Towards a formalised metatheory of session types in Lean

Wojciech Nawrocki

4th Year Project Report
Computer Science and Physics
School of Informatics
University of Edinburgh

2019

Abstract

Synchronous GV is a minimal, session-typed linear λ-calculus able to describe the protocol-based communication of networked systems. It has strong metatheoretical properties. It is type-sound, free from deadlocks, confluent and terminating. I take first steps towards establishing fully formal proofs of these properties in the Lean theorem prover. To do so, in order to support linearity I recreate a variant of Quantitative Type Theory in Lean by translating a formulation from "Programming Language Foundations in Agda". Next, I write a formal specification of Synchronous GV in Lean and establish a proof of type soundness of its sequential terms. I also formalize some aspects of its concurrent syntax and prove the well-formedness preservation of Synchronous GV configurations (processes) under reduction. I discuss the challenges arising in the use of Lean and theorem provers more generally, as well as solutions to some of them and possible directions for future work.

Acknowledgements

I would like to thank my supervisor, Paul Jackson, for introducing me to Lean, as well as his guidance, encouragement and feedback. Edinburgh-PL researchers, in particular Wen Kokke, Simon Fowler and Sam Lindley for helping me find a clear path through hundreds of research articles. Members of the Lean community for their patience and helpfulness answering swarms of my noob questions on the Zulip forums. Finally, my parents for their unconditional support.

Contents

Chapter 1

Introduction

Writing correct software is hard. Writing correct, concurrent software is more or less impossible. (Un)fortunately, it is also necessary - every modern computer, from phones and tablets to desktops, is multicore. On top of this, most software running today is part of large, distributed systems communicating over the internet in server-client and peer-to-peer models. Consequences of programming errors in networked systems can range from loss of data and database inconsistencies to security breaches and the leakage of secret information. For example, see CVE-2018-10933 - an authentication bug in libssh due to a misimplemented SSH protocol state machine.

Mitigating some of these issues through programming language design has been the subject of a line of work dating back to the 1980s. **Linear logic**, introduced by Girard [22] in 1987, provides an unorthodox framework for reasoning about bounded resources. Interesting on its own, it has found many uses in areas ranging from quantum physics [9] to programming languages. The latter is of particular interest to this project - Abramsky [1] has demonstrated how linear logic corresponds to the π-calculus of concurrent processes by Milner [28], giving it an early computational interpretation. In parallel, Honda [23] introduced **session types** (Section 4.1) - a typing discipline related to the π-calculus, which allows us to describe communication protocols in data types. The implementation of all endpoints communicating over that protocol can then be checked against its session type by the compiler or a language runtime.[1] This goes a long way towards preventing bugs like the one in libssh.

Recently, Caires and Pfenning [6] have tied a number of previous discoveries together by showing a correspondence between all three - linear logic, the π-calculus and session types. While partly a purely academic exercise, it enables designing session-typed languages with a number of strong properties very much applicable to the real world. One such language, Synchronous GV, is the subject of this project.

[1] Frameworks such as Node.js allow programming both client and server implementations in the same language, but on their own they do not guarantee that the endpoints will adhere to the communication protocol - this is where session types come into play.

In 2014, inspired by these advances, Wadler [34] introduced *GV*, a minimal session-typed λ-calculus based on previous work by Gay and Vasconcelos [21] (hence the name). It has since been reformulated by Lindley and Morris [25] and then by Fowler [19] as **Synchronous GV** (SGV) (Chapter 4). I chose this particular language for my project because it is both interesting and useful enough to be worthwile, while also being minimal enough for me to be able to develop its (partial) machine-checked specification in the time available.

The research, however, has not stopped on SGV. A realistic programming language needs more than just adherence to protocols - asynchronous operation and exception handling are only some of the features programmers often find useful. Very recently, Fowler et al. [20] have introduced Exceptional GV (EGV) - an extension of SGV combining these features with session types. EGV has been fully implemented as an extension to Links [10], a language for programming web applications.

Both SGV and EGV, largely due to their correspondence with linear logic and process calculi, enjoy a number of strong metatheoretical properties. They are type-sound, deadlock free, confluent and terminating. Although I will not have the time or space to discuss all of these here and will focus on **type soundness** (Section 2.2), they all either eliminate entire classes of programming errors or allow for powerful reasoning about the behaviour of programs written in EGV/Links.

*

SGV's properties have been proven by humans, on paper. This form of proof has been accepted in mathematics and formal logic for as long as the fields have existed. However, as proofs become more and more complicated, our confidence in them becomes increasingly based on trust in the tiny groups of experts able to understand the reasoning. An extreme example of this is the classification of finite simple groups, the proof of which runs for "over 10,000 pages, spread across 500 or so journal articles, by over 100 different authors"[2]. Humans make mistakes, a number of which has been found in the proof. Even now, there is no real guarantee beyond the participants' expertise that no more mistakes are left.

This is especially important when attempting to reason mathematically about safety-critical software. If we are to trust computer programs, for example those controlling self-driving cars, with human lives, even a formal but human-checked proof of their safety properties is insufficient. Fortunately, similar concerns about the trustworthiness of mathematical knowledge have given rise to the field of machine-verifiable formal reasoning.

Interactive [35] and **automated [14] theorem provers** are two closely related classes of computer programs which can verify and construct formal proofs of mathematical theorems. Most of these systems are *foundational* in the sense that they are based on core logical frameworks. All statements expressible in a given

[2] https://plus.maths.org/content/enormous-theorem-classification-finite-simple-groups [Accessed 2019-04-03]

prover are reducible to the minimal language of its framework. Thanks to this, we only really have to trust the *kernel* which verifies core-language statements of proofs in terms of the system's axioms - an orders-of-magnitude improvement over trusting the manual work of hundreds of people over years of research. One such framework is dependent type theory and the Calculus of Inductive Constructions (CiC) [13]. Coq [11] and Lean (Section 2.3) are two examples of theorem provers based on the CiC.

In order to further the state of current knowledge on session types, automated/interactive theorem proving and establish more trust in safe programming languages, I undertook the formal verification of the metatheoretical properties of SGV to be the subject of this project. Lean, being a promising new language, was my tool of choice for the formal development as well as the target of evaluation.

1.1 Contributions

- The primary contribution of the project is a formal, machine-verifiable specification of Synchronous GV and a proof of type soundness for its sequential components (Section 4.4.3) in the Lean theorem prover. Moreover, I take initial steps towards a mechanized type soundness result for the whole language, including its concurrent parts, by showing the well-formedness preservation of SGV's configuration reduction (Section 5.2.3). To my best knowledge, this is the first fully formal treatment of a significant subset of SGV.
- The secondary contribution is a Lean module for non-dependent QTT (Section 3.2) upon which the SGV specification is based. The definitions, their organisation and crucial lemmas are translated verbatim from "Programming Language Foundations in Agda" [24] but all the proofs have been written from scratch in Lean's tactic mode due to stark differences between how one proves things in Agda and in Lean.
- The third contribution is an evaluation of the design decisions and conceptual problems arising in the process of formalizing a linear calculus, and programming languages more generally in a theorem prover based on the Calculus of Inductive Constructions.

1.2 Overview

During the course of the project, I wrote around 2000 lines of Lean code (not counting empty lines), the majority of which comprises proofs. The longest proof is that of progress for SGV terms (Section 4.4.3) at over 600 lines. The associated repository contains all Lean source code, which can be independently checked by the reader.[3]

[3]Also available at https://github.com/Vtec234/lean-sesh .

In order to carry out the project, I also had to become familiar to some extent with all the presented topics, including but not limited to Synchronous GV, Quantitative Type Theory, linear types, session types, concurrent process calculi, interactive theorem proving, dependent type theory, CiC, Lean, Agda and Coq.

The organisation of this report roughly follows the chronological order of work carried out during the project:

- In chapter 3, linear λ-calculus and Quantitative Type Theory are explained and non-dependent QTT is formalized following a development from the online book "Programming Language Foundations in Agda" [24].
- In chapter 4, sequential terms of Synchronous GV are specified using the language of QTT and their type soundness proven.
- In chapter 5, concurrent configurations of SGV are specified using the previously formulated type of terms and well-formedness preservation of configuration reduction proven.
- In chapter 6, I evaluate the ease of use of the Lean prover as a tool for formalizing programming languages and suggest possible improvements. I also contrast Lean with Coq and Agda. I delay all discussion of proof methods from previous chapters until this chapter.

Chapter 2

Background

2.1 Inductive definitions and the λ-calculus

Inductive definitions. One way of specifying mathematical objects - sets, for example - is via an inductive definition. For example, let's say we want to define the set $\mathbb{N}$ of natural numbers. We can start by stating that $0 \in \mathbb{N}$. Moreover, if $n \in \mathbb{N}$, then $S(n) \in \mathbb{N}$. S is the *successor function*, which simply adds 1. Applying this procedure repeatedly ad infinitum will generate the set of natural numbers: start with $\{0\}$, apply the second rule to get $\{0, 1\}$, $\{0, 1, 2\}$, etc. The well-known *Peano axioms* define natural numbers in a very similar way.

From a set-theoretic perspective, inductive definitions can be viewed as specifying a function from sets to sets whose least fixpoint is the set being defined [2] [12]. For natural numbers, the rules above define $F(X) = \{0\} \cup \{S(n) \mid n \in X\}$ with type $F : U \to U$, where U is the universe of sets. By the Knaster-Tarski theorem, if U is a complete lattice and F is monotone, there exists a least fixed point of F in U - a set $\mathbb{N}$ such that $F(\mathbb{N}) = \mathbb{N}$ and for all Y, if $F(Y) = Y$ then $\mathbb{N} \subseteq Y$. That set is the natural numbers.

The fact of its existence, however, does not guarantee that we can ever reach a given fixpoint via a particular construction such as repeated application of the rule function - it works in the case of natural numbers, but not generally.

Inductive definitions turn out to be extremely useful for specifying and understanding the semantics of programming languages. For example, as we shall see shortly, typing rules are defined in this way. In this area, the usual notation for an inductive definition is to use logical inference rules, which specify that the *conclusion* below the line is implied by zero or more *premises* above the line. On the example of $\mathbb{N}$:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}}$$

The left rule is an *axiom* stating that zero is in the set. The right one claims that

if n is in the set, so is $S\ n$. We can see that this is simply different notation for the description above.

Lambda calculus. Devised by Alonzo Church, lambda calculus (henceforth λ-calculus) is a formal language able to describe arbitrary computation[1]. Given by the BNF

$$M, N\ ::=\ x \mid \lambda x.M \mid M\ N$$

λ-calculus is extremely primitive, containing only function abstraction and application: $\lambda x.M$ abstracts M as a function of the variable x, while $M\ N$ applies N as an argument to the function M. Natural numbers, for example, need not be added as primitives to the language since they can be encoded as *Church numerals*.

To make it somewhat more well-behaved, we can require the untyped λ-calculus to assign each of its terms a type. The simplest system with this restriction is the **simply-typed** λ-calculus (STLC) [29] which, notably, is *not* Turing-complete. While this means that it can no longer express every algorithm, it also gives STLC some nice properties such as strong normalization, i.e. the fact that any sequence of rewrites (e.g. β-reductions) applied to an STLC term will eventually bring it into a normal form that is not further reducible.

Typing STLC. To build up STLC types, we need at least one base type, that is a type which is not inductively constructed out of other types. For example, the unit type $\mathbf{1}$ with exactly one inhabitant - $()$. With $\mathbf{1}$, STLC is given by:

Variables	x, y
Types	$A, B ::= \mathbf{1} \mid A \to B$
Terms	$M, N ::= x \mid \lambda x : A.M \mid M\ N \mid ()$
Type Environments	$\Gamma ::= \mathsf{nil} \mid \Gamma, x : A$

In STLC, the typing environment is a partial map from variable names to the universe of STLC types. It can also be thought of as a set of *named variable bindings* $x : T$, read as "the variable x has type T", where each x is unique. We define the type of each term by specifying *typing rules*. One way to think about these is as an inductive definition for the relation $\vdash$ on triples $(\Gamma,\ M,\ A)$, where $\Gamma \vdash M : A$ can be read as "M has type A under the typing environment Γ". In a way analogous to defining $\mathbb{N}$, we start with the empty set of typing triples and extend it - first with axioms, then with rules that depend on other triples being in the set:

[1] If one believes the Church-Turing thesis.

Typing Rules for STLC $\boxed{\Gamma \vdash M : A}$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \text{T-One} \qquad \frac{}{\Gamma, x : A \vdash x : A}\ \text{T-Var} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : A \to B}\ \text{T-Abs}\ (x \notin \Gamma)$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}\ \text{T-App}$$

If an expression M has some type under environment Γ, then it is *well-typed* (under Γ). An annoying issue is that it is unclear what to do when trying to type an abstraction which introduces a variable x under an environment that already contains x. This can be dealt with by introducing fresh variables and via the use of α-renaming. In this project, however, I will shortly introduce de Bruijn indices which solve such problems. For now it is sufficient to add a side condition to T-Abs specifying that x must be a new name, not yet in Γ.

While too simple to be useful outside of theoretical deliberation, λ-calculus can easily be extended to model more realistic programming languages. In particular, I will use it in the following chapters as a basis upon which to define Synchronous GV.

2.2 Type soundness

As mentioned in the introduction, SGV enjoys a number of strong *metatheoretical* properties. Proving one of these - **type soundness** of SGV terms - formally is one of the main contributions of this project. It therefore bears mentioning what type soundness *is*, exactly. The phrase Milner used when introducing the notion in his 1978 article [27] is still relevant today - type soundness means that "well-typed programs cannot go wrong". In particular, they cannot get stuck during execution - either the program has halted or the next step to take is well-defined. Type disciplines are inherently static - verified at compile-time, while the semantics of execution are inherently dynamic - describing runtime. Type soundness connects the two, making a firm statement about any program's runtime behaviour given only its static properties and is almost always the first major theorem proven in any serious programming language development.

Type soundness is usually implied by two weaker notions - *type preservation* and *progress*. Both of these will be stated formally in Chapter 4.

2.3 The L∃∀N theorem prover

Lean[17] is an open source theorem prover and programming language developed at Microsoft Research. It aims to improve on existing provers by bridging inter-

active, human-guided theorem proving with automation able to resolve tedious details.

The Lean language is dependently-typed and purely functional. For an introduction to dependent type theory and Lean, see for example "Theorem Proving in Lean" [4]. Its type theory, like that of Coq, is based on the Calculus of Inductive Constructions [13]. One difference is that the Lean kernel also includes the axiom of proof irrelevance, which has consequences for equivalence relations (Section 6.1.1). Here I will briefly review some of the Lean features and syntax.

Basic definitions:

```
/- Comments looks like this.. -/
-- ..or this.

/- Functions are defined with
   'def name (args): type := body'. -/
def a_string: string := "Hello world!"
def identity (n: ℕ): ℕ := n

/- Interactive commands provide immediate feedback.
   #check prints the type of the expression in question. -/
#check identity -- ℕ → ℕ
```

Lean provides bidirectional type inference via its elaboration engine [16]:

```
/- Simple type inference. -/
def a_string := "Hello world!"
#check a_string -- a_string : string

/- Unification of a metavariable in a type. -/
def nats: list _ := [0, 1, 2]
#check nats -- list ℕ

/- Arguments can be marked as implicit through
   the use of curly brackets. Implicit arguments
   are always inferred by the elaborator, rather
   than specified explicitly. -/
def swap {α β: Type} (pair: α × β): β × α :=
  (pair.snd, pair.fst)

/- Types of the pair components are inferred. -/
#check swap (0, "string") -- string × ℕ
```

Inductive definitions form the basis of all constructions:

```
/- Inductive definitions are specified using
   the 'inductive' keyword followed by zero
   or more constructors. -/
inductive nat: Type
| zero: nat
| succ (n: nat): nat

/- Structures are really syntax sugar for
   an inductive type with one constructor
   'mk' and a projection for each member. -/
structure pair (α β: Type) :=
```

```
(a: α)
(b: β)

/- Here ?M_i are metavariables - variables of the
   elaboration process which can be inferred from
   the surrounding context. -/
#check pair.mk -- pair.mk : ?M_1 → ?M_2 → pair ?M_1 ?M_2
#check pair.a -- pair.a : pair ?M_1 ?M_2 → ?M_1
#check pair.b -- pair.b : pair ?M_1 ?M_2 → ?M_2
```

Pattern matching compiles via the *equation compiler*:

```
/- Pattern matching. -/
def is_the_answer: ℕ → bool
| 42 := tt
| _  := ff

/- Lean definitions are terminating and so recursion
   must be well-founded, but the equation compiler
   can often automatically find a proof of this. -/
def fib: ℕ → ℕ
| 0     := 0
| 1     := 1
| (n+2) := fib (n+1) + fib n
```

Via the Curry-Howard correspondence, propositions are types in the `Prop` universe. The elements of proposition types are their proofs:

```
/- An example functional proof. -/
def and_commutative (P Q: Prop): P ∧ Q → Q ∧ P :=
  λ h, ⟨h.right, h.left⟩
```

Finally, Lean supports type classes, which we will encounter in greater detail in the upcoming chapters:

```
class has_add (α : Type u) := (add : α → α → α)

/- Require that α have addition defined by enforcing
   that it is a member of the has_add type class.
   The instance [has_add α] can be found via an
   automatic search procedure. -/
def twice {α: Type} [has_add α] (x: α) := x + x
```

Lean also features tactic-mode proofs and a metaprogramming language, which I describe in Section 6.2.1.

2.4 Representations of a language

Just as we can look at one concept from many points of view, theorem proving environments like Lean provide many ways to express the same idea. In particular, stating the syntax and semantics of STLC, or of any **target-language** more generally, can be achieved via various means, requiring the programmer to choose one out of many metatheoretical frameworks. Some desidarata guiding how one should go about formalizing their target-language of interest are:

- the ability to express and prove all desired properties of the target-language

- the amenability of the formalization to automatic proof techniques which could significantly reduce the workload
- how extensibile and easy to modify the formalization would be
- how understandable the mechanised definitions and proofs end up being

A large part of my work in this project was to explore various ways of expressing SGV, evaluate them according to the above criteria and make an informed design decision about which one to use. Since the behaviour of SGV is stated in terms of small-step operational semantics, I skipped many interesting embeddings such as Parametric Higher-Order Abstract Syntax [8], which is more appropriate for denotational semantics.

Instead, I shortly explored explored a **weakly-typed** formulation following Pierce et al. [30] up to a proof of progress for STLC and then switched to a **strongly-typed** one.[2] The choice between these two options can be phrased as: "Do I want the host-language type system to check the well-formedness of terms in the target-language?" [5].

In a strongly-typed formulation, only well-formed target-language terms are representable. Usually, well-formed in this context means well-typed, but if the host-language type system is expressive enough, arbitrary invariants can be encoded in host-language types. Ill-formed target-language terms will be rejected by the host (Lean, in this case) typechecker. In a weakly-typed formulation, terms of the target-language are defined purely syntactically, separately from typing rules, and can therefore be constructed incorrectly without restriction.

2.4.1 Weakly-typed terms

To see this in practice, let us first consider a weakly-typed formulation of STLC in Lean:

```
/- STLC types -/
inductive tp
| Unit: tp
| Fn: tp → tp → tp
open tp

/- STLC terms -/
inductive term
| One: term
| Var: string → term
| Abs: string → tp → term → term
| App: term → term → term
open term

/- Typing environments are represented as a partial map from variable
   names to types. Different representations, for example as a set
   of pairs, are also possible. -/
```

[2] There is no standard terminology here - extrinsic vs. intrinsic, raw vs. inherently-typed, etc.

```
def env := string → option tp
/- The empty environment. -/
def env.nil: env := λ x, none
/- A way to extend environments. -/
def env.ext (Γ: env) (x: string) (A: tp)
  : env := λ x', if x = x' then A else Γ x'

/- (typeof Γ M A) is the typing judgment Γ ⊢ M:A. In the weakly-typed
   formulation, it is stated separately from the `term` definition. -/
inductive typeof: env → term → tp → Prop
| TOne: ∀ (Γ: env),
  -----------------
  typeof Γ One Unit

| TVar: ∀ (Γ: env) x A,
  ---------------------------
  typeof (Γ.ext x A) (Var x) A

| TAbs: ∀ (Γ: env) x M A B,
  (Γ x).is_none
→ typeof (Γ.ext x A) M B
  ----------------------------
→ typeof Γ (Abs x A M) (Fn A B)

| TApp: ∀ {Γ: env} {M N: term} {A B: tp},
  typeof Γ M (Fn A B)
→ typeof Γ N A
  --------------------
→ typeof Γ (App M N) B

/- An expression which applies the unit constant as if it were a function
   is valid in the host language, even though it is ill-typed in the target
   language. There is no way to actually construct an instance of Typeof
   for this expression. -/
#check (App One (Abs "x" Unit One): term)
```

As demonstrated here, weak typing grants more freedom in what is expressible, allowing for easy manipulation of target-language terms. This comes at a price, though, of having to specify large numbers of auxilliary lemmas just to verify the correctness of various definitions in terms of the semantics of the target-language. For example, any transformation `f: term → term` over terms that is supposed to preserve their types needs to have an associated lemma of the form `∀ {Γ: env} {M: term} {A: tp}, typeof Γ M A → typeof Γ (f M) A`. This generally results in longer specifications and proofs.

2.4.2 De Bruijn indices

Another issue of note in the above code listing is the difficulty of dealing with typing environments. The side condition $x \notin \Gamma$ from T-Abs is represented as `(Γ x).is_none`. More abstractly, a definition of environments using either functions or sets does not have a nice inductive structure, which is generally desirable in CiC-based theorem provers due to its compatibility with various proof techniques. These problems can be disposed of through the use of **de Bruijn indices**.

Named binding defines the typing environment as a partial map from names to types, while de Bruijn indices replace variable names with numbers [15], s.t. the variable n refers to the n-th λ binder working outwards from its use site (zero-indexed). For example, $\lambda.\lambda.1$ is equivalent to $\lambda x.\lambda y.x$ and $\lambda.\lambda.\lambda.2\,0$ to $\lambda x.\lambda y.\lambda z.x\,z$. De Bruijn indices feel less natural, but turn out to be far more convenient to work with, therefore most automatic reasoning systems use them instead of named bindings. α-equivalence is not at all a concern, because all expressions have a unique, canonical de Bruijn form. Substitution of terms into lambda abstractions with de Bruijn involves simply "lifting" all variable indices i.e. shifting their numeric values by an appropriate amount. Substitution using named bindings, on the other hand, requires keeping track of which new names are introduced and renaming them in order to avoid clashes if the substituted term has free variables.

In Lean, we can represent a strongly-typed version of de Bruijn indices.[3] Besides selecting the n-th outermost binder, this representation also encodes the entire typing environment and the type of the variable selected. This is useful because it provides a witness that such a binding does, in fact, exist in the typing environment. While using a representation of de Bruijn indices which does not store this information would still be an improvement over named bindings thanks to removing concerns about α-equivalence, it would require manually carrying through proofs - that at every variable use site the number of enclosing binders is greater than the used index, as well as that the type of the n-th binder is the right one. In a strongly-typed representation, both of these are embedded in the type of de Bruijn indices:

```
/- The typing environment is now just a list. -/
def env := list tp

/- Indexes a binding in the typing context. Definition follows
   the inductive structure of natural numbers. -/
inductive debruijn_idx: env → tp → Type
infix ` ∋ `:40 := debruijn_idx
| ZVar: Π Γ A, A::Γ ∋ A
| SVar: Π {Γ A} B, Γ ∋ A → B::Γ ∋ A
open debruijn_idx

/- An index of Lean Type `Γ ∋ A` selects some variable
   of STLC type A out of the environment Γ. -/
local infix ` ∋ `:40 := debruijn_idx

/- Note that a `debruijn_idx` "knows" its type and the environment
   under which it is well-formed. In this case, the Unit type under
   the environment [Unit]. -/
def zero: [Unit] ∋ Unit :=
  ZVar [] Unit

/- This is equivalent to x in λx:(1→1).λy:1.λz:1.x z -/
def two: [Unit, Unit, Fn Unit Unit] ∋ Fn Unit Unit :=
  SVar Unit $ SVar Unit $ ZVar [] (Fn Unit Unit)
```

[3] Unfortunately understanding this representation is not easy, but it will hopefully become clearer as we go along and use the indices.

```
/- This is the y in λx:(1→1).λy:1.y . Notice that the index
   can also include the types of binders further away than
   the one it references. This makes it possible to constrain
   the environment of the index to be exactly the environment
   of the entire expression, even though the index itself
   does not really use all of it (but it still provides a witness
   that the variable's type exists in the environment). -/
def zero_free: [Unit, Fn Unit Unit] ∋ Unit :=
  ZVar [Fn Unit Unit] Unit
```

2.4.3 Strongly-typed terms

Strongly-typed terms use the host language's typechecker to validate embedded terms. This guarantees correctness by construction and is especially appealing because it tends to guide proofs of statements about the embedded language by eliminating large classes of incorrect expressions without user intervention. This representation is, however, also more difficult to understand and stresses the limits of the host-language, its type system and proof automation.

An example strongly-typed formulation of STLC with **1** could look as follows:

```
/- A term can only be built given its own typing derivation,
   therefore any expression of type (term _ A) is a valid
   STLC term by definition. (term Γ A) can be read as
   "the set of STLC terms that under environment Γ have
   STLC type A". -/
inductive term: env → tp → type
| One: ∀ Γ, term Γ Unit
| Var: ∀ {Γ A}, Γ ∋ A → term Γ A
| Abs: ∀ {Γ A B}, term (A::Γ) B → term Γ (Fn A B)
| App: ∀ {Γ A B}, term Γ (Fn A B) → term Γ A → term Γ B
open term

/- Valid terms typecheck. -/
#check (App (Abs $ Var $ ZVar [] Unit) (One []): term [] Unit)

/- type mismatch at application
  App (One ?m_4)
term
  One ?m_1
has type
  term ?m_1 Unit
but is expected to have type
  term ?m_1 (Fn ?m_2 ?m_3) -/
#check App (One _) (One _)
```

What happens here is that the typing relation (previously `typeof`) and the definition of STLC terms are conflated into a single object - `term`. The inductive definition of a target-language term "knows" its own typing derivation and is impossible to construct without it. Thanks to this, well-typedness is encoded in a host-language type and malformed terms cannot be constructed. Moreover, any function operating over STLC terms can express target-language type preservation in its type without any auxilliary lemmas: `f: ∀ {Γ}, term Γ A → term Γ`

A. This is great, but as we shall see later, it also significantly stresses the limits of Lean proof automation and its type theory.

Chapter 3

Linear λ-calculus and Quantitative Type Theory

3.1 Linear λ-calculus

An extension of STLC of particular interest to this project is the simply-typed, *linear* λ-calculus, the typing rules of which are as follows[1]:

Typing Rules for Linear STLC $\boxed{\Gamma \vdash M : A}$

$$\frac{}{x : A \vdash x : A}\ \text{T-Var} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : A \multimap B}\ \text{T-Abs} \qquad \frac{\Gamma \vdash M : A \multimap B \qquad \Delta \vdash N : A}{\Gamma, \Delta \vdash M\, N : B}\ \text{T-App}$$

Where, importantly, a variable x can only be typed (T-Var) in an environment containing nothing but the type of x. Moreover, in function application (T-App) the typing environment Γ, Δ is a union of two **disjoint** environments Γ and Δ. Γ has to be *exhausted*, i.e. used up, by M and Δ by N. The effect of these rules is that every variable has to be used (i.e. referenced within the body of the abstraction it was introduced in) exactly once - no more, no less. This is known as **linear typing**.

As part of a more general family of substructural type systems, linear types are useful when dealing with resource-constrained situations. For example, we might want to ensure that a memory region is deallocated, but not more than once. If the only way to consume the region resource is by deallocating it, a linear discipline would force the programmer to do so and ensure freedom from memory leaks by design. We will see later that session types in SGV are an extension of linear types.

[1] Assuming some base type(s), the rules for which are not given.

Specifying linear λ-calculus in a theorem prover has historically been challenging [citation needed] - most representations of the above constraints on typing environments are cumbersome to work with. Fairly recently, McBride [26] and Atkey [3] have developed **Quantitative Type Theory** (QTT). Although its goal is to combine linear and dependent types into a "dependent lollipop" $(\mathsf{x} : A) \multimap B[x]$, i.e. a linear *and* dependent function type, it turns out to also be helpful for the mechanisation of non-dependent linear λ-calculus, where we mostly ignore the dependent part of QTT - henceforth I shall call this simplified framework *non-dependent QTT*. It also has built-in support for both linear and regular expressions in a single calculus, to a similar effect as what was done by e.g. Wadler [33].

3.2 Quantitative Type Theory

This section follows the Lean development of QTT in tandem with its theoretical description. The implementation is translated from "Programming Language Foundations in Agda" [24].

The core idea of QTT is to equip every variable binding $\mathsf{x} : T$ in the typing environment with a resource usage annotation, or **multiplicity** $\pi \in R$:

$$\Gamma \ ::= \ \mathsf{nil} \mid \Gamma, x \overset{\pi}{:} T$$

where R is an arbitrary **partially ordered semiring**. The annotation on a binding specifies how many units of the resource associated with that binding the environment contains. Since any semiring must have a 0 and a 1, we can give these multiplicity values special interpretations. A binding annotated with 1 can be used at most once and is thus affine. Zero is more interesting - the original reason for its introduction was to allow considering variables non-computationally for the formation of types in dependent QTT. This type of usage consumes 0 resources and can thus be repeated indefinitely. In a non-dependent context, a 0-resourced binding can be thought of as not existing at all. As will become clearer after specifying QTT typing rules, this is useful because it allows us to extend typing environments with arbitrary 0-resourced binding without affecting the well-typedness of expressions in any way.

Fixing a particular R gives the resulting type system specific properties. In particular, $R = \{0, 1, \omega\}$ (read as *zero-one-many*) with an appropriate partial order will make the calculus linear.

3.2.1 Typing contexts

To specify typing rules for QTT, we will need to construct Lean objects corresponding to typing contexts. Let a **precontext**[2] γ be the standard typing environment without usage annotations:

[2]This use of terminology differs from the QTT papers but matches that of Kokke and Wadler [24] and the Lean implementation.

$\gamma ::= \mathsf{nil} \mid \gamma, x : T$

In Lean, we represent this in the *nameless*, de Bruijn convention as a list of types:

```
@[reducible]
def precontext := list tp
```

Something to note is that neither `precontext` nor other definitions in the module are parametric over `tp`, the Lean `Type` of types of the language under consideration, even though they could be, as QTT is a general theory. Instead, they are specific from the beginning - `tp` here is the set of SGV types. This is essentially for performance reasons - I found that the parametric version slows down interactive proof tactics and elaboration to the point of making them unusable due to frequent timeouts. With that in mind, the formalization is written so that it can be made generic with a few textual substitutions.

Henceforth, elements of R are given by the `Type mult`. On top of precontexts, we define **γ-contexts** - assignments of multiplicity to each binding in γ:

```
/- Follows the usual recursive list definition. -/
inductive context: precontext → Type
| nil: context []
| cons {γ: precontext} (π: mult) (T: tp): context γ → context (T::γ)
/- If γ is a precontext and Γ is a γ-context,
   then ⟦π·T⟧::Γ is a (T::γ)-context, where the head
   binding has T available for use π times. -/
notation `⟦`π`·`T`⟧::`Γ:90 := context.cons π T Γ
```

These serve the role of the typing environment from before. In the de Bruijn convention, we can identify each γ-context for a particular γ with a vector of multiplicities $(\pi_1, \pi_2, .., \pi_n)$, where $n =$ `γ.size`.[3] Because R is a semiring, such vectors have a few useful properties. Firstly, they make a commutative monoid with pointwise addition. In Lean, I represent this by defining vector addition and proving that γ-contexts are a member (`instance`) of the `add_comm_monoid` type class:

```
/- Multiplicities in γ-contexts can be added pointwise. -/
protected def add: Π {γ}, context γ → context γ → context γ
| _ nil nil := nil
/- T occurs twice, but Lean cannot match the same variable twice,
   so its first occurrence is marked as an inaccessible term .(T).
   This occurrence does not participate in pattern matching. -/
| _ (⟦π₁·.(T)⟧::Γ₁) (⟦π₂·T⟧::Γ₂) := ⟦(π₁+π₂)·T⟧::(add Γ₁ Γ₂)

/- Tell Lean to use the above function whenever it sees
   (+) used with contexts. -/
instance {γ: precontext} : has_add (context γ) :=
  ⟨context.add⟩

/- The monoid zero is a γ-context containing only zeros. -/
def zeros: Π (γ: precontext), context γ
| [] := nil
| (T::δ) := ⟦0·T⟧::(zeros δ)
```

[3] A named formulation could use a partial map from names to multiplicities.

```
/- Use (zeros γ) whenever a 0 appears with expected type
   context γ. -/
instance {γ: precontext} : has_zero (context γ) :=
  ⟨zeros γ⟩

/- Some lemmas here. -/

/- Instantiating type class membership requires providing proofs
   of all additive commutative monoid properties. -/
instance {γ: precontext} : add_comm_monoid (context γ) :=
{ add := context.add, /- monoid addition -/
  zero := zeros γ, /- monoid zero -/
  zero_add := @zero_add γ, /- 0 + Γ = Γ -/
  add_zero := @add_zero γ, /- Γ + 0 = Γ -/
  add_comm := @add_comm γ, /- Γ₁ + Γ₂ = Γ₂ + Γ₁ -/
  add_assoc := @add_assoc γ /- (Γ₁ + Γ₂) + Γ₃ = Γ₁ + (Γ₂ + Γ₃) -/ }
```

Secondly, the vectors make an R-semimodule with scalar multiplication $\pi\Gamma$, which multiplies Γ by π pointwise using the "times" operation of R. Again, this is represented by a type class `instance`, this time in class `semimodule mult` (where R is `mult`):

```
/- Scalar-vector multiplication. Scalars are the multiplicities,
   i.e. elements of R. -/
protected def smul: Π {γ}, mult → context γ → context γ
| _ π nil := nil
| _ π (⟦π'·T⟧::Γ) := ⟦(π*π')·T⟧::(smul π Γ)

/- Tell Lean to use `smul` whenever it encounters π • Γ
   for some π: mult and some Γ: context _. -/
instance {γ: precontext} : has_scalar mult (context γ) :=
  ⟨context.smul⟩

/- Semimodule lemmas here. -/

instance {γ: precontext} : semimodule mult (context γ) :=
{ one_smul := @one_smul γ, /- 1 • Γ = Γ -/
  zero_smul := @zero_smul γ, /- 0 • Γ = 0 -/
  smul_zero := @smul_zero γ, /- π • 0 = 0 -/
  smul_add := @smul_add γ, /- π • (Γ₁ + Γ₂) = π•Γ₁ + π•Γ₂ -/
  add_smul := @add_smul γ, /- (π₁ + π₂) • Γ = π₁•Γ + π₂•Γ -/
  mul_smul := @mul_smul γ /- (π * π') • Γ = π • (π' • Γ) -/ }
```

Besides neatly packaging groups of lemmas for a particular algebraic structure into a Lean `structure`, type classes provide a powerful mechanism for generic programming, whereby statements that hold of an algebraic structure can be proven once and specialized to a particular instance of that structure on demand. Having only proven that `add_comm_monoid (context γ)`, we could apply any theorem of the form `theorem some_thm {α: Type} [add_comm_monoid α]: ∀ {a: α}, P a` where `P {α: Type}: α → Prop`. Moreover, generic interactive tactics such as `ring`, which simplifies expressions involving elements of any commutative semiring, can be written.

I show and elaborate on some of the lemmas omitted from above in Chapter 6.

3.2.2 Typing rules

With an understanding of γ-contexts, we can reformulate STLC in terms of non-dependent QTT. Firstly, the syntax needs to be amended as follows:

$$\begin{aligned}\text{Types}\quad & A, B ::= \mathbf{1} \mid (x \overset{\pi}{:} A) \to B \\ \text{Terms}\quad & M, N ::= x \mid \lambda x \overset{\pi}{:} A.M \mid M\,N \mid ()\end{aligned}$$

In this version, the function type has to explictly state how many (π) instances of the argument are consumed by the body M. Notice that B is *not* allowed to depend on A since, as I mentioned, we do not need dependent types in this work.

Given this syntax, we can establish the following base set of typing rules:

Typing Rules for Non-Dependent QTT $\boxed{\Gamma \vdash M \overset{\pi}{:} A}$

$$\text{T-Var}\ \frac{}{0\Gamma, x \overset{\pi}{:} A \vdash x \overset{\pi}{:} A} \qquad \text{T-Abs}\ \frac{\Gamma, x \overset{\pi\sigma}{:} A \vdash M \overset{\sigma}{:} B}{\Gamma \vdash (\lambda x \overset{\pi}{:} A.M) \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B}$$

$$\text{T-App}\ \frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} A) \to B \qquad \Gamma_2 \vdash N \overset{\sigma}{:} A}{\Gamma_1 + \pi\Gamma_2 \vdash M\,N \overset{\sigma}{:} B}$$

The typing judgment $\Gamma \vdash M \overset{\pi}{:} A$ can be read as "under the γ-context Γ, the term M has type A and is available for use π times". The precontexts are implied by the contexts - Γ_i are γ-contexts, Δ_i are δ-contexts, etc. Whenever two contexts appear in an algebraic operation such as the pointwise addition of their multiplicities in T-App, they must have the same precontext for the operation to be defined, therefore they are both γ-contexts for some γ.

Similarly to the behaviour of linear STLC, the rule for typing variables (T-Var) requires that the context only contain $x : A$. However, unlike previously, it can contain other bindings 0 times. It can be read as "under a γ-context containing $(x : A)$ π times and possibly other bindings 0 times, there exist π xs with type A".

How can the other 0-resourced bindings appear?

When the (T-App) rule is applied, the typing context is not split into disjoint unions as before. Instead, we say that the application $M\,N$ is well-typed under the γ-context $(\Gamma_1 + \pi\Gamma_2)$ if there exist Γ_1 and Γ_2 s.t. there is some way to type M under Γ_1 and to type N under Γ_2. The splitting is achieved using addition on the multiplicity vectors. For example, $(1,0) + \pi(0,1)$ means that the first variable is used in M and the second in N. As in the syntax, π states how many times the function M will use its argument and can be seen in the rule T-Abs.

The fact that in these (and other) rules, the environment can be a γ-context for *any* γ as long as it contains all variables, i.e. $\forall i,\ \gamma \ni A_i$, massively simplifies

a strongly-typed mechanisation in a theorem prover like Lean. As mentioned earlier, a γ-context can be extended to a (B,γ)-context for any B by appending a 0-resourced binding, which will useful when defining renaming.

The rules presented here are too general for our needs, so I will not provide a Lean definition of them. Instead, in Chapter 4 I will list simpler rules restricted to Synchronous GV.

3.2.3 Substitution

An interesting insight of Kokke and Wood, which follows from the QTT typing rules, is that substitution can be defined as matrix multiplication of contexts. Suppose we have a term s with n_γ free variables, well-typed under the γ-context Γ. Then, say we want to perform **simultaneous substitution** of n_γ terms e_1, e_2, .., e_{n_γ} into s, replacing each of its free variables. If every e_i is well-typed under a δ-context Δ_i, then we can compute the resource requirements of $s\{e_1/1,\ e_2/2,\ \ldots,\ e_n/n\}$ as follows.

Let Ξ be an $n_\gamma \times n_\delta$ matrix whose i-th row is Δ_i. Multiplying Γ by Ξ (denoted as $\circledast$) will then result in a γ-context specifying exactly how many resources we need for the resulting expression with substitutions applied. For example, substitution into a term with two free variables could require the following resources:

$$\Gamma \circledast \Xi = \begin{pmatrix} 1 & \omega \end{pmatrix} \circledast \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & \omega \end{pmatrix}$$

In Lean, we define matrices as (again strongly-typed) maps from de Bruijn indices to their corresponding rows:

```
def matrix (γ δ: precontext): Type :=
  Π (T: tp), γ ∋ T → context δ
```

The identity matrix is one that returns a γ-context containing a single 1 at the n-th position and zeros everywhere else when given the de Bruijn index n. It is equivalent to the usual definition:

$$\mathbb{1} = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

```
def identity: Π γ, matrix γ γ
| (T::_) _ (ZVar _ _) := ⟦1·T⟧::0
| (T::γ') U (SVar .(T) x) := ⟦0·T⟧::(identity γ' U x)
```

Vector-matrix multiplication is also defined:

```
/- Although the definition below is fairly impenetrable,
   it behaves exactly like usual vector-matrix multiplication. -/
def vmul: ∀ {γ δ}, context γ → matrix γ δ → context δ
| _ _ context.nil _ := 0
| (_::γ) _ (⟦π·T⟧::Γ) Ξ := (π • Ξ T $ ZVar γ T)
                         + (vmul Γ $ λ U x, Ξ U $ SVar T x)
```

```
infix ` ⊛ `:70 := vmul

/- On top of this, we prove several useful lemmas,
   but unlike in the previous cases, do not specify
   contexts with vmul to be part of any particular
   type class. For example: -/
@[sop_form] lemma vmul_right_distrib
  : ∀ {γ δ} {Γ₁ Γ₂: context γ} {Ξ: matrix γ δ},
    (Γ₁ + Γ₂) ⊛ Ξ = (Γ₁ ⊛ Ξ) + (Γ₂ ⊛ Ξ) := /- proof -/
```

This concludes the formalization of non-dependent QTT. The next chapter will look into SGV proper.

Chapter 4

SGV terms

Synchronous GV (SGV) [34] [25] [19] is a linear λ-calculus with session types and synchronous communication.

Both its linearity and communication constructs pose a challenge for mechanisation in a system like Lean. In this chapter, I will deal with linearity (using non-dependent QTT) and single-threaded behaviour, while communication will be investigated later. The syntax [1] of SGV is given by:

$$
\begin{array}{lll}
\text{Types} & A, B, C & ::= \mathbf{1} \mid A \multimap B \mid A + B \mid A \times B \mid S \mid S^\sharp \\
\text{Session Types} & S & ::= !A.S \mid ?A.S \mid \mathsf{End}_! \mid \mathsf{End}_? \\
\\
\text{Variables} & x, y, z & \\
\text{Terms} & L, M, N & ::= x \mid \lambda x.M \mid M\,N \\
 & & \mid \ () \mid \textsf{let}\,() = M\,\textsf{in}\,N \\
 & & \mid \ (M, N) \mid \textsf{let}\,(x, y) = M\,\textsf{in}\,N \\
 & & \mid \ \textsf{inl}\,M \mid \textsf{inr}\,M \mid \textsf{case}\,L\,\textsf{of}\,\{\textsf{inl}\,x \mapsto M; \textsf{inr}\,y \mapsto N\} \\
 & & \mid \ \textsf{fork}\,M \mid \textsf{send}\,M\,N \mid \textsf{receive}\,M \mid \textsf{wait}\,M \\
\\
\text{Type Environments} & \Gamma & ::= \cdot \mid \Gamma, x \overset{\pi}{:} A
\end{array}
$$

The expressions in **blue** describe standard programming constructs like pairs, let-bindings, etc. and can be considered extensions to linear STLC that have no significant impact on its metatheory. **Red** constructs, on the other hand, deal with communication over session-typed *channels* (Section 5.2). These require semantics for the reduction of parallel compositions in order to make sense, so I will describe them in Chapter 5.

There are two departures from previous presentations of SGV here. Firstly, in order to support non-dependent QTT, bindings in type environments are enriched with a multiplicity π. Function abstraction $\lambda x.M$, however, is not allowed to

[1] All SGV syntax and semantics figures reproduced and altered with permission from Fowler [19].

specify a custom resource usage. Instead, in order to disallow multiple uses of any linear variable, it is always implied to be 1. Secondly, the runtime type of channels $S^\sharp$ is conflated with regular types in order to simplify the formalisation.

4.1 SGV types

Regular SGV types A, B, C include the unit type $\mathbf{1}$, product $A \times B$, sum $A + B$ and linear function $A \multimap B$, which is equivalent to $(x \overset{1}{:} A) \to B$ in non-dependent QTT.

Session types S make up the more interesting part of the type system. As mentioned in the introduction, their core idea is to encode a communication protocol directly in the type of a λ-expression. The two most basic session types are $!A.S$ and $?A.S$. The former reads "send a value of type A and then continue with S", while the latter reads "receive a value of type A and then continue with S". These two are in direct correspondence - if one endpoint sends $(V : A)$, the other should expect to receive $(V : A)$.

This is represented via the concept of **duality**. Put shortly, given a session type S encoding all operations in some protocol from the point of view of one endpoint (e.g. the client), $\overline{S}$ is the type of the opposite endpoint (e.g. the server) whose behaviour will exactly mirror that of the first one. Formally:

Duality $\boxed{\overline{S}}$

$$\overline{!A.S} = ?A.\overline{S} \qquad \overline{?A.S} = !A.\overline{S} \qquad \overline{\mathsf{End}_!} = \mathsf{End}_? \qquad \overline{\mathsf{End}_?} = \mathsf{End}_!$$

For a more concrete example, let's say we would like to define a one-shot echo protocol - the server simply echoes back whatever the client has sent once and then exits. Its type could then be $S_s = ?A.!A.\mathsf{End}_!$ - "receive A, send A, finish" - while the client would be typed as $S_c = \overline{S}_s = !A.?A.\mathsf{End}_?$ - "send A, receive A, finish" - for some A, the type of what's being echoed. Note that the session type *cannot* enforce that the server send back what it received[2], only that it adheres to the communication protocol and sends back a value of the right type. Additionally, because SGV is terminating, a loop-forever echo server is *not* expressible - a modification to the language would be necessary.

$\mathsf{End}_!$ and $\mathsf{End}_?$ are the terminators for send and receive operations, respectively. They are always the last operation in a protocol, and hence the rightmost subtype of a session type. Finally, the "hash" type $S^\sharp$ describes both endpoints in a single type.

Regular and session types are defined in a *mutually inductive* way - a session type is a regular type and some session types can include regular types. To represent this, I used the mutual induction feature of Lean:

[2]Parametricity aside.

```
mutual inductive tp, sesh_tp
/- A, B, C -/
with tp: Type
| unit: tp
| fn: tp → tp → tp /- A⊸B -/
| sum: tp → tp → tp /- A+B -/
| prod: tp → tp → tp /- AxB -/
| sesh: sesh_tp → tp /- S -/
| hash: sesh_tp → tp /- S♯ -/
/- S, T -/
with sesh_tp: Type
| send: tp → sesh_tp → sesh_tp /- !A.S -/
| recv: tp → sesh_tp → sesh_tp /- ?A.S -/
| end_send: sesh_tp /- End! -/
| end_recv: sesh_tp /- End? -/

/- Notation: -/
infix `⊸`:90 := tp.fn
notation S`♯`:90 := tp.hash S
notation `End!` := sesh_tp.end_send
notation `End?` := sesh_tp.end_recv
notation `!`A`·`S:90 := sesh_tp.send A S
notation `?`A`·`S:90 := sesh_tp.recv A S
```

An interesting feature of Lean is that it can automatically convert from type `α` to type `β` if it can find an instance of the `has_coe α β` type class. The instance below allows passing in a `sesh_tp` directly whenever a `tp` is expected:

```
/- Automatically coerce session types to types. -/
instance sesh_tp_to_tp : has_coe sesh_tp tp :=
  ⟨λ S, tp.sesh S⟩
```

Duality is specified as a recursive function:

```
def sesh_tp.dual: sesh_tp → sesh_tp
| !A·S := ?A·(dual S)
| ?A·S := !A·(dual S)
| End! := End?
| End? := End!

/- Duality is its own inverse. -/
lemma sesh_tp.dual_dual (S: sesh_tp)
  : dual (dual S) = S := /- proof -/
```

4.2 Choice of semiring

To establish typing rules, we need to choose a semiring over which to instantiate QTT. As mentioned earlier, $R = (0, 1, \omega)$ is a good choice which, with some restrictions, provides linearity. The Lean formalization of R defines basic operations and ends with a proof that `mult` is a commutative semiring (this is stronger than required by QTT, which doesn't need commutativity):

```
@[derive decidable_eq]
inductive mult: Type
| Zero: mult
| One: mult
```

```
| Many: mult

protected def add: mult → mult → mult
| 0 π := π
| π 0 := π
| 1 1 := ω
| ω _ := ω
| _ ω := ω

protected def mul: mult → mult → mult
| _ 0 := 0
| 0 _ := 0
| π 1 := π
| _ ω := ω

/- some lemmas -/

instance : comm_semiring mult := /- instance definition -/
```

An important thing to note is that $1+1=\omega$. In principle then, it could be possible to type an expression which uses the same variable twice or more, breaking linearity. However, in the following section I define the SGV typing rules in a way that only permits variable bindings with a multiplicity of 1, making it impossible for ω to appear when applying the rules. In addition to this, any top-level context must only contain ones - this is taken as a convention.

4.3 Typing rules

Typing Rules for Terms $\boxed{\Gamma \vdash M : A}$

$$\textsc{T-Var}\ \frac{}{0\Gamma_1, x \overset{1}{:} A, 0\Gamma_2 \vdash x : A} \qquad \textsc{T-Abs}\ \frac{\Gamma, x \overset{1}{:} A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1 + \Gamma_2 \vdash M\,N : B}$$

$$\textsc{T-Unit}\ \frac{}{0\Gamma \vdash () : \mathbf{1}} \qquad \textsc{T-LetUnit}\ \frac{\Gamma_1 \vdash M : \mathbf{1} \quad \Gamma_2 \vdash N : A}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,() = M\,\mathbf{in}\,N : A}$$

$$\textsc{T-Pair}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B}{\Gamma_1 + \Gamma_2 \vdash (M, N) : A \times B} \qquad \textsc{T-LetPair}\ \frac{\Gamma_1 \vdash M : A \times B \quad \Gamma_2, x \overset{1}{:} A, y \overset{1}{:} B \vdash N : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{let}\,(x, y) = M\,\mathbf{in}\,N : C}$$

$$\textsc{T-Inl}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathbf{inl}\,M : A + B} \qquad \textsc{T-Inr}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathbf{inr}\,M : A + B}$$

$$\textsc{T-Case}\ \frac{\Gamma_1 \vdash L : A + B \quad \Gamma_2, x \overset{1}{:} A \vdash M : C \quad \Gamma_2, y \overset{1}{:} B \vdash N : C}{\Gamma_1 + \Gamma_2 \vdash \mathbf{case}\,L\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} : C}$$

$$\textsc{T-Fork}\ \frac{\Gamma \vdash M : S \multimap \mathsf{End}_!}{\Gamma \vdash \mathbf{fork}\,M : \overline{S}} \qquad \textsc{T-Send}\ \frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : !A.S}{\Gamma_1 + \Gamma_2 \vdash \mathbf{send}\,M\,N : S} \qquad \textsc{T-Recv}\ \frac{\Gamma \vdash M : ?A.S}{\Gamma \vdash \mathbf{receive}\,M : (A \times S)}$$

$$\textsc{T-Wait}\ \frac{\Gamma \vdash M : \mathsf{End}_?}{\Gamma \vdash \mathbf{wait}\,M : \mathbf{1}}$$

The rules are essentially those of Fowler's formulation, except that disjoint unions of contexts Γ_1, Γ_2 have all been replaced with γ-context sums $\Gamma_1 + \Gamma_2$. Moreover, all variables (e.g. in T-Var) are bound with a multiplicity of 1, ensuring linearity. Multiplicities are elided from the right-hand-side of typing judgments and are simply always implied to be 1.

Do note that it is possible to type a non-linear expression by starting with a top-level context containing ωs. It is taken as a convention not do that.

4.4 Representing terms

In my formalization, SGV terms are represented as a strongly-typed Lean type using de Bruijn indices for variable binding:

```
inductive term: Π {γ}, context γ → tp → Type
```

Here I will go over some of its constructors in detail. First, variables. Let's start with a correct, but unwieldy definition and find how to improve it:

```
| Var:
  Π {γ} {A: tp}
  (x: γ ∋ A),
  ----------------------
  term (identity γ A x) A
```

The first explicit argument is `x`, a de Bruijn index which picks out a binding of type `A` out of the precontext `γ`. Thanks to its strong typing, it also certifies that such a binding exists - otherwise a value of type `γ ∋ A` would be impossible to construct.

Given a valid `x`, we can construct a term that under γ-context `(identity γ A x)` has type A. Recall from Chapter 3 that `(identity γ)` is the identity matrix. Selecting the `x`-th row out of this matrix returns a γ-context full of zeroes and a single 1 in the `x`-th position. This matches the T-Var rule, which similarly requires all-zeroes around a single instance of $x : A$.

So what's wrong with this definition? In principle nothing, as it is a valid encoding of T-Var. The issue, however, is that an application of the `identity` function appears in the return type. More often than not, when Lean expects a `(term <complicated_expression> A)` and is given a `(term <other_expression> A)`, it will not be able to automatically prove to itself that the expressions are equal and will fail to typecheck. I will defer a discussion of why this is so to Section 6.1.1, but suffice it to say for now that this is one of **the** main issues with using strongly-typed terms and dependent types in CiC more generally.

Since `identity γ A x` falls under the category of "complicated expressions", we may redefine `Var` as follows:

```
| Var:
  Π {γ} {A: tp}
  (Γ: context γ)
  (x: γ ∋ A)
  (_: auto_param (Γ = identity γ A x) ``solve_context),
  ------------------------------------------------------
  term Γ A
```

In this version, the first explicit argument `Γ` is the γ-context **that Lean expects the term which we are constructing to have**. This usually depends on the surrounding context. Because we must still need to meet the constraints of T-Var, the constructor argument expects a proof that `Γ = identity γ A x`. Notably, this is *not* the same as inlining `identity` into the return type - the proof is a propositional equality, while in the latter case Lean has to establish a definitional equality. If propositional equality was not required explicitly, in most cases the constructor would have to be wrapped in a conversion which establishes that same fact - threading it through an argument makes for a simpler usage. Detailed explanation of the problem is again deferred to Section 6.1.1.

The last argument is an `auto_param` wrapper around the required proof. An argument of type `auto_param P ``tac` attempts to use the tactic *tac* to prove the proposition $P$. The process is automatic and the argument acts as if it were implicit, with the goal of making it a little easier to use the constructor - its user usually does not have to give a proof, since `auto_param` can find one automatically. The `solve_context` tactic more or less tries to apply various previously proven lemmas using Lean's simplifier in order to prove the equality.

A note about `Γ` being an explicit argument - while the expected γ-context `Γ` can sometimes be inferred, in my experience inference usually fails or the inferred γ-context is not the right one, so I elected to make the parameter explicit.

Other term constructors work in a similar way - they always require parameters of the right types for the resulting term to be well-typed and whenever a non-inductive expression appears in a return type, it is moved into an `auto_param` argument:

```
| Abs:
  Π {γ} {Γ: context γ} {A B: tp},
  term (⟦1·A⟧::Γ) B
  ------------------
→ term Γ (A⊸B)

| App:
  Π {γ} {Γ₁ Γ₂: context γ} {A B: tp}
  (Γ: context γ)
  (M: term Γ₁ (A⊸B))
  (N: term Γ₂ A)
  (_: auto_param (Γ = Γ₁ + Γ₂) ``solve_context),
  ------------------------------------------------
  term Γ B

| Unit:
  Π {γ}
  (Γ: context γ)
  (_: auto_param (Γ = 0) ``solve_context),
  ----------------------------------------
  term Γ tp.unit

/- Other constructors elided. -/
```

4.4.1 Renaming and substitution

An important property we would like to have in any language is that renaming and substitution are admissible. The former means that it should be possible to change names of all bound variables in a term and its typing environment and end up with the same, well-typed term. In the case of de Bruijn indices, this simply involves adding a constant number n to all indices, which in Lean is representable as applying the `SVar` constructor (aka the successor function) n times. This also corresponds to multiplication by a carefully constructed matrix:

Lemma 1 (Renaming) *If* $\Gamma \vdash M : A$*, then for all* $(\gamma \times \delta)$ *matrices* Ξ *which consist of a* $\gamma \times \gamma$ *identity matrix and* $(\delta - \gamma)$ *additional columns of zeroes,* $\Gamma \circledast \Xi \vdash$

$M : A$.

In Lean:

```
/- The type of renaming functions which transport indices
   from one precontext to another. -/
def ren_fn (γ δ) := Π (A: tp), γ ∋ A → δ ∋ A

/- Applies the renaming ρ to a term M. The resulting term
   is the same one and has the same type, but its context
   is Γ times an identity matrix shifted by ρ. This has
   the effect of simply appending one or more zero-resourced
   bindings to Γ, and since we treat those as non-existent,
   the context is essentially the same. -/
def rename: Π {γ δ: precontext} {Γ: context γ} {A: tp}
  (ρ: ren_fn γ δ)
  (Δ: context δ)
  (M: term Γ A)
  (h: auto_param
    (Δ = Γ ⊛ (λ B x, matrix.identity δ B $ ρ B x))
    ``solve_context),
  term Δ A
| _ _ _ _ ρ _ (Var Γ x _) h  := Var _ (ρ _ x)
| _ _ _ _ ρ _ (Abs e) h := Abs (rename (ρ.ext _) _ e)
/- etc. -/
```

Given renaming, we can define *simultaneous substitution*, that is the action of substituting a term for every free variable in another term.

As a reminder, the resource requirements of such an operation can be expressed as a matrix Ξ where every row is a δ-context and the post-substitution term is well-typed under the δ-context $\Gamma \circledast \Xi$.

Lemma 2 (Simultaneous substitution) *If $\Gamma \vdash M : A$ and for all i, $\Delta_i \vdash N_i : B_i$, then $\Gamma \circledast \Xi \vdash M\{N_1/x_1, \ldots, N_n/x_n\} : A$ where $\Xi_i = \Delta_i$ is the i-th row of Ξ.*

In Lean:

```
/- For every variable x in γ, (σ _ x) (where σ: sub_fn Ξ) returns
   the term which should be substituted into that variable. The
   resource requirements of that term can be extracted out of
   the matrix Ξ. -/
def sub_fn {γ δ} (Ξ: matrix γ δ) :=
  Π (A: tp) (x: γ ∋ A), term (Ξ A x) A

/- Applies simultaneous substitution, replacing all variables
   in a term with whatever the matrix Ξ contains. -/
def subst
  : Π {γ δ} {Γ: context γ} {Ξ: matrix γ δ} {A}
    (σ: sub_fn Ξ)
    (Δ: context δ)
    (M: term Γ A)
    (h: auto_param (Δ = Γ ⊛ Ξ) ``solve_context),
    term Δ A
| _ _ _ _ _ σ _ (Var _ x _) _ := cast (by solve_context) $ σ _ x
| _ _ _ _ _ σ _ (Abs M) _ := Abs $ subst (σ.ext _) _ M
/- etc -/
```

Finally, we can define substitution of a single variable for convenience. In formal syntax, this is denoted as $M\{N/x\}$ and read as "M with N substituted for x".

Lemma 3 (Substitution) *Suppose* $\Gamma, x \overset{\pi}{:} B \vdash M : A$ *and* $\Gamma' \vdash N : B$*, where* Γ*,* Γ' *are both* γ*-contexts. Then* $\Gamma + \pi\Gamma' \vdash M\{N/x\} : A$*.*

In Lean:

```
def ssubst {γ} {Γ Γ': context γ} {A B: tp} {π: mult}
  (Δ: context γ)
  (e: term (⟦π·B⟧::Γ) A) -- The term being sub'd into requires π Bs
  (s: term Γ' B) -- The sub'd term requires Γ' resources
  (_: auto_param (Δ = Γ + π•Γ') ``solve_context)
  : term Δ A :=
subst /- sub_fn constructed from s -/ _ e
```

4.4.2 Term reduction

The final piece of SGV term semantics is to define how they undergo computation, i.e. how they can be reduced by taking computational steps. Because SGV supports communication between processes, there are two ways in which a term can get into an irreducible form, i.e. for term reduction to halt. Firstly, it can become a **value**, which can be thought of as a result of some computation. Values are themselves irreducible, but can still participate in reduction in conjunction with other terms. The BNF for values is:

$$\text{Values } U, V, W ::= x \mid \lambda x.M \mid () \mid (V, W) \mid \mathbf{inl}\, V \mid \mathbf{inr}\, V$$

In Lean:

```
inductive value: ∀ {γ} {Γ: context γ} {A: tp}, term Γ A → Prop
| VVar: ∀ {γ} {A: tp}
  (Γ': context γ)
  (x: γ ∋ A)
  (_: auto_param (Γ' = identity γ A x) ``solve_context),
  ------------------------------------------------------
  value (Var Γ' x)

| VAbs:
  ∀ {γ} {Γ: context γ} {A B: tp}
  (M: term (⟦1·A⟧::Γ) B),
  ----------------------
  value (Abs M)

| VUnit:
  ∀ {γ}
  (Γ: context γ)
  (_: auto_param (Γ = 0) ``solve_context),
  ---------------------------------------
  value (Unit Γ)

/- etc. -/
```

Secondly, reduction can block on a communication operation which needs to be carried out before term reduction can proceed. To specify this and reduction in general, we will first need to formalize evaluation contexts.

4.4.2.1 Evaluation contexts

An **evaluation context** is a term with a *hole*, where another term of an appropriate type can be plugged in. A hole is essentially like a free variable, and in fact evaluation contexts can be treated like abstractions, but I distinguish them for clarity.[3] The evaluation contexts in SGV are defined as follows:

$$\begin{aligned}\text{Evaluation Contexts } E ::= {} & [\,] \mid E\,M \mid V\,E \\ \mid {} & \textbf{let}\,() = E\,\textbf{in}\,M \mid \textbf{let}\,(x,y) = E\,\textbf{in}\,M \mid (E,V) \mid (V,E) \\ \mid {} & \textbf{inl}\,E \mid \textbf{inr}\,E \mid \textbf{case}\,E\,\textbf{of}\,\{\textbf{inl}\,x \mapsto M; \textbf{inr}\,y \mapsto N\} \\ \mid {} & \textbf{fork}\,E \mid \textbf{send}\,E\,M \mid \textbf{send}\,V\,E \mid \textbf{receive}\,E \mid \textbf{wait}\,E\end{aligned}$$

The basic context is the identity containing just a hole $[\,]$, with other contexts defined inductively on top of that. The point of using evaluation contexts is to express many reduction rules curtly and avoiding combinatorial explosion. The E-Lift reduction rule in Section 4.4.2.2 serves to do just that - a context $E[M]$, where $[M]$ denotes that the term M was substituted for the hole, reduces to $E[N]$ if M reduces to N. Note that $M\,E$ is not a context, only $V\,E$ is - this implicitly defines the order of evaluation.

In Lean, the definition of evaluation contexts is based on a type `eval_ctx_fn` of functions which take in a term argument with which to replace the hole and return the whole context term. Then, `eval_ctx` certifies that a particular function is, in fact, a valid evaluation context:

```
/- The context except the hole consumes Γₑ resources, while the
   hole can consume arbitrary resources Γ. The term plugged in
   for the hole must have type A' (hence the hole is "typed") and
   the resulting term has type A. In every evaluation context,
   the hole has the same precontext as the overall expression,
   since GV never evaluates under binders. Therefore I don't have
   to allow a fully general (i.e. arbitrary precontext) typing
   context for the hole. -/
@[reducible]
def eval_ctx_fn {γ} (Γₑ: context γ) (A' A: tp): Type :=
  Π (Γ: context γ), term Γ A' → term (Γ + Γₑ) A

/- A value of type (eval_ctx f) specifies that f is a valid function
   for an evaluation context. It's a Type rather than a Prop, because
   Prop can only eliminate into Prop but sometimes I need to extract
   the actual value of a constructor argument out of an eval_ctx. -/
inductive eval_ctx
  : Π {γ} {A' A: tp} (Γₑ: context γ), eval_ctx_fn Γₑ A' A → Type
| EHole:
  Π (γ: precontext) (A: tp),
  eval_ctx 0 (λ (Γ: context γ) (M: term Γ A),
    begin convert M, solve_context end)
```

[3] It could be an interesting exploration to see if one can specify evaluation contexts as simply `Abs term`s.

```
| EAppLeft:
  Π {γ} {Γ₁ Γ₂: context γ} {A B: tp}
    {C'}
  (Γₑ: context γ)
  (hΓ: Γₑ = Γ₁ + Γ₂)
  (N: term Γ₂ A)
  (E: eval_ctx_fn Γ₁ C' $ A⊸B),
  eval_ctx Γ₁ E
  --------------------------------
→ eval_ctx Γₑ (λ Γ M, App _ (E Γ M) N)

| EAppRight:
  Π {γ} {Γ₁ Γ₂: context γ} {A B: tp}
    {A'} {V: term Γ₁ $ A⊸B}
  (Γₑ: context γ)
  (hΓ: Γₑ = Γ₁ + Γ₂)
  (hV: value V)
  (E: eval_ctx_fn Γ₂ A' A),
  eval_ctx Γ₂ E
  --------------------------------
→ eval_ctx Γₑ (λ Γ M, App _ V $ E Γ M)

/- Other constructors. -/

/- The function and its certificate can be packed into
   a single structure. -/
structure eval_ctx' {γ} (Γₑ: context γ) (A' A: tp) :=
(f: eval_ctx_fn Γₑ A' A)
(h: eval_ctx Γₑ f)

/- Similarly to term renaming, evaluation contexts support
   being extended to a larger typing context. -/
def ext {γ δ: precontext} {Γ: context γ} {A' A: tp}
  (ρ: ren_fn γ δ) (E: eval_ctx' Γ A' A)
  : eval_ctx' (Γ ⊛ (λ B x, identity δ B $ ρ B x)) A' A :=
  /- implementation -/
```

4.4.2.2 Semantics and type preservation

Now that everything necessary has been defined, we can specify how terms undergo reduction:

Term Reduction $\boxed{M \longrightarrow_{\mathsf{M}} N}$

$$
\begin{array}{lr}
\text{E-Lam} & (\lambda x.M)\,V \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\text{E-Unit} & \mathbf{let}\,() = ()\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M \\
\text{E-Pair} & \mathbf{let}\,(x,y) = (V,W)\,\mathbf{in}\,M \longrightarrow_{\mathsf{M}} M\{V/x,W/y\} \\
\text{E-Inl} & \mathbf{case}\,\mathbf{inl}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} M\{V/x\} \\
\text{E-Inr} & \mathbf{case}\,\mathbf{inr}\,V\,\mathbf{of}\,\{\mathbf{inl}\,x \mapsto M; \mathbf{inr}\,y \mapsto N\} \longrightarrow_{\mathsf{M}} N\{V/y\} \\
\text{E-Lift} & E[M] \longrightarrow_{\mathsf{M}} E[N], \quad \text{if } M \longrightarrow_{\mathsf{M}} N
\end{array}
$$

In Lean, $M \longrightarrow_{\mathsf{M}} N$ is formalized as `term_reduces M N` or `M ⟶M N`:

```
inductive term_reduces
  : ∀ {γ} {Γ: context γ} {A: tp}, term Γ A → term Γ A → Prop
infix ` ⟶M `:55 := term_reduces
/- constructors shown in snippets below -/
```

An extremely useful property of this definition is that both the initial and reduced terms have type `term Γ A`, as opposed to allowing something like `term_reduces: ∀ .., term Γ A → term Γ' A' → Prop`. This means that the typing relation is *preserved* along the reduction relation - the only way we can express reduction from M to N is if they both belong to the same set of terms which have type A under γ-context Γ. Intuitively, it means that the type of a term never changes when it undergoes reduction.

Lemma 4 (Preservation (SGV Terms)) *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_M N$, *then* $\Gamma \vdash N : A$.

Thanks to the use of strongly-typed terms, preservation is established simply by defining reduction - it is implicit in the `term_reduces` relation. If a weakly-typed formalism was used, reduction would likely be defined as `term_reduces: term → term → Prop`. Then, I would have had to prove a separate preservation lemma such as:

```
lemma preservation:
  ∀ {γ} {Γ: context γ} {A: tp}
    {M N: term},
  typeof Γ M A
→ term_reduces M N
  ----------------
→ typeof Γ N A
```

This conciseness, together with by-design correctness, is an argument for strong typing. However, in Section 4.4.3 strong typing will prove rather troublesome.

The E-Lift reduction rule is first. Like its paper counterpart, it can wrap arbitrary reduction statements in an evaluation context, generating a new statement:

```
| EvalLift:
  ∀ {γ} {Γ Γₑ: context γ} {A A': tp}
    {M M': term Γ A}
  (Γ': context γ)
  (E: eval_ctx' Γₑ A A')
  (EM EN: term Γ' A')
  (hΓ': Γ' = Γ+Γₑ)
  (hStep: M ⟶M N)
  (hEM: EM = E.f.apply Γ' M)
  (hEN: EN = E.f.apply Γ' N),
  --------------------------
  EM ⟶M EN
```

The constructor for `EvalLift` unfortunately contains a lot of irrelevant noise. This, again, is due to problems with showing equality between dependent types. I settled on the form above after a lot of experimentation - while unwieldy and difficult to read, it is still easier to work with than certain other possible representations.

Other constructors follow the rest of the rules:

```
| EvalLam:
  ∀ {γ} {Γ₁ Γ₂: context γ} {A B: tp}
    {V: term Γ₂ A}
  (Γ: context γ)
  (M: term (⟦1·A⟧::Γ₁) B)
  (hV: value V)
  (_: auto_param (Γ = Γ₁ + Γ₂) ``solve_context),
  ------------------------------------------
  (App Γ (Abs M) V) ⟶M ssubst _ M V

| EvalUnit:
  ∀ {γ} {Γ: context γ} {A: tp}
  (M: term Γ A),
  ------------------------------------
  (LetUnit Γ (Unit 0) M $ by simp) ⟶M M

/- etc. -/
```

4.4.3 Progress and type soundness

The final theorem we need to prove in order to establish type soundness is **progress**. Informally, it means that either reduction on a term has finished or the program knows what the next step to take is. Formally:[4]

Lemma 5 (Progress (SGV terms)) *If* $0\Gamma \vdash M : A$*, then either:*

1. M *is a value.*
2. *There exists some* N *such that* $M \longrightarrow_M N$*.*
3. *There exist some* E *and* N *such that* M *may be written* $E[N]$*, where* N *is a communication and concurrency construct, i.e.,* ***fork*** V*,* ***send*** $V\,W$*,* ***receive*** V*, or* ***wait*** V*.*

This can be encoded in Lean as an inductive proposition with one constructor for each of the cases:

```
inductive progress {γ} {Γ: context γ} {A: tp} (M: term Γ A): Prop
| Done:
  value M
→ progress

| Step:
  ∀ (M': term Γ A),
  M ⟶M M'
→ progress
```

[4]The definition here is slightly weaker than that stated by Fowler [19]. Only completely empty contexts are allowed, while Fowler also allows contexts containing nothing but channel names. Because expressing such an invariant in a theorem prover when a single context is used both for variables and channel names is troublesome, I elected not to do it in interest of time, but it does mean that not all cases have been handled (even though the omitted cases are simple). Splitting the context into two, one for variables and one for channels, could make this more tractable.

```
| Conc:
  comms_blocked M
→ progress
```

Where `comms_blocked M` is a predicate certifying that M can be expressed as $E[N]$ for some evaluation context E and communication construct N:

```
inductive comms_blocked {γ} {Γ: context γ} {A: tp} (M: term Γ A): Prop
| mk: ∀ {A': tp} {Γ' Γₑ: context γ} {N: term Γ' A'}
  (hN: comms_construct N)
  (E: eval_ctx' Γₑ A' A)
  (hΓ: Γ = Γ'+Γₑ)
  (h: M = E.f.apply Γ N),
  comms_blocked

inductive comms_construct: ∀ {γ} {Γ: context γ} {A: tp}, term Γ A → Prop
| CCFork: /- ... -/ comms_construct (Fork V)
| CCSend: /- ... -/ comms_construct (Send Γ V W)
| CCRecv: /- ... -/ comms_construct (Recv V)
| CCWait: /- ... -/ comms_construct (Wait V)
```

A proof of progress then follows by defining a function which can return a value of type `progress M` for all M:

```
def mk_progress'
  : Π {γ: precontext} {A: tp}
  (Γ: context γ) (M: term Γ A)
  (hΓ: auto_param (Γ = 0) ``solve_context),
  progress M
```

The proof[5] itself is over 600 lines long and proceeds by cases on term constructors, as well as recursion - if M can step then $\textbf{let}\,() = M\,\textbf{in}\,N$ can step via E-Lift, etc. The majority of operations involve casting terms back and forth in order to convince the typechecker that the expressions are type-correct. This is, in my opinion, a fairly strong argument *against* the use of strongly-typed terms - in a weakly-typed formalism, invariants that have been established in separate proofs do not pollute proofs of other properties and can be used if needed, but do not need to be enforced at every step. This makes one have to consider more cases at times when a strongly-typed formulation would have excluded them via the invariant, but these cases are usually more straightforward than the complicated operations needed to show type-correctness.

This concludes the formalisation of the sequential subset of SGV and establishes the type soundness of its terms.

[5] Available in `progress.lean` in the associated source repository.

Chapter 5

SGV processes

Having established the sequential semantics of SGV terms, we can now proceed with defining the behaviour of processes running in parallel and communicating with each other.

5.1 Threads

A **thread** is an object that only exists at runtime, rather than when the program is being written. Its purpose is to wrap a term, specifying whether the term is running as the unique **main** ($\bullet$) thread or one of zero or more **child** ($\circ$) threads. Thread contexts are an extension of evaluation contexts that also include the thread **flag** (main or child) - in the rules below, E is the evaluation context from Section 4.4.2.1. Addition is defined on thread flags - it will be given an interpretation shortly in terms of typing rules for configurations. Formally:

$$\begin{array}{ll} \text{Thread Flags} & \phi ::= \bullet \mid \circ \\ \text{Thread Contexts} & \mathcal{F} ::= \phi E \end{array}$$

Combination of Flags $\boxed{\phi_1 + \phi_2}$

$\bullet + \circ = \bullet$ $\quad \circ + \bullet = \bullet$ $\quad \circ + \circ = \circ$ $\quad \bullet + \bullet$ undefined

Here, E is the evaluation context from Section 4.4.2.1. In Lean:

```
inductive thread_flag: Type
| Main: thread_flag
| Child: thread_flag

namespace thread_flag

/- Instead of as a function, addition is represented as
   an inductive relation on flags. This makes it easier
   to work with in the type theory. -/
inductive add: thread_flag → thread_flag → thread_flag → Prop
| CC: add Child Child Child
```

```
| CM: add Child Main Main
| MC: add Main Child Main

/- Establish some simple lemmas about addition. -/
lemma child_add: ∀ {Φ}, add Child Φ Φ
| Child := add.CC
| Main := add.CM

lemma add_child: ∀ {Φ}, add Φ Child Φ
| Child := add.CC
| Main := add.MC

end thread_flag

/- A tactic that can solve most goals involving thread flags. -/
meta def solve_flag: tactic unit :=
  `[ exact add.CC <|> exact add.CM <|> exact add.MC
     <|> assumption <|> exact child_add <|> exact add_child
     <|> tactic.fail "Failed to solve flags goal." ]
```

And thread contexts are simple wrappers around `eval_ctx`:

```
@[reducible]
def thread_ctx_fn {γ} (Γₑ: context γ) (A': tp) (Φ: thread_flag) :=
  Π (Γ: context γ), term Γ A' → config (Γ + Γₑ) Φ

inductive thread_ctx
  : Π {γ} {A': tp} {Φ: thread_flag} (Γₑ: context γ),
    thread_ctx_fn Γₑ A' Φ → Type
| FMain:
  ∀ {γ} {Γₑ: context γ} {A' A: tp}
  (E: eval_ctx' Γₑ A' A),
  ---------------------------------
  thread_ctx Γₑ (λ Γ M, •(E.f Γ M))

| FChild:
  ∀ {γ} {Γₑ: context γ} {A': tp}
  (E: eval_ctx' Γₑ A' End!),
  ---------------------------------
  thread_ctx Γₑ (λ Γ M, ○(E.f Γ M))
```

5.2 Configurations

The core parallel behaviour of SGV is defined using **configurations**. Configurations are runtime wrappers around terms, which specify how the terms are composed together into a system with intercommunicating parts. Threads from the previous section are the base variant of a configuration.

The second type of configuration, $(\nu a)\mathcal{C}$ is a sort of λ-abstraction, except it binds a **channel name** a into the typing context. A channel name always has type $S^\sharp$ for some session type S - the channel type described both dual endpoints. These can be distinguished from standard variables if one wishes, but in my formalisation I *pun* them with variable names into a single syntactic object.[1] The third type is

[1]This has consequences for the formalisation of progress in Section 4.4.3. The notion of

a parallel composition of two configurations, $C \parallel \mathcal{D}$, which can be read as "$\mathcal{C}$ is running in parallel with $\mathcal{D}$" (or vice-versa, because this relation is commutative). The SGV formalism by Fowler [19] also includes configuration contexts, but I elected to encode configuration reduction directly, as there are not many rules for it and therefore little benefit to using contexts. Formally:

$$\text{Configurations } \mathcal{C}, \mathcal{D}, \mathcal{E} ::= (\nu a)\mathcal{C} \mid \mathcal{C} \parallel \mathcal{D} \mid \phi M$$

Configurations also include a notion of equivalence, which I did not have enough time to formalize in Lean:

Configuration Equivalence $\boxed{\mathcal{C} \equiv \mathcal{D}}$

$$\mathcal{C} \parallel (\mathcal{D} \parallel \mathcal{E}) \equiv (\mathcal{C} \parallel \mathcal{D}) \parallel \mathcal{E} \qquad \mathcal{C} \parallel \mathcal{D} \equiv \mathcal{D} \parallel \mathcal{C} \qquad (\nu a)(\nu b)\mathcal{C} \equiv (\nu b)(\nu a)\mathcal{C}$$

$$\mathcal{C} \parallel (\nu a)\mathcal{D} \equiv (\nu a)(\mathcal{C} \parallel \mathcal{D}), \quad \text{if } a \notin \mathsf{fn}(\mathcal{C})$$

5.2.1 Configuration typing

While configurations do not have types per se, they do have rules for well-formedness. These are given by the relation $\Gamma \vdash^{\phi} \mathcal{C}$, which can be read as "the configuration $\mathcal{C}$ is well-formed under Γ and its maximal thread flag is ϕ". The maximal thread flag ϕ is determined by addition of flags of a configuration's components and is $\bullet$ if the components include the main thread, $\circ$ otherwise. Formally:

Well-Formedness of Configurations $\boxed{\Gamma \vdash^{\phi} \mathcal{C}}$

T-Nu
$$\frac{\Gamma, a : S^{\sharp} \vdash^{\phi} \mathcal{C}}{\Gamma \vdash^{\phi} (\nu a)\mathcal{C}}$$

T-Connect
$$\frac{\Gamma_1, a : S \vdash^{\phi_1} \mathcal{C} \quad \Gamma_2, a : \overline{S} \vdash^{\phi_2} \mathcal{D}}{\Gamma_1, \Gamma_2, a : S^{\sharp} \vdash^{\phi_1 + \phi_2} \mathcal{C} \parallel \mathcal{D}}$$

T-Child
$$\frac{\Gamma \vdash M : \mathsf{End}_!}{\Gamma \vdash^{\circ} \circ M}$$

T-Main
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash^{\bullet} \bullet M}$$

An important departure from previous formulations of SGV in the above rules is that here there is only one rule for parallel composition T-Connect, while Fowler [19] uses two. The reason for having two is that the rule is asymmetric - $\mathcal{C}$ gets S, while $\mathcal{D}$ gets $\overline{S}$ as the channel endpoint to use, but they might in fact each need the opposite. For example, $\mathcal{C}$ might need the client endpoint and $\mathcal{D}$ the server, but due to T-Connect the server can only go to $\mathcal{C}$ and the client only to $\mathcal{D}$. Even though configuration equivalence allows for commutativity, it can be shown that particular configurations are untypable after applying equivalence rules. A mirror rule which simply has the dual type on the opposite side solves

progress should work under any context containing only channel names, but for simplicity my definition for now works only on empty contexts.

this issue. I wondered, however, if this could be amended by instead saying that channel types $S^\sharp$ are equivalent to their duals, i.e. $S^\sharp \equiv_T \overline{S}^\sharp$ where $\equiv_T$ is a new equivalence relation on types. Unfortunately I did not have enough time to investigate this further.

In Lean, similarly to terms I use a strongly-typed object for configurations which enforces their well-formedness at construction time. $\Gamma \vdash^\phi \mathcal{C}$ is written as `C: config Γ Φ`:

```
inductive config: Π {γ}, context γ → thread_flag → Type
| CNu:
  Π {γ} {Γ: context γ}
    {Φ: thread_flag} {S: sesh_tp},
  /- Channel names, being treated the same as standard variables,
     are added to the usual typing context. -/
  config (⟦1·S♯⟧::Γ) Φ
  --------------------
→ config Γ Φ

| CComp:
  Π {γ} {Γ₁ Γ₂: context γ}
    {Φ₁ Φ₂: thread_flag} {S: sesh_tp}
  (Γ: context (S♯::γ))
  (Φ: thread_flag)
  (C: config (⟦1·S⟧::Γ₁) Φ₁)
  (D: config (⟦1·(sesh_tp.dual S)⟧::Γ₂) Φ₂)
  /- The same auto_param technique is used for the resulting
     context and thread flag. -/
  (_: auto_param (Γ = (⟦1·S♯⟧::(Γ₁ + Γ₂))) ``solve_context)
  (_: auto_param (add Φ₁ Φ₂ Φ) ``solve_flag),
  ----------
  config Γ Φ

| CMain:
  Π {γ} {Γ: context γ} {A: tp},
  term Γ A
  -------------
→ config Γ Main

| CChild:
  Π {γ} {Γ: context γ},
  term Γ End!
  --------------
→ config Γ Child

notation `●`C:90 := CMain C
notation `○`C:90 := CChild C
```

5.2.2 Configuration reduction

The final piece of SGV semantics is to specify how processes communicate. This is defined via the reduction of configurations containing **communication constructs**:

Configuration Reduction $\boxed{\mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D}}$

$$
\begin{array}{lrl}
\text{E-Fork} & \mathcal{F}[\textsf{fork}\,\lambda x.M] \longrightarrow_{\mathsf{C}} (\nu a)(\mathcal{F}[a] \parallel \circ M\{a/x\}) & (a \text{ is fresh}) \\
\text{E-Comm} & \mathcal{F}[\textsf{send}\,V\,a] \parallel \mathcal{F}'[\textsf{receive}\,a] \longrightarrow_{\mathsf{C}} \mathcal{F}[a] \parallel \mathcal{F}'[(V,a)] & \\
\text{E-Wait} & (\nu a)(\mathcal{F}[\textsf{wait}\,a] \parallel \circ a) \longrightarrow_{\mathsf{C}} \mathcal{F}[()] & \\
\text{E-Main} & \bullet M \longrightarrow_{\mathsf{C}} \bullet N \quad (\text{if } M \longrightarrow_{\mathsf{M}} N) & \\
\text{E-Child} & \circ M \longrightarrow_{\mathsf{C}} \circ N \quad (\text{if } M \longrightarrow_{\mathsf{M}} N) & \\
\text{E-Nu} & (\nu a)\mathcal{C} \longrightarrow_{\mathsf{C}} (\nu a)\mathcal{D} \quad (\text{if } \mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D}) & \\
\text{E-Comp} & \mathcal{C} \parallel \mathcal{E} \longrightarrow_{\mathsf{C}} \mathcal{D} \parallel \mathcal{E} \quad (\text{if } \mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D}) &
\end{array}
$$

In E-Fork, **fork** M spawns a new child thread running the term M as well as creates a new channel a under a ν-binder to use for communicating with the child. In E-Comm, **send** V A and **receive** a, which have dual session types, pass the value V from the sending process to the receiving one. Finally, in E-Wait **wait** a waits on a child thread to finish operation and kills it. Other reduction rules *lift* the reduction of subexpressions. If a term can reduce, then so can the thread running it (E-Main, E-Child) and if a subconfiguration can reduce, then so can the configuration containing it (E-Nu, E-Comp).

When considered from a typing perspective, the above rules seem to violate preservation. For example, in E-Comm, the type of the channel name a must be different before and after reduction - recall that holes have a unique type, so the type of **send** V a before reduction must equal that of a after to "fit" in $\mathcal{F}$. Perhaps counterintuitively, this makes sense - session types follow the pattern of e.g. $!A.S$ for "send A and continue with S", so after **send** completes, the channel type should be S, not $!A.S$. This is achieved via the reduction of session types and typing contexts:

Reduction on Session Types and Typing Contexts $\boxed{S \longrightarrow_S S'}$ $\boxed{\Gamma \longrightarrow_\Gamma \Gamma'}$

$$
\frac{}{!A.S \longrightarrow_S S} \qquad \frac{}{?A.S \longrightarrow_S S} \qquad \frac{S \longrightarrow_S S'}{\Gamma, a \overset{\pi}{:} S^\sharp \longrightarrow_\Gamma \Gamma, a \overset{\pi}{:} S'^\sharp}
$$

In Lean, I merged the two into a single definition. $\Gamma \longrightarrow_\Gamma \Gamma'$ is denoted as `context_reduces Γ Γ'`:

```
inductive context_reduces: ∀ {γ γ'}, context γ → context γ' → Prop
/- This reduction is not in the written rules
   but it makes the formalization simpler. -/
| ΓId:
  ∀ {γ} {Γ: context γ},
  context_reduces Γ Γ

| ΓSend:
    ∀ {γ} {Γ: context γ} {π: mult}
```

```
      {A: tp} {S: sesh_tp},
    context_reduces (⟦π·(!A·S)♯⟧::Γ) (⟦π·S♯⟧::Γ)

| ΓRecv:
    ∀ {γ} {Γ: context γ} {π: mult}
      {A: tp} {S: sesh_tp},
    context_reduces (⟦π·(?A·S)♯⟧::Γ) (⟦π·S♯⟧::Γ)
```

5.2.3 Preservation

The above ingredients together allow finally stating a strongly-typed definition of configuration reduction. Similarly to the case of terms, preservation is proven by construction of the reduction relation, but in this case it only holds modulo reduction of typing contexts:

Lemma 6 (Preservation (SGV Configurations)) *If* $\Gamma \vdash^{\phi} \mathcal{C}$ *and* $\mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D}$, *then there exists some* $\Gamma \longrightarrow_{\Gamma} \Gamma'$ *such that* $\Gamma' \vdash^{\phi} \mathcal{D}$.

$\mathcal{C} \longrightarrow_{\mathsf{C}} \mathcal{D}$ is denoted by `config_reduces Γ_reduces C D` (or `C -Γ_reduces⟶D`), where `Γ_reduces` is an element of $\Gamma \longrightarrow_{\Gamma} \Gamma'$ specifying how $\mathcal{C}$'s typing context reduces to $\mathcal{D}$'s typing context. I sincerely apologize for having unleashed the following monstrosity upon the world:

```
inductive config_reduces
  : ∀ {γ γ'} {Γ: context γ} {Γ': context γ'} {Φ},
  context_reduces Γ Γ' → config Γ Φ → config Γ' Φ → Prop
notation C` -`h`⟶C `C':55 := config_reduces h C C'
| CEvalNu:
  ∀ {γ} {Γ: context γ} {S: sesh_tp} {Φ: thread_flag}
    {C C': config (⟦1·S♯⟧::Γ) Φ},
  C -ΓId⟶C C'
  ---------------------
→ ((CNu C) -ΓId⟶C (CNu C'))

| CEvalComp:
  ∀ {γ} {Γ₁ Γ₂: context γ} {S: sesh_tp} {Φ₁ Φ₂: thread_flag}
    {C C': config (⟦1·S⟧::Γ₁) Φ₁}
  (Γ: context $ S♯::γ)
  (hΓ: Γ = ⟦1·S♯⟧::(Γ₁ + Γ₂))
  (Φ: thread_flag)
  (hΦ: add Φ₁ Φ₂ Φ)
  (D: config (⟦1·sesh_tp.dual S⟧::Γ₂) Φ₂),
  C -ΓId⟶C C'
  ------------------------------------------------------------
→ ((CComp Γ Φ C D) -ΓId⟶C (CComp Γ Φ C' D))

| CEvalChild:
  ∀ {γ} {Γ: context γ}
    {M M': term Γ End!},
  M ⟶M M'
  -------------------------------
→ (○M -ΓId⟶C ○M)

| CEvalMain:
  ∀ {γ} {Γ: context γ} {A: tp}
    {M M': term Γ A},
```

```
  M ⟶M M'
  --------------------------------
→ (•M -ΓId⟶C •M')

| CEvalFork:
  ∀ {γ} {Γₑ: context γ} {S: sesh_tp} {Φ}
  (F: thread_ctx' Γₑ (sesh_tp.dual S) Φ)
  (M: term (⟦1·S⟧::(0: context γ)) End!),
  ---------------------------------------
  ((F.f.apply Γₑ $ Fork $ Abs M)
  -ΓId⟶C
  (CNu
    $ CComp (⟦1·S♯⟧::Γₑ) Φ
      (CChild M)
      $ (F.ext $ ren_fn.lift_once $ sesh_tp.dual S).f.apply
        (⟦1·sesh_tp.dual S⟧::Γₑ)
        (Var
          (⟦1·sesh_tp.dual S⟧::0)
          $ ZVar _ $ sesh_tp.dual S)
        $ by solve_context))

| CEvalComm:
  ∀ {γ} {Γv: context γ} {A: tp} {S: sesh_tp}
    {Φ₁ Φ₂: thread_flag}
  (V: term Γv A)
  (hV: value V)
  (Φ: thread_flag)
  (hΦ: add Φ₁ Φ₂ Φ)
  (F: thread_ctx' (0: context γ) S Φ₁)
  (F': thread_ctx' (0: context γ) (tp.prod A $ sesh_tp.dual S) Φ₂),
  ------------------------------------------------------------------
  ((CComp (⟦1·(!A·S)♯⟧::Γv) Φ
    ((F.ext $ ren_fn.lift_once $ !A·S).f.apply
      (⟦1·!A·S⟧::Γv)
      (Send
        (⟦1·(!A·S)⟧::Γv)
        (term.rename (ren_fn.lift_once $ !A·S) _ V)
        (Var
          (⟦1·(!A·S)⟧::0)
          $ ZVar γ $ !A·S)
        $ by solve_context)
      $ by solve_context)
    $ (F'.ext $ ren_fn.lift_once $ sesh_tp.dual $ !A·S).f.apply
      (⟦1·sesh_tp.dual (!A·S)⟧::0)
      (Recv $
        Var
          (⟦1·sesh_tp.dual (!A·S)⟧::0)
          (begin convert
            (ZVar γ $ sesh_tp.dual $ !A·S),
            rw [sesh_tp.dual],
          end)
        $ begin
          simp [*, identity] with unfold_,
          h_generalize Hx: (ZVar γ $ sesh_tp.dual $ !A·S) == x,
          apply eq_of_heq, lmao
        end))
  -ΓSend⟶C
  (CComp (⟦1·S♯⟧::Γv) Φ
```

```
    ((F.ext $ ren_fn.lift_once S).f.apply
      (⟦1·S⟧::0)
      (Var
        (⟦1·S⟧::0)
        (ZVar γ S))
      $ by solve_context)
    $ (F'.ext $ ren_fn.lift_once $ sesh_tp.dual S).f.apply
      (⟦1·sesh_tp.dual S⟧::Γv)
      $ Pair
        (⟦1·sesh_tp.dual S⟧::Γv)
        (term.rename (ren_fn.lift_once $ sesh_tp.dual S) _ V)
        (Var
          (⟦1·sesh_tp.dual S⟧::0)
          $ ZVar γ $ sesh_tp.dual S)
        $ by solve_context))

| CEvalWait:
  ∀ {γ} {Φ}
  (F: thread_ctx' (0: context γ) tp.unit Φ),
  ------------------------------------------
  ((CNu
    $ CComp (⟦1·End?♯⟧::0) Φ
      ((F.ext $ ren_fn.lift_once $ End?).f.apply
        (⟦1·End?⟧::0)
        (Wait
          $ Var
            (⟦1·End?⟧::0)
            $ ZVar γ End?)
        $ by solve_context)
      $ CChild
        $ Var (⟦1·End!⟧::0)
          (ZVar γ $ sesh_tp.dual End?)
          $ begin
            show ⟦1·End!⟧::0 = identity (End!::γ) End! (ZVar γ End!),
            simp with unfold_,
          end)
  -ΓId⟶C
  (F.f.apply (0: context γ) $ Unit 0))
```

Part of the reason for this definition's exponential increase in length compared to the informal, written rules is the much greater level of detail necessary for a formal description of any theory. The primary reason, however, is the need to cast between expressions depending on propositionally, but not definitionally equal typing contexts. Essentially every proof in the above within `begin .. end` blocks and `by ..` expressions does just that. This issue is discussed in the next chapter.

The above type-preserving definition of configuration reduction is the last thing I was able to state in the available timeframe and concludes the presented Lean formalization of SGV. Formalizing a soundness result for SGV *configurations* would require proving a number of stepping-stone lemmas about the behaviour of configurations under reduction modulo their equivalence relation, leading to the notions of *open* and *global* progress [19].

Chapter 6

Evaluation

The final chapter is a collection of observations about Lean, formalizing programming languages and theorem proving more generally.

6.1 Encoding invariants in types

Throughout the project, I extensively use Lean's dependent types to encode various invariants that objects of that type should uphold. Strongly-typed terms (Section 2.4.3) are an example of this technique - the target-language type of a `term` is part of the host-language `Type`. At various points I mentioned that this representation becomes more and more difficult to work with as the invariants become more complicated - here, I will explain why that is so and what exactly "complicated" means in this context.

6.1.1 The trouble with equality

Suppose that we would like to define a term of type `T a`, where `T: ℕ → Type`. However, the expression which we want to use for this purpose has type `T b`, where `a` is not *exactly* `b`, syntactically speaking - their form in source code is distinguishable. For example, `a = b+0`. Clearly in this case they *are* the same, but how can a computer know that? This question is surprisingly difficult to answer - Lean itself includes at least four different notions of equality.

Firstly, there is **definitional equality** $A \equiv B$. This notion is built into the CiC, Lean's core type theory - any two expressions which are definitionally equal are interchangeable within the kernel. Definitional equality considers operations such as β-reduction, extensionality and proof irrelevance (any two proofs of the same proposition are equal) in order to find its congruence classes. Unfortunately, because this notion is undecidable [7], what Lean actually computes is **algorithmic** definitional equality, denoted $A \Leftrightarrow B$. While it implies standard definitional equality, it is slightly weaker and hence can show fewer things equal.

On top of these, we have two notions of user-definable equality. Firstly, **propositional equality** `a = b`:

```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
infix ` = `:50 := eq
```

It has exactly one constructor, reflexivity - after all, the only way two things can be equal is if they are the same thing. Because `eq` is not a builtin of the type theory, it can be constructed via arbitrary reasoning, using sophisticated proofs. Thanks to this, far more things can be shown propositionally equal than definitional equality could ever support.

While more powerful, `eq` is still not the most general notion of equality - that title can be bestowed upon **heterogeneous equality** `a == b`:

```
inductive heq {α : Sort u} (a : α) : Π {β : Sort u}, β → Prop
| refl : heq a
infix ` == `:50 := heq
```

Its only constructor also is reflexivity, but it supports expressing statements of equality between elements of different types - for example `2 == "a string"`. While that could never be true, the key idea here is that it is *expressible* in the first place - such a statement could not even be written down with `eq`. Of course to be equal, two things must have the same type. Why, then, would we ever try to prove things of different types equal?

This brings us back to the initial example. The question of which types are equal is subtle - the Lean kernel only really accepts (⇔) as a proof of that. However, because of its limitations, not many things are definitionally equal. Consider the following example of a type `T` depending on a natural number:

```
inductive T: ℕ → Type
| mk: Π n, T n
```

We can return an expression of type `T (x+1)` when `T (nat.succ x)` is expected:

```
def foo_inductive (x): T (nat.succ x) :=
  T.mk (x+1) /- ok -/
```

Why is this so? Because `x+1` is *defined* to be `nat.succ x`. (⇔) respects β-reduction and can unfold the definiton of addition, arriving at the desired type. In general, if the value to be determined equal can be reduced to a constructor of some inductive type, then it will most likely be accepted. It is therefore good practice to make types depend *only* on such inductive expressions. Strongly-typed programming language developments such as that of Benton et al. [5] try to go in this direction.

Functional expressions, on the other hand, are ”complicated”, and will not pass typechecking if not (⇔)-equal to the expected value, which is rather often:

```
/-
type mismatch, term
  T.mk (x + y + 2)
has type
  T (x + y + 2)
but is expected to have type
  T (nat.succ x + nat.succ y)
```

```
-/
def foo_func (x y): T ((nat.succ x) + (nat.succ y)) :=
  T.mk (x+y+2) /- bad -/
```

To fix this, we need to explicitly show that the type is correct via the use of a `cast`, which can convert terms using a proof of propositional equality:

```
@[inline] def cast {α β : Sort u} (h : α = β) (a : α) : β :=
eq.rec a h

def foo_cast (x y): T ((nat.succ x) + (nat.succ y)) :=
  cast (begin congr' 1, show x+y+2 = (x+1)+(y+1), ring end)
    $ T.mk (x+y+2) /- ok -/
```

In particularly difficult cases, the proof of type equality using `eq` may *itself not be type-correct*. This is where heterogeneous equality becomes useful - after complicated transformations have been carried out, it can be transported back to an `eq` form using `eq_of_heq`:

```
lemma eq_of_heq {α : Sort u} {a a' : α} (h : a == a')
  : a = a' := /- proof using the axiom of proof irrelevance -/
```

6.1.2 Inserting casts

The need to insert `cast`s vastly complicates certain expression - in this project, `config_reduces` is just one example. While I can use `auto_param` parameters in carefully chosen positions to alleviate some of these issues, they are still quite painful. Other provers provide more mature solutions. For example, Coq's `Program` command [32] can automatically generate type equality goals. Coq will prove them using automation where possible and otherwise leave them to be proven outside the expression, allowing for a much cleaner specification with formal details resolved externally. To do this, `Program` utilizes predicate subtyping, present also in systems such as PVS [31].

6.2 Writing proofs in Lean

In this section I will walk through some of the lemmas omitted from previous sections, discussing generally how I went about proving them.

6.2.1 Tactics

Like Coq and Isabelle but unlike e.g. Agda, Lean definitions and proofs can be constructed in tactic mode. Briefly, this mode proceeds by applying a series of algorithmic transformations called *tactics* to a *context*. The context includes all available hypotheses and one or more *goals* - the types of expressions which need to be constructed. The initial goal is the type of what's being defined and the initial hypotheses are the definition's arguments. For example, the `lemma gt_2_of_gt_3 {n: ℕ} (h: n > 3): n > 2` would has following initial context:

```
1 goal
n : ℕ,
```

```
h : n > 3
⊢ n > 2
```

To support writing tactics, Lean features a sophisticated metaprogramming language [18]. A major difference from Coq and Isabelle is that in Lean, this language is not in any way distinguished from the language of definitions. Metaprogram definitions, denoted `meta def`, can access all the same constructs and techniques as regular `def` definitions, but they are not restricted by termination checking and hence can encode arbitrary control flow including infinite loops. For this reason, `meta def` definitions can use `def` objects, but not the other way around. Metaprograms are generally written in Haskell-style `do`-notation, with a `tactic` monad exposing the context so that tactics can modify it. An example from the Lean core library:

```
meta def constructor (cfg : apply_cfg := {})
  : tactic (list (name × expr)) :=
target' >>= get_constructors_for >>= try_constructors cfg
```

A simple way of proving facts about inductive types with finitely many elements is a proof by brute-force - enumerate all cases, and then if the goal is true, it will follow trivially via reflexivity ($a = a$). I use it when proving facts involving elements of the (finite) semiring $(0, 1, \omega)$, but that's about the only place in which it works:

```
lemma mul_comm (π₁ π₂: mult)
  : π₁ * π₂ = π₂ * π₁ :=
by { cases π₁; cases π₂; refl }
```

The `cases` tactic is one of the most basic and useful ones in general. It carries out a split over the possible constructors of a value and generates a subgoal for each constructor (case). Semicolon is an example of a tactic *combinator*, where `tac_a; tac_b` reads "run `tac_a` and then run `tac_b` on each subgoal produced by `tac_a`".

6.2.2 Using the simplifier for normalization

After `cases`, probably the most useful tools are the rewriter and simplifier.

Briefly, `rewrite [h] at e` uses the propositional equality `(h: a=b)` where `a`, `b` are arbitrary Lean terms to replace every occurrence of `a` in `e` with `b`. A curious implication of the fact that rewriting uses equations left-to-right is that they are, in this context, not symmetric.

The simplifier, accessible via `simp`, keeps a database of previously proven *simplification lemmas* and applies them iteratively as rewrites until no simplification lemma matches. Which lemmas are actually simplifications is decided by the programmer, who places a special attribute:

```
/- RHS clearly simpler than LHS. -/
@[simp] lemma add_zero
  : ∀ (x: ℕ), x + 0 = 0 := by ..

/- The RHS is not really simpler, just flipped.
```

```
    Hence, not a simp lemma. -/
lemma add_comm
  : ∀ (x y: ℕ), a + b = b + a := by ..
```

This presents an interesting dilemma - adding more lemmas to the simplification database can have unpredictable consequences, as well as considerably slow down the simplifier, but it might also make a lot of proofs trivial thanks to the `simp *` tactic, which tries every possible rewrite.

Throughout the formalization work, I've found how `simp` attributes can be leveraged to make proofs easier without tagging everything with `@[simp]`. Namely, `simp` admits custom attributes. These can be used to defined simplification classes - for example, lemmas tagged `@[sop_norm]` can be expected to bring expressions to a sum-of-products normal form:

```
/- Define the sop_form attribute. -/
run_cmd mk_simp_attr `sop_form [`simp]

...

/- Vector addition is distributive over scalar
   multiplication. Going left-to-right, this
   equality produces a sum-of-products form. -/
@[sop_form] lemma smul_add
  : ∀ {γ} {π: mult} {Γ₁ Γ₂: context γ},
    π • (Γ₁ + Γ₂) = π•Γ₁ + π•Γ₂ :=
begin
  intros,
  /- proof by induction -/
  induction Γ₁ with γ₁ π₁ T₁ Γ₁ ih₁,
  { cases Γ₂, refl },
  { cases Γ₂ with _ π₂,
    /- simplify using another attribute - "unfold_" -/
    simp [*, left_distrib] with unfold_ },
end

...

/- Scalar multiplication commutes with
   vector-matrix multiplication. -/
lemma smul_vmul
  : ∀ {γ δ} {Γ: context γ} {Ξ: matrix γ δ} {π: mult},
    (π • Γ) ⊛ Ξ = π • (Γ ⊛ Ξ) :=
begin
  intros,
  induction Γ with γ' π' T Γ ih;
  /- simplify using "sop_form" -/
  { simp [*, context.mul_smul]
    with unfold_ sop_form },
end
```

Lean also provides SMT-based tactics such as congruence closure and E-matching using the `[smt]` annotation, but I did not ever use them in this project.

6.2.3 Comparison with Agda

Since the definitions and lemmas of non-dependent QTT are translated from Agda, it bears mentioning how Lean compares with Agda. While Agda definitions are arguably more readable thanks to its focus on pattern-matching syntax, I found that proofs tend to be much shorter in Lean (as they would be in other tactic-based languages, e.g. Coq) thanks to the use of automation. Agda requires building equational proof terms explicitly, for example (Agda from [24]):

```
⊛-distribʳ-⋈ : ∀ {γ} {δ} (Γ₁ Γ₂ : Context γ) (Ξ : Matrix γ δ)

  ---------------------------------
  → (Γ₁ ⋈ Γ₂) ⊛ Ξ ≔ Γ₁ ⊛ Ξ ⋈ Γ₂ ⊛ Ξ

⊛-distribʳ-⋈   Ξ =
  begin
    0s
  ≡⟨ sym (⋈-identityʳ 0s) ⟩
    0s ⋈ 0s
  ∎

⊛-distribʳ-⋈ (Γ₁ , π₁ · A) (Γ₂ , π₂ · .A) Ξ =
  begin
    (π₁ + π₂) ** Ξ Z ⋈ (Γ₁ ⋈ Γ₂) ⊛ (Ξ ∘ S_)
  ≡⟨ ⊛-distribʳ-⋈ Γ₁ Γ₂ (Ξ ∘ S_) |> cong ((π₁ + π₂) ** Ξ Z ⋈_) ⟩
    (π₁ + π₂) ** Ξ Z ⋈ (Γ₁ ⊛ (Ξ ∘ S_) ⋈ Γ₂ ⊛ (Ξ ∘ S_))
  ≡⟨ **-distribʳ-⋈ (Ξ Z) π₁ π₂ |> cong (_⋈ Γ₁ ⊛ (Ξ ∘ S_) ⋈ Γ₂ ⊛ (Ξ ∘
  ↪  S_)) ⟩
    (π₁ ** Ξ Z ⋈ π₂ ** Ξ Z) ⋈ (Γ₁ ⊛ (Ξ ∘ S_) ⋈ Γ₂ ⊛ (Ξ ∘ S_))
  ≡⟨ ⋈-assoc (π₁ ** Ξ Z ⋈ π₂ ** Ξ Z) (Γ₁ ⊛ (Ξ ∘ S_)) (Γ₂ ⊛ (Ξ ∘ S_)) |>
  ↪  sym ⟩
    ((π₁ ** Ξ Z ⋈ π₂ ** Ξ Z) ⋈ Γ₁ ⊛ (Ξ ∘ S_)) ⋈ Γ₂ ⊛ (Ξ ∘ S_)
  ≡⟨ ⋈-assoc (π₁ ** Ξ Z) (π₂ ** Ξ Z) (Γ₁ ⊛ (Ξ ∘ S_)) |> cong (_⋈ Γ₂ ⊛ (Ξ
  ↪  ∘ S_)) ⟩
    (π₁ ** Ξ Z ⋈ (π₂ ** Ξ Z ⋈ Γ₁ ⊛ (Ξ ∘ S_))) ⋈ Γ₂ ⊛ (Ξ ∘ S_)
  ≡⟨ ⋈-comm (π₂ ** Ξ Z) (Γ₁ ⊛ (Ξ ∘ S_)) |> cong ((_⋈ Γ₂ ⊛ (Ξ ∘ S_)) ∘ (π₁
  ↪  ** Ξ Z ⋈_)) ⟩
    (π₁ ** Ξ Z ⋈ (Γ₁ ⊛ (Ξ ∘ S_) ⋈ π₂ ** Ξ Z)) ⋈ Γ₂ ⊛ (Ξ ∘ S_)
  ≡⟨ ⋈-assoc (π₁ ** Ξ Z) (Γ₁ ⊛ (Ξ ∘ S_)) (π₂ ** Ξ Z) |> sym ∘ cong (_⋈ Γ₂
  ↪  ⊛ (Ξ ∘ S_)) ⟩
    ((π₁ ** Ξ Z ⋈ Γ₁ ⊛ (Ξ ∘ S_)) ⋈ π₂ ** Ξ Z) ⋈ Γ₂ ⊛ (Ξ ∘ S_)
  ≡⟨ ⋈-assoc (π₁ ** Ξ Z ⋈ Γ₁ ⊛ (Ξ ∘ S_)) (π₂ ** Ξ Z) (Γ₂ ⊛ (Ξ ∘ S_)) ⟩
    (π₁ ** Ξ Z ⋈ Γ₁ ⊛ (Ξ ∘ S_)) ⋈ (π₂ ** Ξ Z ⋈ Γ₂ ⊛ (Ξ ∘ S_))
  ∎
```

While Lean can build the proof term without much assistance:

```
lemma vmul_right_distrib
  : ∀ {γ δ} {Γ₁ Γ₂: context γ} {Ξ: matrix γ δ},
    vmul (Γ₁ + Γ₂) Ξ = (vmul Γ₁ Ξ) + (vmul Γ₂ Ξ) :=
begin
  intros,
  induction Γ₁ with γ₁ π₁ T₁ Γ₁ ih₁;
  { cases Γ₂,
    unfold vmul,
    simp * },
end
```

It is also worth mentioning that Agda's Unified Type Theory seems to admit more complicated equalities, solving some of the issues of Section 6.1.1. This is something I would like to investigate further.

6.3 Other facets of Lean

Pros:

- The standard library is based on strong type class support. For example, proving that a custom structure `foo` makes a commutative semiring is enough to gain access to a powerful `ring` tactic which can manipulate arithmetic involving `foo`.
- A single language is used both for definitions and tactic meta-definitions, making it easier to program both interchangeably and alleviating the need to learn multiple sub-languages.
- Friendly and extraordinarily helpful community.

Cons:

- Restricted unification in pattern matching. Lean's equation compiler needs to be told explicitly which variables should be unified:

```
/- The definition of substitution on lambda terms. -/
def subst
  : Π {γ δ} {Γ: context γ} {Ξ: matrix γ δ} {A}
    (σ: sub_fn Ξ)
    (Δ: context δ)
    (M: term Γ A)
    (h: auto_param (Δ = Γ ⊛ Ξ) ``solve_context),
    term Δ A
/- Note how I have to match on *everything* for the dependent
   type of terms to unify correctly. -/
| _ _ _ _ _ σ _ (Var _ x _) _ := cast (by solve_context) $ σ _ x
/- etc. -/
```

  This is in contrast to Coq, which with the help of the `Program` command [32] can unify them automatically. Adding such a command to Lean would be a very interesting avenue for future work.

- Largely due to its young age, Lean lacks many common and powerful tactics such as hammers, linear algebra solvers, etc.

6.4 Conclusion

During the course of the project, I have made progress towards a fully formal specification of Synchronous GV. In order to do that, I have recreated the formulation of non-dependent QTT by Kokke and Wadler [24] in the Lean prover. Based on that, I have written a formal specification of Synchronous GV terms and configurations. I have proven the type soundness of SGV terms in Lean, as well as the well-formedness preservation of its configurations, laying foundations

for future work aiming at formally proving all of its properties. Writing down a full formal proof of progress and deadlock-freedom for SGV remains an open problem.

Working in a proof assistant gave rise to many conceptual problems which don't appear when working on paper - from which out of many equivalent formulations of SGV to use, to how to resolve issues with different kinds of equality and working with "very dependent" types. The resulting Lean code provides evidence supporting certain choices over others.

It seems that encoding strongly-typed invariants with functional terms in a CiC-based theorem prover leads to fundamental issues with what is expressible in this type theory. Two alternative paths worth pursuing in future work are to either try using weakly-typed terms or find a way to express objects such as linear typing contexts neatly, using purely inductive relations. An alternative solution would be to try and automate some of the pain away by porting Coq's `Program` and enhancing support for automatic resolution of heterogeneous equalities.

Overall, this work answered some questions but gave rise to many others, illuminating avenues for future work.

Bibliography

[1] Samson Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993. doi: 10.1016/0304-3975(93)90181-R. URL https://doi.org/10.1016/0304-3975(93)90181-R.

[2] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739 – 782. Elsevier, 1977. doi: https://doi.org/10.1016/S0049-237X(08)71120-0. URL http://www.sciencedirect.com/science/article/pii/S0049237X08711200.

[3] Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi: 10.1145/3209108.3209189. URL https://doi.org/10.1145/3209108.3209189.

[4] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Electronic textbook, April 2018. URL https://leanprover.github.io/theorem_proving_in_lean/. Release 3.4.0.

[5] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2): 141–159, 2012. doi: 10.1007/s10817-011-9219-0. URL https://doi.org/10.1007/s10817-011-9219-0.

[6] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. ISBN 978-3-642-15374-7. doi: 10.1007/978-3-642-15375-4_16. URL https://doi.org/10.1007/978-3-642-15375-4_16.

[7] Mario Carneiro. The type theory of lean. 2019. URL https://github.com/digama0/lean-type-theory.

[8] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming,*

ICFP 2008, Victoria, BC, Canada, September 20-28, 2008, pages 143–156. ACM, 2008. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411226. URL https://doi.org/10.1145/1411204.1411226.

[9] Bob Coecke. De-linearizing linearity: Projective quantum axiomatics from strong compact closure. *Electronic Notes in Theoretical Computer Science*, 170:49 – 72, 2007. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2006.12.011. Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005).

[10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006. ISBN 978-3-540-74791-8. doi: 10.1007/978-3-540-74792-5_12. URL https://doi.org/10.1007/978-3-540-74792-5_12.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.9*, January 2019. URL http://coq.inria.fr.

[12] Thierry Coquand and Peter Dybjer. Inductive definitions and type theory: an introduction (preliminary version). In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science, 14th Conference, Madras, India, December 15-17, 1994, Proceedings*, volume 880 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 1994. ISBN 3-540-58715-2. doi: 10.1007/3-540-58715-2_114. URL https://doi.org/10.1007/3-540-58715-2_114.

[13] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. ISBN 3-540-52335-9. doi: 10.1007/3-540-52335-9_47. URL https://doi.org/10.1007/3-540-52335-9_47.

[14] Martin Davis. The early history of automated deduction. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 3–15. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

[15] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972. ISSN 1385-7258. doi: https://doi.org/10.1016/1385-7258(72)90034-0. URL http://www.sciencedirect.com/science/article/pii/1385725872900340.

[16] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody

Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015. URL `http://arxiv.org/abs/1505.04324`.

[17] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. ISBN 978-3-319-21400-9. doi: 10.1007/978-3-319-21401-6_26. URL `https://doi.org/10.1007/978-3-319-21401-6_26`.

[18] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi: 10.1145/3110278. URL `https://doi.org/10.1145/3110278`.

[19] Simon Fowler. *Typed Concurrent Functional Programming with Channels, Actors, and Sessions*. PhD thesis, University of Edinburgh, 2018. Under review.

[20] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *PACMPL*, 3(POPL): 28:1–28:29, 2019. URL `https://dl.acm.org/citation.cfm?id=3290341`.

[21] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010. doi: 10.1017/S0956796809990268. URL `https://doi.org/10.1017/S0956796809990268`.

[22] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi: 10.1016/0304-3975(87)90045-4. URL `https://doi.org/10.1016/0304-3975(87)90045-4`.

[23] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. ISBN 3-540-57208-2. doi: 10.1007/3-540-57208-2_35. URL `https://doi.org/10.1007/3-540-57208-2_35`.

[24] Wen Kokke and Philip Wadler. *Programming Language Foundations in Agda*. Electronic textbook, April 2019. URL `https://plfa.github.io/`.

[25] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015. ISBN

978-3-662-46668-1. doi: 10.1007/978-3-662-46669-8_23. URL https://doi.org/10.1007/978-3-662-46669-8_23.

[26] Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. ISBN 978-3-319-30935-4. doi: 10.1007/978-3-319-30936-1_12. URL https://doi.org/10.1007/978-3-319-30936-1_12.

[27] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL https://doi.org/10.1016/0022-0000(78)90014-4.

[28] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN 978-0-521-65869-0.

[29] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

[30] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018. Version 5.5.

[31] John M. Rushby, Sam Owre, and Natarajan Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24(9): 709–720, 1998. doi: 10.1109/32.713327. URL https://doi.org/10.1109/32.713327.

[32] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2006. ISBN 978-3-540-74463-4. doi: 10.1007/978-3-540-74464-1_16. URL https://doi.org/10.1007/978-3-540-74464-1_16.

[33] Philip Wadler. Linear types can change the world! In Manfred Broy, editor, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990. ISBN 0-444-88545-5.

[34] Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi: 10.1017/S095679681400001X. URL https://doi.org/10.1017/S095679681400001X.

[35] H. Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, Jan 1960. ISSN 0018-8646. doi: 10.1147/rd.41.0002.