Verifying Type- and Scope-Safe Program Transformations

Piotr Jander

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2019

Abstract

There is an ongoing effort in the programming languages community to verify correctness of compilers. A typical compiler consist of several compilation passes which use different intermediate languages. Type-and-scope safe representation is a commonly used encoding for such intermediate languages; it facilitates proofs of correctness of compilation phases, including proofs by logical relations. However, using such representation requires repeating considerable meta-theoretical boilerplate for each intermediate language used by the compiler.

This project formalises an intermediate language with closures, implements a closure conversion algorithm, and mechanises two proofs of its correctness: with bisimulations and with Kripke logical relations.

This work builds on a line of research which gave rise to a state-of-the-art framework for representing languages with binders and generically proving their meta-theoretical properties [2]. While this technique is useful for certain intermediate langauges, this project shows that an otherwise appealing representation of an intermediate language with closures is not compatible with the framework.

Table of Contents

Chapter 1

Introduction

This project, at its most general, concerns verifying transformations of functional programs in compilers.

Functional language, with their rich semantics, expressive types, and restrictions such as purity, are particularly well-suited to verification, or formally proving that a program conforms to a specification. However, verification is performed with respect to the semantics of the source program, whereas guarantees are needed about the compiled machine code which is actually executed. To bridge the gap between verification of the source code and assurances about the executable code, the compiler should be proved to preserve the semantics of the source language.

Verification of compilers is typically achieved by mechanising a proof of their correctness in a proof assistant like Coq, Isabelle/HOL, or Agda. Reasoning about individual compilation phases depends on formalising the semantics of the source and target languages of the phase. But most languages have a notion of a variable binder and binding, and a language representation which facilitates reasoning about the binding structure greatly simplifies the task of showing semantics preservation.

This project applies a state-of-the-art technique for representing languages with binders to implement a type-preserving compiler transformation and prove its correctness with two distinct techniques: bisimulation and logical relations.

1.1 Motivation

Closure conversion is a compilation phase present in a typical compiler which takes a language with first-class, nested functions to machine code. Suppose the source and target languages are given by:

$\mathsf{S} ::= \mathsf{x} \mid \mathsf{S}\ \mathsf{S} \mid \lambda\mathsf{x}.\mathsf{S}$ $\qquad\qquad$ $\mathsf{T} ::= \mathsf{x} \mid \mathsf{T}\ \mathsf{T} \mid \langle\langle\ \lambda\mathsf{x}.\lambda\mathsf{e}.\mathsf{T}\ ,\ \mathsf{E}\ \rangle\rangle$

The source language is a variant of simply typed lambda calculus, and the target language is similar, except that it does not allow abstractions with free variables. Instead, it has closures, i.e. records which consist of (a) a function which take an argument and an environment record and have their bodies defined in terms of the argument and the environment, and (b) an environment record. A closure is a first-class value which simulates a function with free variables.

Suppose we implement a translation function which transforms a source program to a target program with closures. We would like to show that the transformation is correct according to a selected definition of correctness, e.g. that each reduction of the source program is simulated by a reduction of the target program.

To formalise this claim, we need to define operational semantics for both languages. And since our representations of the source and target langauge are type- and scope-safe, we need to define substitution, and hence renaming. Futhermore, the correctness result depends on multiple smaller results about the interplay between renaming, substitution, the translation function, and others.

This project reports on the experience of mechanising two different correctness proofs for closure conversion. It concludes with a discussion of possibilities of reducing the mechanisation effort.

1.2 Goals and contributions

The main **goal** of this work is to mechanise a proof of correctness of a compiler transformation using type- and scope-safe representation for the intermediate languages. Specifically, the **objectives** of this project, all of which have been achieved, are:

1. To implement a compiler transformation for a variant of simply-typed lambda calculus in Agda.
2. To use scope-safe and well-typed representation for the object languages.
3. To prove that the transformation is correct: that the output program of the transformation behaves “the same” as the input program.
4. To use generic programming techniques from ACMM.

Additionally, a **contribution** of this project is to demonstrate that languagues with closures and closure conversion are problematic for current state-of-the-art techniques for mechanising language meta-theory.

1.3 Overview and organisation

This report is organised as follows:

Chapter 2 contains a literature review and evaluation of exisiting work.

Chapter 3 explains background topics which are needed in later sections.

Chapter 4 defines the language with closures and explains the implementation of the type-preserving, environment-minimising closure conversion function.

Chapter 5 describes a mechanisation of a proof of bisimulation between the source and target languages of closure conversion.

Chapter 6 describes a mechanisation of a proof of correctness of closure conversion by logical relations.

Chapter 7 evaluates the work by comparing it to the state-of-the-art, looks at possible improvements, and explains how this project demonstrates limitations of existing generic proving techniques.

Chapter 8 explains the relation between this project and the UG4 project, and applies skills learned this year to improve last year's solutions. It also discusses differences and similarities between compiler transformations and program derivations.

Finally, Chapter 9 contains a summary of this work.

Chapter 2

Related work

There are several topics within the broad field of programming languages and verification which have special relevance to this project: (a) representations of languages with bindings, (b) generic proving of properties of program traversals, (c) compiler verification, and (d) the theory of languages similarity and equivalence. This chapter provides an overview of work on those topics.

2.1 Representations of languages with bindings

There are clear benefits to mechanising proofs about programming languages (PL) theory. A typical PL proof is relatively simple in terms of deep concepts used, but complex in terms of the number of cases and bookkeeping burden. An error in a pen-and-paper proof can invalidate a whole theory. Proof assistants provide means of checking proofs and doing the bookkeeping for the user, thus addressing the problems. And yet, currently a large proportion of papers submitted to PL conferences do not have an accompanying mechanisation.

In 2005, a group of PL researchers put forward a hypothesis that limited adoption of mechanised proofs stems from a lack of agreement about good ways of mechanising meta-theory of languages, in particular, languages with bindings. They issued a challenge [**?**] whose goal was to try to compare different representations a particular language with bindings, $F_{<:}$. In response, a dozen solution were submitted, using techniques for representing binders and bindings like named variables, de Bruijn variables, (parametric) higher-order abstract syntax [**?**] [**?**], nominal sets [**?**], and other.

This project follows ACMM [3] in using well-typed and scoped de Bruijn indices [4]. Discussed in the background section, it gives rise to a deep embedding of the object language, which can be inspected and modified. However, to ensure that transformations on programs preserve well-typedness and scopedness, this representation relies on the operations renaming and substitution. In proofs about meta-theory, there frequently arises a need to prove correctness lemmas about interactions between renaming, substitution, and other traversals. The next section discusses research in proving these kind of results generically.

2.2 Generic transformations of and proofs about type-and-scope safe programs

McBride's observation [6] that in a type-and-scope safe language the operations of renaming and substitution share a common structrure gave rise to a line of research on generic implementation of such traversals, and generic proofs of their properties.

A paper by Allais et al. which we will refer to as ACMM [3] deals with those problems in the setting of simply typed lambda calculus (STLC). It introduces a notion of a *semantics*, which is a record defining a traversal in terms of (a) the result value, (b) values mapped to the variables in the environment, (c) semantic counterparts to the syntactic constructors of STLC, and (d) the operation of *weaking* which ensured that the traversal remains well-typed and scoped when it recurses on a term under a binder.

In a dependently-typed proof assistant like Agda, the structure of the proof often mirrors the structure of the program whose properties are being proved. Therefore, when different traversals share a common structure, proofs which relate such traversals can be treated generically. ACMM exploits this fact and provides a generic way to prove certain classes of properties relating traverals on programs in STLC.

A follow-up paper by Allais et al., which we will refer to as AACMM, generalises the contributions of ACMM from the setting of STLC to a family of languages (syntaxes) which satisfy appropriate constrainst. A framework accompanying the AACMM paper allows the user to describe a syntax, and then uses generic programming and proving to generate the operations of renaming and substitutions for the syntax, together with correctness lemmas describing interactions between different traversals of the language.

The repository accompanying AACMM has an example demonstrating its contributions in action. A problem is: given two variants of STLC, with and without a *let* construct, implement a traversal which inlines *let* expressions, and prove it correct with a simulation. Two solution are given. A naive solution contains manual proofs of correctness lemmas relating different traversals. A solution using the AACMM framework is able to use generic proofs, and is many times shorter.

This work relies on results from ACMM, but does not attempt to use AACMM. However, Chapter 7 provides reflection on the feasibility of applying AACMM-like techniques to closure conversion.

2.3 Verified compilation

Closure conversion is just one possible verification phase, and its verification constitutes part of a wider effort to verify compilation end-to-end, which usually entails verifying operational correctness of all compilation phases.

As far as type safety is concerned, the reference is a paper by Morrisett et al., "From System F to Typed Assembly Language" [?]. It builds upon previous results in type safety of compilation phases (like the aforementioned [?]) and describes a typed RISC-like assembly (named TAL), which is the target of the final phases of compilation. As a whole, the paper proves type safety for a compilation pipeline from System F to TAL. It does not, however, prove end-to-end operational correctness.

An compiler which was verified for end-to-end operational correctness was described by Adam Chlipala in his paper "A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language". The source is a variant of the simply-typed lambda calculus (STLC). Compilation proceeds through six phases, eventually yielding idealised assembly code. The compiler is implemented in Coq, where terms and functions on terms are dependently typed, guaranteeing type preservation. This is also the approach taken in this project, except that we use Agda instead of Coq [?]. Operational correctness is proved by adopting denotational semantics, unlike in this project, which uses operational semantics. Due to unfamiliarity with operational semantics, we cannot comment on which approach is better (TODO or can we?).

Another example of a certified compiler is CompCert [?], which is the result of the first successful attempt to implement a certified compiler of a real-world (TODO wording) language. Even compared with the simply-typed lambda calculus (STLC), which was the source language in Chlipala's work [?], the C language is in some ways simpler, especially since it does not have first-class functions with free variables (TODO wording: scoping?). But, being a fully-fledged language, C presents enough challenges as the source language of a verified compiler.

2.4 Closure conversion

Closure conversion is a compilation phase where functions or lambda abstractions with free variables are transformed to /closures/. A closure consists of a body (code) and the /environment/, which is a record holding the values corresponding to the free variables in the body (code). Closure conversion transforms abstractions to closures, and replaces references to variables with lookups in the environment.

Closure conversion was necessarily used in every compiler for a language which supports functions with free variables (TODO wording: scope?). But the first work which provided a rigorous treatment of closure conversion was the paper "Typed Closure Conversion" by Minamide et al. [?]. It demonstrated type-preserving closure conversion, where closure environments have existential types (TODO wording). On top of a proof of type-safety, the paper contains a proof of operational correctness of the typed closure conversion algorithm by logical relations.

Another notable paper about closure conversion is "Typed Closure Conversion Preserves Observational Equivalence" by Ahmed and Blum [?]. The paper's title explains its main result, so we should explain the title.

(TODO bring up the Reynolds' paper) Within a language L, we have a program P =

C[A], where A is an implementation of an abstraction and C is the "context", or "the rest of the program". Given some other implementation A' of the abstraction, we say that A and A' are contextually equivalent when for all possible contexts C, programs P = C[A] and P' = C[A'] behave identically.

We say that another abstraction A' is contextually equivalent to A if for all contexts C, programs C[A] and C[A'] are equivalent. This corresponds to a programmer's intuition that A and A' behave in the same way in all possible programs.

TODO OE matters for security and safety: If an attack would be possible by exposing a certain implementation detail, then this detail is made inaccessible / private, for example by using an existential type.

Why this matters: modern software systems are made up of multiple components, of which some might not be trusted.

// To ensure reliable and secure operation, it is important to defend against faulty or malicious code. Language-based security is built upon the concept of abstraction: if access to some private implementation detail might enable an attack, then this detail is made inaccessible by hiding it behind an abstract interface, for example using an existential type. //

TODO I have quite a bit about the paper and we don't want to duplicate the paper's introduction: how do I make it shorter?

Chapter 3

Background

This chapter will introduce the relevant concepts. It will start with closure conversion, then discuss compilation phases and intermediate languages, and finally explain the Agda definitions and encodings which were borrowed from ACMM and PLFA.

3.1 Closure conversion

TODO explain and give an example

TODO explain why existential types

3.2 Compilation phases and intermediate representations

In all but the most trivial compilers, compilation proceeds in phases, or transformations. A compilation phase transforms the compilation unit to bring it one step closer from the source code to the target representation.

[diagram here]

Intermediate representations As illustrated in the figure, each compilation phase takes a source representation to a target representation [relate to diagram]. An intermediate representation can also be called an intermediate language, and abbreviated to IR or IL. For some phases, the source and target representation may be the same. Arguably, this is the case for constant expression folding.

However, other phases benefit from using different source and target representations. An example of such transformation is closure conversion, which as the reader may recall from [section], transforms abstractions with free variables to so called closures, which take an explit environement and can only reference values from that environment.

Typed and untyped IRs To question of whether closure conversion must necessarily use different source and target languages hinges on the distinction between typed and untyped intermediate language. Using a typed IR requires that at each point along the compilation pipeline, intermediate representantions are well-typed.

Suppose that closure conversion is performed on simply typed lambda calculus (STLC). One of the two in necessary for the target language of closure conversion: either it should have first-class closures, or existential types. Neither is true of STLC, so another intermediate language is needed.

[TODO unintellegible comment about this paragraph] On the other hand, if the source and target representations are untyped, then the compiler architect might get away with using the same intermediate language as both source and target (for example Scheme, which is sometimes used as a compilation target). But even in this case, compilation process might benefit if the abstract syntax has explicit closures.

IRs in this project This project uses a dependently typed meta language (Agda) to implement compilation phases (specifically, closure conversion), so typed intermediate representations are a natural choice. Therefore, in the following sections, we will describe two intermediate representations, which are both variants of lambda calculus. The source representation will be simply typed lambda calculus, which we will refer to as λst. The target representation will be simply typed lambda calculus with closures, denoted with λcl.

The two intermediate representations are similar, and differ mainly in having either abstractions with free variables in λst or closures with environments in λcl. Unfortunately, this means that formalisations of λst and λcl share a lot of duplication. This is a common problem in formalising languages which has recently been addressed by [2]. Whether techniques from Allais et al. are applicable to this work will be discussed in [related work]. On the other hand, [section] demonstrates that while two intermediate languages can only differ in a handful of syntactic constructs and reduction steps, they can behave very differently with respect to the ubiquitious operations of renaming and substitution.

3.3 Type- and scope-safe representation of simply typed lambda calculus λst

This section will discuss the encoding of simply typed lambda calculus (abbreviated as STLC, denoted with λst), which is the source language of closure conversion. Typing and reduction rules are standard for call-by-value lambda calculus, so it is the encoding in Agda which is of interest in this section. As similar encoding is used for the closure language λcl.

Using dependently typed Agda as the meta language allows us to encode certain invariants in the representation. Two such invariants are scope and type safety. The representation is scope-safe in the sense that all variables in a term are either bound by some binder in the term, or explicitly accounted for in the context. It is type-safe in the sense that terms are synonymous with their typing derivations, which makes ill-typed terms unrepresentable. This kind of scope and type safety is due to [4]. The rest of this section shows how this is achieved in Agda; the Agda encoding is based on the one used in [3], [2], and [10].

TODO STLC as a figure here

To start with, λst typed are defined as follows.

```
data Type : Set where
  α      : Type
  _⇒_    : Type → Type → Type
```

The context is simply a list of types.

```
Context : Set
Context = List Type
```

Variables are synonymous with proofs of context membership. Since a variable is identified by its position in the context, it is appropriate to call it a de Bruijn variable. Accordingly, the constructors of Var are named after *zero* and *successor*. Notice that the definition assumes that the leftmost type in the context corresponds to the most recently bound variable.

```
data Var : Type → Context → Set where
  z   : ∀ {σ Γ}     → Var σ (σ :: Γ)
  s   : ∀ {σ τ Γ}   → Var σ Γ    → Var σ (τ :: Γ)
```

We can now present the formulation of λst terms, which is synonymous with their typing derivations:

```
data Lam : Type → Context → Set where
  V   : ∀ {Γ σ}     → Var σ Γ           → Lam σ Γ
  A   : ∀ {Γ σ τ}   → Lam (σ ⇒ τ) Γ     → Lam σ Γ    → Lam τ Γ
  L   : ∀ {Γ σ τ}   → Lam τ (σ :: Γ)    → Lam (σ ⇒ τ) Γ
```

The syntactic variable V constructor takes a de Bruijn variable to a term. The abstraction constructor L requires that the body is well-typed in the context Γ extended with the type σ of the variable bound by the abstraction. The application constructor A follows the typing rule for application.

3.4 Type- and scope-safe programs

Many useful traversals of the abstract syntax tree involve maintaining a mapping from free variables to appropriate values. Two such traversals are simultaneous renaming and substitution.

Simultaneous renaming takes a term N in the context Γ. It maintains a mapping ρ from variables in the original context Γ to *variables* in some other context Δ. It produces a term in Δ, which is N with variables renamed with ρ.

Similarly, simultaneous substitution takes a term N in the context Γ. It maintains a mapping σ from variables in the original context Γ to *terms* in some other context Δ. It produces a term in Δ, which is N with variables substitution for with σ.

Before we can demonstrate an implementation of renaming and substitution, we need to formalise the notion of a mapping from free variables to appropriate values, which we call the *environment*.

```
record _–Env (Γ : Context) (𝒱 : Type → Context → Set) (Δ : Context) : Set where
  constructor pack
  field lookup : ∀  → Var σ Γ → 𝒱 σ Δ
```

A environment (Γ –Env) 𝒱 Δ encapsulates a mapping from variables in Γ to values 𝒱 (variables for renaming, terms for substitution) which are well-typed and -scoped in Δ.

An environment which maps variables to variables is important enough to deserve its own name.

```
Thinning : Context → Context → Set
Thinning Γ Δ = (Γ –Env) Var Δ
```

There is a notion of an empty environment ε, of extending an environment ρ with a value v: ρ • v, and of mapping a function f over an environment ρ: f <$> ρ, corresponding to the analogous operations on contexts (which are just lists). Finally, select ren ρ renames a variable with ren before looking it up in ρ.

```
ε : ∀ {𝒱 Δ} → ([] –Env) 𝒱 Δ
lookup ε ()

_•_ : ∀ {Γ Δ σ 𝒱} → (Γ –Env) 𝒱 Δ → 𝒱 σ Δ → (σ :: Γ –Env) 𝒱 Δ
lookup (ρ • v) Z = v
lookup (ρ • v) (S x) = lookup ρ x

_<$>_   : ∀ {Γ Δ Θ 𝒱₁ 𝒱₂}
        → (∀  → 𝒱₁ σ Δ → 𝒱₂ σ Θ) → (Γ –Env) 𝒱₁ Δ → (Γ –Env) 𝒱₂ Θ
lookup (f <$> ρ) x = f (lookup ρ x)

select : ∀ {Γ Δ Θ 𝒱} → Thinning Γ Δ → (Δ –Env) 𝒱 Θ → (Γ –Env) 𝒱 Θ
lookup (select ren ρ) k = lookup ρ (lookup ren k)
```

Notice that those four operations on environments are defined using copatterns [1] by "observing" the behaviour of lookup.

Equipped with the notion of environments, we can give an implementation of renaming and substitution:

```
ext : ∀ {Γ Δ} {σ : Type} → Thinning Γ Δ → Thinning (σ :: Γ) (σ :: Δ)
ext ρ = s <$> ρ • z

rename : ∀ {Γ Δ σ} → Thinning Γ Δ → Lam σ Γ → Lam σ Δ
rename ρ (V x)      =   V (lookup ρ x)
rename ρ (L N)      =   L (rename (ext ρ) N)
rename ρ (A M N)    =   A (rename ρ M) (rename ρ N)

exts : ∀ {Γ Δ} {τ : Type} → (Γ –Env) Lam Δ → (τ :: Γ –Env) Lam (τ :: Δ)
exts σ = rename (pack s) <$> σ • V z

subst : ∀ {Γ Δ σ} → (Γ –Env) Lam Δ → Lam σ Γ → Lam σ Δ
subst σ (V x)      =   lookup σ x
subst σ (L N)      =   L (subst (exts σ) N)
subst σ (A M N)    =   A (subst σ M) (subst σ N)
```

Notice that those two traversals are indentical except (1) *renaming* wraps the result of lookup ρ x in V, and *renaming* and *substitution* extend the environment in a different way: s <$> ρ • z vs rename (pack s) <$> σ • V z. The observation that renaming and substitution for STLC share a common structure was a basis was the unpublished manuscript by McBride [6], and subsequently motivated the ACMM paper [3]. In [section], we will show how ACMM abstracts this common structure of renaming and substitution into a notion of a semantics.

Also notice how the functions ext and exts extend the environment when the traversal goes under a binder.

An example instantiation of simultaneous substitution is single substitution. Single substitution replaces occurrences of the last-bound variable in the context, and it is useful for defining the beta reduction for abstractions. Single substitution environment is an identity substitution environment extended with a single value:

```
id-subst : ∀  → Subst Γ Γ
lookup id-subst x = V x

_/_ : ∀ {Γ σ τ} → Lam τ (σ :: Γ) → Lam σ Γ → Lam τ Γ
_/_  {_}  N M = subst (id-subst • M) N
```

3.5 ACMM's notion of a semantics

TODO ACMM, synch, fusions

3.6 Small-step operational semantics

The formalisation of small step semantics for a call-by-value lambda calculus is adapted from [10].

Values are terms which do not reduce further. In this most basic version of lambda calculus language, the only values are abstractions:

```
data Value : ∀ {Γ σ} → Lam σ Γ → Set where

  V-L   : ∀ {Γ σ τ} {N : Lam τ (σ :: Γ)}
          -----------------------------
    → Value (L N)
```

Our operational semantics include two kinds of reduction rules. Compatibility rules, whose names start with ξ, reduce parts of the term (specifically, the LHS and RHS of application). Beta reduction β-L, on the other hand, describes what an abstraction applied to a value reduces to.

```
data _—→_ : ∀ {Γ σ} → (Lam σ Γ) → (Lam σ Γ) → Set where

  ξ-A₁ : ∀ {Γ σ τ} {M M′ : Lam (σ ⇒ τ) Γ} {N : Lam σ Γ}
    →   M —→ M′
        ----------------
    → A M N —→ A M′ N

  ξ-A₂ : ∀ {Γ σ τ} {V : Lam (σ ⇒ τ) Γ} {N N′ : Lam σ Γ}
    → Value V
    →   N —→ N′
        ----------------
    → A V N —→ A V N′

  β-L : ∀ {Γ σ τ} {N : Lam τ (σ :: Γ)} {V : Lam σ Γ}
    →   Value V
        ---------------------
    → A (L N) V —→ N / V
```

A term which can take a reduction step is called a reducible expression, or a redex. A property of a language that every well-typed term is either a value or a redux is called type-safety. This property is captured by a slogan “well-typed terms don’t get stuck” and can be proved by techniques like *progress and preservation* or *logical relations*. Simply typed lambda calculus is type-safe, and so is this formalisation. For a proof of type safety for a similar formalisation of STLC, cf. [10].

Operational semantics are needed for the treatment of bisimulation.

Chapter 4

Formalising closure conversion

This chapter presents this project's formalisation of closure conversion. It starts by discussing the closure language λcl, an intermediate language which is like STLC but with abstractions replaced by closures. Then it demonstrates a type-preserving conversion for λst to λcl which has the property that the obtained closure environments are *minimal*. Finally, several properties about interactions between renaming and substitution in λcl are formally established — they are needed in proofs of correctness in subsequent chapters.

4.1 Closure language λcl

As discussed in the Background [or maybe Intro?] chapter, some compilation phases must use different source and target intermediate representations. This is the case with closure conversion, and this section presents a formalisation of an intermediate language with closures. The language is very similar the formalised simply typed lambda calculus, except that abstraction with free variables are replaced by closures with environments. What might seem like a simple change has interesting implications for traversals like renaming and substitution.

The closure language λcl shares types, contexts, and de-Bruijn-variables-as-proofs-of-context-membership, and their respective Agda formalisations, with the source representation. In general, two different intermediate representations do not need to share the same type system, but if they do, this simplifies formalisation. The descriptions of those formalisations can be found in Section [TODO].

4.1.1 Terms

The definition of terms of λcl differs from terms of λst in the L constructor, which, in λcl, holds the closure body and the closure environment.

```
data Lam : Type → Context → Set where
  V   : ∀ {Γ σ}        → Var σ Γ           → Lam σ Γ
```

```
A  : ∀ {Γ σ τ}     → Lam (σ ⇒ τ) Γ   → Lam σ Γ          → Lam τ Γ
L  : ∀ {Γ Δ σ τ}   → Lam τ (σ :: Δ)  → (Δ –Env) Lam Γ   → Lam (σ ⇒ τ) Γ
```

Notice that the typing rule for the closure constructor L mentions two contexts, Γ and Δ. We call Γ the *outer context* and Δ the *inner context* of a closure.

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma . e : \sigma \to \tau}\text{T-abs} \qquad \frac{e_{ev} = subst(\Delta \subseteq \Gamma) \qquad \Delta, x : \sigma \vdash e : \tau}{\Gamma \vdash \langle\langle \lambda x : \sigma . e \ , \ e_{ev} \rangle\rangle : \sigma \to \tau}\text{T-clos}$$

The closure as a whole is typed in Γ, but the closure body (also called the *closure code*) is typed in σ :: Δ. The relationship between Γ and Δ is given by the closure environment.

A closure environment is traditionally implemented as a record, and variables in the closure code reference fields of that record. In this development, on the other hand, the environment is represented as a substitution environment, that is, a mapping from variables in Δ to terms in Γ. This representation is isomorphic to the one using a record, and it has several benefits, especially eliminating the need for products in the language, and overall simplification of the formalisation.

Finally, recall from [section] that in order for a closure-converted program to be well-typed, a closure environment should have an existential type. It is important to note that in this formalisation, existential typing is achieved in the meta language Agda, not in the object language λcl, which does not have existential types. Indeed, existential quantification (including over types) can in achieved in Agda through dependent products, a datatype constructor is a dependent product, and the environment is a parameter to the L constructor.

4.1.2 Renaming and substitution

Consider the case for the constructor L of renaming and substitution in λcl and how it is different from the corresponding definition in λst.

```
rename : ∀ {Γ Δ σ} → Thinning Γ Δ → Lam σ Γ → Lam σ Δ
rename ρ (V x)      =   V (lookup ρ x)
rename ρ (A M N)    =   A (rename ρ M) (rename ρ N)
rename ρ (L N E)    =   L N (rename ρ <$> E)
```

```
subst : ∀ {Γ Δ σ} → Subst Γ Δ → Lam σ Γ → Lam σ Δ
subst ρ (V x)      =   lookup ρ x
subst ρ (A M N)    =   A (subst ρ M) (subst ρ N)
subst ρ (L N E)    =   L N (subst ρ <$> E)
```

Unlike in λst, renaming and substitution in λcl *do not go under binders* (do not change the closure body). This is because renaming and substitution take a term in a context Γ to a term in a context Γ'. But the code (body) of a closure is typed in a different context Δ. So upon recursing on a closure, renaming and substitution adjust the closure environment and leave the closure body unchanged. The adjustment to the environment is rename ρ <$> E in the case of renaming and subst ρ <$> E in the case of substitution. In either case, the adjustment consists of mapping the renaming/substitution over the values in the environment.

Just like in λst, we also define functions ext and exts which extend the environment when renaming or substitution goes under a binder:

```
ext : ∀ {Γ Δ} {σ : Type} → Thinning Γ Δ    → Thinning (σ :: Γ) (σ :: Δ)
ext ρ = s <$> ρ • z


exts : ∀ {Γ Δ σ} → Subst Γ Δ → Subst (σ :: Γ) (σ :: Δ)
exts ρ = rename (pack s) <$> ρ • V z
```

4.1.3 Operational semantics

Operational semantics are similar to the semantics for λst, except for adjustments for closures. Values in λcl are closures, and the rule for beta reduction is different:

```
infix 2 _—→_
data _—→_ : ∀ {Γ σ} → (Lam σ Γ) → (Lam σ Γ) → Set where

  β-L : ∀ {Γ Δ σ τ} {N : Lam τ (σ :: Δ)} {E : Subst Δ Γ} {V : Lam σ Γ}
    →   Value V
        --------------------
    → A (L N E) V —→ subst (E • V) N
```

Recall that a closure is a function without free variables, partially applied to an environment. When the closure argument reduces to a value, the argument and the values in the environment get simultaneously substituted into the closure body. The simplicity of this reduction rule is another benefit of representing environments as substitution environments.

4.1.4 Conversion from λst to λcl

This project's approach to typed, or type-preserving, closure conversion follows [8]. An important point here is that the specification of typed closure conversion allows for different implementations which might differ in their treatment of environments. The only requirement in the specification is that

1. If the source term is an abstraction typed in the context Γ;
2. if the body of the source abstraction can be typed in a smaller context Δ, such that $\Delta \subseteq \Gamma$;

3. then the target terms is a closure whose environment is a substitution from Δ to Γ.

This is given by the following conversion rule:

$$\frac{e_{ev} = subst(\Delta \subseteq \Gamma) \qquad \Delta, x : \sigma \vdash e \rightsquigarrow e' : \tau}{\Gamma \vdash \lambda x : \sigma.e \rightsquigarrow \langle\langle \lambda x : \sigma.e' \ , \ e_{ev} \rangle\rangle : \sigma \to \tau}$$

It is up to the implementation of closure conversion to decide how big to make Δ, on the spectrum between (1) Δ being equal to Γ, and (2) Δ being *minimal*, i.e. only containing the parts of Γ which are necessary to type the term. We present two Agda implementation of closure conversion, corresponding to the two ends of the spectrum.

Closure conversion where Δ is the same as Γ is a simple transformation:

```
simple-cc : ∀ {Γ σ} → S.Lam σ Γ → T.Lam σ Γ
simple-cc (S.V x) = T.V x
simple-cc (S.A M N) = T.A (simple-cc M) (simple-cc N)
simple-cc (S.L N) = T.L (simple-cc N) T.id-subst
```

where T.id-subst is the identity substitution which maps a term in Γ to itself, defined as:

```
id-subst : ∀  → Subst Γ Γ
lookup id-subst x = V x
```

We call the other end of the spectrum *minimising closure conversion*. Its implementation in Agda is rather more involved and is described in the next section.

4.1.5 Minimising closure conversion

Minimising closure conversion is given by the following deduction rules, where a statement $\Gamma \vdash \mathsf{e} : \sigma \rightsquigarrow \Delta \vdash \mathsf{e'} : \sigma$ should be read as: “the term e of type σ in the context Γ can be closure converted to the term e’ in Δ”:

$$\frac{}{\Gamma \vdash x : \sigma \rightsquigarrow \emptyset, x : \sigma \vdash x : \sigma}\ \text{(min-V)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \to \tau \rightsquigarrow \Delta_1 \vdash e'_1 : \sigma \to \tau \\ \Gamma \vdash e_2 : \sigma \rightsquigarrow \Delta_2 \vdash e'_2 : \sigma \\ \Delta = merge\ \Delta_1\ \Delta_2\end{array}}{\Gamma \vdash e_1 e_2 : \tau \rightsquigarrow \Delta \vdash e'_1 e'_2 : \tau}\ \text{(min-A)}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau \rightsquigarrow \Delta, x : \tau \vdash e : \tau \qquad e_{id} = subst(\Delta \subseteq \Delta)}{\Gamma \vdash \lambda x : \sigma.e : \sigma \to \tau \rightsquigarrow \Delta \vdash \langle\langle \lambda x : \sigma.e \ , \ e_{id} \rangle\rangle : \sigma \to \tau}\ \text{(min-L)}$$

min-V: Any variables can be typed in a singleton context containing just the type of the variable.

min-A: If the conversion e_1' of e_1 can be typed in Δ_1, and the conversion e_2' of e_2 can be typed in Δ_2, then the application e_1' e_2' can be typed in Δ, where Δ is the result of merging Δ_1 and Δ_2.

min-L: If the conversion e' of the abstraction body e can be typed in context $\sigma :: \Delta$ (or $\Delta, x : \sigma$, using the notation with names), then the closure resulting from the conversion of the abstraction can be typed in Δ, and it has the identity environment $\Delta \subseteq \Delta$.

To formalise this conversion in Agda, we need several helper definitions.

4.1.5.1 Merging subcontexts

The deduction rules for minimising closure conversion contained statements of the form $\Delta \subseteq \Gamma$, which reads: "Δ is a subcontext of Γ". Since in this development, a context is just a list of types, the notion of subcontexts can be captured with the _⊆_ (sublist) relation from Agda's standard library. The inductive definition of the relation is:

```
data _⊆_ : List A → List A → Set where
  base  : [] ⊆ []
  skip  : ∀ {xs y ys}   → xs ⊆ ys   → xs ⊆ (y :: ys)
  keep  : ∀ {x xs ys}   → xs ⊆ ys   → (x :: xs) ⊆ (x :: ys)
```

This project's contribution is to define the operation of merging two subcontexts. Given contexts Γ, Δ, and Δ_1 such that $\Delta \subseteq \Gamma$ and $\Delta_1 \subseteq \Gamma$, the result of merging the subcontexts Δ and Δ_1 is a context Γ_1 which satisfies the following conditions:

1. It is contained in the big context: $\Gamma_1 \subseteq \Gamma$.
2. It contains the small contexts: $\Delta \subseteq \Gamma_1$ and $\Delta_1 \subseteq \Gamma_1$.
3. The proof that $\Delta \subseteq \Gamma$ obtained by transitivity from $\Delta \subseteq \Gamma_1$ and $\Gamma_1 \subseteq \Gamma$ is the same as the input proof that $\Delta \subseteq \Gamma$; similarly for $\Delta_1 \subseteq \Gamma$.

All those requirements are captured by the following dependent record in Agda:

```
record SubListSum {Γ Δ Δ₁ : List A} (Δ⊆Γ : Δ ⊆ Γ) (Δ₁⊆Γ : Δ₁ ⊆ Γ) : Set where
  constructor subListSum
  field
    Γ₁      : List A
    Γ₁⊆Γ    : Γ₁ ⊆ Γ
    Δ⊆Γ₁    : Δ ⊆ Γ₁
    Δ₁⊆Γ₁   : Δ₁ ⊆ Γ₁
    well    : ⊆-trans Δ⊆Γ₁   Γ₁⊆Γ ≡ Δ⊆Γ
    well₁   : ⊆-trans Δ₁⊆Γ₁ Γ₁⊆Γ ≡ Δ₁⊆Γ
```

The type of the function which merges two subcontexts can be stated as:

```
merge : ∀ {Γ Δ Δ₁} → (Δ⊆Γ : Δ ⊆ Γ) → (Δ₁⊆Γ : Δ₁ ⊆ Γ) → SubListSum Δ⊆Γ Δ₁⊆Γ
```

We argue that the type of the function completely captures its behaviour (TODO how would we prove this?). The fact that a type can completely capture the behaviour of a function is a remarkable feature of programming with dependent types. Even more remarkable is the fact that the logical properties of Γ_1 are useful computationally. E.g the proof that $\Delta \subseteq \Gamma_1$ determines a renaming from Δ to Γ_1, which is used in the minimising closure conversion algorithm. A further example: the fact that ⊆-trans $\Delta{\subseteq}\Gamma_1$ $\Gamma_1{\subseteq}\Gamma \equiv \Delta{\subseteq}\Gamma$ is used in proofs of certain equivalences involving subcontexts and renaming.

4.1.6 Agda implementation of minimising closure conversion

Recall that terms of our intermediate languages are explicitly typed in a given context. For that reason, the result type of minimising closure conversion must be existentially quatified over a context. In fact, the context should be a subcontext of the input context Γ. This is captured with the dependent record _⊩_:

```
record _⊩_ (Γ : Context) (A : Type) : Set where
  constructor ∃[_]_∧_
  field
    Δ : Context
    Δ⊆Γ : Δ ⊆ Γ
    N : T.Lam A Δ
```

For example, a term N in a context Δ which is a subcontext of Γ by $\Delta{\subseteq}\Gamma$, would be constructed as ∃[Δ] Δ⊆Γ ∧ N.

With this data type, the type of the minimising closure conversion function is:

```
cc : ∀ {Γ A} → S.Lam A Γ → Γ ⊩ A
```

The function definition is by cases:

Variable case

```
cc {A = A} (S.V x) = ∃[ A :: [] ] Var→⊆ x ∧ T.V z
```

Following *min-V*, a variable is typed in a singleton context. The proof of the subcontext relation is computed from the proof of the context membership by a function Var→⊆.

Application case

```
cc (S.A M N) with cc M | cc N
cc (S.A M N) | ∃[ Δ ] Δ⊆Γ ∧ M† | ∃[ Δ₁ ] Δ₁⊆Γ ∧ N† with merge Δ⊆Γ Δ₁⊆Γ
cc (S.A M N) | ∃[ Δ ] Δ⊆Γ ∧ M† | ∃[ Δ₁ ] Δ₁⊆Γ ∧ N† | subListSum Γ₁ Γ₁⊆Γ Δ⊆Γ₁ Δ₁⊆Γ₁ _ _
  = ∃[ Γ₁ ] Γ₁⊆Γ ∧ (T.A (T.rename (⊆→ρ Δ⊆Γ₁) M†) (T.rename (⊆→ρ Δ₁⊆Γ₁) N†))
```

Given an application e_1 e_2, e_1 and e_2 are closure converted recursively, resulting in terms e_1' and e_2', which are typed in Δ_1 and Δ_2, respectively. Following *app-V*, the result of closure-converting the application is typed in the context Δ, which is the result of merging Δ_1 and Δ_2. As terms are explicitly typed in a context, e_1' and e_2' have to be renamed from Δ_1 to Δ, and from Δ_2 to Δ, respectively. A renaming environment is computed from a subcontext relation proof by the function ⊆→ρ which is given by:

```
⊆→ρ : {Γ Δ : Context} → Γ ⊆ Δ → Thinning Γ Δ
lookup (⊆→ρ base) ()
lookup (⊆→ρ (skip Γ⊆Δ)) x = s (lookup (⊆→ρ Γ⊆Δ) x)
lookup (⊆→ρ (keep Γ⊆Δ)) z = z
lookup (⊆→ρ (keep Γ⊆Δ)) (s x) = s (lookup (⊆→ρ Γ⊆Δ) x)
```

Abstraction case

```
cc (S.L N) with cc N
cc (S.L N) | ∃[ Δ ] Δ⊆Γ ∧ N† with adjust-context Δ⊆Γ
cc (S.L N) | ∃[ Δ ] Δ⊆Γ ∧ N† | adjust Δ₁ Δ₁⊆Γ Δ⊆AΔ₁ _
  = ∃[ Δ₁ ] Δ₁⊆Γ ∧ (T.L (T.rename (⊆→ρ Δ⊆AΔ₁) N†) T.id-subst)
```

Following *min-A*, the result of closure-converting an abstraction depends on the result N† of closure-clonverting its body. A recursive call on the body of the abstraction yields a term typed in some context Δ. But looking at the typing rule for closures (*T-clos*), the closure body is typed in a context $\sigma :: \Delta_1$ (or $\Delta_1, \mathsf{x} : \sigma$ using named variables), where σ is the type of the last bound variable and Δ_1 is the context corresponding to the closure environment. Thus, we need a way of decomposing Δ into σ and Δ_1, together with an appropriate proof of membership in the input context Γ.

This task is achieved by the function adjust-context:

```
adjust-context : ∀ {Γ Δ A} → (Δ⊆A::Γ : Δ ⊆ A :: Γ) → AdjustContext Δ⊆A::Γ
```

whose specification is captured by its return type which uses the dependent record AdjustContext:

```
record AdjustContext {A Γ Δ} (Δ⊆A::Γ : Δ ⊆ A :: Γ) : Set where
  constructor adjust
  field
    Δ₁      : Context
    Δ₁⊆Γ    : Δ₁ ⊆ Γ
    Δ⊆AΔ₁   : Δ ⊆ A :: Δ₁
    well    : Δ⊆A::Γ ≡ ⊆-trans Δ⊆AΔ₁ (keep Δ₁⊆Γ)
```

The specification is: given $\Delta \subseteq \mathsf{A} :: \Gamma$, there exists a context Δ_1 such that $\Delta_1 \subseteq \Gamma$ and $\Delta \subseteq \mathsf{A} :: \Delta_1$, such that the proof $\Delta \subseteq \mathsf{A} :: \Gamma$ obtained by transitivity is the same as the input proof.

The evidence that $\Delta \subseteq \mathsf{A} :: \Delta_1$ is used to rename N† so that the final inherently-typed term is well-typed.

***

We also provide a wrapper function _†:

```
_† : ∀ {Γ A} → S.Lam A Γ → T.Lam A Γ
M † with cc M
M † | ∃[ Δ ] Δ⊆Γ ∧ N = T.rename (⊆→ρ Δ⊆Γ) N
```

This function is a wrapper over the min-cc function which undoes the minimisation on the outer level. In other words, all closures in the term are still minimised, but the outer term is typed in the same context as the input source term. This is useful when we need to compare the input and output of closure conversion, and need to ensure that they are typed in the same context.

4.1.7 Fusion lemmas for the closure language λcl

One distinct kind of lemmas about interactions between different traversals, or semantics, is a fusion lemma. A fusion lemma relates three traversals: the pair we sequence and their sequential composition. The two traversals which have to be fused in this work's proofs are renaming and substitution. There are four ways we can sequence renaming and substitution, and each of those four sequencing can be expressed as a single renaming or substitution:

1. A renaming followed by a renaming is a renaming
2. A renaming followed by a substitution is a substitution,
3. A substitution followed by a renaming is a substitution,
4. A substitution followed by a substitution is a substitution.

We state the results as signatures of Agda functions, using the environment combinators _<$>_ and select which are described in Section 3.4.

```
rename∘rename : ∀ {Γ Δ Θ τ} (ρ₁ : Thinning Γ Δ) (ρ₂ : Thinning Δ Θ) (N : Lam τ Γ)
  → rename ρ₂ (rename ρ₁ N) ≡ rename (select ρ₁ ρ₂) N

subst∘rename : ∀ {Γ Δ Θ τ} (ρσ : Subst Γ Θ) (ρρ : Thinning Δ Γ) (N : Lam τ Δ)
  → subst ρσ (rename ρρ N) ≡ subst (select ρρ ρσ) N

rename∘subst : ∀ {Γ Δ Θ τ} (ρρ : Thinning Γ Θ) (ρσ : Subst Δ Γ) (N : Lam τ Δ)
  → rename ρρ (subst ρσ N) ≡ subst (rename ρρ <$> ρσ) N

subst∘subst : ∀ {Γ Δ Θ τ} (ρ₁ : Subst Γ Θ) (ρ₂ : Subst Δ Γ) (N : Lam τ Δ)
  → subst ρ₁ (subst ρ₂ N) ≡ subst (subst ρ₁ <$> ρ₂) N
```

Rather than include Agda proofs of all four lemmas, here we outline the proof structure, analyse just one of the four proofs, and compare fusion lemmas for λcl with the corresponding lemmas for λst.

A generic technique to prove fusion lemmas for STLC, including the ones about renaming and substitution, is one of the main contributions of ACMM [3]. Their proof uses Kripke logical relations and it relies on the invariant that corresponding environment values are in appropriate relations, including when environments are extended when going under a binder. Maintaining this invariant is possible thanks to the generic framework for writing traversals introduced by ACMM.

As it turns out, fusion lemmas for the closure language are simpler, as they do not require the logical relation machinery of ACMM. This is because renaming and substitution in λcl *do not happen under binders*, as can be seen from their definitions in Section 4.1.2. For both renaming and substitution, in the closure case (L), the closure body is left untouched; only the closure environment is modified.

We are now ready to take a closer look at the proof of the fusion lemma stating that a renaming followed by a subsitution is a substitution:

```
subst∘rename : ∀ {Γ Δ Θ τ} (ρσ : Subst Γ Θ) (ρρ : Thinning Δ Γ) (N : Lam τ Δ)
  → subst ρσ (rename ρρ N) ≡ subst (select ρρ ρσ) N


subst∘rename ρσ ρρ (V x)      =  refl
subst∘rename ρσ ρρ (A M N)    =  cong₂ A  (subst∘rename ρσ ρρ M)
                                          (subst∘rename ρσ ρρ N)
subst∘rename ρσ ρρ (L N E)    =  cong₂ L refl (env-extensionality h)
  where  h :  (_<$>_ {𝒲 = Lam} (subst ρσ) (_<$>_ {𝒲 = Lam} (rename ρρ) E))
              ≡ᴱ (subst (select ρρ ρσ) <$> E)
         h =  beginᴱ
               _<$>_ {𝒲 = Lam} (subst ρσ) (_<$>_ {𝒲 = Lam} (rename ρρ) E)
              ≡ᴱ⟨ <$>-distr {𝒲 = Lam} (rename ρρ) (subst ρσ) E ⟩
               _<$>_ {𝒲 = Lam} (subst ρσ ∘ rename ρρ) E
              ≡ᴱ⟨ <$>-fun {𝒲 = Lam} (λ e → subst∘rename ρσ ρρ e) E ⟩
               subst (select ρρ ρσ) <$> E
              ■ᴱ
```

The proof is by induction on the typing derivation of the term:

- In the variable case, the LHS and the RHS normalise to the same term, so refl suffices.
- In the application case, the proof is by induction and congruence.
- In the closure case, the proof is also by congruence, but an equational proof is required to show that the LHS and RHS act in the same way on the environment E.

The equational proof proceeds as follows:

1. It uses the fact that function composition _∘_ distributes through mapping over environments _<$>_: we have f <$> g <$> E ≡ f ∘ g <$> E which is capture by the lemma < $>-distr,

2. It uses the fact that when f and g are extensionally equal ($\forall$ {x} $\rightarrow$ f x $\equiv$ g x), then f <$> E $\equiv$ g <$> E which is captured by the lemma <$>-fun,
3. <$>-fun is instantiated with the inductive hypothesis.

Unfortunately, Agda does not recognise this project's fusion lemmas as terminating, and we were unable to provide a termination proof. Still, we believe that the function does in fact terminate.

Chapter 5

Proving correctness of closure conversion with bisimulation

The preceding sections defined the source and target languages of closure conversion, λst and λcl, together with reduction rules for each, and a closure conversion function min-cc from λst to λcl.

The min-cc closure conversion is type- and scope-preserving by construction. The property of type preservation provides confidence in the compilation process, but in this theoretical development which deals with a small, toy language, it is within the reach of this project to prove properties about operational correctness.

One such operational correctness property of a pair of languages is **bisimulation**. Intuition about bisimulation is captured by a slogan: related terms reduce to related terms.

This chapter starts by defining a relation between terms of λst and terms of λcl, which we call a *compatibility relation*. The compatibility relation is syntactic: in general, two terms are compatible when their subterms are compatible.

Then, we define what it means for a relation to be a bisimulation. A bisimulation is a relation which has a semantic property which relates reduction steps of source and target terms. Next, we will show that the compatibility relation is a bisimulation.

Finally, we will link the compatibility relation to closure conversion: we will argue that the graph relation of every sensible closure conversion function is contained in the compatibility relation. In particular, we will prove that this is the case for min-cc.

Overall, correctness of the minimising closure conversion is established: first, by showing that the input and output of closure conversion are related by a syntactic relation, and second, by showing that this syntactic relation is also a semantic relation. Thus, soundness of our closure conversion is established.

The part which shows that the compatibility relation is a bisimulation is inspired by the "Bisimulation" chapter from [10].

5.0.1 Compatibility relation

The compatibility relation is defined as follows:
Definition. Given a term M in λst and a term M† in λcl, the compatibilty relation M ∼ M† is defined inductively as follows:

- (*Variable*) For any given variable (proof of context membership) x, we have S.‘ x ∼ T.‘ x.
- (*Application*) If M ∼ M† and N ∼ N†, then M · N ∼ M† · N†.
- (*Abstraction*) If N   T.subst (T.exts E) N†, then S.L N ∼ T.L N† E.

Recall that λst and λcl share types, contexts, and variables (proofs of context membership). In fact, compatibility is only defined for source and target terms of the same type in the same context (this is explicit in the Agda definition).

While the variable and application cases are straightforward, the abstraction / closure case needs some explanation. Since the body N of the abstraction is defined in σ :: Γ, and the body of the closure N† is defined in σ :: Δ, they cannot be compatible. However, N can be compatible with the result of substituting the environment E in N† (the environment is extended with a variable corresponding to the σ in the context). The intuition for the abstraction/closure case is that substituting the environment “undoes” the effect of closure conversion on the context.

The compatibility relation is defined in Agda as follows:

```
data _~_ : ∀ {Γ σ} → S.Lam σ Γ → T.Lam σ Γ → Set where

  ~V : ∀ {Γ σ} {x : Var σ Γ}
    → S.V x ~ T.V x

  ~A : ∀   {Γ σ τ} {L : S.Lam (σ ⇒ τ) Γ} {L† : T.Lam (σ ⇒ τ) Γ}
           {M : S.Lam σ Γ} {M† : T.Lam σ Γ}
    → L ~ L† → M ~ M†
    → S.A L M ~ T.A L† M†

  ~L : ∀   {Γ Δ σ τ} {N : S.Lam τ (σ :: Γ)}
           {N† : T.Lam τ (σ :: Δ)} {E : T.Subst Δ Γ}
    → N ~ T.subst (T.exts E) N†
    → S.L N ~ T.L N† E
```

We have defined the syntactic compatibility relation. The next section defines what it means for a relation to be a bisimulation.

5.1 Bisimulation

Bisimulation, as the name implies, is defined in terms on two simulations: one from source to target terms, and the other one from target to source terms.

In the following definitions speak about a two languages, A and B. Also, whenever simulations or bisimulations are mentioned, they are implicitly *lock-step*. The literature has example of more general simulations.

Definition. Given a relation ≈ between terms of A and terms of B, we say that ≈ is a **simulation** from A to B if and only if for all terms M and N in A, and M† in B, if M reduces in a single step to N, and M and M† are in the ≈ relation (M ≈ M†), then there exists a term N† in B such that M† reduces to N† in a single step, and N is in the ≈ relation with N†: N ≈ N†.

The essence of simulation can be captured in a diagram.

$$\begin{tikzcd}
M \arrow[r, "\longrightarrow"] \arrow[d, "\approx"'] & N \arrow[d, "\approx"] \\
M\dagger \arrow[r, "\longrightarrow"] & N\dagger
\end{tikzcd}$$

Recall that the *converse* of the relation ≈ is a relation ≈' defined by y ≈' x whenever x ≈ y.

Definition. A relation ≈ is a **bisimulation** if and only if it is a simulation and its converse is also a simulation.

In Agda, we instantiate the definition of simulation twice: once for a simulation from λst to λcl, and again for a simulation from λcl to λst:

```
ST-Rel = ∀ {Γ σ} → S.Lam σ Γ → T.Lam σ Γ → Set

ST-Simulation : ST-Rel → Set
ST-Simulation _≈_ = ∀ {Γ σ} {M N : S.Lam σ Γ} {M† : T.Lam σ Γ}
  →  M ≈ M† → M S.—→ N
     ---------
  → ∃[ N† ] ((N ≈ N†) × (M† T.—→ N†))

TS-Simulation : ST-Rel → Set
TS-Simulation _≈_ = ∀ {Γ σ} {M : S.Lam σ Γ} {M† N† : T.Lam σ Γ}
  →  M ≈ M† → M† T.—→ N†
     ------------------------------
  → ∃[ N ] ((N ≈ N†) × (M S.—→ N))
```

Then we can provide an Agda definition of a bisimulation:

```
Bisimulation : ST-Rel → Set
Bisimulation _≈_ = ST-Simulation _≈_ × TS-Simulation _≈_
```

To show that the compatibility relation is a bisimulation, we need to obtain lemmas about the interactions between the compatibility relation, values, renaming, and substitution.

5.2 Compatibility, values, renaming, and substitution

As discussed in [TODO], mechanising the meta-theory of a language involves proving lemmas about the interactions between various traversals and transformations, including renaming, substitution, and compilation phases. This is also the case for proving correctness with bisimulation, which requires establishing lemmas about the interplay between the compatibility relation, values, renaming, and substitution. In fact, proving those lemmas often constitutes the biggest effort in the entire proof. In Chapter 7, we reflect on the possibiity of automating this effort with generic proving.

For each relevant property, we state it as an informal lemma, give its Agda statement, and its Agda proof.
Lemma. *Values commute with compatibility. If $M \sim M\dagger$ and M is a value, then $M\dagger$ is also a value.*

The proof is by cases of term constructors.

```
~val : ∀ {Γ σ} {M : S.Lam σ Γ} {M† : T.Lam σ Γ}
  → M ~ M† → S.Value M
  → T.Value M†
~val ~V          ()
~val (~L ~N)    S.V-L   =   T.V-L
~val (~A ~M ~N) ()
```

Lemma. *Renaming commutes with compatibility. If ρ is a renaming from Γ to Δ, and $M \sim M\dagger$ are compatible terms in the context Γ, then the results of renaming M and $M\dagger$ with ρ are also compatible: S.rename ρ $M \sim$ T.rename ρ $M\dagger$.*

The proof is by induction on the similarity relation.

```
~rename : ∀ {Γ Δ σ} {M : S.Lam σ Γ} {M† : T.Lam σ Γ}
  → (ρ : Thinning Γ Δ) → M ~ M†
  → S.rename ρ M ~ T.rename ρ M†
~rename ρ ~V                                = ~V
~rename ρ (~A ~M ~N)                        = ~A (~rename ρ ~M) (~rename ρ ~N)
~rename ρ (~L {N = N}   ~N) with ~rename (T.ext ρ) ~N
... | ~ρN rewrite TT.lemma-~ren-L ρ E N†    =    ~L ~ρN
```

The variable and application cases are straightforward, but as ever, the abstraction case is more involved: it requires rewriting with an instantiation of the fusion lemma rename∘subst.

```
lemma-~ren-L : ∀ {Γ Δ Θ σ τ} (ρρ : Thinning Γ Θ) (ρσ : Subst Δ Γ) (N : Lam τ (σ :: Δ))
  → rename (ext ρρ) (subst (exts ρσ) N) ≡ subst (exts (rename ρρ <$> ρσ)) N
```

The final lemma is about the interplay between compatibility and substitution.
Definition. Suppose ρ and ρ† are two substitutions which take variables x in Γ to terms in Δ, such that for all x we have that lookup ρ x ∼ lookup ρ† x. Then we say that ρ and ρ† are *pointwise compatible*.
Lemma. *Substitution commutes with compatibility. Suppose ρ and ρ† are two pointwise compatible substitutions. Then given compatible terms M ∼M† in Γ, the results of applying ρ to M and ρ† to M† are also compatible: S.subst ρ M ∼T.subst ρ† M†.*

Pointwise similarity relation between substitutions ρ and ρ† is defined in Agda with ∼σ:

```
record _~σ_ {Γ Δ : Context} (ρ : S.Subst Γ Δ) (ρ† : T.Subst Γ Δ) : Set where
  field ρ~ρ† : ∀  → (x : Var σ Γ) → lookup ρ x ~ lookup ρ† x
```

We can show that pointwise similarity is preserved by applying exts to both substitutions:

```
~exts : ∀ {Γ Δ} {σ : Type} {ρ    : S.Subst Γ Δ} {ρ† : T.Subst Γ Δ}
  → ρ ~σ ρ†
  → S.exts {τ = σ} ρ ~σ T.exts ρ†
ρ~ρ† (~exts ~ρ) z    = ~V
ρ~ρ† (~exts {σ = σ} {ρ = ρ}  ~ρ) (s x)
  = ~rename E.extend (ρ~ρ† ~ρ x)
```

In fact, exteding pointwise-similar substitutions with similar terms preserves pointwise similarity:

```
_~●_ : ∀   {Γ Δ σ} {ρ    : S.Subst Γ Δ} {ρ† : T.Subst Γ Δ}
           {M : S.Lam σ Δ} {M† : T.Lam σ Δ}
  → ρ ~σ ρ† → M ~ M†
  → ρ ● M ~σ ρ† ● M†
ρ~ρ† (ρ~σρ† ~● M~M†) z = M~M†
ρ~ρ† (ρ~σρ† ~● M~M†) (s x) = ρ~ρ† ρ~σρ† x
```

With the notion of pointwise similarity, we can prove that substitution commutes with similarity:

```
~subst : ∀   {Γ Δ τ} {ρ    : S.Subst Γ Δ} {ρ† : T.Subst Γ Δ}
             {M : S.Lam τ Γ} {M† : T.Lam τ Γ}
  → ρ ~σ ρ† → M ~ M†
  → S.subst ρ M ~ T.subst ρ† M†
~subst ~ρ (~V {x = x}) = ρ~ρ† ~ρ x
~subst ~ρ (~A ~M ~N) = ~A (~subst ~ρ ~M) (~subst ~ρ ~N)
~subst {ρ† = ρ†} ~ρ (~L {N = N}   ~N) with ~subst (~exts ~ρ) ~N
... | ~ρN rewrite TT.lemma-~subst-L ρ† E N† = ~L ~ρN
```

Just like in the lemma that renaming commutes with compatibility, the only non-trivial case is the one about abstractions/closures, which requires rewriting by an instatiation of the fusion lemma subst∘subst.

```
lemma-~subst-L : ∀ {Γ Δ Θ σ τ} (ρ₁ : Subst Γ Θ) (ρ₂ : Subst Δ Γ) (N : Lam τ (σ :: Δ))
  → subst (exts ρ₁) (subst (exts ρ₂) N) ≡ subst (exts (subst ρ₁ <$> ρ₂)) N
```

With those three lemmas, showing that the compatibility relation is a bisimulation becomes straightforward.

5.3 Compatibility is a bisimulation

The proof that the compatibility relation ∼ is a bisimulation consists of two proofs of simulations. Given:

st-sim : ST-Simulation _∼_   ts-sim : TS-Simulation _∼_

we have that ∼ is a bisimulation:

```
bisim : Bisimulation _~_
bisim = st-sim , ts-sim
```

The proofs of both st-sim and ts-sim are by case analysis on all possible instances of the compabiity relation ∼ and all possible instances of the reduction relation. The Agda mechanisation of the proof can be found in the technical appendix.

5.4 Compatibility and closure conversion

We have showed that the compability relation is a bisimulation. The connection between closure conversion and the compatibility relation is that we require that the graph relation of every-well behaved closure conversion function _† is contained in the compatibility relation: M ∼ M †. We cannot quantify over all closure conversion function, so instead, we must show that this is the case for every function which we claim is a well-behaved closure conversion. In this section, we will show that this property is possessed by the trivial closure conversion simple-cc and the minimising closure conversion _†.

While there are many possible closure conversion function, which differ by how big environments they construct, there is a unique backtranslation from λcl to λst, which we call undo. We can show that the converse of the graph relation of undo is contained in the compatibility relation: undo N ∼ N.

```
{-# TERMINATING #-}
undo : ∀ {Γ A} → T.Lam A Γ → S.Lam A Γ
undo (T.V x)       = S.V x
undo (T.A M N)    = S.A (undo M) (undo N)
undo (T.L M E)    = S.L (undo (T.subst (T.exts E) M))

{-# TERMINATING #-}
undo-compat : ∀ {Γ σ} (N : T.Lam σ Γ) → undo N ~ N
undo-compat (T.V x)        = ~V
undo-compat (T.A M N)      = ~A (undo-compat M) (undo-compat N)
undo-compat (T.L N E)      = ~L (undo-compat _)
```

Side note about proving termination of proofs. Notice that several functions in this development have been annotated as TERMINATING. This annotation is not checked, and if a function is annotated incorrectly, it could cause Agda to loop forever during typechecking. Furthermore, non-terminating function does not corresponds to a proof, and allowing such functions makes Agda's logic inconsistent.

In general, a function terminates if it strictly decreases in one of its arguments, and the type of that argument cannot decrease infinitely: e.g. natural numbers are bounded from below by zero. Agda can tell that an argument decreases when it is evident syntactically, but in more complicated cases, an explicit proof needs to be provided.

We can tell by inspection that the undo function terminates. We can define a size measure (function) on target terms which is defined as the number of term constructors in the term, including in the environment. Then it is easy to see that in the closure case, the argument to the recursive call to undo, T.subst (T.exts E) M, has a smaller measure than the input argument T.L M E.

An explicit proof of termination is not of interest in this project. The reason for mechanising proofs is to help with bookkeeing and prevent errors. When we are certain about a property such as termination, there is little value in mechanising its proof.

The trivial closure conversion simple-cc uses full contexts as environments (through identity substitutions id-subst):

```
simple-cc : ∀ {Γ σ} → S.Lam σ Γ → T.Lam σ Γ
simple-cc (S.V x) = T.V x
simple-cc (S.A M N) = T.A (simple-cc M) (simple-cc N)
simple-cc (S.L N) = T.L (simple-cc N) T.id-subst
```

simple-cc is well-behaved as its graph relation is contained in the compatibility relation. The proof is by straightforward induction; in the abstraction case, we need to argue that applying an identity substitution leaves the argument term unchanged.

```
~simple-cc : ∀ {Γ σ} (N : S.Lam σ Γ)
  → N ~ simple-cc N
~simple-cc (S.V x) = ~V
~simple-cc (S.A f e) = ~A (~simple-cc f) (~simple-cc e)
~simple-cc (S.L b) = ~L g
  where
  h : ∀ {Γ σ τ} (M : T.Lam σ (τ :: Γ)) → T.subst (T.exts T.id-subst) M ≡ M
  h   M =
    begin
      T.subst (T.exts T.id-subst) M
    ≡⟨ cong (λ e → T.subst e M) (sym (env-extensionality TT.exts-id-subst)) ⟩
      T.subst T.id-subst M
    ≡⟨ TT.subst-id-id M ⟩
      M
    ∎
  g : b ~ T.subst (T.exts T.id-subst) (simple-cc b)
  g rewrite h (simple-cc b) = ~simple-cc b
```

The minimising closure conversion _† is also well-behaved:

```
N~N† : ∀ {Γ A} (N : S.Lam A Γ)
  → N ~ N †
```

The proof of this is too long to discuss here, but the reader can find it in the technical appendix of this report.

***

This concludes the argument that our closure conversion is correct. We have shown that the graph relation of our closure conversion function is contained in the compatibility relation, and that the compatibility relation is a bisimulation. This means that when a source term and its closure converted target term both take a reduction step, then the terms they reduce to are also compatible.

[TODO example that reductions may take us out of the graph relation of a specific closure conversion function]

Chapter 6

Proving correctness of closure conversion by logical relations

The previous chapter showed a correctness property of closure conversion: the source and target of our closure conversion are in a relation which is a bisimulation. This chapter demonstrates another technique for showing correctness properties: Kripke-style [??] type-indexed logical relations. Type-indexed logical relations are characterised by using induction on the type structure of terms.

The outline of this chapter is similar to that of the previous one about bisimulations. First, we introduce a modified representations of simply typed lambda calculus (λst') and the language with closures (λcl'), where terms are explicitly labelled as values or reducible expressions (this helps with mechanisation as logical relations treat values and reducible terms differently). Unlike λst and λcl, λst' and λcl' semantics are defined as big step [TODO wording].

Then, the syntactic compatibility relation between λst' and λcl' is redefined. Finally, we define a logical relation between λst' and λcl', and we formulate the fundamental theorem for the that logical relation. As a corollary, it follows that the compatibility relation implies the logical relation for closed terms.

The proof by logical relations is based on [8], but the Agda mechanisation is this project's contribution.

6.1 Alternative representation of languages

This section presents an alternative representation of the source and target languages of closure conversion. We call the new formalisation of the source language λst', and the new formalisation of the target language - λcl'. Compared with λst and λcl in Chapter 4, λst' and λcl' are different in two ways. Firstly, the distinction between values and non-values is made explicit in the definition of terms in λst' and λcl', replacing a predicate on terms in λst and λcl. Secondly, we give big-step semantics for λst' and λcl', in contrast to small-step semantics for λst and λcl. These two differences simplify mechanisation of a proof by logica relation.

These improvements in formalisation are inspired by an Agda formalisation accompanying [7].

[TODO maybe only discuss STLC?]

The definitions of types, contexts, variables as proofs of context membership, and environments, are the same as for λst and λcl in the previous chapter. The definition of language expressions is different, however, in that it makes an explicit distinction between values Val and non-values Trm. This is achieved by indexing the Exp data type by a Kind:

```
data Kind : Set where
  ‘val ‘trm : Kind


data Exp : Kind → Type → Context → Set

Trm : Type → Context → Set
Trm = Exp ‘trm

Val : Type → Context → Set
Val = Exp ‘val

infixl 5 _‘$_

data Exp where

  -- values
  ‘var : ∀ {Γ σ} → Var σ Γ → Val σ Γ
  ‘λ : ∀ {Γ σ τ} → Trm τ (σ :: Γ) → Val (σ ⇒ τ) Γ

  -- non-values (a.k.a.  terms)
  _‘$_ : ∀ {Γ σ τ} → Val (σ ⇒ τ) Γ → Val σ Γ → Trm τ Γ
  ‘let : ∀ {Γ σ τ} → Trm σ Γ → Trm τ (σ :: Γ) → Trm τ Γ
  ‘val : ∀ {Γ σ} → Val σ Γ → Trm σ Γ
```

Notice that there are two new constructors for language expressions. The first one is ‘val, which takes a value Val to a term Trm and thus makes it possible to use values in positions where terms are expected. The second is ‘let, which is a standard let construct. The let is necessary to make the evaluation order explicit: function application applies a value to a value, so nested computations need to be factored out and bound as values by a let expression. This representation is known as A-normal form [9]; its other benefit is that it simplifies the definition as big-step semantics.

Definition of renaming and substitution are similar to those for λst, so we do not include the updated versions here.

We define aliases for closed values Val_0 and closed terms Trm_0 (typed in an empty context):

```
Exp₀ : Kind → Type → Set
Exp₀ k τ = Exp k τ []

Trm₀ : Type → Set
Trm₀ = Exp₀ ‘trm

Val₀ : Type → Set
Val₀ = Exp₀ ‘val
```

Like it was mentioned, the semantics of λst are defined as big-step semantics [TODO wording]. Given a term M and a value V, the inductive definition M ⇓ V states the conditions for M to reduce to a value V:

```
data _→₁_ : ∀ → Trm₀ σ → Trm₀ σ → Set where
  →₁app : ∀ {σ τ} {M : Trm τ (σ :: [])} {V : Val₀ σ} → ‘λ M ‘$ V →₁ M [ V ]

data _⇓_ : ∀ → Trm₀ σ → Val₀ σ → Set where
  ⇓val    : ∀ {V : Val₀ σ} → ‘val V ⇓ V
  ⇓app    : ∀ {σ τ} {M : Trm τ (σ :: [])} {V : Val₀ σ} {U : Val₀ τ}
          → M [ V ] ⇓ U → ‘λ M ‘$ V ⇓ U
  ⇓let    : ∀ {σ τ} {M : Trm₀ σ} {N : Trm τ (σ :: [])} {U : Val₀ σ} {V : Val₀ τ}
          → M ⇓ U → N [ U ] ⇓ V → ‘let M N ⇓ V
  ⇓step   : ∀ {M M’ : Trm₀ σ} {V : Val₀ σ} → M →₁ M’ → M’ ⇓ V → M ⇓ V
```

It is worth explaining the ⇓step constructor and the M $\rightarrow_1$ M’ data type. The M $\rightarrow_1$ M’ data type describes part of the small-step reducton relation and has a single constructor which captures beta reduction for functions. The ⇓step constructor is similar to the transitive closure of the small-step reduction relation: if M reduces to M’ in a single step, and M’ reduces to V in multiple steps, then M reduces to V in multiple steps.

Finally, ... [TODO what to make of a non-terminating proof of termination?]

```
{-# TERMINATING #-}
sn : ∀ (N : Trm₀ σ) → Σ[ V ∈ Val₀ σ ] (N ⇓ V)
sn (‘var () ‘$ _)
sn (‘λ M ‘$ V) with sn (M [ V ])
sn (‘λ M ‘$ V) | U , M[V]⇓U = U , ⇓step →₁app M[V]⇓U
sn (‘let M N) with sn M
sn (‘let M N) | U , M⇓U with sn (N [ U ])
sn (‘let M N) | U , M⇓U | V , N⇓V = V , ⇓let M⇓U N⇓V
sn (‘val V) = V , ⇓val
```

Differences between λcl and λcl’ are analogous.

6.2 Correctness by logical relations

This section defines two relations between terms of λst’ and λcl’. The first is just a reformulation of the compatibility relation from Chapter 5, which, as the reader may remember, subsumes the graph relation of any closure conversion. The other is a *logical relation*, which captures the notion that related terms reduce to related values.

We set up a fundamental theorem for the logical relation, and as a corollary, we obtain the result that for closed terms, the compability relation implies the logical relation.
Lemma. Correctness property for the compatibility relation: *Given a closed source term M, and a closed target term M†, if M and M† are compatible, then they reduce to values which are in the logical relation.*

From this, a correctness property for closure conversion follows: given any well-behaved closure conversion function _†, a closed source term M, and a closed target term M† which is the result of closure converting M, since M and M† are compatible, they reduce to related values.

The proof is inspired by a sketch of an argument from [8].

6.2.1 The compatibility relation

We denote the compatibility relation with ≅. In general, given a term M_1 in λst’ and a term M_2 in λcl’, M_1 and M_2 are compatible ($M_1 \cong M_2$) when their subterms are compatible. In the special case of abstractions/closures, the closure body is renamed with the environment in the premise of the rule.

```
infix   4 _≅_
data _≅_ : ∀ {Γ σ k} → S.Exp k σ Γ → T.Exp k σ Γ → Set where

  -- values

  ~var : ∀ {Γ σ} {x : Var σ Γ}
    ----------------
    → S.‘var x ≅ T.‘var x

  ~λ : ∀ {Γ Δ σ τ} {N₁ : S.Trm τ (σ :: Γ)} {N₂ : T.Trm τ (σ :: Δ)} {E : T.Subst Δ Γ}
    →   N₁ ≅ T.subst (T.exts E) N₂
        -----------------
    → S.‘λ N₁ ≅ T.‘λ N₂ E

  -- terms

  _~$_ : ∀   {Γ σ τ} {L : S.Val (σ ⇒ τ) Γ} {L† : T.Val (σ ⇒ τ) Γ}
             {M : S.Val σ Γ} {M† : T.Val σ Γ}
    → L ≅ L†
    →   M ≅ M†
```

```
        ----------------------
      → L S.‘$ M ≅ L† T.‘$ M†

  ~let : ∀   {Γ σ τ} {M₁ : S.Trm σ Γ} {M₂ : T.Trm σ Γ}
              {N₁ : S.Trm τ (σ :: Γ)} {N₂ : T.Trm τ (σ :: Γ)}
      → M₁ ≅ M₂
      →   N₁ ≅ N₂
        ------------------------------
      → S.‘let M₁ N₁ ≅ T.‘let M₂ N₂

  ~val : ∀ {Γ σ} {M₁ : S.Val σ Γ} {M₂ : T.Val σ Γ}
      →   M₁ ≅ M₂
        ------------------------
      → S.‘val M₁ ≅ T.‘val M₂
```

For brevity, we do not include translation functions from λst’ to λcl’. The reader should convince themselves that the minimising closure conversion from λst and λcl could be ported to λst’ and λcl’, and that its graph relation woud be contained in ≅.

6.2.2 The logical relation

While the compatibility relation captures syntactic correspondence, we need another relation on (closed) language expressions which captures operational correspondence. We define a family of logical relation ⇔ relating closed source terms (reducible expressions) to closed target terms (∼) and closed source values to closed target values (≈). The relations are defined by induction on types. In the definition, we write $\tau \ni \mathsf{M}_1\ \ \mathsf{M}_2$ or $\tau \ni \mathsf{M}_1 \approx \mathsf{M}_2$ to mean that M_1 and M_2 are related at type τ:

$$\begin{array}{rlll} \tau \ni & \mathsf{M}_1 \sim \mathsf{M}_2 & \text{iff} & \mathsf{M}_1 \Downarrow \mathsf{V}_1,\ \mathsf{M}_2 \Downarrow \mathsf{V}_2, \text{ and } \tau \ni \mathsf{V}_1 \approx \mathsf{V}_2 \\ \sigma \Rightarrow \tau \ni & \mathsf{U}_1 \approx \mathsf{U}_2 & \text{iff} & \text{for all } \sigma \ni \mathsf{V}_1 \approx \mathsf{V}_2,\ \tau \ni \mathsf{U}_1\ \text{‘\$}\ \mathsf{V}_2 \sim \mathsf{U}_2\ \text{‘\$}\ \mathsf{V}_2 \end{array}$$

There is no case for ≈ at the ground type α as only variables can have the ground type, and the values in ≈ are closed.

In Agda, ∼ and ≈ are defined as specialisations of the ⇔ relation on closed expressions of λst and λcl.

```
{-# NO_POSITIVITY_CHECK #-}
data _⇔_ : ∀ {k τ} → S.Exp₀ k τ → T.Exp₀ k τ → Set

_~_ : ∀  → S.Trm₀ τ → T.Trm₀ τ → Set
_~_ = _⇔_

_≈_ : ∀  → S.Val₀ τ → T.Val₀ τ → Set
_≈_ = _⇔_

data _⇔_ where
```

```
-- values

≈λ :  ∀  {Δ σ τ} {M₁ : S.Trm τ (σ :: [])}
          {M₂ : T.Trm τ (σ :: Δ)} {E : T.Subst Δ []}
      →  ({V₁   : S.Val₀ σ} {V₂ : T.Val₀ σ}
               → V₁ ≈ V₂ → M₁ [ V₁ ] ~ T.subst (E • V₂) M₂)
          -----------------------------------------------------
      → S.‘λ M₁ ≈ T.‘λ M₂ E

-- terms

~Trm : ∀  {N₁ : S.Trm₀ σ} {N₂ : T.Trm₀ σ}
           {V₁ : S.Val₀ σ} {V₂ : T.Val₀ σ}
   → N₁ S.⇓ V₁
   → N₂ T.⇓ V₂
   →   V₁ ≈ V₂
       -------
   → N₁ ~ N₂
```

We define a pointwise version of the ≈ relation which relates source and target substitution environments, similar to what we did in Section 5.2:

```
record _•≈_ {Γ : List Type}
  (ρˢ : S.Subst Γ []) (ρᵗ : T.Subst Γ []) : Set where
  constructor packᴿ
  field lookupᴿ   : {σ : Type} (v : Var σ Γ)
                  → lookup ρˢ v ≈ lookup ρᵗ v
```

We also provide a function •ᴿ which extends two related substitution environments with a pair of related values:

```
_•ᴿ_   :   ∀ {Γ τ}
            {ρˢ : S.Subst Γ []} {ρᵗ : T.Subst Γ []}
            {N₁ : S.Val₀ τ} {N₂ : T.Val₀ τ}
        → ρˢ •≈ ρᵗ
        →   N₁ ≈ N₂
            ---------------------------------------
        → ρˢ • N₁ •≈ ρᵗ • N₂
lookupᴿ (ρᴿ •ᴿ ≈N) z       = ≈N
lookupᴿ (ρᴿ •ᴿ ≈N) (s x)   = lookupᴿ ρᴿ x
```

Finally, we can state the fundamental theorem for our logical relation [TODO wording]. The theorem generalised the Correctness property for the compatibility relation to open terms, as long as there are substitution environments in the pointwise relation.

Lemma. Fundamental theorem of logical relations. *Given a source term M, a target term M†, a source substitution ρ^s, and a target substitution ρ^t, if M is compatible with M†, and for all variables x in the context, the corresponding values in the substitution environments are in the logical relation ($\rho^s(x) \approx \rho^t(x)$), then S.subst ρ^s M and T.subst ρ^t M† are in the logical relation.*

```
  fund :   ∀ {Γ σ k} {M₁ : S.Exp k σ Γ} {M₂ : T.Exp k σ Γ}
           {ρˢ : S.Subst Γ []} {ρᵗ : T.Subst Γ []}
    → ρˢ •≈ ρᵗ
    →    M₁ ≅ M₂
         --------------------------------
    → S.subst ρˢ M₁ ⇔ T.subst ρᵗ M₂
```

Observe that the Fundamental theorem, instatiated from closed terms, is equivalent to the Correctness property for the compatibility relation.

We do not include the Agda proof here as it is not very readable; instead, we present several cases on paper: TODO

[TODO let case in the proof]

Chapter 7

Reflections and evaluation

This project is a case study on verification of transformations of functional programs using two different techniques: bisimulations and logical relations. The implemented transformation is closure conversion. Both proofs of operational correctness are mechanised with state-of-the-art techniques.

Recall that the **objectives** set forth and achieved in the project were:

1. To implement a compiler transformation for a variant of simply-typed lambda calculus in Agda.
2. To use scope-safe and well-typed representation for the object languages.
3. To prove that the transformation is correct: that the output program of the transformation behaves “the same” as the input program.
4. To use generic programming techniques from ACMM.

[TODO split eval and refl, highlight achievements]

(Objective 1) Capturing the essence of closure conversion The implemented tranformation — closure conversion — requires a different source and target language. While the formalisation of the source language is largely borrowed from ACMM, and the formalisation of the target language is similar except for the difference between abstractions and closures, this project’s contribution was to capture the essence of closure conversion in what we believe is the simplest and most elegant way possible.

Inherently typed closures A traditional representation of closure conversion replaces variables in the source program with references to a record containing the environment in the target program. This project’s use of scope-safe and well-typed terms allowed for a more elegant solution where the closure body is typed in a context corresponding to the closure’s environment, and variables remain variables.

Closure environments as substitution environments Furthermore, while a closure environment is traditionally represented as a record which stores environment values, this project captures the essence of an environment by representing it as a substitution environment, i.e. a mapping from variables to values.

Existential types for closure environments As this report points out, closure environment must have existential types in order for a program with closures to be well-typed. This observation was made by [8], which deals with this fact by equipping the closure language with existential types. This project uses a different, arguably simpler approach, whereby closure environment are existentially typed *in the meta language (Agda)*, which allows us to keep object langauge types simple.

Comparison with traditional closure conversion In comparison with traditional closure conversion which represents environments as records, this formulation, which represents closure environments as substitution environments, i.e. meta-language functions, is further removed from the eventual target, which is machine code. But one can imagine a subsequent compilation phase which replaces substitution environments with records, and variables with record lookups (the object language would need existential types then). In general, splitting the compilation process into many specialised passes facilitates verification, as each compilation phase is easier to verify, and composing correctness results about phases gives rise to a end-to-end correctness result.

(Objetive 2) Scope-safe and well-typed representation Both the source and target language have scope-safe and well-typed representation, which were possible thanks to using dependently-typed Agda as the meta language. Using inherently scoped and typed terms has many benefits, which include the fact that when programs are synonymous with their typing derivations, transformations on programs are synonymous with proofs of type preservation. Additionally, many techniques for reasoning about operational correctness are type directed, e.g. the type-indexed logical relations which we used, and inherently typed representations are well-matched to such techniques.

(Objetive 3) The closure conversion preserves operational correctness This project uses two standard techniques to show that the implemented closure conversion is correct: bisimulation and logical relations. In an informal setting of pen-and-paper proofs, both of those techniques have rather straighforward proofs. However, mechanisation of those proofs involves proving several lemmas about the interactions between renaming, substitution, closure conversion, and the compatibility relation.

Mechanising the meta-theory of a language As observed in ACMM, mechanising the meta-theory of a language most often requires proving lemmas about the interactions between different transformations, or semantics, like renaming and substitutions. ACMM singles out synchronisation lemmas, which relate two semantics (e.g. for every renaming there exists a substitution which behaves the same), and fusion lemmas (e.g.

for every composition of two substitutions, there exists a substitution which behaves the same).

(Objective 4) ACMM ACMM exploit similarity between various traversals (semantics) on simply typed lambda calculus (STLC) to come up with a generic way to prove synchronisation and fusion lemmas for STLC.

It should be noted that the objective of using generic proving from ACMM was met partially. This project does borrow a type- and scope-safe representation from ACMM, but it does not duplicate ACMM's effort of setting up the machinery for generic proofs of synchronisation and fusion lemmas. That machinery depends on the concrete representation of STLC, and since ours is slightly different, we just postulate the synchronisation and fusion lemmas for STLC — ACMM shows that their their mechanical proofs exist and can be made generic.

Intermediate languages other than STLC require their own definitions of renaming and substitution, and proofs of correctness lemmas. For example, the proofs of operational correctness with bisimulations and logical relations depend on four fusion lemmas relating renaming and substitution for the language with closures. Since ACMM did not show anything about a language with closures, we proved the necessary lemmas manually. In fact, mechanising those lemmas constituted the biggest effort in the whole proof.

Possible remedy: AACMM and generic programming The problem of having to define renaming and substitution for each new language, and proving correctness lemmas about the interactions between renaming, substitution, and transformations, is address by a follow-up paper, which we will refer to as AACMM [2]. AACMM provides a way to supply a definition of a syntax with bindings, and then derives meta-theoretical correctness lemmas from that definition. The paper repository contains an example of an elaboration whose source is a language with a let construct, and whose target is simply-typed lambda calculus with let-expressions inlined. The example demonstrates how a proof of simulation is drastically simplified thanks to the use of the AACMM library and its generic programing capabilities.

Feasibility of closure conversion in AACMM AACMM demonstrates that transformations like let-inlining and CPS conversion can be expresses in their generic framework. They pose an open question about which compilation passes can be implemented generically. Unfortunately, this work suggests that closure conversion might not fit well into the AACMM framework. Specifically, the closure language in this project — with aforementioned features like syntax being mutually dependent on substitution environments, or environments being existentially quatified in the meta language (Agda) — is not expressible as an AACMM generic syntax. The traditional representation of a languages with closures — with environments as records and existential types in the object language — would not fit either as syntaxes in AACMM cannot contain existential types.

Bisimulation vs logical relations [TODO what could I say about the pros and cons of both?] [Derek Dreyer's paper]

Summary This work and ACMM/AACMM are both concerned with mechanising the meta-theory of languages, and applying this metatheory to reason about the language. While AACMM shows that a certain class of languages / syntaxes can be treated generically, this work contains a negative result which indicates that a language with closures might not benefit from current techniques for relieving the burden of mechanising meta-theory. This is an open question, however, whether there exist feasible generic syntaxes which would encompass a language with closures, or whether an alternative formalisation of a language with closures exists which is compatible with AACMM.

[TODO the conclusions of this chapter depend on my understanding of AACMM — is it correct?]

Chapter 8

Relationship to the UG4 project

The UG4 explored one particular kind of program transformations, which we refer to as program derivation. In program derivation, a tranformation is specified by a source program (or specification), and a recipe for obtaining a result program. The UG4 project assumed a particular style of derivations known as Bird-Meertens Formalism [5]. In BMF, derivations are based on equational reasoning, and consist of a sequence of intermediate forms of the program, where the steps are annotated with rules justifying each step. This is illustrated by the template below:

```
specification
  = { justification }
intermediate form 1
  = { justification }
...
  = { justification }
intermediate form n
  = { justification }
implementation
```

This year's work explores the other ends of the spectrum of transformations on programs, which encompasses compilation phases.

This chapter attempts to find common characteristics between compilation phases and program derivations, but also highlight their differences. It also looks at the middle of the spectrum, where certain transformations on programs could be classified either way. It ends with a reflections on lessons learned from this year's project which would have been helpful for last year's work.

In literature, what we refer to as program derivation is sometimes called program construction or calculation, and a program derivation described an instance of the process. Here, we use the term "derivation" to describe a family of transformations on programs, in order to avoid confusion with program transformations within compilers.

8.1 Compilation phases and program derivations

Compilation phases and derivations can be described and compared in terms of several criteria: (a) the objective of the transformation, (b) what the source and the target of the transformation is, (b) required expertise from the user, (c) user's expertise and input needed to guide the transformation, and (d) whether rules apply in all or only selected possible places.

The following characterises compilation phases:

1. (**Objective** and **Source and target of transformation**) The objective is to generate low-level code from a program in the source language.
2. (**Required expertise**) Programmer only needs to know the source language
3. (**Input from programmer and required expertise**) Little or none, except for specialised annotations which are used by the compiler
4. (**Totality and selectivity**) If a compilation phases can be specified as a rule, then the rule is typically applied in all the places where its premises match
5. (**Examples**) Closure conversion, CPS transformation, lambda lifting, type-checking, constant expression folding, code generation, dead code elimination, inlining.

The following are features of program derivation:

1. (**Objective**) Enable the programmer to write clear, concise, understandable programs which serve as specification. Methodically derive a correct, efficient implementation. Possibly mechanise the tranformation described by the derivation.
2. (**Source and target of transformation**) The source could be an executable functional program or a non-executable specification (e.g. in the categorical calculus of relations; or as a solution to an equation). The target is an efficient functional or imperative program.
3. (**Input from programmer and required expertise**) A derivation is a description of a transformation which can be calculated mechanically, so by definition, non-trivial derivations require input fromt the programmer. This might require expertise beyond the capabilities of an average programmer.
4. (**Totality and selectivity**) Rules are applied selectively: in arbitrary places and order.
5. (**Example realisation**) Bird-Meertens formalism: equational reasoning, where rules justify correctness of each step [5].
6. (**Obstacles to adoption**) Few tools, hard to learn, hard to use, hard to understand, hard to maintain, writing the implementation by hand can be easier than writing the derivation.

Benefits of employing a sort of program derivation (more or less formal) for the programmer include: (a) a structured process of obtaining an implementation from specification, (b) greater confidence in the correctness of the implementation, (c) possibility of discovering further optimisations, and finally (d) a framework for a proof of correctness. This last use case could be explained as follows: suppose we can prove correctness of the "specification" program, and that each step preserves the meaning of the program. Then we can show correctness of the "implementation" program.

8.2 Rewrite rules

When equational reasoning is employed, derivation steps can be justified with *rewrite rules*. A basic example of a rewrite rule in functional programming is *map-comp*:

$\forall$ {f g xs} $\rightarrow$ map f (map g xs) $\equiv$ map (f $\circ$ g) xs

where f, g, xs are metavariables. An application of a rewrite rule consists of unifying the LHS of the rule with a subterm of the program (and thus obtaining a substitution σ), and then replacing the subterm with the RHS of the rule, instantiated with the substitution σ.

TODO fix Notably, such simple form of a rewrite rule only supports first-order abstract syntax trees, but not higher-order abstract syntax. (TODO elaborate on what it would mean to have a context and go under a binder in a rewrite rule). (TODO can't express conditions)

8.3 Program derivations in compilers and their limitations

There are program transformations in existing compilers which, other than being compilation phases, have features of program derivation. A good example of this is the support for rewrite rules in GHC, a Haskell compiler. A Haskell programmer may specify a rule like *map-comp* as part of the code, and in one of early compilation passes, GHC will apply the rule wherever possible (i.e. replace the occurrences of the LHS with the RHS). Rewriting in GHC is a compilation phase in the sense that rules are applied in all places they match, but it also resembles derivations since the transformation is guided by input from the programmer, namely, the specified rewrite rules. Note that GHC makes no attempt to ensure that the rules preserves the meaning, or that rewriting would terminate: a programmer could externally check these properties, e.g. using a proof assistant.

Yet another ambiguous situation is where program transformation becomes a search problem. A compiler could try to find a transformation by applying rewrite rules in a selective, non-deterministic manner, and thus perform a search over the space of possible derivations. The search could be guided by an objective function, for example, a sort of static analysis of the running time, or the compiler could evaluate programs by running them on sample inputs. Exploring the space of derivations is the approach taken by the Lift compiler [**?**].

Programmer's input may or may not be involved in such search. For example, the UG4 project delivered a graphical user interface which allows the user to interact with the derivation process.

Unfortunately, derivation search needs a fixed set of rewrite rules, either hardcoded in the compiler or provided by the user. This precudes derivations which use original rewrite rules, coming up with which would require human insight. Section 8.6, which discusses some of the rewrite steps from the UG4 project, has examples of rules which were invented for a specific derivation, and it is difficult to conceive that they could all be provided to the compiler in advance.

8.4 Program derivations in the UG4 project

The UG4 project analyses two instances of program derivation in detail. The first one is a derivation, described in [5], of an efficient implementation of the maximum segment sum problem (MSS). While the input specification (which is also a runnable program) runs in cubic time in the length of the input list, the output program runs in linear time. The asymptotic speed-up is achieved by applying several rewrite rules involving higher-order functions on lists such as map, foldr, and filter.

The second case study involved an original derivation of a program for matrix-vector multiplication. The input program takes a dense matrix, and the output program takes a sparse matrix in the compressed sparse row (CSR) format. Or, to be precise, the input program which acts on a dense matrix, is transformed into a composition of two programs: (a) a conversion from a dense to a CSR-sparse matrix, and (b) a matrix-vector multiplication program which acts on a CSR-sparse matrix. This is because, as a rule, the input and output types of the program must stay the same in the course of the derivation. This second derivation was similarly accomplished with rewrite rules involving higher-order functions.

8.5 Implementation of rewriting in the UG4 project

The UG4 project included a purpose-built framework for specifying derivations. The framework included:

1. A simple functional language with parametric polymorphism. The language is point-free, that is, based on function composition rather than variable binders. This is because variables and abstraction are difficult to implement correctly, as demonstrated by this UG5 project, and even more difficult to rewrite.
2. A type-checker for the language.
3. Rewriting functionality and declaring derivations as sequences of rewrites.
4. An interpreter for the language, which was used to empirically verify claims about performance gains from derivations.

Writing the framework was a good exercise in implementing routine parts of compiler front-ends, such as type checking and unification. Writing it in Scala made sense given the stretch objective of compiling the language to Lift, which was not realised, however.

Rewrite rules were stated without justification, much as postulates in Agda. One could prove the rules externally – but then one is pressed to ask, why not express a derivation in a proof assistant, which supports unification and rewriting natively? Indeed, with hindsight, we can say with certainty that a proof assistant is perfectly suited for the job, its only downside being that it requires considerable expertise, which I did not have during my fourth year. [TODO I/me?] The next section expresses a part of the derivation from the UG4 project in Agda, and evaluates the benefits of doing so, compared with the approach from last year's project.

8.6 Program derivation in Agda: sparse matrix-vector multiplication

To complete last year's work, we conduct a derivation of the program for matrix-vector multiplication which acts on CSR-sparse matrices. Unlike last year, we can now provide proofs of individual rewrite rules. Indeed, some proofs are quite involved. TODO whether and how to do it.

Bibliography

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 27–38. ACM, 2013.

[2] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *PACMPL*, 2(ICFP):90:1–90:30, 2018.

[3] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 195–207. ACM, 2017.

[4] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999.

[5] Jeremy Gibbons. An introduction to the bird- meertens formalism. 1994.

[6] Conor McBride. Type-preserving renaming and substitution. 2005.

[7] Craig McLaughlin, James McKinna, and Ian Stark. Triangulating context lemmas. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 102–114. ACM, 2018.

[8] Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 271–283. ACM Press, 1996.

[9] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992.

[10] Philip Wadler. Programming language foundations in agda. In Tiago Massoni and Mohammad Reza Mousavi, editors, *Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings*, volume 11254 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2018.

Chapter 9

Conclusion

The main deliverables of this project are (a) an elegant representation of a language with closures, (b) a type-preserving closure conversion algorithm which minimises closure environments, and (c) mechanisations of proofs of correctness of closure conversion using two different techniques: bisimulations and logical relations.

This work builds on a long line of research in several areas: higher-order abstract syntax representation, type-preserving compilation, and compiler verification. In particular, the style of the Agda development is influenced by ACMM and PLFA.

By mechanising two different proofs of correctness of the transfomation, the project provides a reference for comparing the methods of bisimulation and logical relations.

It was confirmed that when the languages have a type- and scope-safe representation, a large amount of effort in mechanising meta-theoretical proofs goes into correctness lemmas about interactions between renaming, substitution, and other traversals.

A natural continuation for this project would be to try to apply generic proving techniques, e.g. from AACMM, to prove meta-theoretical lemmas for a family of intermediate languages (syntaxes) all at once. However, this project's insights seem to imply that closure conversion does not fit into the framework provided by AACMM, and that further developments in generic proving would be needed to support closure conversion.

Another possible extension of this work would be to prove correctness properties of more complex languages with features like higher-order functions, polymorphism, abstract data types, recursive types, mutable state and control effects. In that case, however, use of a proof language with tactics and automation would be more appropriate, as scaling manual proof techniques to complex languages is known to be extremely tedious.

In summary, the objectives of the project were met, if slightly altered. Just one compilation phase was implemented, but it was proved correct with two different methods. The generic proving techniques from ACMM, although not ported to our representation of simply typed lambda calculus, provided a basis for postulating correctness lemmas about STLC. Using generic programming solutions from AACMM was beyond the scope of this project, but like it was mentioned multiple times, they would be inadequate for a languauge with closures anyway.

Chapter 10

Technical appendix

10.1 Minimising closure conversion and the compatibility relation

Below is the Agda proof that the graph relation of the minimising closure conversion function is contained in the compatibility relation.

```
_† : ∀ {Γ A} → S.Lam A Γ → T.Lam A Γ
M † with cc M
M † | ∃[ Δ ] Δ⊆Γ ∧ N = T.rename (⊆→ρ Δ⊆Γ) N


{-# TERMINATING #-}
undo : ∀ {Γ A} → T.Lam A Γ → S.Lam A Γ
undo (T.V x)       = S.V x
undo (T.A M N)     = S.A (undo M) (undo N)
undo (T.L M E)     = S.L (undo (T.subst (T.exts E) M))

{-# TERMINATING #-}
undo-compat : ∀ {Γ σ} (N : T.Lam σ Γ) → undo N ~ N
undo-compat (T.V x)       = ~V
undo-compat (T.A M N)     = ~A (undo-compat M) (undo-compat N)
undo-compat (T.L N E)     = ~L (undo-compat _)


helper-2 : ∀ {Γ A} (x : Var A Γ)
  → lookup (⊆→ρ (Var→⊆ x)) z ≡ x
helper-2 z = refl
helper-2 (s x) = cong s (helper-2 x)

helper-3 : ∀ {Δ₁ Γ₁ Γ} (Δ₁⊆Γ₁ : Δ₁ ⊆ Γ₁) (Γ₁⊆Γ : Γ₁ ⊆ Γ)
  → select (⊆→ρ Δ₁⊆Γ₁) (⊆→ρ Γ₁⊆Γ) ≡ᴱ ⊆→ρ (⊆-trans Δ₁⊆Γ₁ Γ₁⊆Γ)
eq (helper-3 base base) ()
eq (helper-3 Δ₁⊆Γ₁ (skip Γ₁⊆Γ)) x
  = cong s (eq (helper-3 Δ₁⊆Γ₁ Γ₁⊆Γ) x)
eq (helper-3 (skip Δ₁⊆Γ₁) (keep Γ₁⊆Γ)) x
  = cong s (eq (helper-3 Δ₁⊆Γ₁ Γ₁⊆Γ) x)
eq (helper-3 (keep Δ₁⊆Γ₁) (keep Γ₁⊆Γ)) z
  = refl
eq (helper-3 (keep Δ₁⊆Γ₁) (keep Γ₁⊆Γ)) (s x)
  = cong s (eq (helper-3 Δ₁⊆Γ₁ Γ₁⊆Γ) x)

helper-4 : ∀ {Δ₁ Γ₁ Γ τ}
  (Δ₁⊆Γ₁ : Δ₁ ⊆ Γ₁) (Γ₁⊆Γ : Γ₁ ⊆ Γ)
  (Δ₁⊆Γ : Δ₁ ⊆ Γ) (M† : T.Lam τ Δ₁)
  → ⊆-trans Δ₁⊆Γ₁ Γ₁⊆Γ ≡ Δ₁⊆Γ
  →   T.rename (⊆→ρ Γ₁⊆Γ) (T.rename (⊆→ρ Δ₁⊆Γ₁) M†)
      ≡ T.rename (⊆→ρ Δ₁⊆Γ) M†
helper-4 Δ₁⊆Γ₁ Γ₁⊆Γ Δ₁⊆Γ M† well =
  begin
      T.rename (⊆→ρ Γ₁⊆Γ) (T.rename (⊆→ρ Δ₁⊆Γ₁) M†)
  ≡⟨ rename∘rename (⊆→ρ Δ₁⊆Γ₁) (⊆→ρ Γ₁⊆Γ) M† ⟩
      T.rename (select (⊆→ρ Δ₁⊆Γ₁) (⊆→ρ Γ₁⊆Γ)) M†
  ≡⟨ cong   (λ e → T.rename e M†)
            (env-extensionality (helper-3 Δ₁⊆Γ₁ Γ₁⊆Γ)) ⟩
      T.rename (⊆→ρ (⊆-trans Δ₁⊆Γ₁ Γ₁⊆Γ)) M†
  ≡⟨ cong (λ e → T.rename (⊆→ρ e) M†) well ⟩
      T.rename (⊆→ρ Δ₁⊆Γ) M†
  ∎

{-# TERMINATING #-}
helper-5 : ∀ {Γ Δ σ τ} (Δ⊆Γ : Δ ⊆ Γ) (N : T.Lam σ (τ :: Δ))
  →   T.subst (T.exts (T.rename (⊆→ρ Δ⊆Γ) <$> T.id-subst)) N
      ≡ T.rename (⊆→ρ (keep Δ⊆Γ)) N
helper-5 Δ⊆Γ (T.V x) with x
helper-5 Δ⊆Γ (T.V x) | z = refl
helper-5 Δ⊆Γ (T.V x) | s x’ = refl
helper-5 Δ⊆Γ (T.A M N)
  = cong₂ T.A (helper-5 Δ⊆Γ M) (helper-5 Δ⊆Γ N)
helper-5 Δ⊆Γ (T.L N E) = cong (T.L N) h
  where
  h :   T.subst (T.exts (T.rename (⊆→ρ Δ⊆Γ) <$> T.id-subst)) <$> E
          ≡ _<$>_ {𝒲 = T.Lam} (T.rename (⊆→ρ (keep Δ⊆Γ))) E
  h   =
      begin
        T.subst (T.exts (T.rename (⊆→ρ Δ⊆Γ) <$> T.id-subst)) <$> E
      ≡⟨ env-extensionality (<$>-fun (helper-5 Δ⊆Γ) E) ⟩
        _<$>_ {𝒲 = T.Lam} (T.rename (⊆→ρ (keep Δ⊆Γ))) E
      ∎
```

```
N~N† : ∀ {Γ A} (N : S.Lam A Γ)
  → N ~ N †


N~N† (S.V x) with cc (S.V x)
N~N† (S.V x) | ∃[ Δ ] Δ⊆Γ ∧ N rewrite helper-2 x = ~V
N~N† (S.A M N) with cc M | cc N | inspect _† M | inspect _† N
N~N† (S.A M N) | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M† | ∃[ Δ₂ ] Δ₂⊆Γ ∧ N†
  | [ p ] | [ q ] with merge Δ₁⊆Γ Δ₂⊆Γ
N~N† (S.A M N) | ∃[ Δ₁ ] Δ₁⊆Γ ∧ M† | ∃[ Δ₂ ] Δ₂⊆Γ ∧ N†
  | [ p ] | [ q ] | subListSum Γ₁ Γ₁⊆Γ Δ₁⊆Γ₁ Δ₂⊆Γ₁ well well₁
  rewrite helper-4 Δ₁⊆Γ₁ Γ₁⊆Γ Δ₁⊆Γ M† well
  | helper-4 Δ₂⊆Γ₁ Γ₁⊆Γ Δ₂⊆Γ N† well₁ | sym p | sym q
  = ~A (N~N† M) (N~N† N)
N~N† (S.L N) with cc N | inspect _† N
N~N† (S.L N) | ∃[ Δ ] Δ⊆Γ ∧ N’ | [ p ]
  with adjust-context Δ⊆Γ
N~N† (S.L N) | ∃[ Δ ] Δ⊆Γ ∧ N’ | [ p ]
  | adjust Δ₁ Δ₁⊆Γ Δ⊆AΔ₁ well = ~L g
  where
  h :   T.subst (T.exts (T.rename (⊆→ρ Δ₁⊆Γ) <$> T.id-subst))
          (T.rename (⊆→ρ Δ⊆AΔ₁) N’) ≡ T.rename (⊆→ρ Δ⊆Γ) N’
  h   =
      begin
        T.subst (T.exts (T.rename (⊆→ρ Δ₁⊆Γ) <$> T.id-subst))
          (T.rename (⊆→ρ Δ⊆AΔ₁) N’)
      ≡⟨ helper-5 Δ₁⊆Γ (T.rename (⊆→ρ Δ⊆AΔ₁) N’) ⟩
        T.rename (⊆→ρ (keep Δ₁⊆Γ)) (T.rename (⊆→ρ Δ⊆AΔ₁) N’)
      ≡⟨ rename∘rename (⊆→ρ Δ⊆AΔ₁) (⊆→ρ (keep Δ₁⊆Γ)) N’ ⟩
        T.rename (select (⊆→ρ Δ⊆AΔ₁) (⊆→ρ (keep Δ₁⊆Γ))) N’
      ≡⟨   cong (λ e → T.rename e N’)
             (env-extensionality (helper-3 Δ⊆AΔ₁ (keep Δ₁⊆Γ))) ⟩
        T.rename (⊆→ρ (⊆-trans Δ⊆AΔ₁ (keep Δ₁⊆Γ))) N’
      ≡⟨ cong (λ e → T.rename (⊆→ρ e) N’) (sym well) ⟩
        T.rename (⊆→ρ Δ⊆Γ) N’
      ∎
  g : N ~ T.subst   (T.exts (T.rename (⊆→ρ Δ₁⊆Γ) <$> T.id-subst))
                    (T.rename (⊆→ρ Δ⊆AΔ₁) N’)
  g rewrite h | sym p = N~N† N
```

10.2 Compatibility relation is a bisimulation

```
st-sim : ST-Simulation _~_
st-sim ~V ()
```

```
st-sim (~L ~N) ()
st-sim (~A ~M ~N) (S.ξ-A₁ M—→)
  with st-sim ~M M—→
... | _ , ~M’ , M†—→ = _ , ~A ~M’ ~N , T.ξ-A₁ M†—→
st-sim (~A ~M ~N) (S.ξ-A₂ VV N—→)
  with st-sim ~N N—→
... | _ , ~N′ , N†—→ = _ , ~A ~M ~N′ , T.ξ-A₂ (~val ~M VV) N†—→
st-sim (~A (~L {N = N}  ~N) ~VV) (S.β-L VV)
  = _ , /V≡E•V† {N = N}  ~N ~VV , T.β-L (~val ~VV VV)

ts-sim : TS-Simulation _~_
ts-sim ~V ()
ts-sim (~L ~N) ()
ts-sim (~A ~M ~N) (T.ξ-A₁ M†—→) with ts-sim ~M M†—→
... | _ , ~M’ , M—→ = _ , ~A ~M’ ~N , S.ξ-A₁ M—→
ts-sim (~A ~M ~N) (T.ξ-A₂ VV† N†—→) with ts-sim ~N N†—→
... | _ , ~N’ , N—→ = _ , ~A ~M ~N’ , S.ξ-A₂ (~ts-val ~M VV†) N—→
ts-sim (~A (~L {N = N}  ~N) ~VV) (T.β-L VV†)
  = _ , /V≡E•V† {N = N}  ~N ~VV , S.β-L (~ts-val ~VV VV†)


bisim : Bisimulation _~_
bisim = st-sim , ts-sim
```