for the total WP connective

In the following, we define the semantic typing judgement $\Gamma \vDash e : \tau$ in terms of the total WP. As such, proving our adequacy theorem is indeed as trivial as promised in our proof outline (§3.1), as SN then follows immediately from $\emptyset \vDash e : \tau$.

Definition 3.3.1 (Total WP)**.** We define the total weakest precondition as follows:

$$\begin{array}{rcl} \mathsf{wp}\ (-)\ [-] & : & \mathsf{GExp} \to (\mathsf{GVal} \to \mathsf{siProp}) \to \mathsf{siProp} \\ \mathsf{wp}\ e\ [\Phi] & \triangleq & \exists\, v.\ (e \mapsto^* v) \wedge \Phi(v) \end{array}$$

The definition encodes exactly the informal statement above: e must reduce to some value v, which in turn satisfies the postcondition Φ. The safety property follows from determinism of $\mapsto^*$ (Lemma 2.2.1), which implies that $e \mapsto^* v$ is the only reduction for e, hence there can be no other reduction path to a stuck term, and e is safe to reduce.

Proving the semantic typing rules to establish the Fundamental Lemma will frequently require reasoning about the total WP connective. In those proofs, we wish to handle the total WP abstractly, and avoid unfolding its definition each time. We do so by deriving a number of proof rules for our total WP, all of which are standard for WP connectives.

Proof rules. The rules for the total weakest precondition are given in Figure 3.3. We write $\mathsf{wp}\ e\ [v.\ P]$ as a shorthand for $\mathsf{wp}\ e\ [\lambda\, v.\ P]$. Let us consider the rules in detail:

- Rule TWP-Val can be derived immediately from our definition of $\mathsf{wp}\ e\ [\Phi]$: given $v \in \mathit{GVal}$ with $\Phi(v)$, then also $\mathsf{wp}\ v\ [\Phi]$, since the value v trivially steps to itself.
- Rule TWP-Step states that we can step expressions under the weakest precondition. Given $\mathsf{wp}\ e_2\ [\Phi]$, there exists a value v such that $e_2 \mapsto^* v$ and $\Phi(v)$. Since $e_1 \mapsto e_2$, we also have that $e_1 \mapsto^* v$, and thus $\mathsf{wp}\ e_1\ [\Phi]$.
- Rule TWP-Bind is derivable from Lemma 2.2.2, by which the reduction $e_1 \mapsto^* e_2$ implies $K[e_1] \mapsto^* K[e_2]$ for some context K. To prove a weakest precondition of the form $\mathsf{wp}\ K[e]\ [\Phi]$, it suffices to show that e steps to some v which in turn satisfies $\mathsf{wp}\ K[v]\ [\Phi]$. The assumption states that $e \mapsto^* v$, where v satisfies $\mathsf{wp}\ K[v]\ [\Phi]$. Since $e \mapsto^* v$ implies $K[e] \mapsto^* K[v]$, we get (by repeated application of TWP-Step) that $\mathsf{wp}\ K[e]\ [\Phi]$.

- Rule TWP-Impl states that the total weakest precondition is monotone in the postcondition. This rule is also immediate by applying the assumption $\forall\, v.\ \Phi(v) \Rightarrow \Psi(v)$ in $\exists\, v.\ (e \mapsto^* v) \wedge \Phi(v)$.

3.4 Semantic Interpretation of gλ-Types

As mentioned above, our normalisation proof involves first defining a semantic version of the gλ typing judgement $\Gamma \vDash e : \tau$. Again, we closely follow the outline of Timany et al. [2024], where the remainder of the present section defines our semantic interpretation of types for values, and how we lift it to expressions.

We begin by constructing a *logical relation*, corresponding to the semantic typing judgement on closed terms. It is important to note that our logical relation is defined only on *well-formed* gλ-types (by the type formation judgement of Figure 2.1), that is, recursive types $\mu\alpha.\tau$ must satisfy guardedness of α in τ by $\blacktriangleright$.

Using closing substitutions, the logical relation for closed terms can then be lifted to the semantic typing relation on open terms. The logical relation corresponds to the semantic interpretation of types mentioned previously. More specifically, the logical relation encompasses two semantic interpretations of types, whose mutually recursive definitions are given in Figure 3.4.

- The *value interpretation* $[\![\tau]\!]_\delta : \mathsf{GVal} \to \mathsf{siProp}$, which corresponds to the semantic interpretation of types discussed previously, is constructed by structural recursion on the grammar of types. The proposition $[\![\tau]\!]_\delta(v)$ expresses that the value v behaves as a valid inhabitant of type τ.

- The *expression interpretation* $[\![\tau]\!]^{\mathsf{e}}_\delta : \mathsf{GExp} \to \mathsf{siProp}$ states the semantics required of a closed expression of type τ. It is defined in terms of $[\![\tau]\!]_\delta$.

Both interpretations are annotated with a *semantic environment* δ due to the fact that gλ features μ-bound type variables. The semantic environment maps type variables α to their semantic value interpretation. As such, the value interpretation for a type variable α is given by $[\![\alpha]\!]_\delta = \delta(\alpha)$. The full type of the value interpretation is given by

$$[\![-]\!]_{(-)} : \mathit{GType} \to (\mathit{TVar} \xrightarrow{\mathsf{fin}} (\mathsf{GVal} \to \mathsf{siProp})) \to \mathsf{GVal} \to \mathsf{siProp}.$$

The semantic environment $\delta : \mathit{TVar} \xrightarrow{\mathsf{fin}} (\mathsf{GVal} \to \mathsf{siProp})$ is a finite map, where $\mathrm{dom}(\delta)$ implicitly matches the set of free type variables in the first input $\tau : \mathit{GType}$.

We lift the interpretation $[\![\tau]\!]_\delta$ on values v to expressions e by stating that e evaluates to some value v, which in turn satisfies the value interpretation. This property is captured by our notion of total weakest precondition from Definition 3.3.1:

$$[\![\tau]\!]^{\mathsf{e}}_\delta \triangleq \lambda\, e.\ \mathsf{wp}\ e\ \left[\![\tau]\!]_\delta\right]$$

Aside from the expression interpretation, Figure 3.4 defines the value interpretation by structural recursion on types, the cases of which we discuss in the following.

$$
\begin{aligned}
[\![\tau]\!]^{\mathrm{e}}_{\delta} &\triangleq \lambda\, e.\ \mathsf{wp}\ e\ \left[\!\left[\tau\right]\!\right]_{\delta}\right] \\
[\![\alpha]\!]_{\delta} &\triangleq \delta(\alpha) \\
[\![\mathbf{N}]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, n \in \mathbb{N}.\ v = \mathtt{lit}\ n \\
[\![\mathbf{1}]\!]_{\delta} &\triangleq \lambda\, v.\ v = \langle\rangle \\
[\![\tau_1 \times \tau_2]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, e_1, e_2.\ (v = \langle e_1, e_2\rangle) \wedge [\![\tau_1]\!]^{\mathrm{e}}_{\delta}(e_1) \wedge [\![\tau_2]\!]^{\mathrm{e}}_{\delta}(e_2) \\
[\![\mathbf{0}]\!]_{\delta} &\triangleq \lambda\ \_.\ \mathsf{False} \\
[\![\tau_1 + \tau_2]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, i \in \{1,2\}, e.\ (v = \mathtt{inj}_i\ e) \wedge [\![\tau_i]\!]^{\mathrm{e}}_{\delta}(e) \\
[\![\tau_1 \to \tau_2]\!]_{\delta} &\triangleq \lambda\, v.\ \left(\forall\, e.\ [\![\tau_1]\!]^{\mathrm{e}}_{\delta}(e) \Rightarrow [\![\tau_2]\!]^{\mathrm{e}}_{\delta}(v\ e)\right) \\
[\![\mu\alpha.\tau]\!]_{\delta} &\triangleq \mu\ (\Psi : \mathsf{GVal} \to \mathsf{siProp}).\ \lambda\, v.\ \exists\, e.\ (v = \mathtt{fold}\ e) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e) \\
[\![\blacktriangleright\tau]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, e.\ (v = \mathtt{next}\ e) \wedge \triangleright [\![\tau]\!]^{\mathrm{e}}_{\delta}(e)
\end{aligned}
$$

Figure 3.4: Unary logical relation on the gλ-calculus

3.4.1 Base Types

The cases for the base types are the simplest, as we can just give the set of values directly, and define the value interpretations as functions stating when a value is in said set. For $\mathbf{0}$, there are no inhabitants, hence the interpretation is just defined as the constant function returning False.

$$[\![\mathbf{1}]\!]_{\delta} \triangleq \lambda\, v.\ v = \langle\rangle \qquad [\![\mathbf{0}]\!]_{\delta} \triangleq \lambda\ \_.\ \mathsf{False} \qquad [\![\mathbf{N}]\!]_{\delta} \triangleq \lambda\, v.\ \exists\, n \in \mathbb{N}.\ v = \mathtt{lit}\ n$$

3.4.2 Product and Sum Types

Product types $\tau_1 \times \tau_2$ and sum types $\tau_1 + \tau_2$ are more or less interpreted pointwise. Here we encounter the first notable deviation from Timany et al. [2024]: since pairs and injections are product forms which are not evaluated under, we require a value of a product type to be of the form $\langle e_1, e_2\rangle$, where e_1 and e_2 should behave as expressions—but not necessarily values—of type τ_1 and τ_2, respectively. As such, the expression interpretation $[\![-]\!]^{\mathrm{e}}_{\delta}$ is used, stating that e.g. e_1 must reduce to some value v_1, which in turn satisfies the value interpretation of τ_1. Similarly, a value of type $\tau_1 + \tau_2$ must be an injection of the form $\mathtt{inj}_i\ e$ for $i \in \{1,2\}$, and e must be in the expression interpretation of τ_i.

$$
\begin{aligned}
[\![\tau_1 \times \tau_2]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, e_1, e_2.\ (v = \langle e_1, e_2\rangle) \wedge [\![\tau_1]\!]^{\mathrm{e}}_{\delta}(e_1) \wedge [\![\tau_2]\!]^{\mathrm{e}}_{\delta}(e_2) \\
[\![\tau_1 + \tau_2]\!]_{\delta} &\triangleq \lambda\, v.\ \exists\, i \in \{1,2\}, e.\ (v = \mathtt{inj}_i\ e) \wedge [\![\tau_i]\!]^{\mathrm{e}}_{\delta}(e)
\end{aligned}
$$

3.4.3 Function Types

Values of a function type $\tau_1 \to \tau_2$ are not required to have any specific syntactic form. Instead, we require that they can be applied to an expression e in the expression interpretation of the input type τ_1, where the application $v\ e$ must be in the expression

interpretation of τ_2. Given the call-by-name semantics of gλ, the argument is not necessarily a value, as β-reduction is performed without first reducing the argument.

$$[\![\tau_1 \to \tau_2]\!]_\delta \triangleq \lambda\, v.\ \left(\forall\, e.\ [\![\tau_1]\!]^{\mathrm{e}}_\delta(e) \Rightarrow [\![\tau_2]\!]^{\mathrm{e}}_\delta(v\ e)\right)$$

From the fact that $(v\ e)$ can step and that $v \in \mathit{GVal}$, it follows that v is of the form $\lambda x.\ e$, but we need not explicitly encode this property in the logical relation.

An important detail is that the definition features $[\![\tau_1]\!]^{\mathrm{e}}_\delta$ in a negative (contravariant) position, on the left side of the implication $\Rightarrow$. It is this negative occurrence that motivates the definition of the logical relation $[\![\tau]\!]_\delta$ by structural recursion on τ, as it would break well-foundedness of a definition via an inductive predicate. Defining $[\![\tau]\!]_\delta$ instead by structural recursion on τ ensures well-foundedness by the fact that in the case $\tau = \tau_1 \to \tau_2$, the type τ_1 is smaller than τ.

3.4.4 Guarded Types

Next, we look at the interpretation of guarded types. A value of type $\blacktriangleright\tau$ is expected to be of the form $\mathtt{next}\ e$ for some e satisfying the expression interpretation of τ:

$$[\![\blacktriangleright\tau]\!]_\delta \triangleq \lambda\, v.\ \exists\, e.\ (v = \mathtt{next}\ e) \wedge \triangleright [\![\tau]\!]^{\mathrm{e}}_\delta(e)$$

This case is rather straightforward, but features another key detail: in the same way that the guarded type $\blacktriangleright\tau$ expresses that e has type τ 'later', or after one time step, we similarly require that $[\![\tau]\!]^{\mathrm{e}}_\delta(e)$ holds only 'later', by placing the proposition under a $\triangleright$, the logic-level later modality that we introduced in §3.2. Aside from the $\triangleright$ connective relating to the $\blacktriangleright$ type former by the Curry-Howard correspondence, we motivate this choice further in just a moment, when we shed more light on why we must include a $\triangleright$ here.

3.4.5 Recursive Types

Recall that the syntactic typing rule T-Fold states that $\mathtt{fold}\ e$ is of type $\mu\alpha.\tau$ if e is well-typed with the larger type $\tau[\mu\alpha.\tau/\alpha]$:

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathtt{fold}\ e : \mu\alpha.\tau}$$

As such, we wish to define $[\![\mu\alpha.\tau]\!]_\delta$ such that we have $[\![\mu\alpha.\tau]\!]_\delta(v)$ if $v = \mathtt{fold}\ e$ for some $e \in \mathit{GExp}$, where $[\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(e)$. If we try to encode this requirement directly, we run into an issue: interpreting $\mu\alpha.\tau$ via its syntactically larger unfolding $\tau[\mu\alpha.\tau/\alpha]$ violates structural recursion, and the definition of $[\![-]\!]$ would no longer be well-founded.

Fortunately, Theorem 3.2.1 lets us define recursive predicates of type GVal $\to$ siProp with unrestricted self-references via the guarded fixpoint combinator μ, i.e. in the form $\mu\,\Phi : \mathsf{GVal} \to \mathsf{siProp}.\ P$, provided that occurrences of Φ in P are guarded by $\triangleright$. In that case, we get that $(\mu\ \Phi.\ P) = P[\mu\ \Phi.\ P/\Phi]$. With this in mind, we define $[\![\mu\alpha.\tau]\!]_\delta$ as a guarded fixpoint, where we extend the semantic environment δ with the self-reference:

$$[\![\mu\alpha.\tau]\!]_\delta \triangleq \mu\ (\Psi : \mathsf{GVal} \to \mathsf{siProp}).\ \lambda\, v.\ \exists\, e.\ (v = \mathtt{fold}\ e) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)$$

There is just one slight issue: while we just stated that all self-references must be guarded, it seems that Ψ is not guarded by a $\triangleright$ above! Comparing to Timany et al. [2024], their interpretation of recursive types features a $\triangleright$ on the recursive interpretation of τ, which guards the occurrence of Ψ. However, adding a $\triangleright$ explicitly would require eliminating a later when proving the semantic rule corresponding to T-Unfold. Our total WP does not allow later eliminations through program steps, and a WP that does is necessarily partial (see e.g. Krebbers et al. [2025]), hence we cannot include a $\triangleright$ here.

To explain why our definition is still permissible, we must consider when and how Ψ can appear in the μ-body above, specifically in the expression interpretation for τ. The semantic environment δ passed to $[\![\tau]\!]^{\mathrm{e}}$ is extended with the mapping $\alpha \mapsto \Psi$, so Ψ can only occur in $[\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}$ when we encounter α inside τ. In that case, α is interpreted by $[\![\alpha]\!]_{\delta,\alpha\mapsto\Psi} \triangleq (\delta, \alpha \mapsto \Psi)(\alpha) = \Psi$.[2]

Now let us briefly return to the type formation judgement $\Delta \vdash \tau$ on gλ-types defined in Figure 2.1. Since the type $\mu\alpha.\tau$ under interpretation is well-formed, we can conclude (by WF-Rec) that $\mathrm{guarded}_\alpha\ \tau$, and thus that α only occurs in τ under the $\blacktriangleright$ type former of gλ. Given that the $\blacktriangleright$ of gλ is interpreted to *siProp*'s $\triangleright$, we will also come across a $\triangleright$ in $[\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}$ before we ever reach the interpretation of α, that is, Ψ. In other words, the judgement $\mathrm{guarded}_\alpha\ \tau$ implies that Ψ is guarded (by $\triangleright$) in $[\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}$, and therefore that Ψ is guarded in the body of the fixpoint with μ, as required.

Lemma 3.4.1 (Guardedness Property of $[\![-]\!]$)**.** *For all δ and well-formed types τ, we have that Ψ is guarded in $[\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)$ if* $\mathrm{guarded}_\alpha\ \tau$.

The proof is mutual with the definition of $[\![\tau]\!]_\delta$ according to Figure 3.4, as the definition uses contractiveness of rec_pre in the guarded fixpoint in the interpretation of $\mu\alpha.\tau$. We discuss in §6 how we realise this mutual definition and proof in Rocq.

In summary, the guardedness restriction of gλ, which ensures well-foundedness of our object-level types, carries over to our meta language *siProp* to sanction the guarded fixpoint construction used to define $[\![\mu\alpha.\tau]\!]_\delta$.

3.4.6 Semantic Typing Judgement

While the expression interpretation $[\![-]\!]^{\mathrm{e}}$ is defined on closed gλ-terms, the semantic typing judgement should express semantic well-typedness for open terms. To this end, we lift $[\![-]\!]^{\mathrm{e}}$ to open terms via *closing substitutions*, which map term variables to closed terms $e \in \mathit{GExp}$:

$$\mathit{GSubst} \triangleq \mathit{Var} \xrightarrow{\mathsf{fin}} \mathit{GExp}$$

Note that as we are working with a call-by-name semantics, β-reduction performs substitution with expressions rather than values, and our closing substitutions must therefore also replace free variables in e with expressions. The application of a closing substitution $\gamma \in \mathit{GSubst}$ to $e \in \mathit{GExp}$ is denoted by $\gamma(e)$.

[2] By Barendregt's variable convention, we may assume that possible further μ-bindings in τ only bind type variables distinct from α.

Context interpretation. To define the semantic typing judgement $\Gamma \vDash e : \tau$, we wish to express that some $\gamma \in \mathit{GSubst}$ maps term variables to expressions whose behaviour agrees with the types given in Γ. We do so by defining a *context interpretation* $[\![\Gamma]\!]^{\mathrm{c}}_{\delta} : \mathit{GSubst} \to \mathrm{siProp}$, where for each $x : \tau$ in Γ, the expression $\gamma(x)$ must be in $[\![\tau]\!]^{\mathrm{e}}_{\delta}$:

$$[\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma) \triangleq \mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma) \land \forall x.\, \big(\Gamma(x) = \tau \Rightarrow [\![\tau]\!]^{\mathrm{e}}_{\delta}(\gamma(x))\big)\,.$$

With the context interpretation in place, we can finally give the definition of the semantic typing judgement:

$$\Gamma \vDash e : \tau \triangleq \forall \delta, \gamma.\, [\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma) \Rightarrow [\![\tau]\!]^{\mathrm{e}}_{\delta}(\gamma(e))$$

The judgement $\Gamma \vDash e : \tau$ states that in any semantic environment δ, the expression interpretation $[\![\tau]\!]^{\mathrm{e}}_{\delta}$ is satisfied by e under any closing substitution γ that is semantically well-typed, as expressed by $[\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma)$.

3.5 Proof of Strong Normalisation

Having defined the semantic typing judgement $\Gamma \vDash e : \tau$, we now proceed to show in §3.5.1 how our semantic typing judgement implies normalisation, thanks to our use of the total weakest precondition. In §3.5.2 we then sketch out how we prove the semantic typing rules, again closely following Timany et al. [2024]. From the semantic typing rules, we prove the fundamental theorem, stating that our semantic typing judgement $\Gamma \vDash e : \tau$ is implied by the syntactic typing judgement $\Gamma \vdash e : \tau$.

3.5.1 Adequacy

As stated previously in §3.4, adequacy of the semantic typing judgement is more or less immediate, as the total weakest precondition we use in the definition of $[\![-]\!]^{\mathrm{e}}$ implies normalisation. Here, we require adequacy of *siProp* (Theorem 3.2.3).

Lemma 3.5.1 (Adequacy of Total Weakest Precondition)**.** *For any $e \in \mathit{GExp}$ and postcondition $\Phi : \mathit{GVal} \to \mathit{siProp}$, it follows from* $\mathsf{True} \vdash \mathsf{wp}\ e\ [\Phi]$ *that e is strongly normalising.*

Proof. By the definition of the total weakest precondition $\mathsf{wp}\ e\ [\Phi]$ in §3.4, there exists some $v \in \mathit{GVal}$ such that $e \mapsto^* v$. That is, e has a reduction path to a value, which is the only reduction path by determinism of $\mapsto$ (Lemma 2.2.1), and we conclude that e is strongly normalising. Theorem 3.2.3 lets us lift this result to the meta-level. □

Theorem 3.5.2 (Adequacy of Semantic Typing)**.** *Let e be a closed term that is semantically well-typed by $\emptyset \vDash e : \tau$, then e is strongly normalising.*

Proof. By the definition of the semantic typing judgement, $\emptyset \vDash e : \tau$ is equivalent to $[\![\tau]\!]^{\mathrm{e}}_{\delta}(e)$. Unfolding the definition of $[\![\tau]\!]^{\mathrm{e}}_{\delta}(e)$, we get that $\mathsf{wp}\ e\ \big[[\![\tau]\!]_{\delta}\big]$, so e is strongly normalising by Lemma 3.5.1. □

3.5.2 Semantic Typing Rules

In the following, we go over the syntactic typing rules of Figure 2.2, sketching the proofs of their semantic versions. For most of the rules, we keep the discussion brief, and refer the reader to Timany et al. [2024] for further explanations and detailed proof outlines. In our setup, the interesting rules are those for `fold` and `unfold` (i.e. S-Fold and S-Unfold), as well as for the 'later' operations `next` and $\circledast$ (S-Next and S-Star). Once we have all the semantic typing rules established, the *fundamental property*, stating that $\Gamma \vdash e : \tau$ implies $\Gamma \vDash e : \tau$, becomes straightforward to prove by induction on the typing derivation, where each inductive step holds by the respective semantic typing rule.

Contexts and closing substitutions. In most rules, the context Γ remains unchanged between the hypotheses and the conclusion. The goal then gives us $[\![\Gamma]\!]^{\mathrm{c}}_\delta(\gamma)$, with which we specialise the hypotheses to the same γ. We thus prove such rules only for closed expressions, as the desired semantic typing rule then follows immediately.

Monadic rules for the expression interpretation. For the subsequent proofs of the semantic typing rules, we introduce two new derived proof rules for the expression interpretation $[\![-]\!]^{\mathrm{e}}$, which resemble the types of the monadic unit (or return) and bind operator.

$$\frac{[\![-]\!]^{\mathrm{e}}\text{-Val}}{\dfrac{[\![\tau]\!]_\delta(v)}{[\![\tau]\!]^{\mathrm{e}}_\delta(v)}} \qquad \frac{[\![-]\!]^{\mathrm{e}}\text{-Bind}}{\dfrac{[\![\tau]\!]^{\mathrm{e}}_\delta(e) \qquad \forall\, v.\ [\![\tau]\!]_\delta(v) \Rightarrow [\![\tau']\!]^{\mathrm{e}}_\delta(K[v])}{[\![\tau']\!]^{\mathrm{e}}_\delta(K[e])}}$$

The first rule, $[\![-]\!]^{\mathrm{e}}$-Val, is immediate from the definition of $[\![-]\!]^{\mathrm{e}}$ and TWP-Val, while $[\![-]\!]^{\mathrm{e}}$-Bind is a version of the bind rule specialised to the expression interpretation, derived directly from TWP-Bind and TWP-Impl. These rules streamline the following proofs, where we frequently use $[\![-]\!]^{\mathrm{e}}$-Bind to replace a (universally quantified) semantically well-typed subterm e with a semantically well-typed value v. We then obtain the assumption $[\![\tau]\!]_\delta(v)$, which typically provides information about the form of v with which we can reason about $K[v]$ in the new goal. Once we have reduced a term e to a value v under the expression interpretation of some type τ, we can use $[\![-]\!]^{\mathrm{e}}$-Val to switch to the value interpretation of τ and unfold the definition of $[\![\tau]\!]$.

With these rules in place, we are ready to start proving the semantic typing rules.

Base types. For the base types $\mathbf{N}$, $\mathbf{1}$, and $\mathbf{0}$, we must show semantic versions of T-Lit, T-Unit and T-Fail, which we refer to as S-Lit, S-Unit and S-Fail.

$$\frac{\text{S-Lit}}{\Gamma \vDash \mathtt{lit}\ n : \mathbf{N}} \qquad \frac{\text{S-Unit}}{\Gamma \vDash \langle\rangle : \mathbf{1}} \qquad \frac{\text{S-Fail}\quad \Gamma \vDash e : \mathbf{0}}{\Gamma \vDash \mathtt{fail}\ e : \tau}$$

The rule for $\mathbf{N}$ is follows from TWP-Val: $\mathtt{lit}\ n$ is a value form, hence it suffices to show that $[\![\mathbf{N}]\!]_\delta(\mathtt{lit}\ n)$, that is, $\exists\, n' \in \mathbb{N}, \mathtt{lit}\ n = \mathtt{lit}\ n'$, where we simply instantiate n' with n. For $\mathbf{1}$, we proceed analogously, and $[\![\mathbf{1}]\!]_\delta(\langle\rangle)$ again holds trivially.

In the rule for $\mathbf{0}$, we thread through the unchanged context interpretation and ignore the context, as discussed above. The assumption $\vDash e : \mathbf{0}$ then gives us that $e \mapsto^* v$ for some $v \in \mathit{GVal}$ such that $[\![\mathbf{0}]\!]_\delta(v)$. Since $[\![\mathbf{0}]\!]_\delta$ is defined as $\lambda \_.$ False, we have a proof of falsity, and may conclude $[\![\tau]\!]_\delta(\texttt{fail}\ e)$.

Pairs and injections. The semantic versions of T-Pair and T-Inj are given by the following rules:

$$\frac{\Gamma \vDash e_1 : \tau_1 \qquad \Gamma \vDash e_2 : \tau_2}{\Gamma \vDash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \text{S-Pair} \qquad \frac{\Gamma \vDash e : \tau_i \qquad i \in \{1, 2\}}{\Gamma \vDash \texttt{inj}_i\ e : \tau_1 + \tau_2}\ \text{S-Inj}$$

Pairs $\langle e_1, e_2 \rangle$ are values, so we can use rule TWP-Val, by which we must show (again, ignoring contexts) that $[\![\tau_1 \times \tau_2]\!]_\delta\ (\langle e_1, e_2 \rangle)$ for arbitrary but fixed δ. By the definition of $[\![\tau_1 \times \tau_2]\!]$, it remains to show that

$$\exists\, e_1', e_2'.\ \left(\langle e_1, e_2 \rangle = \langle e_1', e_2' \rangle\right) \wedge [\![\tau_1]\!]^{\mathrm{e}}_\delta(e_1') \wedge [\![\tau_2]\!]^{\mathrm{e}}_\delta(e_2').$$

We simply instantiate e_1' and e_2' to satisfy the equality on the left, which leaves us to prove $[\![\tau_1]\!]^{\mathrm{e}}_\delta(e_1)$ and $[\![\tau_2]\!]^{\mathrm{e}}_\delta(e_2)$, both of which are immediate from the hypotheses.

Similarly, injections $\texttt{inj}_i\ e$ are also values, so we proceed in much the same way using TWP-Val and the hypothesis for either $i = 1$ or $i = 2$.

Variables. In the rules so far, the closing substitution γ has just been 'threaded through' unchanged. For T-Var, however, we must show that for a given $\gamma \in \mathit{GSubst}$ with $[\![\Gamma]\!]^{\mathrm{c}}_\delta(\gamma)$, the substitution γ maps the variable x to a term that is semantically of type $\Gamma(x) = \tau$.

$$\frac{\Gamma(x) = \tau}{\Gamma \vDash x : \tau}\ \text{S-Var}$$

This property is more or less 'baked into' the definition of $[\![-]\!]^{\mathrm{c}}$, which tells us that there exists some e such that $\gamma(x) = e$ and $[\![\tau]\!]^{\mathrm{e}}(\gamma(x))$, by which e semantically inhabits τ.

Functions. For λ-abstractions and function application we have the following semantic versions of T-Abs and T-App:

$$\frac{\Gamma, x : \tau_1 \vDash e : \tau_2}{\Gamma \vDash \lambda x.\ e : \tau_1 \to \tau_2}\ \text{S-Abs} \qquad \frac{\Gamma \vDash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vDash e_2 : \tau_1}{\Gamma \vDash e_1\ e_2 : \tau_2}\ \text{S-App}$$

Let us first look at S-Abs: As always, we unfold the definition of the semantic typing judgement, giving us a γ with $[\![\Gamma]\!]^{\mathrm{c}}_\delta(\gamma)$. Since $\lambda x.\ e$ is a value, we again use TWP-Val, by which we must show that $[\![\tau_1 \to \tau_2]\!]_\delta(\lambda x.\ \gamma(e))$. The interpretation of the function type unfolds to

$$\forall e'.\ [\![\tau_1]\!]^{\mathrm{e}}_\delta(e') \Rightarrow [\![\tau_2]\!]^{\mathrm{e}}_\delta((\lambda x.\ \gamma(e))\ e'),$$

where we can step the β-redex on the right-hand side to $\gamma(e)[e'/x]$ according to rule TWP-Step. We can write this expression as $\gamma'(e)$, where $\gamma' \triangleq (\gamma, x \mapsto e')$ merges the γ and the substitution of x with e'. We then have that $[\![\Gamma, x : \tau_1]\!]^{\mathrm{c}}_\delta(\gamma')$ by the fact that $\gamma'(x) = e'$ and $[\![\tau_1]\!]^{\mathrm{e}}_\delta(e')$, as well as $[\![\Gamma]\!]^{\mathrm{c}}_\delta(\gamma)$. As such, we can apply the hypothesis $\Gamma, x : \tau_1 \vDash e : \tau_2$ with δ and γ', which gives us precisely $[\![\tau_2]\!]^{\mathrm{e}}_\delta(\gamma'(e))$.

Now for S-App: here, Γ again remains unchanged between the hypotheses and the conclusion, allowing us to prove the rule via an auxiliary result for closed terms. Here, we finally get to see $[\![-]\!]^{\mathrm{e}}$-Bind in action, which we use with the context $K \triangleq \cdot\ e_2$ to replace e_1 with a value. We use the left hypothesis, which gives us some v in $[\![\tau_1 \to \tau_2]\!]^{\mathrm{e}}_\delta$, and leaves us to show that $[\![\tau_2]\!]^{\mathrm{e}}_\delta(v\ e_2)$. This time, the interpretation of the function type unfolds to

$$\forall e.\ [\![\tau_1]\!]^{\mathrm{e}}_\delta(e) \Rightarrow [\![\tau_2]\!]^{\mathrm{e}}_\delta(v\ e),$$

which—when specialised to e_2—gives us exactly the proof step we need. All that remains to show is the left side, that is, that e_2 is in $[\![\tau_2]\!]^{\mathrm{e}}_\delta$, which is precisely what the right hypothesis states.

Projections and pattern matches. Next, we have the rules for projections $\mathtt{proj}_i\ e$ and pattern matches $\mathtt{case}\ e\ \mathtt{of}\ x_1.e_1\,;x_2.e_2$. The semantic version of T-Proj is given by

$$\textsc{S-Proj}\quad \frac{\Gamma \vDash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Gamma \vDash \mathtt{proj}_i\ e : \tau_i}$$

The proof resembles those we have seen so far, so we keep the discussion brief. By rule $[\![-]\!]^{\mathrm{e}}$-Bind, we use the assumption to take e to a value v in $[\![\tau_1 \times \tau_2]\!]_\delta$. The value interpretation of the product type states that v is then a pair of the form $\langle e_1, e_2\rangle$ such that $[\![\tau_1]\!]^{\mathrm{e}}_\delta(e_1)$ and $[\![\tau_2]\!]^{\mathrm{e}}_\delta(e_2)$. We thus have the redex $\mathtt{proj}_i\ \langle e_1, e_2\rangle$, which we can—by case distinction on i—reduce to either e_1 or e_2 in a single step by TWP-Step.

For the semantic version of T-Case, the proof closely resembles that of S-Abs, since the (semantic) typing context Γ is extended in the hypotheses.

$$\textsc{S-Case}\quad \frac{\Gamma \vDash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vDash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vDash e_2 : \tau}{\Gamma \vDash \mathtt{case}\ e\ \mathtt{of}\ x_1.e_1\,;x_2.e_2 : \tau}$$

Again, $[\![-]\!]^{\mathrm{e}}$-Bind is used in combination with the left assumption to create a redex $\mathtt{case}\ \mathtt{inj}_i\ e\ \mathtt{of}\ x_1.e_1\,;x_2.e_2$, which can be stepped to either $e_1[e/x]$ or $e_2[e/x]$, depending on i. From there, we simply apply one of the two rightmost hypotheses with the closing substitution extended with $x_1 \mapsto e$, or $x_2 \mapsto e$, respectively.

Folds and unfolds. Proving the semantic version of T-Fold and T-Unfold—shown below—again involves much of the same proofs steps we have seen so far. There are, however, two auxiliary results we must introduce.

$$\textsc{S-Fold}\quad \frac{\Gamma \vDash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vDash \mathtt{fold}\ e : \mu\alpha.\tau} \qquad \textsc{S-Unfold}\quad \frac{\Gamma \vDash e : \mu\alpha.\tau}{\Gamma \vDash \mathtt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

Firstly, we need that the interpretation of a recursive type $\mu\alpha.\tau$ can be unrolled by one step as follows.

Lemma 3.5.3. *For all $v \in \mathit{GVal}$, we have that*

$$[\![\mu\alpha.\tau]\!]_\delta(v) \dashv\vdash \exists\, e.\ (v = \mathtt{fold}\ e) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto[\![\mu\alpha.\tau]\!]_\delta}(e).$$

Proof. The bi-implication is just μ-Unfold, specialised to the definition of $[\![\mu\alpha.\tau]\!]_\delta$. □

Secondly, we need the following standard substitution lemma for logical relations.

Lemma 3.5.4. *For all $v \in \mathit{GVal}$, we have that*

$$[\![\tau]\!]_{\delta,\alpha\mapsto[\![\tau']\!]_\delta}(v) \dashv\vdash [\![\tau[\tau'/\alpha]]\!]_\delta(v).$$

Proof. The above result is proven by induction on τ. □

Intuitively, the above states that substitution in types under the value interpretation $[\![-]\!]$ corresponds to extending the semantic environment supplied to $[\![-]\!]$. Combining the previous two results, we get the following bi-implication.

Lemma 3.5.5. *For $v \in \mathit{GVal}$, we can unroll $[\![\mu\alpha.\tau]\!]$ as follows:*

$$[\![\mu\alpha.\tau]\!]_\delta(v) \dashv\vdash \exists\, e.\ (v = \mathtt{fold}\ e) \wedge [\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(e)$$

Proof. In the statement of Lemma 3.5.3 we apply Lemma 3.5.4 to replace the extension of δ with a substitution in τ. □

In the proof of S-Fold, we apply TWP-Val since $\mathtt{fold}\ e$ is a value form. We must then show $[\![\mu\alpha.\tau]\!]_\delta(\mathtt{fold}\ e)$, which unrolls by Lemma 3.5.5 to

$$\exists\, e'.\ (\mathtt{fold}\ e = \mathtt{fold}\ e') \wedge [\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(e').$$

The left equality holds by instantiating e' to e, while the right is exactly the hypothesis.

For S-Unfold, we again use $[\![-]\!]^{\mathrm{e}}$-Bind, giving us the term $\mathtt{unfold}\ v$, where unrolling the hypothesis by Lemma 3.5.5 tells us that $v = \mathtt{fold}\ e$ for some e with $[\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(e)$. As such, we must show that $\mathtt{unfold}\ (\mathtt{fold}\ e)$ is in the interpretation of $\tau[\mu\alpha.\tau/\alpha]$, and since $\mathtt{unfold}\ (\mathtt{fold}\ e) \mapsto e$, we are done by TWP-Step.

Note that neither proof involves reasoning about the later modality $\triangleright$, and this instead occurs in the rules for $\mathtt{next}$ and $\circledast$, as we detail in the following.

Later operations. Finally, we come to the semantic versions of T-Next and T-Star.

$$\text{S-Next}\ \frac{\Gamma \vDash e : \tau}{\Gamma \vDash \mathtt{next}\ e : \blacktriangleright\tau} \qquad \text{S-Star}\ \frac{\Gamma \vDash e_1 : \blacktriangleright(\tau_1 \to \tau_2) \qquad \Gamma \vDash e_2 : \blacktriangleright\tau_1}{\Gamma \vDash e_1 \circledast e_2 : \blacktriangleright\tau_2}$$

In S-Next, the term $\mathtt{next}\ e$ is a value form, so we proceed with TWP-Val, by which we must show $[\![\blacktriangleright\tau]\!]_\delta(\mathtt{next}\ e)$. Unfolding the definition of $[\![\blacktriangleright\tau]\!]$, we get $\triangleright[\![\tau]\!]^{\mathrm{e}}_\delta(e)$, which is immediate from the hypothesis and $\triangleright$-Intro.

The proof of S-Star again resembles the previous proofs: we use $[\![-]\!]^{\mathrm{e}}$-Bind to bring $e_1 \circledast e_2$ into the form $v_1 \circledast v_2$, where the hypotheses give us that both values are next's, so our goal is of the form $[\![\tau_2]\!]^{\mathrm{e}}_{\delta}\left((\mathtt{next}\ e_1') \circledast (\mathtt{next}\ e_2')\right)$. Here, we can reduce by TWP-Step to $[\![\tau_2]\!]^{\mathrm{e}}_{\delta}\left(\mathtt{next}(e_1'\ e_2')\right)$, which by TWP-Val follows from $\triangleright[\![\tau_2]\!](e_1'\ e_2')$. At this point, we can strip away the $\triangleright$ on the goal using the $\triangleright$'s on both hypotheses by $\triangleright$-And along with $\triangleright$-Mono. With the $\triangleright$'s gone, the remaining steps are identical to those for S-App.

3.5.3 Strong Normalisation Result

With all the semantic typing rules at hand, the fundamental property follows more or less immediately, as promised in §3.1. By virtue of inducting on the typing derivation, each case of the induction precisely matches one of the rules from the previous section.

Lemma 3.5.6 (Fundamental Property)**.** *If $e \in \mathit{GExp}$ is syntactically well-typed in context Γ with type τ, that is, $\Gamma \vdash e : \tau$, it follows that e is also semantically well-typed by $\Gamma \vDash e : \tau$.*

Proof. We proceed by induction on $\Gamma \vdash e : \tau$. Let us consider the case for T-App:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

We may assume the inductive hypothesis for the assumptions of the rule, namely the judgements $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$, giving us their semantic versions, $\Gamma \vDash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vDash e_2 : \tau_1$, and we must show that $\Gamma \vDash e_1\ e_2 : \tau_2$. This is exactly what rule S-App states, so we just apply it with the inductive hypotheses. The remaining cases proceed in the same way, using each of our semantic typing rules. □

By composing the fundamental property with the adequacy proof of Theorem 3.5.2, we finally show our desired result of strong normalisation for well-typed gλ-programs.

Theorem 3.5.7 (Strong Normalisation of gλ)**.** *Let $e \in \mathit{GExp}$. If e is closed and well-typed by $\emptyset \vdash e : \tau$, then e is strongly normalising.*

Proof. The fundamental property states that the syntactic typing judgement $\emptyset \vdash e : \tau$ implies semantic well-typedness $\emptyset \vDash e : \tau$. By Theorem 3.5.2 (Adequacy), we then have that e is strongly normalising. □

Chapter 4

Translation of gλ-Terms

The type system of the gλ-calculus allows us to write self-referential programs with the guarantee of productivity and strong normalisation, provided that we satisfy the guardedness restriction on recursive types. As such, programming in the gλ-calculus, or a more fleshed-out language with the same mechanism of guarded self-references, could offer some appealing properties, especially when working with infinite data structures such as streams. However, it is not immediately obvious how we could execute our programs efficiently.

To this end, we propose a program translation from the gλ-calculus to a version of the untyped call-by-value λ-calculus featuring products, sums, and natural numbers. We will proceed to show that this program translation, or compilation, preserves both program behaviour and normalisation, the proof of which we have mechanised in Rocq using the Iris framework. Our program translation effectively 'erases' the guarded constructs from the source program, yielding a target program that we could compile to efficient low-level code by leveraging existing infrastructure for functional languages. For instance, our target language is essentially a small subset of the Malfunction language of Dolan [2016], a wrapper around OCaml's untyped intermediate representation `Lambda`, which strongly resembles the untyped λ-calculus.

In the following, we will define the syntax and operational semantics of our untyped target language in §4.1, while §4.2 specifies our program translation from the gλ-calculus to our target language. §5 then covers how we state the correctness of our program translation, and how we prove it using a binary logical relation on typed gλ-terms and untyped terms of the target language.

4.1 Target Language

As our target language, we choose an extension of the untyped λ-calculus with pairs and projections, as well as injections and pattern matches, corresponding to the constructs for products and sums present in the gλ-calculus. We similarly include natural number literals. While we could of course represent all the above using Church encodings in a more minimal calculus, doing so would neither result in efficient code, nor would it be straightforward to relate the encodings to the constructs of the source language when

reasoning about our translation. Importantly, our target language does not feature the 'later' operations `next` and $\circledast$, as these are the constructs we erase in our translation.

Aside from the absence of `next` and $\circledast$, a key deviation from the gλ-calculus is the evaluation strategy: where the gλ-calculus uses lazy evaluation, specifically *call-by-name*, our target language is instead *strict*, or *eager*, employing a *call-by-value* evaluation order. It would arguably be simpler to reason about translating from gλ to another call-by-name language, but we choose an eager language as our target for two main reasons.

- For one, it is—generally speaking—particularly challenging to produce efficient low-level code from lazy functional languages, which involves storing unevaluated expressions (*thunks*) on the heap [Johnsson, 1987; Smetsers et al., 1991].
- Secondly, if we wish to extend our translation to a verified pipeline targeting a 'real-world' language, a strict target is a more suitable candidate, as it brings us closer to intermediate representations, such as OCaml's `Lambda`, or MALFUNCTION.

Having motivated some of our choices regarding the target for our translation, let us now give a formal definition of the target language.

4.1.1 Syntax

Much of the syntax overlaps with that of gλ, and in the following, we will frequently see both languages side-by-side, though it is (nearly) always clear from the context which term belongs to which language. To make the distinction explicit without adding confusing alternative notations and keywords for each of the constructs, we mark the constructs of the target language in red for a clear visual differentiation, and will use blue for gλ's term formers from now on.

The term and value syntax of our target language is presented below.

$$
\begin{array}{lcll}
\mathit{UExp} \ni e & ::= & x & \textit{Variables} \\
& | & \mathtt{lit}\ n & \textit{Natural numbers} \\
& | & \langle\rangle \mid \langle e,e\rangle \mid \mathtt{proj}_1\ e \mid \mathtt{proj}_2\ e & \textit{Products} \\
& | & \mathtt{fail}\ e \mid \mathtt{inj}_1\ e \mid \mathtt{inj}_2\ e & \textit{Sums} \\
& | & \mathtt{case}\ e\ \mathtt{of}\ x_1.e\,;x_2.e & \\
& | & \lambda x.e \mid e\ e & \textit{Functions} \\
\\
\mathit{UVal} \ni v & ::= & \mathtt{lit}\ n \mid \langle\rangle \mid \langle v,v\rangle & \textit{Values} \\
& | & \mathtt{inj}_1\ v \mid \mathtt{inj}_2\ v \mid \lambda x.e &
\end{array}
$$

Note that our change of evaluation strategy compared to gλ also impacts the value forms, where pairs and injections are now values only if their subterms are themselves also values.

4.1.2 Operational Semantics

The value forms are a consequence of the strict evaluation on pairs and injections: where gλ does not evaluate under the term formers $\langle -, - \rangle$ and $\mathtt{inj}_i\ -$, we do evaluate under

the corresponding constructs in the target language, as shown in the following grammar for evaluation contexts K.

$$\begin{array}{rcl} K & ::= & \cdot \mid \langle K,e\rangle \mid \langle v,K\rangle \mid \mathtt{proj}_1\, K \mid \mathtt{proj}_2\, K \quad \textit{Evaluation contexts} \\ & \mid & \mathtt{fail}\, K \mid \mathtt{inj}_1\, K \mid \mathtt{inj}_2\, K \\ & \mid & \mathtt{case}\, K\, \mathtt{of}\, x_1.e_1\,;x_2.e_2 \mid K\, e \mid v\, K \end{array}$$

The last two cases state that we reduce function arguments prior to β-reduction of function application, as is standard for call-by-value evaluation. We also point out the fixed left-to-right evaluation order, as seen in the evaluation contexts for pairs and application.

The base reduction rules, given below, closely resemble those of gλ, but without the β-rule for `fold`/`unfold`, and the special rule concerning $\circledast$. In addition, the rules make explicit that function arguments and pair/injection entries must be reduced to values prior to applying the rules for β-reduction, thereby fixing the strict evaluation order, and ensuring determinism of the language.

$$\begin{array}{rcll} (\lambda x.e)\, v & \mapsto_{\mathrm{b}} & e[v/x] & \\ \mathtt{proj}_i\, \langle v_1,v_2\rangle & \mapsto_{\mathrm{b}} & v_i & i \in \{1,2\} \\ \mathtt{case}\, \mathtt{inj}_i\, v\, \mathtt{of}\, x_1.e_1\,;x_2.e_2 & \mapsto_{\mathrm{b}} & e_i[v/x_i] & i \in \{1,2\} \end{array}$$

Again, the step relation is given by $K[e_1] \mapsto K[e_2]$ for any context K, where $e_1 \mapsto_{\mathrm{b}} e_2$ holds by the base step relation defined above, and we write $\mapsto^*$ for the reflexive-transitive closure of $\mapsto$.

Lemma 4.1.1 (Determinism of $\mapsto$)**.** *The reduction relation $\mapsto$ on the target language is deterministic: Given $e, e', e'' \in \mathit{UExp}$, if both $e \mapsto e'$ and $e \mapsto e''$, we have that $e' = e''$.*

Lemma 4.1.2 (Stepping under contexts)**.** *For any evaluation context K and $e, e' \in \mathit{UExp}$, if $e \mapsto e'$, then also $K[e] \mapsto K[e']$.*

Lemma 4.1.3 (Substitutivity of $\mapsto$)**.** *For $e, e' \in \mathit{UExp}$ and $\gamma = x_1 \mapsto e_1, \ldots, x_n \mapsto e_n$ a substitution, we have that*

$$e \mapsto e' \quad \Rightarrow \quad \gamma(e) \mapsto \gamma(e').$$

We need the latter property to show that forcing a thunk under a substitution yields the delayed term under the same substitution, a result we use in §5.4.

In gλ, the value form `next` e has the key role of 'wrapping' an arbitrary expression as a value, thereby preventing its evaluation. This property, together with the guardedness requirement for $\blacktriangleright$, ensures that infinite structures, such as streams, are only ever evaluated for a finite number of steps. While well-typed gλ-programs enjoy the strong normalisation result of §3, the untyped target language does not satisfy normalisation, and one can easily construct diverging terms, such as the canonical example

$$\Omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x).$$

Despite this, we are still able to compile from gλ to the untyped language in such a way that we retain the normalisation property of the gλ type system, as we will show in §5. A key ingredient for this result is the translation of `next`'s in the source language to *thunks* in the target language, as we will see in the following section.

4.2 Translation

We now show how we translate to our target language from the gλ-calculus. In order to maintain a clear distinction when talking about terms from both languages, we will use ge as the meta-symbol for gλ-terms, and ue for terms of the untyped target.

In gλ, the `next` term former prevents evaluation of its subterm, which plays a vital role in ensuring termination of the language. When the type system states that data is only available 'later', this fact is enforced at the term level by `next`. More specifically, `next` is the introduction form of the later type modality ▶, and a program of type $\blacktriangleright\tau$ always reduces to a value of the form `next` ge, as captured by our semantic interpretation of gλ-types defined in §3.4. It is in this sense that `next` acts as a guard at the term level, that is, it delays the evaluation for (sub)terms of a guarded type.

As such, we must take care to compile expressions of the form `next` ge such that evaluation of the subexpression ge is delayed in the target language. We achieve this behaviour by compiling occurrences of `next` in the source program to *thunks* in the target language: by wrapping an untyped expression ue in a function $\lambda x.ue$, we obtain a value, and prevent evaluation of the function body. The meta-theoretic function $delay : UExp \to UExp$, defined by

$$delay(ue) \triangleq \lambda x.ue \text{ where } x \notin FV(ue),$$

does precisely such, using an arbitrary variable x that does not occur unbound in e. To force evaluation of ue, we simply supply the unit value $\langle\rangle$ as a dummy argument to the thunk, as described by $force : UExp \to UExp$, defined as follows:

$$force(ue) \triangleq ue\ \langle\rangle.$$

Figure 4.1 defines the translation of gλ-terms to our untyped target language. To avoid overloading the Scott brackets $[\![-]\!]$ yet again, we write $(\!|-|\!)$, where $(\!|ge|\!) \in UExp$ denotes the result of translating $ge \in GExp$. The translation $(\!|-|\!)$ is expressed as a function defined by straightforward structural recursion on the source expression ge.

Most of the lines in Figure 4.1 are much as expected, as the source language constructs are simply translated to their untyped counterparts, where the subterms are replaced by the result of their (recursive) translation. The `fold` and `unfold` term formers are not present in the target language; their role is that of folding or unfolding the μ-type of their subterm, hence we simply erase them during translation.

For `next` and ⊛, we take a closer look: as mentioned before, gλ-terms of the form `next` ge are thunked in the target language, and similarly for ⊛. The operator ⊛ facilitates application under a `next`, where the function ge_1 and argument ge_2 both reduce to values of the form `next` ge in the source language. As such, the compiled terms ue_1 and ue_2 will reduce to thunks in the target language, both of which we must force before 're-thunking' their application, in the same way that the reduction rule of gλ re-wraps the application of the subterms in a `next`:

$$(\texttt{next}\ ge_1) \circledast (\texttt{next}\ ge_2) \mapsto \texttt{next}\ (ge_1\ ge_2).$$

$$
\begin{array}{rcl}
\mathit{delay}(\mathit{ue}) & \triangleq & \lambda x.\mathit{ue} \text{ where } x \notin \mathit{FV}(\mathit{ue}) \\
\mathit{force}(\mathit{ue}) & \triangleq & \mathit{ue}\ \langle\rangle \\
\\
(\!|x|\!) & \triangleq & x \\
(\!|\langle\rangle|\!) & \triangleq & \langle\rangle \\
(\!|\texttt{lit}\ n|\!) & \triangleq & \texttt{lit}\ n \\
(\!|\langle \mathit{ge}_1, \mathit{ge}_2 \rangle|\!) & \triangleq & \langle (\!|\mathit{ge}_1|\!), (\!|\mathit{ge}_2|\!) \rangle \\
(\!|\texttt{proj}_i\ \mathit{ge}|\!) & \triangleq & \texttt{proj}_i\ (\!|\mathit{ge}|\!) \\
(\!|\texttt{fail}\ \mathit{ge}|\!) & \triangleq & \texttt{fail}\ (\!|\mathit{ge}|\!) \\
(\!|\texttt{inj}_i\ \mathit{ge}|\!) & \triangleq & \texttt{inj}_i\ (\!|\mathit{ge}|\!) \\
(\!|\texttt{case}\ \mathit{ge}\ \texttt{of}\ x_1.\mathit{ge}_1 ; x_2.\mathit{ge}_2|\!) & \triangleq & \texttt{case}\ (\!|\mathit{ge}|\!)\ \texttt{of}\ x_1.(\!|\mathit{ge}_1|\!) ; x_2.(\!|\mathit{ge}_2|\!) \\
(\!|\lambda x.\mathit{ge}|\!) & \triangleq & \lambda x.(\!|\mathit{ge}|\!) \\
(\!|\mathit{ge}_1\ \mathit{ge}_2|\!) & \triangleq & (\!|\mathit{ge}_1|\!)\ (\!|\mathit{ge}_2|\!) \\
(\!|\texttt{fold}\ \mathit{ge}|\!) & \triangleq & (\!|\mathit{ge}|\!) \\
(\!|\texttt{unfold}\ \mathit{ge}|\!) & \triangleq & (\!|\mathit{ge}|\!) \\
(\!|\texttt{next}\ \mathit{ge}|\!) & \triangleq & \mathit{delay}((\!|\mathit{ge}|\!)) \\
(\!|\mathit{ge}_1 \circledast \mathit{ge}_2|\!) & \triangleq & \mathit{delay}(\mathit{force}((\!|\mathit{ge}_1|\!))\ \mathit{force}((\!|\mathit{ge}_2|\!)))
\end{array}
$$

Figure 4.1: Translation of gλ-terms to untyped target language

In essence, the translation thus boils down to replacing `next`'s with thunks, and converting $\circledast$ to a thunked function application, where the function and argument are both thunks that must first be forced; all other cases are trivial. However, verifying the translation formally is not so trivial, as we will see in the following chapter.

Chapter 5

Correctness of Program Translation

Having defined the translation of typed gλ-terms to our untyped target language, we will now look at properties of said translation (or compilation). In particular, we are interested in the preservation of program behaviour, in the sense that if some gλ-term e reduces to a given value, e.g. to the literal `lit 5`, the result of its compilation $(\!|e|\!)$ should reduce to the corresponding value according to the semantics of the target language.

To show that our compiler satisfies the desired properties, we will (again) use a logical relation, closely following our approach in §3. However, a notable distinction to our strong normalisation proof for gλ is that we now use a *binary* logical relation, which relates gλ-terms and target language terms at a given gλ-type. The logical relation expresses a form of semantic equivalence between source and target terms at the respective type. Using the logical relation, we show concretely that at type $\mathbf{N}$, both terms must evaluate to literals in the respective languages, and the values of the literals must match.

Proof overview. We formally state the properties we prove of the translation and motivate their choice in §5.1. The proof itself features the same steps as the strong normalisation proof (see §3.1), again following Timany et al. [2024]:

- We reuse the *siProp* logic introduced in §3.2, but now employ a binary program logic, which we derive in §5.2. The program logic features a binary WP connective that relates a source and target term, along with corresponding proof rules.
- In §5.3, we use this binary WP connective to construct the binary logical relation $[\![\tau]\!]_\delta : \mathit{GVal} \times \mathit{UVal} \to \mathrm{siProp}$, which follows the unary logical relation of our SN proof. Using closing substitutions, we lift the logical relation to open terms in our binary semantic typing judgement of the form $\Gamma \vDash \mathit{ge} \approx \mathit{ue} : \tau$.
- Finally, in §5.4, we again prove *adequacy*, i.e., that semantic typing for closed terms $\emptyset \vDash \mathit{ge} \approx \mathit{ue} : \tau$ implies our desired property of behavioural equivalence, as well as the *Fundamental Property*. The latter is specific to $(\!|-|\!)$, stating that well-typed gλ-terms $\Gamma \vdash \mathit{ge} : \tau$ satisfy $\Gamma \vDash \mathit{ge} \approx (\!|\mathit{ge}|\!) : \tau$. We prove binary semantic variants of gλ's typing rules, from which the Fundamental Property (again) follows by induction on the typing judgement of gλ. Combined with the adequacy proof, we can then show our correctness result for $(\!|-|\!)$.

5.1 Correctness Properties

Stating that a compiler or program translation is *correct* is taken to mean a number of different properties in different contexts, so we must first make precise what properties we (aim to) prove of our translation from gλ to the target language. We are interested in the preservation of semantics or program behaviour, in particular the termination of programs with a given value. Ideally, all possible behaviours of a source program should be preserved in the target, while at the same time, the target's behaviours should be restricted to those of the source, i.e., no new behaviours are introduced under translation. Our source language is both pure and total, since well-typed programs in gλ are strongly normalising, as shown in Theorem 3.5.7. Accordingly, the only behaviour of interest is that of termination with a given result value, which we expect to be preserved under translation as the only possible behaviour of the target.

Comparing values is not equally meaningful for all types though: for instance, given a gλ-term of some function type $\tau_1 \to \tau_2$, we expect both the source and target program to evaluate to a λ-abstraction, which we cannot (syntactically) compare between the two languages. By contrast, a gλ-term ge of type $\mathbf{N}$ reduces to a literal $\texttt{lit}\ n$, while we expect the target term $(\!|ge|\!)$ to reduce to $\texttt{lit}\ m$, where we can compare n and m directly.

Indeed, the property that we show is specific to results of type $\mathbf{N}$, where we require the literals to match, both from the source to the target and vice versa:

(1) For a closed source expression ge of type $\mathbf{N}$, i.e. with $\emptyset \vdash ge : \mathbf{N}$, if $ge \mapsto^* \texttt{lit}\ n$, then also $(\!|ge|\!) \mapsto^* \texttt{lit}\ n$.

(2) Vice versa, again for ge with $\emptyset \vdash ge : \mathbf{N}$, if $(\!|ge|\!) \mapsto^* \texttt{lit}\ n$, then also $ge \mapsto^* \texttt{lit}\ n$.

Viewing the result value as a behaviour of the program, the first property constitutes a *forward simulation* in the terminology of Leroy [2009b][1], since we require that the behaviour present in the source is also present in the target. Analogously, we can say that the second property is a form of *backwards simulation*. In each case, we are only considering source programs which are safe, since the source term is closed and well-typed, and thus normalises without getting stuck by Theorem 3.5.7.

Importantly, both gλ and our target language are also deterministic, meaning that properties (1) and (2) actually imply each other, given the fact that for $\emptyset \vdash ge : \mathbf{N}$, we have that $ge \mapsto^* \texttt{lit}\ n$ for some $n \in \mathbb{N}$, which follows from Lemma 3.5.6. As a result, (1) and (2) are indeed both equivalent to the *bisimulation* property, stating that the behaviours of the source and compiled program are exactly identical. More formally, both (1) and (2) are equivalent to stating that for $\emptyset \vdash ge : \mathbf{N}$, we have the bi-implication

$$ge \mapsto^* \texttt{lit}\ n \iff (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Note that the above statements only consider those terms of the target language obtained by compiling a closed, well-typed gλ-term. As pointed out in §4.1.2, our target

[1] Leroy employs a big-step trace semantics for the source and target. Since we consider only termination and not other behaviours such as I/O and system calls, we equate big-step reduction with multistep reduction ($\mapsto^*$) to a value for our languages.

language does not satisfy normalisation, and features diverging terms such as Ω. It does not satisfy safety either, as the target language is untyped, and thus permits stuck terms such as $\texttt{case}\ (\texttt{lit}\ 5)\ \texttt{of}\ x_1.ue_1\,;x_2.ue$, which is neither a value, nor can it be reduced according to $\mapsto$. However, for closed and well-typed gλ-terms as inputs, we expect the terms returned by $(\!|-|\!)$ to satisfy not only safety, but also strong normalisation.

For the above correctness property, well-typedness in the source is a necessary condition: if we consider arbitrary gλ-terms, we have e.g. $(\lambda x.\texttt{lit}\ 2)\ \Omega$, which normalises in the call-by-name gλ-calculus, while its translation $(\lambda x.\texttt{lit}\ 2)\ \Omega$ diverges in the strict target language. We also have terms such as $(\texttt{unfold}\ (\lambda x.\texttt{fold}\ \texttt{lit}\ 2))\ \texttt{lit}\ 3$, which is stuck in g$\lambda$, though its translation $(\lambda x.\texttt{lit}\ 2)\ \texttt{lit}\ 3$ reduces to $\texttt{lit}\ 2$ in the target.

5.2 Binary Program Logic

For our binary logical relation, we again work in *siProp*, and refer back to §3.2 for the definition, as well as the proof rules and semantic model. In the following, we wish to reason about terms from the source and target language in relation to each other. We define a binary weakest precondition on a pair of expressions $ge \in \mathit{GExp}$ and $ue \in \mathit{UExp}$, which we denote by $\mathsf{wp}\ ge \sim ue\ [\Phi]$. As before, we will show proof rules for the binary weakest precondition, which correspond to those mentioned above, allowing us to reason modularly and abstractly about the new WP connective.

5.2.1 Binary Total Weakest Precondition

In §3.3, we defined a total weakest precondition in *siProp*, denoted by $\mathsf{wp}\ e\ [\Phi]$, which states that the expression e reduces to some value v such that $\Phi(v)$ holds. For our binary logical relation, we define a total binary WP connective $\mathsf{wp}\ ge \sim ue\ [\Phi]$, where the postcondition $\Phi : \mathit{GVal} \times \mathit{UVal} \to \mathit{siProp}$ is a binary predicate on a value from each language. The binary WP should state that ge and ue reduce to values gv and uv, respectively, which satisfy the postcondition by $\Phi(uv, uv)$.

Definition 5.2.1 (Binary Total Weakest Precondition)**.** We define the binary total WP connective as follows:

$$\begin{array}{rcl} \mathsf{wp}\ (-) \sim (-)\ [-] & : & \mathit{GExp} \to \mathit{UExp} \to (\mathit{GVal} \times \mathit{UVal} \to \mathsf{siProp}) \to \mathsf{siProp} \\ \mathsf{wp}\ ge \sim ue\ [\Phi] & \triangleq & \exists\, gv, uv.\ (ge \mapsto^* gv) \land (ue \mapsto^* uv) \land \Phi(gv, uv) \end{array}$$

Lemmas 2.2.1 and 4.1.1 state that the stepping relations for both languages are deterministic, meaning that the reductions $ge \mapsto^* gv$ and $ue \mapsto^* uv$ are indeed the only reduction paths for ge and ue. Since there are no other reduction paths by which these expressions can get stuck, they are safe to reduce by their respective step relations.

Alternatively, we could define the binary WP by nesting unary WPs for the two languages. We defer further discussion on this choice to §6, but in short, the custom definition above lets us more easily step the two terms independently and in any order, which matters for proving the semantic typing rules.

Rules for the binary total weakest precondition:

$$\frac{\text{wp } ge' \sim ue\ [\Phi] \qquad ge \mapsto ge'}{\text{wp } ge \sim ue\ [\Phi]}\ \textsc{TwpBin-StepL} \qquad \frac{\text{wp } ge \sim ue'\ [\Phi] \qquad ue \mapsto ue'}{\text{wp } ge \sim ue\ [\Phi]}\ \textsc{TwpBin-StepR}$$

$$\frac{\Phi(gv, uv)}{\text{wp } gv \sim uv\ [\Phi]}\ \textsc{TwpBin-Val} \qquad \frac{\text{wp } ge \sim ue\ \left[(gv, uv).\ \text{wp } K_g[gv] \sim K_u[uv]\ [\Phi]\right]}{\text{wp } K_g[ge] \sim K_u[ue]\ [\Phi]}\ \textsc{TwpBin-Bind}$$

$$\frac{\text{wp } ge \sim ue\ [\Phi] \qquad \forall\, gv, uv.\ \Phi(gv, uv) \Rightarrow \Psi(gv, uv)}{\text{wp } ge \sim ue\ [\Psi]}\ \textsc{TwpBin-Impl}$$

Figure 5.1: Derived rules for the binary total WP connective

5.2.2 Proof Rules

With the definition of the total binary WP in place, we derive proof rules analogous to those given for our unary WP in §3.3, i.e., matching TWP-Step, TWP-Val, TWP-Bind, and TWP-Impl. We write wp $ge \sim ue\ [(gv, uv).\ P]$ to mean wp $ge \sim ue\ [\lambda\ (gv, uv).\ P]$, and write $\lambda\ (gv, uv).\ P$ as a shorthand for $\lambda\ (v : GVal \times UVal).\ P[\pi_1(v)/gv, \pi_2(v)/uv]$.

The derived rules for wp $ge \sim ue\ [\Phi]$, shown in Figure 5.1, are explained below:

- TwpBin-StepL and TwpBin-StepR let us reduce the left or right term by one step. When proving the semantic typing rules, we cannot always step both terms in parallel, hence we need two separate rules. Let us consider TwpBin-StepL: unfolding the WP's definition, the left assumption states that ge' and ue step to some values gv and uv satisfying Φ. Since $ge \mapsto ge'$, we also have that $ge \mapsto^* gv$, and thus that wp $ge \sim ue\ [\Phi]$. The same reasoning applies to TwpBin-StepR.

- The rule TwpBin-Val, is immediate from our definition of the binary WP, since for gv and uv with $\Phi(gv, uv)$, we have wp $gv \sim uv\ [\Phi]$ (by $gv \mapsto^* gv$ and $uv \mapsto^* uv$).

- For TwpBin-Bind, we have by the assumption that $ge \mapsto^* gv$ and $ue \mapsto^* uv$, from which (by Lemma 2.2.2) also $K_g[ge] \mapsto^* K_g[gv]$ for all contexts K_g, and likewise for $K_u[ue] \mapsto^* K_u[uv]$ (by Lemma 4.1.2). Repeated application of TwpBin-StepL and TwpBin-StepR then gives us that wp $K_g[ge] \sim K_u[ue]\ [\Phi]$.

- Finally, TwpBin-Impl again expresses monotonicity of the WP connective in the postcondition, that is, we can weaken the postcondition Φ, replacing it with Ψ if $\Phi(gv, uv) \Rightarrow \Psi(gv, uv)$ for all gv and uv.

With the binary WP at hand, we are ready to define our logical relation and the binary semantic typing judgement, after which we again prove semantic versions of the typing rules, where the above derived rules come into play.

5.3 Constructing the Binary Logical Relation

Much as in §3.4, we proceed by constructing a value interpretation $[\![-]\!]$ and expression interpretation $[\![-]\!]^{\mathsf{e}}$ on well-formed gλ-types, for which we reuse the notations of our unary logical relation. The binary logical relation closely resembles that given in Figure 3.4, as it is again defined by structural recursion on types $\tau \in \mathit{GType}$, and specified in terms of the operational semantics, while interpreting into the step-indexed logic introduced in §3.2, that is, propositions in *siProp*. However, while the unary logical relation in our proof of strong normalisation merely expresses when a value/expression semantically inhabits a type, the *binary* logical relation we construct now additionally expresses a notion of behavioural equivalence between the two terms it relates. The binary logical relation is, in principle, independent of the translation $(\!|-|\!)$. It is, however, constructed specifically such that it relates well-typed gλ-terms with the result of their translation.

5.3.1 Value and Expression Interpretation

The value interpretation $[\![-]\!]$ relates a gλ-value $\mathit{gv} \in \mathit{GVal}$ and a target language value $\mathit{uv} \in \mathit{UVal}$ at a gλ-type $\tau \in \mathit{GType}$, and likewise for the expression relation $[\![-]\!]^{\mathsf{e}}$ with some $\mathit{ge} \in \mathit{GExp}$ and $\mathit{ue} \in \mathit{UExp}$. Accordingly, $[\![-]\!]$ relates values that exhibit equivalent behaviour, while $[\![-]\!]^{\mathsf{e}}$ does likewise for expressions. For instance, for terms ge and ue related at type $\mathbf{N}$, we expect that there exists $n \in \mathbb{N}$ such that the terms reduce to the literal `lit` n, or `lit` n, respectively.

As before, the value and expression interpretation are both parameterised with a semantic environment δ, which maps type variables $\alpha \in \mathit{TVar}$ of gλ to their semantic interpretation. The type of the value interpretation is given by

$$[\![-]\!]_{(-)} : \mathit{GType} \to (\mathit{TVar} \xrightarrow{\mathsf{fin}} (\mathit{GVal} \times \mathit{UVal} \to \mathsf{siProp})) \to \mathit{GVal} \times \mathit{UVal} \to \mathsf{siProp},$$

where implicitly, the domain of δ matches the free type variables in the first argument of type *GType*. We note that the value interpretation is uncurried, matching the postcondition type for our binary WP connective, i.e. $\mathit{GVal} \times \mathit{UVal} \to \mathsf{siProp}$.

The value and expression interpretation are defined in Figure 5.2.

The expression interpretation is defined in terms of the value interpretation, once again following the unary logical relation. It is here that our binary total WP connective comes into use, allowing us to express that the terms ge and ue reduce to values which in turn satisfy the value interpretation for the same τ and δ:

$$[\![\tau]\!]^{\mathsf{e}}_{\delta} \triangleq \lambda\, (\mathit{ge}, \mathit{ue}).\ \mathsf{wp}\ \mathit{ge} \sim \mathit{ue}\ \left[[\![\tau]\!]_{\delta}\right].$$

Intuitively, the (closed) expressions ge and ue are related at type $\tau \in \mathit{GType}$ if they step to the values gv and uv, respectively, which in turn satisfy $[\![\tau]\!]_{\delta}$.

For the cases of the value interpretation, comprising the remainder of Figure 5.2, we keep the discussion brief, as the cases closely follow those of the unary value interpretation, which we treated in detail in §§3.4.1 to 3.4.5.

$$
\begin{aligned}
[\![\tau]\!]^{\mathrm{e}}_{\delta} &\triangleq \lambda\ (ge, ue).\ \mathsf{wp}\ ge \sim ue\ \left[[\![\tau]\!]_{\delta}\right] \\
[\![\alpha]\!]_{\delta} &\triangleq \delta(\alpha) \\
[\![\mathbf{N}]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ \exists\, n \in \mathbb{N}.\ gv = \mathtt{lit}\ n \wedge uv = \mathtt{lit}\ n \\
[\![\mathbf{1}]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ gv = \langle\rangle \wedge uv = \langle\rangle \\
[\![\tau_1 \times \tau_2]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ \exists\, ge_1, ge_2, uv_1, uv_2.\ (gv = \langle ge_1, ge_2\rangle) \wedge (uv = \langle uv_1, uv_2\rangle) \wedge \\
&\qquad [\![\tau_1]\!]^{\mathrm{e}}_{\delta}(ge_1, uv_1) \wedge [\![\tau_2]\!]^{\mathrm{e}}_{\delta}(ge_2, uv_2) \\
[\![\mathbf{0}]\!]_{\delta} &\triangleq \lambda\ (\_, \_).\ \mathsf{False} \\
[\![\tau_1 + \tau_2]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ \exists\, i \in \{1, 2\}, ge, uv'.\ (gv = \mathtt{inj}_i\ ge) \wedge (uv = \mathtt{inj}_i\ uv') \wedge \\
&\qquad [\![\tau_i]\!]^{\mathrm{e}}_{\delta}(ge, uv') \\
[\![\tau_1 \to \tau_2]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ \left(\forall ge, ue.\ [\![\tau_1]\!]^{\mathrm{e}}_{\delta}(ge, ue) \Rrightarrow [\![\tau_2]\!]^{\mathrm{e}}_{\delta}(gv\ ge,\ uv\ ue)\right) \\
[\![\mu\alpha.\tau]\!]_{\delta} &\triangleq \mu\ (\Psi : GVal \times UVal \to \mathsf{siProp}).\ \lambda\ (gv, uv).\ \exists\, ge.\ (gv = \mathtt{fold}\ ge) \wedge \\
&\qquad [\![\tau]\!]^{\mathrm{e}}_{\delta, \alpha \mapsto \Psi}(ge, uv) \\
[\![\blacktriangleright \tau]\!]_{\delta} &\triangleq \lambda\ (gv, uv).\ \exists\, ge, (gv = \mathtt{next}\ ge) \wedge \triangleright [\![\tau]\!]^{\mathrm{e}}_{\delta}(ge, uv\ \langle\rangle)
\end{aligned}
$$

Figure 5.2: Binary logical relation on gλ-terms and target language terms

- **Base types.** We expect values that semantically inhabit **1** to be exactly the unit value in the respective language ($\langle\rangle$ or $\langle\rangle$), while for **N**, we expect the values to be literals of the form $\mathtt{lit}\ n$ and $\mathtt{lit}\ n$, where—importantly—the $n \in \mathbb{N}$ must match between both literals. The 'void' type **0** should again be uninhabited, as is expressed by the constant False predicate.

- **Product and sum types.** The treatment for the gλ-value $gv \in GVal$ is much the same as for the unary logical relation. The main point of interest is that pairs and injections are values in the target language only if their subterms are themselves values, while $\langle ge_1, ge_2\rangle$ and $\mathtt{inj}_i\ ge$ are in $GVal$ for arbitrary $ge_1, ge_2, ge \in GExp$.

  In the case for $\tau_1 \times \tau_2$, we thus expect uv to be a value pair $\langle uv_1, uv_2\rangle$, and likewise with $uv = \mathtt{inj}_i\ uv'$. Of the two terms we pass to $[\![\tau_1]\!]^{\mathrm{e}}_{\delta}$ and (resp. or) $[\![\tau_2]\!]^{\mathrm{e}}_{\delta}$, the first is thus in $GExp$, while the latter is always a value from $UVal$.

- **Function types.** We treat function types exactly as before, requiring that values inhabiting $[\![\tau_1 \to \tau_2]\!]_{\delta}$, when applied to terms in the interpretation of the argument type $[\![\tau_1]\!]^{\mathrm{e}}_{\delta}$, reduces to a value pair that is semantically of the output type τ_2.

- **Recursive types.** The operations $\mathtt{fold}$ and $\mathtt{unfold}$ are erased during compilation, hence values gv and uv are related at a recursive type $\mu\alpha.\tau$ if gv is of the form $\mathtt{fold}\ ge$, where the expressions ge and uv are related at the recursive type, unfolded by one step. Here, we again use the *siProp*-level fixed-point constructor μ, which

constructs the fixed point of the function parametric in Ψ, provided that said function is contractive.

- **Guarded types.** Again, the treatment for the gλ-value is exactly as in the unary logical relation; in the target language, however, the corresponding construct for a value `next` ge is a *thunk* of the form $\lambda\_.ue$, and the subterm ge then corresponds to the result of forcing the thunk by applying it to $\langle\rangle$, hence we require the terms ge and $uv\ \langle\rangle$ to be related at type τ. As in §3.4, we interpret the object-level later type modality $\blacktriangleright$ to the $\triangleright$ of *siProp*, which guards the interpretation of τ at the logic level and thus admits the fixpoint construction in the interpretation of recursive types.

As in the unary logical relation, using the guarded fixpoint combinator μ with a function of type $(GVal \times UVal \to siProp) \to GVal \times UVal \to siProp$ demands that we show contractiveness of the function. The reasoning here is precisely the same as in §3.4.5, which we refer to for a more detailed treatment. Since we consider only well-formed gλ-types, recursive types $\mu\alpha.\tau$ satisfy guardedness in the object language, where occurrences of α in τ are always under a later type modality $\blacktriangleright$. Since the interpretation of types $\blacktriangleright\tau$ places the interpretation of τ under the *siProp* later modality $\triangleright$, we conclude that Ψ is guarded by $\triangleright$ in $[\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}$, as we only ever interpret α somewhere under a $\triangleright$.

5.3.2 Binary Semantic Typing Judgement

Following the standard procedure, the next step is to lift our expression interpretation $[\![-]\!]^{\mathrm{e}}$ to open terms, for which we use the notion of closing substitutions. We now define *USubst* alongside the set *GSubst* we introduced in §3.4.6:

$$\begin{aligned} \gamma &\in GSubst \triangleq Var \xrightarrow{\mathsf{fin}} GExp \\ \zeta &\in USubst \triangleq Var \xrightarrow{\mathsf{fin}} UVal \end{aligned}$$

Closing substitutions from *GSubst* (resp. *USubst*) map term variables x to closed terms $ge \in GExp$ (resp. closed values $uv \in UVal$). Note that because the target language features a call-by-value semantics, β-reduction substitutes term variables with *values*, rather than with expressions (as in gλ), which is reflected in our definition of *USubst*. However, the following definitions can largely gloss over this difference by injecting values $uv \in UVal$ into the superset of target language expressions *UExp*.

Context interpretation. We now define the (binary) interpretation of typing contexts $[\![\Gamma]\!]^{\mathrm{c}}_{\delta} : GSubst \times USubst \to siProp$, expressing that for all mappings $x : \tau$ in Γ, the specified substitutions $\gamma \in GSubst$ and $\zeta \in USubst$ map term variables x to expressions—respectively values—that semantically inhabit τ.

$$[\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma,\zeta) \triangleq \mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma) = \mathrm{dom}(\zeta) \wedge \forall\, x.\ \big(\Gamma(x) = \tau \Rightarrow [\![\tau]\!]^{\mathrm{e}}(\gamma(x),\zeta(x))\big)$$

Despite the fact that ζ maps to values rather than terms, we use the binary expression interpretation $[\![-]\!]^{\mathrm{e}}$, and inject values $uv \in UVal$ into *UExp*. As for the unary interpretation of contexts, we have that $[\![-]\!]^{\mathrm{c}}$ is trivially satisfied by the empty context and empty substitutions, and can be decomposed entry-by-entry.

Semantic typing judgement. Having specified the binary context interpretation, we now have all notions in place to define the binary semantic typing judgement.

Definition 5.3.1 (Semantic Typing Judgement)**.** For terms ge and ue, typing context Γ and type τ, we define the binary semantic typing judgement by:

$$\Gamma \vDash ge \approx ue : \tau \triangleq \forall\, \delta, \gamma, \zeta.\ [\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma, \zeta) \Rrightarrow [\![\tau]\!]^{\mathrm{e}}_{\delta}(\gamma(ge), \zeta(ue))$$

As suggested by the notation, the judgement $\Gamma \vDash ge \approx ue : \tau$ states that ge and ue both semantically inhabit τ under the context Γ in any semantic environment δ, while simultaneously stating a behavioural equivalence between ge and ue. For $\Gamma \vDash ge \approx ue : \tau$ to hold, the terms ge and ue satisfy the expression interpretation $[\![\tau]\!]^{\mathrm{e}}_{\delta}$ under any closing substitutions γ and ζ satisfying $[\![\Gamma]\!]^{\mathrm{c}}_{\delta}$.

5.4 Correctness Result

We now go on to show our desired correctness result—as stated in §5.1—for the program translation defined in Figure 4.1 by utilising the binary logical relation and semantic typing judgement defined in the preceding sections.

Once again, we proceed in two steps. In §5.4.1, we show *adequacy* of the semantic typing judgement $\Gamma \vDash ge \approx ue : \tau$, i.e. that it implies our correctness property. We then prove the semantic typing rules in §5.4.2, from which we deduce the *Fundamental Property* and the correctness result for our translation of gλ-terms in §5.4.3.

5.4.1 Adequacy

For the semantic typing judgement $\Gamma \vDash ge \approx ue : \tau$, adequacy follows by construction of the logical relation, where the case for $\mathbf{N}$ states that ge and ue must evaluate to literals $\texttt{lit}\ n$ and $\texttt{lit}\ n$, respectively, where the value of n matches between the two languages. In the adequacy proof, we also use the fact that both languages—or more specifically, their reduction relations—are deterministic, by which we conclude that ge uniquely reduces to $\texttt{lit}\ n$, and likewise for ue and $\texttt{lit}\ n$.

We note that the correctness property we consider for the compilation is specific to the $\mathbf{N}$ type: while properties such as termination or safety typically follow immediately from the WP connective used in the expression interpretation, our correctness property does not follow from $\mathrm{wp}\ ge \sim ue\ [\Phi]$ (for some Φ), but is instead specific to $[\![\mathbf{N}]\!]$.

Theorem 5.4.1 (Adequacy of Semantic Typing for $\mathbf{N}$)**.** *Given closed terms $ge \in GExp$ and $ue \in UExp$ such that $\emptyset \vDash ge \approx ue : \mathbf{N}$, then*

$$ge \mapsto^{*} \texttt{lit}\ n \iff ue \mapsto^{*} \texttt{lit}\ n.$$

Proof. By $\emptyset \vDash ge \approx ue : \mathbf{N}$, we have that $[\![\mathbf{N}]\!]^{\mathrm{e}}_{\delta}(ge, ue)$. Unfolding the definition of $[\![-]\!]^{\mathrm{e}}_{\delta}$ and the WP, we get $ge \mapsto^{*} \texttt{lit}\ m$ and $ue \mapsto^{*} \texttt{lit}\ m$, for some $m \in \mathbb{N}$. Using determinism of both step relations, each side of the bi-implication implies that the normal forms match by $n = m$, and the other side follows. Our logical relation is defined in *siProp*, hence we must lift the result to the meta-theory using Theorem 3.2.3. □

Much like the binary logical relation and semantic typing judgement, the adequacy statement is unrelated to the translation function $(\!|-|\!)$, and holds for any ge and ue related at type $\mathbf{N}$. The connection to the translation of §4 is made by the Fundamental Property, stating that the semantic typing judgement relates well-typed gλ-terms with the output of $(\!|-|\!)$ in the target language.

5.4.2 Semantic Typing Rules

With the adequacy proof in place, we proceed to the proof of the Fundamental Property. In order to show our desired result for the program translation $(\!|-|\!)$, the Fundamental Property must state that for a well-typed term $ge \in \mathit{GExp}$ with $\vdash e : \tau$, the term ge is related to the result of its translation $\vDash ge \approx (\!|ge|\!) : \tau$. Using Theorem 5.4.1, we can then conclude that our correctness property holds for all such pairs of well-typed gλ-terms and their translation.

As in §3.5, we proceed by proving semantic versions of the typing rules of gλ, sometimes referred to as *compatibility lemmas* [Pitts, 2005]. There, we now replace the syntactic typing judgement $\Gamma \vdash ge : \tau$ with the binary semantic typing judgement between ge and the translation of ge, that is, $\Gamma \vDash ge \approx (\!|ge|\!) : \tau$. In all rules, we actually prove a slightly stronger result than we need in the proof of the Fundamental Property: we take subterms in the target language to be arbitrary terms ue, rather than the translations of the source subterms $(\!|ge|\!)$. Doing so reduces notational clutter, and the proof of the more general rule is thus clearer than proving the specialised result.

Once the semantic typing rules are in place, the fundamental property can then be shown by induction on the (syntactic) typing judgement, where each case of the induction corresponds to one of the semantic typing rules.

Monadic rules for the expression interpretation. As in §3.5.2, we prove auxiliary rules for the expression interpretation $[\![-]\!]^{\mathrm{e}}$, corresponding to the types of the monadic unit (or return) and bind operations.

$$\frac{}{[\![\tau]\!]^{\mathrm{e}}_{\delta}(gv, uv)}\ [\![-]\!]^{\mathrm{e}}\text{-Val} \qquad \frac{[\![\tau]\!]^{\mathrm{e}}_{\delta}(ge, ue) \qquad \forall\, gv, uv.\ [\![\tau]\!]_{\delta}(gv, uv) \Rightarrow [\![\tau']\!]^{\mathrm{e}}_{\delta}(K_g[gv], K_u[uv])}{[\![\tau']\!]^{\mathrm{e}}_{\delta}(K_g[ge], K_u[ue])}\ [\![-]\!]^{\mathrm{e}}\text{-Bind}$$

The first rule, $[\![-]\!]^{\mathrm{e}}$-Val, is immediate from unfolding the definition of $[\![-]\!]^{\mathrm{e}}$ and applying TwpBin-Val, while $[\![-]\!]^{\mathrm{e}}$-Bind is derived by combining TwpBin-Bind and TwpBin-Impl.

For most of the semantic typing rules, the proof closely follows that given for the corresponding rule of the unary logical relation; we will therefore focus only on the rules that deviate from the unary version, and refer to our treatment in §3.5.2 for the remaining cases. As before, we ignore the context interpretation and closing substitutions in rules where the context Γ remains unchanged between the hypotheses and the conclusion.

Pairs and injections. Based on T-Pair and T-Inj, we must prove the following semantic typing rules:

$$\textsc{S-PairBin}\ \frac{\Gamma \vDash ge_1 \approx ue_1 : \tau_1 \qquad \Gamma \vDash ge_2 \approx ue_2 : \tau_2}{\Gamma \vDash \langle ge_1, ge_2 \rangle \approx \langle ue_1, ue_2 \rangle : \tau_1 \times \tau_2} \qquad \textsc{S-InjBin}\ \frac{\Gamma \vDash ge \approx ue : \tau_i \qquad i \in \{1,2\}}{\Gamma \vDash \mathtt{inj}_i\ ge \approx \mathtt{inj}_i\ ue : \tau_1 + \tau_2}$$

The reason we take a closer look at these two rules is that the value forms differ between the two languages: in the gλ-calculus, pairs and injections are always values, while they only constitute values in the target language when their subterms are also values. Here, the subterms of the target terms are arbitrary expressions, which prevents us from immediately applying $[\![-]\!]^{\mathrm{e}}$-Val, and instead, we must first use TwpBin-StepR (repeatedly) in order to step the target language subterms to values. Only once our goal is of the form $[\![\tau_1 \times \tau_2]\!]^{\mathrm{e}}_\delta(\langle ge_1, ge_2\rangle, \langle uv_1, uv_2\rangle)$—ignoring typing contexts and closing substitutions—or similarly $[\![\tau_1 + \tau_2]\!]^{\mathrm{e}}_\delta(\mathtt{inj}_i\ ge, \mathtt{inj}_i\ uv)$, can we indeed use rule $[\![-]\!]^{\mathrm{e}}$-Val to go from the expression interpretation to the value interpretation. By unfolding the definition of $[\![\tau_1 \times \tau_2]\!]$ (resp. $[\![\tau_1 + \tau_2]\!]$), it remains to show that $[\![\tau_1]\!]^{\mathrm{e}}_\delta(ge_1, uv_1)$ and (resp. or) $[\![\tau_2]\!]^{\mathrm{e}}_\delta(ge_2, uv_2)$, where we now use TwpBin-StepL to step only the gλ-expressions, after which we can again apply $[\![-]\!]^{\mathrm{e}}$-Val and then the (unfolded) hypotheses to complete the proof of each of the rules.

Functions. We encounter a similar situation in the semantic rule for T-Abs:

$$\textsc{S-AbsBin}\ \frac{\Gamma, x : \tau_1 \vDash ge \approx ue : \tau_2}{\Gamma \vDash \lambda x.ge \approx \lambda x.ue : \tau_1 \to \tau_2}$$

This time, we can immediately apply $[\![-]\!]^{\mathrm{e}}$-Val and unfold the value interpretation of the function type $[\![\tau_1 \to \tau_2]\!]$, by which we must show that $[\![\tau_2]\!]^{\mathrm{e}}_\delta((\lambda x.ge)\ ge')((\lambda x.ue)\ ue')$, for all ge' and ue' related at τ_1 by $[\![\tau_1]\!]^{\mathrm{e}}_\delta(ge', ue')$. While $(\lambda x.ge)\ ge'$ forms a reducible expression, the call-by-value target languages only permits β-reduction on values, requiring us to first reduce ue'. As such, we must use TwpBin-StepR with the assumption $[\![\tau_1]\!]^{\mathrm{e}}_\delta(ge', ue')$ to replace ue' with the result of its reduction uv. We then β-reduce both expressions using TwpBin-StepL and TwpBin-StepR, after which it remains to show that

$$[\![\tau_2]\!]^{\mathrm{e}}_\delta(ge[ge'/x], ue[uv/x]).$$

As noted previously, we take Γ and the closing substitutions to be empty for the sake of presentation. The hypothesis of S-AbsBin therefore unfolds to

$$\forall\ ge'', ue''.\ [\![\tau_1]\!]^{\mathrm{e}}(ge'', ue'') \Rightarrow [\![\tau_2]\!]^{\mathrm{e}}(ge[ge''/x], ue[ue''/x]),$$

where we are done by instantiating ge'' to ge', and ue'' to uv.

Folds and unfolds. For the rules corresponding to T-Fold and T-Unfold, we must show binary versions of the helper lemmas that we used in §3.5.2, specifically Lemmas 3.5.3 to 3.5.5. The first lemma corresponds to Lemma 3.5.3, and is immediate by specialising μ-Unfold to the definition of $[\![\mu\alpha.\tau]\!]_\delta$.

Lemma 5.4.2. *For all $gv \in GVal$ and $uv \in UVal$, we have that*

$$[\![\mu\alpha.\tau]\!]_\delta(gv, uv) \dashv\vdash \exists\, ge.\ (gv = \texttt{fold}\ ge) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta, \alpha \mapsto [\![\mu\alpha.\tau]\!]_\delta}(ge, uv).$$

The substitution lemma for the binary logical relation is stated as follows.

Lemma 5.4.3. *For all $gv \in GVal$ and $uv \in UVal$, we have that*

$$[\![\tau]\!]_{\delta, \alpha \mapsto [\![\tau]\!]_\delta}(gv, uv) \dashv\vdash [\![\tau[\tau'/\alpha]]\!]_\delta(gv, uv)$$

By applying the above in Lemma 5.4.2, we then obtain the following biimplication.

Lemma 5.4.4. *The value interpretation of recursive types $\mu\alpha.\tau$ can be unrolled as follows:*

$$[\![\mu\alpha.\tau]\!]_\delta(gv, uv) \dashv\vdash \exists\, ge.\ (gv = \texttt{fold}\ ge) \wedge [\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(ge, uv)$$

The proofs of the semantic typing rules for T-Fold and T-Unfold consist largely of unrolling either the conclusion or hypothesis using this third lemma.

$$\text{S-FoldBin}\ \frac{\Gamma \vDash ge \approx ue : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vDash \texttt{fold}\ ge \approx ue : \mu\alpha.\tau} \qquad \text{S-UnfoldBin}\ \frac{\Gamma \vDash ge \approx ue : \mu\alpha.\tau}{\Gamma \vDash \texttt{unfold}\ ge \approx ue : \tau[\mu\alpha.\tau/\alpha]}$$

In the proof of S-FoldBin, $\texttt{fold}\ ge$ is a value, but the occurrence of $\texttt{fold}$ is erased in the translated target term, meaning that we must first step ue to a value uv. We do so by unfolding the definition of $[\![-]\!]^{\mathrm{e}}$ and the total WP in the hypothesis, which gives us that $ue \mapsto^* uv$ for some uv. We can then apply $[\![-]\!]^{\mathrm{e}}$-Val and unfold the value interpretation of $\mu\alpha.\tau$ according to Lemma 5.4.4, after which we conclude by the hypothesis.

For S-UnfoldBin, we first use $[\![-]\!]^{\mathrm{e}}$-Bind, by which the conclusion becomes

$$\Gamma \vDash \texttt{unfold}\ gv \approx uv : \tau[\mu\alpha.\tau/\alpha],$$

and we gain the assumption $[\![\mu\alpha.\tau]\!]_\delta(gv, uv)$. We then use Lemma 5.4.4 in the hypothesis, by which there exists some ge' such that $gv = \texttt{fold}\ ge'$, and $[\![\tau[\mu\alpha.\tau/\alpha]]\!]^{\mathrm{e}}_\delta(ge', uv)$. The latter lets us conclude the proof by reducing $\texttt{unfold}\ (\texttt{fold}\ ge')$ to ge' with TwpBin-StepL.

Later operations. The most noteworthy cases are those for T-Next and T-Star, since it is here that the 'later' constructs of gλ are translated to λ-abstractions and applications in the target language. Let us first look at the semantic typing rule for T-Next:

$$\text{S-NextBin}\ \frac{\Gamma \vDash ge \approx ue : \tau}{\Gamma \vDash \texttt{next}\ ge \approx delay(ue) : \blacktriangleright\tau}$$

Recall that $delay(ue)$ is defined as $\lambda x.ue$, where x is some variable not free in ue. As such, we can immediately apply $[\![-]\!]^{\mathrm{e}}$-Val and unfold the value interpretation of $\blacktriangleright\tau$, by which the goal becomes

$$\exists\, ge'.\ \texttt{next}\ ge = \texttt{next}\ ge' \wedge \triangleright [\![\tau]\!]^{\mathrm{e}}_\delta(ge', delay(ue)\ \langle\rangle)$$

By instantiating ge' to ge, the left side of the conjunction holds trivially, while on the right, we can β-reduce the application $delay(ue)\ \langle\rangle$. By the definition of $delay$, the argument $\langle\rangle$ is ignored, since it is substituted for a variable of which we explicitly require that it does not appear freely in the body ue.

However, this fact is less obvious when we consider the context Γ and closing substitutions γ and ζ with $[\![\Gamma]\!]^{\mathrm{c}}_{\delta}(\gamma, \zeta)$, where we must show that $\zeta(delay(ue))\ \langle\rangle$ reduces to $\zeta(ue)$. The term $\langle\rangle$ is of course equal to $\zeta(\langle\rangle)$, so we can move the substitution outwards by $\zeta(delay(ue))\ \langle\rangle = \zeta(delay(ue))\ \zeta(\langle\rangle) = \zeta(delay(ue)\ \langle\rangle)$. Applying Lemma 4.1.3 (substitutivity) repeatedly, $delay(ue)\ \langle\rangle \mapsto^* ue$ then implies $\zeta(delay(ue)\ \langle\rangle) \mapsto^* \zeta(ue)$.

Finally, it remains to show that $\triangleright[\![\tau]\!]^{\mathrm{e}}_{\delta}(ge, ue)$, which—after stripping away the $\triangleright$ with $\triangleright$-Intro—precisely matches the hypothesis.

Now for the rule corresponding to T-Star:

$$\textsc{S-StarBin}\quad \frac{\Gamma \vDash ge_1 \approx ue_1 : \blacktriangleright(\tau_1 \to \tau_2) \qquad \Gamma \vDash ge_2 \approx ue_2 : \blacktriangleright\tau_1}{\Gamma \vDash (ge_1 \circledast ge_2) \approx delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle)) : \blacktriangleright\tau_2}$$

While the right (target) term is a thunk, and thus already a value, we must first reduce the subterms ge_1 and ge_2 of the source expression. Since we cannot reduce under the λ-abstraction in $delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle))$, we must use TwpBin-Bind rather than $[\![-]\!]^{\mathrm{e}}$-Bind to reduce the source subterms ge_1 and ge_2. We apply TwpBin-Bind, where we take the source context to be $(\cdot \circledast ge_2)$ and the target context to be empty, by which we must show that

$$\mathrm{wp}\ ge_1 \sim delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle))\ \left[(gv, uv).\ [\![\blacktriangleright\tau_2]\!]^{\mathrm{e}}_{\delta}(gv \circledast ge_2, uv)\right].$$

Using TwpBin-StepL, we can step ge_1 to some value gv_1 using the left hypothesis, and then use TwpBin-Val to resolve the outer WP. We repeat the same procedure for ge_2 using the context $(gv_1 \circledast \cdot)$, resulting in the following:

$$[\![\blacktriangleright\tau_2]\!]^{\mathrm{e}}_{\delta}(gv_1 \circledast gv_2, delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle)))$$

Unfolding the interpretation of $\blacktriangleright(\tau_1 \to \tau_2)$ and $\blacktriangleright\tau_1$ in the assumptions, we get that there exist source terms ge_1', ge_2' satisfying $gv_1 = \texttt{next}\ ge_1'$ and $gv_2 = \texttt{next}\ ge_2'$, and which relate to the target subterms by $\triangleright[\![\tau_1 \to \tau_2]\!]^{\mathrm{e}}_{\delta}(ge_1', ue_1\ \langle\rangle)$ and $\triangleright[\![\tau_1]\!]^{\mathrm{e}}_{\delta}(ge_2', ue_2\ \langle\rangle)$. As such, we can rewrite the source expression to $(\texttt{next}\ ge_1') \circledast (\texttt{next}\ ge_2')$, which we reduce with TwpBin-StepL to $\texttt{next}\ (ge_1'\ ge_2')$. Now, we can finally apply $[\![-]\!]^{\mathrm{e}}$-Val and then unfold the definition of $[\![\blacktriangleright\tau_2]\!]_{\delta}$ to obtain the goal

$$\triangleright[\![\tau_2]\!]^{\mathrm{e}}_{\delta}(ge_1'\ ge_2', delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle))\ \langle\rangle)$$

Since both hypotheses and the goal are all under a later ($\triangleright$), we can strip away all three $\triangleright$'s using $\triangleright$-And in combination with $\triangleright$-Mono. As in the proof for S-NextBin, the term $delay((ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle))\ \langle\rangle$ can be β-reduced to $(ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle)$ using TwpBin-StepR. Our goal is now $[\![\tau_2]\!]^{\mathrm{e}}_{\delta}(ge_1'\ ge_2', (ue_1\ \langle\rangle)\ (ue_2\ \langle\rangle))$, which follows from the left hypothesis $[\![\tau_1 \to \tau_2]\!]^{\mathrm{e}}_{\delta}(ge_1', ue_1\ \langle\rangle)$ and the right hypothesis $[\![\tau_1]\!]^{\mathrm{e}}_{\delta}(ge_2', ue_2\ \langle\rangle)$ using the same steps as the proof of S-App.

The proof of S-StarBin concludes the semantic typing rules for the binary semantic typing judgement $\Gamma \vDash ge \approx ue : \tau$, which allow us to show the fundamental property, and then conclude with our correctness proof for the translation $(\!|-|\!)$.

5.4.3 Correctness of $(\!|-|\!)$

While the previous section presents the semantic typing rules for arbitrary subterms *ue* in the target language, the Fundamental Property is specific to semantic typing judgements of the form $\Gamma \vDash ge \approx (\!|ge|\!) : \tau$. The exact statement is the following.

Lemma 5.4.5 (Fundamental Property)**.** *A syntactically well-typed* gλ*-term* $\Gamma \vdash ge : \tau$ *is also semantically well-typed in relation with* $(\!|ge|\!)$*, that is:*

$$\Gamma \vdash ge : \tau \quad \Rightarrow \quad \Gamma \vDash ge \approx (\!|ge|\!) : \tau.$$

Proof. As for Lemma 3.5.6, we proceed by induction on the typing derivation $\Gamma \vdash e : \tau$, where for each typing rule, we may inductively assume that the semantic versions of the hypotheses hold, and must show the semantic version of the conclusion. Each inductive case thus matches one of the semantic typing rules of the previous section. □

In combination with the adequacy proof of Theorem 5.4.1, the above allows us to prove our desired correctness result for the translation of gλ-programs by $(\!|-|\!)$.

Theorem 5.4.6 (Correctness of $(\!|-|\!)$)**.** *Let* $ge \in GExp$*. If* ge *is closed and well-typed by* $\emptyset \vdash ge : \mathbf{N}$*, then we have that*

$$ge \mapsto^* \texttt{lit}\ n \quad \Longleftrightarrow \quad (\!|ge|\!) \mapsto^* \texttt{lit}\ n.$$

Proof. By Lemma 5.4.5, syntactic well-typedness $\emptyset \vdash ge : \mathbf{N}$ implies that $\emptyset \vDash ge \approx (\!|ge|\!) : \mathbf{N}$. We conclude that $ge \mapsto^* \texttt{lit}\ n \Longleftrightarrow (\!|ge|\!) \mapsto^* \texttt{lit}\ n$ by Theorem 5.4.1. □

Chapter 6

Rocq Formalisation

Our results of §3 and §5 have been mechanised in the Rocq prover using the Iris framework, in particular the strong normalisation result for our source language (Theorem 3.5.7) and our correctness result regarding the program translation $(\!|-|\!)$ (Theorem 5.4.6); the sources can be found in the artifact accompanying this thesis [Läwen, 2025].

The Rocq development comprises around 3000 lines of code, a detailed breakdown of which is given below.

Component	**LOC count**
Language definition for gλ	400
Type system of gλ	340
Unary total WP and derived rules	150
Unary logical relation	240
SN result	240
Target language definition	560
Translation	60
Binary total WP and derived rules	150
Binary logical relation	260
Correctness result	360
Auxiliary proofs and tactics	200

Our formalisation makes use of the Iris framework in defining our source and target language, and for the definition of our step-indexed base logic, which supports interactive proofs using the *Iris Proof Mode* (IPM) [Krebbers et al., 2017, 2018]. To define term substitution in our source and target, we use the Autosubst library [Schäfer et al., 2015], representing term and type variables using de Bruijn indices [de Bruijn, 1972].

Language definitions. The Iris framework is not only generic in the logic used, but also in the object language(s) one reasons about. We instantiate the general-purpose language structures in the `program_logic` module with the base step relation and evaluation contexts for our languages, which then allows us to derive the reduction relations closed under contexts and their reflexive-transitive closures.

Weakest precondition. In §3.3, we define a custom total weakest precondition (Definition 3.3.1), which we use in constructing our unary logical relation.

The Iris framework also provides a definition for the total WP via a least fixpoint construction, as discussed at length by Krebbers et al. [2025, §4]. The total WP of Iris accounts for non-determinism, forked-off threads, and state. However, none of the above are applicable to our source language, and we found it beneficial to work instead with a simpler custom definition in terms of the multistep reduction relation $\mapsto^*$. This choice is made in part to match the definition we use in the correctness proof of §5, which we motivate below, though we also found the custom definition to be more ergonomic than the built-in total WP for our purposes. It simplifies the adequacy proof at little added cost, and the additional capabilities of Iris's `twp` tend to pose more of a hindrance than an advantage for a deterministic, pure, and single-threaded language like gλ.

Regarding our binary WP (§5.2.1), our proofs of the semantic typing rules frequently require stepping the terms of the two languages either simultaneously, or in alternation. To use Iris's total WP, we would need to nest the unary WPs of the two languages, e.g. by wp ge $\left[gv.\ \mathsf{wp}\ ue\ [uv.\ \Phi(gv, uv)]\right]$. However, we can then only step the term ge of the outer WP; to step the inner term ue we would need an additional proof rule stating that we can commute the two WPs, and switch between the two nestings throughout our proofs. Such a commuting rule is also non-trivial, given the least fixpoint construction used to define Iris's `twp`. Consequently, we found a custom definition via the multistep reduction relations of the two languages to be more workable for our purposes.

Mutual definition of the logical relation. The guarded fixpoint construction with which we interpret recursive types $\mu\alpha.\tau$ relies on a guardedness result concerning the interpretation itself. Accordingly, $[\![-]\!]_\delta$ must be defined mutually with the proof of said result, as pointed out in §3.4.5. We proceed analogously for the binary logical relation.

There are two noteworthy discrepancies between our presentation in §3.4.5 and the formalisation. Firstly, guarded fixpoints in the mechanised setting require contractiveness (in the (C)OFE model), rather than a syntactic condition, though contractiveness can usually be derived using Iris's tactic automations when syntactic guardedness is satisfied. Secondly, we use a workaround that lets us first define $[\![-]\!]_\delta$, and then prove the contractiveness result and show equivalence with the desired definition.

Rather than defining $[\![\mu\alpha.\tau]\!]_\delta$ as in §3.4, we take the guarded fixpoint of the function

$$\mathsf{rec\_pre} \triangleq \lambda\,\Psi.\,\lambda\,v.\,\exists\,\Psi'.\,\triangleright(\Psi = \Psi') \wedge (\exists\,e.\,(v = \mathtt{fold}\ e) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi'}).$$

The semantic environment δ is extended not with Ψ, but with Ψ', which is equal to Ψ only under a later. Crucially, contractiveness of rec_pre does not rely on $[\![\tau]\!]_\delta$, which we prove in the semantic model *SIProp*[1]: for all Ψ_1, Ψ_2, we must show that $[\![\mathsf{rec\_pre}\ \Psi_1]\!] \overset{n}{=} [\![\mathsf{rec\_pre}\ \Psi_2]\!]$ follows from $\forall m < n.\ [\![\Psi_1]\!] \overset{m}{=} [\![\Psi_2]\!]$. Since the parameter Ψ appears only on the left of the conjunction, it suffices to show $[\![\triangleright(\Psi_1 = \Psi')]\!] \overset{n}{=} [\![\triangleright(\Psi_2 = \Psi')]\!]$. The latter follows—since $\triangleright$ is contractive—from $\forall m < n.\ [\![\Psi_1 = \Psi']\!] \overset{m}{=} [\![\Psi_2 = \Psi']\!]$, which holds by our assumption.

[1] We break the step-indexing abstraction only in this proof; elsewhere, we work exclusively with $\triangleright$.

After defining the logical relation using rec_pre, we can show equality with our definition of $[\![\mu\alpha.\tau]\!]_\delta$ as in §3.4, that is, without the $\triangleright(\Psi = \Psi')$ wrinkle:

$$(\mu\ \Psi.\ \mathsf{rec\_pre}\ \Psi)\ =\ (\mu\ \Psi.\ \lambda\ v.\ \exists\ e.\ (v = \mathtt{fold}\ e) \wedge [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}(e)). \qquad \text{(Rec-Eq)}$$

To that end, we first show that $\mathsf{guarded}_\alpha\ \tau$ implies that $\lambda\ \Psi.\ [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi}$ is contractive. We proceed by induction on the guardedness judgement (Figure 2.1). There are two interesting cases: type variables β are interpreted independently of Ψ, since G-TVar requires $\beta \neq \alpha$, while the interpretation of $\blacktriangleright\tau$ is trivially contractive; the remaining cases follow from the inductive hypotheses.

Using this contractiveness result, we have $\triangleright(\Psi = \Psi') \vdash ([\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi} = [\![\tau]\!]^{\mathrm{e}}_{\delta,\alpha\mapsto\Psi'})$ when $\mathsf{guarded}_\alpha\ \tau$, allowing us to prove Rec-Eq.

Interactive proofs with step-indexing. We use the *siProp* logic defined in §3.2 as the base logic for both our unary logical relation in §3.4, and the binary logical relation described in §5.3. However, we do not define this logic in Rocq ourselves, and instead use a definition provided in the Iris framework. Aside from helpful notations and lemmas, this definition implements the BI (bunched implications) interface of the framework, allowing proofs in *siProp* to be performed using the Iris Proof Mode (IPM).

In short, the IPM provides custom tactics that allow the user to perform interactive proofs in the embedded logic of a BI instance much as if they were working in `Prop`, that is, in Rocq's native logic. Naively, one could unfold the entailment relation of the object logic (*siProp*, in our case) and work with Rocq's tactics, but for *siProp*, this would mean dealing with explicit step-indices again. Even with proofs of introduction and elimination rules for the connectives of the logic, the issue of *context management* remains, as there is no way to refer to the object logic's hypotheses and manipulate parts of the proof context in isolation.

The IPM remedies the latter point by visualising the context and goal of the object logic with named hypotheses, and by providing general purpose tactics for interacting with said context and operating on the proof goal. These tactics mirror familiar tactics of Rocq, such as `intros`, `destruct`, or `exists`, and support introduction and elimination for all connectives of the object logic, with expressive introduction patterns that also closely resemble those of native Rocq.

We use the IPM whenever we perform proofs in *siProp*, most notably in deriving the proof rules of our program logics (§3.3 and §5.2.2), and in the proofs of the semantic typing rules (§3.5.2 and §5.4.2).

Chapter 7

Related Work

The mechanised proofs that we present in §3 and §5, respectively, connect to a wide range of work on semantic typing, logical relations proofs, and the use of (step-indexed) logical relations in Iris in particular.

Our strong normalisation result of §3 (in Theorem 3.5.7) has previously been proven on paper by Clouston et al. [2016], while Abel and Vezzosi [2014] formalised a normalisation proof for a similar system in Agda. In §7.1, we compare these results to our mechanised proof regarding the proof techniques, and possible advantages of our approach using *siProp* over e.g. the explicit step-indexing that appears in Clouston et al.'s proof.

In §7.2, we regard our formalisation in the broader context of mechanised logical relations proofs, most notably with respect to the 'logical approach' outlined by Timany et al. [2024], as well as the work of Giarrusso et al. [2020] on the *guarded DOT calculus*, which resembles gλ in that it is pure, and features a variant of the 'later' type former.

Our program translation to untyped λ-calculus (in §4) and associated correctness proof (in §5) raise the question of how our results relate to other work on verified program translations and compilation, which we aim to answer in §7.3. Finally, §7.4 briefly discusses how we can recover functions that are productive, but *acausal*, using the 'constant' type former ■ or, alternatively, with so-called *clock quantifiers*.

7.1 Comparison of SN Proofs

The strong normalisation property we show in §3 is an existing result in the literature, with a proof for the gλ-calculus given by Clouston et al. [2016], while Abel and Vezzosi [2014] prove strong normalisation for a similar system that allows reduction under the `next` type former, but only up to a finite depth. The latter proof is mechanised in Agda.

Our approach differs from both existing proofs in that our logical relation is defined using the *operational* semantics of the language, rather than a *denotational* semantics. In addition, our logical relation is defined by structural recursion on the syntax of types, where both existing proofs use a lexicographic ordering that combines indexing with syntactic measures to achieve a well-founded definition. We compare our source language of §2 with the calculi of Clouston et al. and Abel and Vezzosi below, as well as comparing their proof methods with our own SN proof.

Comparison with the gλ-calculus. Clouston et al. [2016] define a denotational semantics for the gλ-calculus, interpreting its types in the *topos of trees*. They show the denotational semantics to be *adequate* using a logical relations argument, where termination falls out as a corollary. In addition, Clouston et al. show how the internal logic of the topos of trees gives rise to a program logic for reasoning about contextual equivalence of gλ-terms.

The topos of trees [Birkedal et al., 2012], denoted by $\mathcal{S}$, has as its objects families of sets $X_0, X_1, X_2, \ldots$ indexed by non-negative integers, which are equipped with similarly indexed restriction functions $r_i^X : X_{i+1} \to X_i$. Types are interpreted to $\mathcal{S}$-objects, where the sets X_i give increasingly refined approximations of the type's inhabitants. Morphisms $f : X \to Y$ are given by indexed families of functions $f_i : X_i \to Y_i$ which must obey the naturality condition of commuting with the restriction maps, that is, $f_i \circ r_i^X = r_i^Y \circ f_{i+1}$. The category $\mathcal{S}$ is Cartesian-closed, with products and coproducts defined pointwise.

Where our SN proof uses a unary logical relation, taking types to predicates on values, the adequacy proof of Clouston et al. relates denotations in $\mathcal{S}$ with gλ-terms. More specifically, the proof uses a family of relations R_i^τ, indexed by both non-negative integers i and closed types τ, where R_i^τ relates elements of the semantics $[\![\tau]\!]_i$ (in $\mathcal{S}$) with closed terms of type τ. Unlike our value interpretation, the definition of R_i^τ is not structurally recursive on τ. The reason lies in the case for $R_i^{\mu\alpha.\tau}$, which refers to $R_i^{\tau[\mu\alpha.\tau/\alpha]}$ in its premise. As a consequence, the definition of R_i^τ requires a somewhat intricate lexicographic ordering, combining the indices i with two syntactic measures on types, namely their *box depth* and *unguarded size*, to ensure well-foundedness.

The fundamental lemma establishes a result about open terms, intuitively stating that closing off a given term by substituting semantic elements on the left and related (by R_i^τ) syntactic elements on the right results in a related denotation and closed term. Specialised to the empty context, this yields the adequacy result, i.e. that closed terms $\vdash e : \tau$ are related to their own denotation in $[\![\tau]\!]_i$ by R_i^τ. Since all cases of R_i^τ demand that the term on the left reduce to a value, strong normalisation follows as a byproduct.

In our logical relation, we reason abstractly about step-indexing using the later modality $\triangleright$ of *siProp*. The topos of trees $\mathcal{S}$ gives rise to an internal logic also featuring a modality '$\triangleright$' that facilitates recursive predicates. However, Clouston et al.'s logical relation R_i^τ is defined using explicit indexing, rather than as a fixpoint using the later modality of the internal logic. The indexing consequently also crops up in e.g. the proof of the fundamental lemma, which requires reasoning about bounds and arithmetic on the indices in a number of the cases. As discussed by e.g. Dreyer et al. [2011], such explicit indexing and index arithmetic can obscure the main idea of a proof through tedious bookkeeping, and can be finicky and error-prone. The later modality [Appel et al., 2007] allows reasoning about the indexing at a higher level of abstraction, as in our proof.

Comparison with $\lambda^{\blacktriangleright}$. Abel and Vezzosi [2014] give a mechanised proof of strong normalisation for a system resembling our source language (§2), which they dub $\lambda^{\blacktriangleright}$. Like Nakano's [2000] original system and our language, $\lambda^{\blacktriangleright}$ lacks the `prev`, `box` and `unbox` term formers and $\blacksquare$ type modality of Clouston et al. [2016]. As such, $\lambda^{\blacktriangleright}$ is restricted to causal functions (like our system). Overall, Abel and Vezzosi's system corresponds to a

simply-typed version of the language considered by Birkedal and Møgelberg [2013].

As in our system, later introduction is explicit via a 'next' term former; however, $\lambda^{\blacktriangleright}$ allows reduction under any finite number of `next`s, whereas our `next` always blocks reduction of its subterm. There are other differences between $\lambda^{\blacktriangleright}$ and our source language, though these are more superficial:

- $\lambda^{\blacktriangleright}$ uses equi-recursive types, with a typing rule for converting between equal types, and a guardedness condition at the meta-level, rather than within the system;
- reduction is non-deterministic, as the reduction strategy is not fixed, and since terms are only reduced up to a certain depth, the system is not confluent either;
- the system permits reduction inside λ-abstractions and pairs.

While our strong normalisation proof interprets types into *siProp*, where we reason about the operational semantics of our language, Abel and Vezzosi instead follow the *saturated sets* approach pioneered by Tait [1967] in proving strong normalisation for STLC. Abel and Vezzosi's SN proof thus uses a *denotational* semantics, interpreting the types of $\lambda^{\blacktriangleright}$ into sets of strongly normalising terms. The saturated sets semantics crucially involves indexing the saturated sets by *depth*, a value $n \in \mathbb{N}$ expressing the number of `next`s under which evaluation can occur. This depth-n indexing differs from Tait's approach, and equips the Kripke model of types with a depth dimension, similar to the step-indexing we employ in our logical relation.

While we can use the fact that our terms have at most one reduction path to a value, the non-deterministic nature of $\lambda^{\blacktriangleright}$ means that all possible reductions must be considered. To characterise strongly normalising terms, Abel and Vezzosi modify a technique originally due to van Raamsdonk et al. [1999], which uses the least set of terms closed under introductions, formation of neutral terms, and *weak head expansion*.

In formalising $\lambda^{\blacktriangleright}$, Abel and Vezzosi take a vastly different approach from ours, modelling types directly as coinductive objects in Agda. More specifically, there is no μ type former in their representation of types, and the types of $\lambda^{\blacktriangleright}$ are described by infinite trees (i.e. infinite type expressions), defined as meta-level (Agda) fixed points. Guarded types are then those infinite trees that have an infinite number of $\blacktriangleright$'s on all infinite paths. This guardedness condition is enforced by piggybacking off of Agda's termination checker, in contrast to our explicit type formation judgement $\Delta \vdash \tau$ (Figure 2.1). To use the infinite type expression as equi-recursive types, they must be equal to their unfolding. Abel and Vezzosi achieve this by defining a bisimulation on types as a coinductive-inductive relation, and postulating that bisimilarity implies (intensional) equality, much like the standard functional extensionality axiom for function types.

Finally, the interpretation of $\lambda^{\blacktriangleright}$-types as saturated sets is again not structurally recursive, as the coinductive representation of types lacks a well-founded recursion principle outright. Like the interpretation of Clouston et al., a lexicographic measure is used, where the top-level induction proceeds on the finite depth n, which decreases at each occurrence of $\blacktriangleright$, while the steps in between decrease the number of the type formers $\to$ and $\times$ up until the following occurrence of $\blacktriangleright$.

Well-foundedness of the logical relation. The reason we can define our logical relation by structural recursion on types in the first place is the fixpoint construction in the case for $\mu\alpha.\tau$. There, the μ combinator of the logic lets us refer to the semantics Ψ of the recursive type, with which we extend the semantic environment δ (see §3.4.5 for details). We must refer to Ψ at a lower step-index though, as enforced by the μ connective, which demands contractiveness of its argument. Crucially, the step-indexing in our construction is hidden away by the later modality $\triangleright$, while it is explicit in the definitions of Clouston et al. and Abel and Vezzosi, and in the lexicographic orderings used to justify those definitions. Both orderings decrease the index when passing an occurrence of $\blacktriangleright$, and use syntactic measures on the type between those occurrences. In our setup, $\blacktriangleright$ is instead interpreted to $\triangleright$ in *siProp*, ensuring that the fixpoint construction is well-founded, while the definition of $[\![-]\!]_{(-)}$ is otherwise well-founded by structural recursion.

7.2 Step-Indexed Logical Relations

Our use of logical relations, following the 'logical approach' [Timany et al., 2024], is based on the work on step-indexed logical relations by Ahmed, Appel, and collaborators [Appel and McAllester, 2001; Ahmed et al., 2002; Ahmed, 2004], as well as the abstraction mechanism provided by the later modality $\triangleright$ [Appel et al., 2007; Dreyer et al., 2011].

However, our presentation is informed by our mechanisation using the Iris framework, e.g. in our use of weakest preconditions and the proof rules we derive for our program logics. As such, we instead compare our use of step-indexed logical relations specifically with other work that is formalised using Iris. Our first candidate is the example application of Timany et al., which we regard as a 'baseline' of sorts. Secondly, we compare to the work of Giarrusso et al. [2020], as the language they consider is pure and features a 'later' type former, while their logical relation also uses a total weakest precondition.

7.2.1 'Logical Approach'

To illustrate the 'logical approach', Timany et al. [2024] use logical relations to prove type soundness and express contextual refinements for their example language **MyLang**, covering typical language features one might encounter in logical relations proof, including polymorphism, existentials, recursive types, concurrency, and higher-order references. Below, we compare our work to this 'typical scenario' to highlight in more detail the distinguishing characteristics of our logical relation(s) touched on in §3.1.

Evaluation order. Where **MyLang** eagerly evaluates function arguments and reduces under all term formers except for λ-abstraction, the gλ-calculus is call-by-name, and does not evaluate under the term formers for pairs $\langle -, - \rangle$ and injections $\mathtt{inj}_i\ -$, as well as $\mathtt{fold}\ -$. As such, values inhabiting the interpretation of products $\tau_1 \times \tau_2$ and sums $\tau_1 + \tau_2$ contain expressions (rather than values) as their subterms, which we treat with the expression interpretation $[\![-]\!]^{\mathrm{e}}$, where Timany et al. can use the value interpretation $[\![-]\!]$; the situation is similar for function types $\tau_1 \to \tau_2$.

Pure vs impure. **MyLang** features a notion of state, with references that can be allocated, read from, and written to using primitive operations, along with compare-and-set and fetch-and-add primitives. A 'fork' primitive allows spawning new threads to execute a given expression. To handle the stateful nature of the language, the logical relation employs the separation logic capabilities of Iris's *iProp*, using $*$ and $-\!\!*$ as the default connectives for conjunction and implication. The value interpretation of reference types $\texttt{ref}\ \tau$ asserts that a value inhabits the type if it is a location ℓ that points to (via *iProp*'s points-to connective) a value inhabiting the type τ.

The type system of **MyLang** is *unrestricted* (or *intuitionistic*), where variables of all types can be used repeatedly, and types do not express ownership of resources. This fact is captured by the value interpretation returning *persistent* Iris propositions, denoted by $\mathit{iProp}_{\square}$. While persistence is inherent to the *siProp* logic that we employ, this is not the case for *iProp*, and the value interpretation for **MyLang** must enforce persistence for occurrences of the $\forall$ quantifier by wrapping them with the persistence modality $\square$, e.g. in the case for $[\![\tau_1 \to \tau_2]\!]$, which quantifies over all possible function arguments. In turn, Iris's *iProp* supports reasoning about substructural type systems via non-persistent propositions [Jung et al., 2018a; Dang et al., 2020; Hinrichsen et al., 2022].

Guarded recursive types. In contrast to the gλ-calculus, **MyLang**'s typing discipline is not guarded, and there is no 'later' type former. The iso-recursive types $\mu\alpha.\tau$ of **MyLang** differ from those of gλ only in the fact that they are not subject to a guardedness requirement. While gλ uses the well-formedness judgement $\Delta \vdash \tau$ (see Figure 2.1) to ensure that a μ-bound type variable α appears only below a later $\blacktriangleright$, there is no restriction on occurrences of μ-bound type variables in **MyLang**. As such, the value interpretation for recursive types $[\![\mu\alpha.\tau]\!]$ is not contractive in α, a fact which we crucially rely on.

In our logical relation, we 'carry over' guardedness from the object level to the logic level to facilitate our use of the (logic-level) μ combinator *without* wrapping the interpretation of τ with a $\triangleright$ in the interpretation of $\mu\alpha.\tau$. The latter would require later eliminations through program steps, which our total WP does not support. For **MyLang**, this is of no concern: the language allows unrestricted recursive types, including e.g. $\mu\alpha.\alpha$, hence termination is out of scope to begin with. The property of interest is instead safety (i.e. type soundness), for which a partial WP suffices. When proving the semantic typing rule corresponding to S-Unfold, the $\triangleright$ on the inner type does not pose an issue, as it can simply be eliminated when stepping $\texttt{unfold}\ (\texttt{fold}\ v)$ to v.

Other applications. Using semantic typing to prove properties besides type safety is by no means novel, and Timany et al. [2024, §8] indeed demonstrate the 'logical approach' in proving contextual refinements, and in showing *representation independence* of abstract data types. Proving termination is also a common use case: an instructive example is the normalisation proof for STLC presented by Pierce [2002, Chapter 12]; a much more involved application is described e.g. by Spies et al. [2021], who use logical relations for proving termination in a language with shared, mutable, higher-order state, using so-called *transfinite* step-indexing. The semantic approach and logical relations have also been applied to compiler correctness, discussion of which we defer to §7.3.

7.2.2 Guarded DOT

Giarrusso et al. [2020] present a formalised type soundness proof in Iris for an extension of Scala's core type system, the *Dependent Object Types* (*DOT*) calculus with a guarded typing discipline, referred to as *guarded DOT* (*gDOT*). The main similarities of the resulting system and the associated type soundness proof with the present work is the appearance of the later modality in the object language, as well as the use of semantic typing and the 'logical approach' in the type soundness proof. Much like the gλ-calculus, gDOT is pure and deterministic. In addition, (g)DOT features a notion of *paths*, which types can depend on; hence, paths must denote a unique value, and it is crucial that paths normalise. In the logical relation, this is ensured by using a total weakest precondition for paths, which strongly resembles our total WP for gλ-terms.

Later and recursive types. While the 'later' type former $\blacktriangleright$ of the gλ-calculus stratifies self-references when taking fixpoints of λ-abstractions, gDOT employs a similar type former $\triangleright$ to preclude the circular use of the 'self' variable for recursive objects $\nu(x : T).\{\overline{d}\}$. As in our semantic typing discipline, the logical relation for gDOT's type soundness proof reflects the object-level later type former into the later modality of the step-indexed logic.

The gDOT calculus also features recursive types of the form $\mu x.T$, though by the dependent nature of the gDOT calculus, x can also bind terms. As such, gDOT's recursive types differ from standard recursive types, and are more akin to a form of recursively dependent signatures [Crary et al., 1999], or self-types [Fu and Stump, 2014], which can be generalised and instantiated by the subject that they type. The recursive types of gDOT are closer to the equi-recursive approach, in the sense that they are self-similar at the meta level, and not via explicit (un)folding operations at the object level. As such, the value interpretation for $\mu x.T$ is simply given by the value interpretation of T in the semantic environment extended with a binding for x, in contrast to $[\![\mu\alpha.\tau]\!]$ in our logical relation, which involves the `fold` term former.

Termination and weakest precondition. The terms of gDOT are non-terminating, and are treated with a partial WP that allows later eliminations through program steps, and is defined as a guarded fixpoint, much like Iris's built-in partial WP. By contrast, paths use a total WP that does not permit later eliminations. While our total WP is defined using the reflexive-transitive closure of the step relation, Giarrusso et al. define their path WP using the one-step reduction relation and a least fixpoint construction, closely following the built-in total WP of Iris.

Base logic and stamps. As (guarded) DOT is pure, the formalisation requires only a subset of the features available in Iris. As in our formalisation, there is no separation logic involved, and ordinary conjunction $\wedge$ and implication $\Rightarrow$ are used everywhere in place of the separating conjunction $*$ and implication $-\!\!*$ typically seen in Iris developments.

Despite this, the formalisation does use the iProp type of Iris propositions, motivated by the use of Iris's *saved predicates* to indirectly represent semantic values. Values contain so-called *stamps*, which are mapped to semantic types using saved predicates.

Stamps allow for a first-order syntax of values, which aids mechanisation in Rocq. Saved predicates are an instance of Iris's higher-order ghost state [Jung et al., 2016], though the gDOT formalisation restricts the ghost state mechanism to saved predicates, thus ensuring that all propositions are persistent, and that the separation logic connectives $*$ and $-\!\!*$ coincide with ordinary conjunction and implication (as in Iris's *siProp*).

7.3 Verified Compilation

We refer to our contribution as a (verified) translation or compiler, suggesting similarity to other work on formally verified compilation. We compare our contributions to such efforts in the following.

Formal end-to-end verification of a realistic compiler was arguably first achieved with the CompCert project [Leroy, 2009a], which considers a feature-rich, optimising compiler for a subset of C, intended to be usable in the production of 'real-world' software, in particular for critical applications. In the last two decades, numerous projects based on CompCert have followed (e.g. [Sevcík et al., 2011; Appel, 2011]), as well as other work on verified compilers, including the CakeML project [Kumar et al., 2014; Tan et al., 2016], which compiles a subset of Standard ML, and its more recent offspring PureCake [Kanabar et al., 2023], for a pure, functional language resembling Haskell.

Our work differs from the aforementioned projects in several major points, which we detail in the following.

Feature-rich verified compilers. For one, projects such as CompCert and CakeML generally consider end-to-end compilation, going all the way from a source string to machine code. CompCert and related efforts focus more so on optimisation passes concerning e.g. instruction selection, register allocation, and data flow analyses, to name just a few examples. CakeML emphasises end-to-end correctness and a minimal trusted computing base, including e.g. verified lexing, parsing, type checking, as well as bootstrapping of the compiler to obtain a verified byte code implementation.

In contrast to such feature-rich verified compilation pipelines, our correctness result concerns a single-pass translation between deep embeddings of the source and target in Rocq, with no optimisations. The languages are both minimal calculi, rather than a language and instruction set that might be used in practice.

Proofs of semantic preservation. Generally speaking, the property of interest for a verified compiler is that of *semantic preservation*: the generated machine code should behave according to the semantics of the source program. Feature-rich, optimising compilers typically use a number of intermediate languages, and possibly also multiple passes within each of those languages. Semantic preservation for such a compiler is typically shown at each step, where transitivity of the semantic preservation property yields a result spanning from the source to the target.

Requiring that the source and target have precisely the same observable behaviours, referred to as *bisimulation* [Leroy, 2009b], is usually too strong a property to be usable, as

it cannot account for non-determinism of the source language, or compiler optimisations such as dead code elimination. In the case of CompCert [Leroy, 2009a], the standard approach for showing semantic preservation involves proving a *simulation diagram*, stating that state transitions in the original program correspond to (a sequence of) transitions in the transformed program of identical observable behaviour, and preserving a binary relation on the execution states. CakeML [Kumar et al., 2014] proves semantic preservation at each layer by induction on the respective big-step reduction relation, using binary relations defined in terms of the environment, and, for the step to byte code, also the semantic context, compiler state and machine state, among other information.

Cross-language logical relations. By contrast, we prove the correctness result for our translation using a binary logical relation on the source and target (via the types of the source). Such a logical relation between two different languages is referred to by Perconti and Ahmed [2014] as a *cross-language logical relation*.

In the context of compiler verification, logical relations have been employed for their ability to show *horizontal compositionality*, or *linking compatibility*, stating that the linking of two program components in the source is refined by linking the separately compiled components in the target. Key proponents of this approach include Benton and Hur [2009], who verify a compiler from STLC with recursion, and Hur and Dreyer [2011], who prove correctness for a compiler from an ML variant to assembly code. Linking is typically defined via language constructs, for which compositionality is captured by the *compatibility lemmas* [Pitts, 2005] used to show the Fundamental Property.

For whole-program semantic preservation, multi-pass compilers can be verified by composing the adequacy results, provided these are chosen to support transitive composition. Proving end-to-end linking compatibility, however, requires transitivity of the logical relations used for each pass.

When used to relate terms of the same language, transitivity of logical relations can be ensured using the technique of *biorthogonality* ($\top\top$-closure) [Dreyer et al., 2012; Pitts and Stark, 1998], or by restriction to well-typed terms, but neither technique scales to cross-language logical relations. More generally, transitive composition of cross-language logical relations has been an open problem for some time, and greatly limits the viability of logical relations for proving linking compatibility of multi-pass compilers.

Perconti and Ahmed [2014] demonstrate how to circumvent the lack of transitivity by embedding the source, target, and intermediate language of a two-pass compiler into a single combined language. Paraskevopoulou et al. [2021] take a different approach, where the end-to-end composed relation is *not* a logical relation; instead, a form of transitive closure is used that suffices for proving adequacy and linking compatibility.

Another technique for verifying multi-pass compilers of higher-order languages is that of *parametric inter-language simulations* (*PILS*), introduced by Neis et al. [2015]. PILSs build on the *parametric bisimulations* of Hur et al. [2012], combining properties of logical relations and bisimulations to support both horizontal and vertical compositionality; Neis et al. expand on this approach to verify a multi-pass optimising compiler. However, PILSs are significantly more involved than standard logical relations, and are defined coinductively, requiring coinductive proof techniques as a consequence.

7.4 Causality and Clock Quantifiers

Although our source language, as presented in §2, closely follows the guarded calculus (gλ) of Clouston et al. [2016], we omit the `prev`, `box` and `unbox` term formers, along with the 'constant' type modality ■ present in their system. These additions of Clouston et al. are motivated by the fact that the system without them is overly restrictive: the type system enforces not only productivity, but also *causality*, which is deemed undesirable (except in specific settings, such as reactive programming [Krishnaswami and Benton, 2011]). Consider, for example, the following function that returns every second element of a stream:

$$\mathsf{every2nd}\ (x :: x' :: xs) \triangleq x :: \mathsf{every2nd}\ xs$$

While every2nd is clearly productive, it does not satisfy *causality*, since elements of the output stream depend on deeper elements of the input stream.

In order to recover acausal functions and achieve merely a restriction to productive functions, Atkey and McBride [2013] propose the notion of *clock quantifiers*, which allow elimination of the later modality in a controlled way. In fact, with Atkey-McBride clock quantifiers it is possible to define types that are denotationally equivalent to standard coinductive types in set-based mathematics without sacrifice of productivity guarantees.

Rather than using clock quantifiers in the style of Atkey and McBride, the gλ-calculus instead employs the unary 'constant' type former, written ■, which corresponds to the use of a single clock. As such, the ■ type modality cannot express nested coinductive types varying on multiple independent time streams, which the clock quantifiers of Atkey and McBride do support. At the same time, the ■ modality is simpler than clock quantifiers, which require reasoning about clock contexts and variables, as well as handling freshness side conditions on said variables.

Chapter 8

Conclusion and Outlook

In this work, we considered an extension of the simply-typed λ-calculus with guarded recursive types, for which we (i) proved strong normalisation, and (ii) described a translation to untyped λ-calculus, along with a proof of semantic preservation. We obtain both results by means of logical relations, which we define in terms of the *operational* semantics of our languages in the step-indexed logic *siProp*. In doing so, we follow the general method used by most logical relations proofs in Iris, as characterised by the 'logical approach' of Timany et al. [2024].

Our correctness result regarding the program translation demonstrates that the *productivity* of functions—as enforced by the typing discipline of our guarded source language—is retained in the translated target programs. More generally, we can erase the guarded constructs of our source language without losing the strong guarantees that they provide. As such, we see potential towards 'practical guarded programming' by compiling a guarded language (such as the gλ-calculus) to a more familiar functional language resembling the untyped λ-calculus, for which one can utilise existing compilation infrastructure.

In the following, we discuss some possible improvements to the present work, and how it might be extended to a (verified) pipeline for running guarded programs.

Acausal functions. While our source language guarantees productivity, it is currently restricted to *causal* functions, thus excluding functions such as every2nd (§2.1), where elements of the output depend on deeper elements of the input. To remedy this limitation and facilitate acausal functions, we could follow Clouston et al. [2016] and amend our source language with the 'constant' type modality (denoted by ■), along with new term formers `prev`, `box`, and `unbox`, which would also allow productive, but non-causal functions to be typed.

Caching of thunks. Currently, the thunks we construct during compilation are re-computed in the target language each time they are forced. An obvious improvement would be to store the result of a thunk at the first evaluation, and thereafter read the stored result, rather than re-computing. We believe this to be a reasonably straightforward extension of our existing formalisation, that could be achieved by amending the target

language with state, and the operational semantics of the target with a notion of heap. We could then use Iris's separation logic capabilities in the logical relation, and to reason about the storing and reading of thunk information in the target language.

Evaluating a thunk only once and then sharing the result is only justified if the given thunk always yields the same result. However, thunks in our target language are generated by translating terms of the pure source language, hence they should never contain stateful operations that could lead to differing results.

A similar scenario is discussed by Kanabar et al. [2023, §5.2] in the context of the PureCake compiler, which uses sharing to efficiently implement call-by-need evaluation of the lazy PureLang source language. Their approach involves a flag that causes stateful operations to fail with a runtime error when forcing a thunk, while executing normally otherwise. It should be noted, however, that other aspects of laziness in PureCake, such as the demand analysis, are not applicable to our translation: we use thunks specifically to enforce the delayed evaluation under `next`, and not to mimic lazy evaluation in our call-by-value target language.

Running the target language. To make the translation of guarded programs usable in practice, we need a way to (efficiently) execute our programs after translation. Our current target language is essentially the untyped call-by-value λ-calculus, and is thus close to (or indeed a subset of) Malfunction [Dolan, 2016], a thin wrapper around the untyped intermediate language of the OCaml compiler. Malfunction features a simple syntax based on s-expressions, and is intended specifically as a compilation target for functional languages. The language is compiled using the tried-and-tested backend of OCaml. A translation to Malfunction would even be amenable to verification, as Forster et al. [2024] contribute a formal semantics for the language.

Another language similar to Malfunction is (untyped) CakeML [Kumar et al., 2014; Tan et al., 2016]; the main difference between the two is that CakeML represents constructors directly while Malfunction exposes their representation as blocks and integers, and CakeML offers native case analysis. In contrast to Malfunction, CakeML boasts a formally verified compiler, with a verified machine code implementation obtained through bootstrapping. By targeting CakeML and composing a correctness result for the translation with the verified CakeML compiler, one could obtain end-to-end correctness down to machine code, as in e.g. the PureCake project [Kanabar et al., 2023].

Inference of guarded type and term formers. Clouston et al. [2016] remark that programming in their gλ-calculus generally involves 'decorating' programs with the novel term and type formers, i.e. adding `nexts` and replacing applications with $\circledast$, and adding $\blacktriangleright$ at the appropriate positions in the types. Although this process is usually straightforward, the programmer is still burdened with elaborating programs in a fairly mechanical fashion. Ideally, this decoration process would be performed automatically.

Severi [2019] addresses this point by presenting a guarded calculus similar to the system of Nakano [2000] (though without subtyping) with a type inference algorithm that automatically handles later introductions and eliminations. The resulting system, denoted by $\lambda^{\bullet}_{\to}$, extends the lambda calculus à la Curry, i.e. lacks type annotations, and, more

importantly, the later operator, which Severi calls ‘delay’ and denotes by •, is silent in the syntax, in the sense that it is introduced and eliminated implicitly, in contrast to e.g. the use of our explicit introduction form `next`. Severi describes a type inference algorithm for $\lambda^{\bullet}_{\to}$ based on standard Hindley-Milner style unification, which is combined with integer linear programming to determine the number of •’s that need to be introduced at the nodes of the syntax tree.

The work of Severi shows that it is feasible to remove the burden of ‘decorating’ programs with the guarded constructs, which greatly increases the viability and practicality of guarded programming.

Bibliography

Andreas Abel and Andrea Vezzosi. 2014. A Formalized Proof of Strong Normalization for Guarded Recursive Types. In *APLAS (LNCS, Vol. 8858)*. 140–158. https://doi.org/10.1007/978-3-319-12736-1_8

Amal J. Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University. https://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References Embeddable in Higher-Order Logic. In *LICS*. 75. https://doi.org/10.1109/LICS.2002.1029818

Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *ESOP 2011 (LNCS, Vol. 6602)*. 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

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

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *POPL 2007*. 109–122. https://doi.org/10.1145/1190216.1190235

Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *ACM SIGPLAN International Conference on Functional Programming*. 197–208. https://doi.org/10.1145/2500365.2500597

Hendrik Pieter Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. Elsevier Science Pub. Co.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, Step-Indexing and Compiler Correctness. In *ICFP 2009*. 97–108. https://doi.org/10.1145/1596550.1596567

Lars Birkedal and Robert Harper. 1999. Relational Interpretations of Recursive Types in an Operational Setting. *Inf. Comput.* 155, 1-2 (1999), 3–63. https://doi.org/10.1006/INCO.1999.2828

Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *LICS 2013*. 213–222. https://doi.org/10.1109/LICS.2013.27

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *Log. Methods Comput. Sci.* 8, 4 (2012), 1–45. https://doi.org/10.2168/LMCS-8(4:1)2012

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2016. The Guarded Lambda-Calculus: Programming and Reasoning with Guarded Recursion for Coinductive Types. *Log. Methods Comput. Sci.* 12, 3 (2016), 1–39. https://doi.org/10.2168/LMCS-12(3:7)2016

Thierry Coquand. 1993. Infinite Objects in Type Theory. In *TYPES'93 (LNCS, Vol. 806)*. 62–78. https://doi.org/10.1007/3-540-58085-9_72

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a Recursive Module?. In *PLDI 1999*. 50–63. https://doi.org/10.1145/301618.301641

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Nils Anders Danielsson. 2010. Beating the Productivity Checker Using Embedded Languages. In *PAR@ITP (EPTCS, Vol. 43)*. 29–48. https://doi.org/10.4204/EPTCS.43.3

Nicolaas Govert de Bruijn. 1972. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Pietro Di Gianantonio and Marino Miculan. 2002. A Unifying Approach to Recursive and Co-recursive Definitions. In *TYPES 2002 (LNCS, Vol. 2646)*. 148–161. https://doi.org/10.1007/3-540-39185-1_9

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Stephen Dolan. 2016. Malfunction Programming. *ML Family Workshop 2016* (2016), 2 pages. https://stedolan.net/talks/2016/malfunction/malfunction.pdf

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *Log. Methods Comput. Sci.* 7, 2 (2011), 1–37. https://doi.org/10.2168/LMCS-7(2:16)2011

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The Impact of Higher-Order State and Control Effects on Local Relational Reasoning. *J. Funct. Program.* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI (2024), 52–75. https://doi.org/10.1145/3656379

Peng Fu and Aaron Stump. 2014. Self Types for Dependently Typed Lambda Encodings. In *RTA-TLCA 2014 (LNCS, Vol. 8560)*. 224–239. https://doi.org/10.1007/978-3-319-08918-8_16

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

Eduardo Giménez. 1994. Codifying Guarded Definitions with Recursive Schemes. In *TYPES'94 (LNCS, Vol. 996)*. 39–59. https://doi.org/10.1007/3-540-60579-7_3

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022), 16:1–16:64. https://doi.org/10.46298/LMCS-18(2:16)2022

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. In *POPL*. 133–146. https://doi.org/10.1145/1926385.1926402

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *POPL 2012*. 59–72. https://doi.org/10.1145/2103656.2103666

Thomas Johnsson. 1987. *Compiling Lazy Functional Programming Languages.* Ph.D. Dissertation. Chalmers University.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-Order Ghost State. In *ICFP*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. 2023. PureCake: A Verified Compiler for a Lazy Functional Language. *Proc. ACM Program. Lang.* 7, PLDI (2023), 952–976. https://doi.org/10.1145/3591259

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL 2017*. 205–217. https://doi.org/10.1145/3009837.3009855

Robbert Krebbers, Luko van der Maas, and Enrico Tassi. 2025. Inductive Predicates via Least Fixed Points in Higher-Order Separation Logic. In *ITP 2025 (LIPIcs)*. 27:1–27:21. https://doi.org/10.4230/LIPICS.ITP.2024.8

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *LICS 2011*. 257–266. https://doi.org/10.1109/LICS.2011.38

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL 2014*. 179–192. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009a. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A Formally Verified Compiler Back-End. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/S10817-009-9155-4

David Läwen. 2025. Artifact of 'Verified Translation of Guarded Programs'. Zenodo. https://doi.org/10.5281/zenodo.17120315

Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science*. 255–266. https://doi.org/10.1109/LICS.2000.855774

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *ICFP 2015*. 166–178. https://doi.org/10.1145/2784731.2784764

Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. 2021. Compositional Optimizations for CertiCoq. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473591

Jamie T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP 2014 (LNCS, Vol. 8410)*. 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Andrew Pitts. 2005. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce (Ed.). The MIT Press, Chapter 7, 245–289. https://doi.org/10.7551/mitpress/1104.003.0011

Andrew Pitts and Ian Stark. 1998. Operational Reasoning for Functions with Local State. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 227–273. http://www.inf.ed.ac.uk/~stark/operfl.html

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *ITP (LNCS, Vol. 9236)*. 359–374. https://doi.org/10.1007/978-3-319-22102-1_24

Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-Memory Concurrency and Verified Compilation. In *POPL 2011*. 43–54. https://doi.org/10.1145/1926385.1926393

Paula Severi. 2019. A Light Modality for Recursion. *Log. Methods Comput. Sci.* 15, 1 (2019), 8:1–8:32. https://doi.org/10.23638/LMCS-15(1:8)2019

Sjaak Smetsers, Eric Nöcker, John H. G. van Groningen, and Marinus J. Plasmeijer. 1991. Generating Efficient Code for Lazy Functional Languages. In *FPCA (LNCS, Vol. 523)*. 592–617. https://doi.org/10.1007/3540543961_28

Simon Spies, Neel Krishnaswami, and Derek Dreyer. 2021. Transfinite Step-Indexing for Termination. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434294

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP (LNCS, Vol. 8410)*. 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP (LNCS, Vol. 7792)*. 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *ICFP 2016*. 60–73. https://doi.org/10.1145/2951913.2951924

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (2024), 40:1–40:75. https://doi.org/10.1145/3676954

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *ICFP*. 377–390. https://doi.org/10.1145/2500365.2500600

Femke van Raamsdonk, Paula Severi, Morten Heine Sørensen, and Hongwei Xi. 1999. Perpetual Reductions in Lambda-Calculus. *Inf. Comput.* 149, 2 (1999), 173–225. https://doi.org/10.1006/INCO.1998.2750