Type-preserving closure conversion of PCF in Agda (more to come)

Piotr Jander

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

2019

Abstract

This is an example of `infthesis` style. The file `skeleton.tex` generates this document and can be used to get a “skeleton” for your thesis. The abstract should summarise your report and fit in the space on the first page. You may, of course, use any other software to write your report, as long as you follow the same style. That means: producing a title page as given here, and including a table of contents and bibliography.

Acknowledgements

Acknowledgements go here.

Table of Contents

1 Introduction **7**
- 1.1 Using Sections . . . 7
- 1.2 Citations . . . 8
- 1.3 Options . . . 8

2 Related literature **9**
- 2.1 Closure conversion . . . 9
- 2.2 Verified compilation . . . 10

3 Background **11**
- 3.1 Closure conversion . . . 11
- 3.2 Compilation phases and intermediate representations . . . 11
- 3.3 Type- and scope-safe representation of simply typed lambda calculus λst 12
- 3.4 Type- and scope-safe programs . . . 14
- 3.5 Small-step operational semantics . . . 15
- 3.6 TODO ACMM and semantics . . . 16
- 3.7 TODO ACMM and synchronisation lemmas . . . 16
- 3.8 TODO ACMM and fusion lemmas . . . 16

4 TODO Target language **17**
- 4.1 Renaming and substitution . . . 17

5 TODO Minimising closure conversion **19**

6 Bisimulation **21**
- 6.1 Similarity relation . . . 21
- 6.2 Bisimulation . . . 23
- 6.3 Fusion lemmas for the closure language λT . . . 25
- 6.4 Back to ~rename and ~subst . . . 27

7 Relationship to the UG4 project **31**
- 7.1 Program transformation vs derivation . . . 31
- 7.2 Program derivations in the UG4 project . . . 32
- 7.3 Rewrite rules . . . 32
- 7.4 Program derivations in compilers and their limitations . . . 33
- 7.5 Implementation of rewriting in the UG4 project . . . 33
- 7.6 Program derivation in Agda: sparse matrix-vector multiplication . . . 34

Bibliography **35**

Chapter 1

Introduction

The document structure should include:

- The title page in the format used above.
- An optional acknowledgements page.
- The table of contents.
- The report text divided into chapters as appropriate.
- The bibliography.

Commands for generating the title page appear in the skeleton file and are self explanatory. The file also includes commands to choose your report type (project report, thesis or dissertation) and degree. These will be placed in the appropriate place in the title page.

The default behaviour of the documentclass is to produce documents typeset in 12 point. Regardless of the formatting system you use, it is recommended that you submit your thesis printed (or copied) double sided.

The report should be printed single-spaced. It should be 30 to 60 pages long, and preferably no shorter than 20 pages. Appendices are in addition to this and you should place detail here which may be too much or not strictly necessary when reading the relevant section.

1.1 Using Sections

Divide your chapters into sub-parts as appropriate.

1.2 Citations

Note that citations (like [**?**] or [**?**]) can be generated using `BibTeX` or by using the `thebibliography` environment. This makes sure that the table of contents includes an entry for the bibliography. Of course you may use any other method as well.

1.3 Options

There are various documentclass options, see the documentation. Here we are using an option (`bsc` or `minf`) to choose the degree type, plus:

- `frontabs` (recommended) to put the abstract on the front page;
- `twoside` (recommended) to format for two-sided printing, with each chapter starting on a right-hand page;
- `singlespacing` (required) for single-spaced formating; and
- `parskip` (a matter of taste) which alters the paragraph formatting so that paragraphs are separated by a vertical space, and there is no indentation at the start of each paragraph.

Chapter 2

Related literature

2.1 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?

2.2 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 type preservation or 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.

The first 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?).

A final 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.

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]

3.2.0.0.1 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.

3.2.0.0.2 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) (without existential types). Then the target representation cannot be the same (STLC), as STLC does not have existential types, and closure environments must be existentially typed in order for programs to be well-typed in general, as discussed in [section]. This is why an intermediate language with existential types is necessary.

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.

3.2.0.0.3 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 STLC or 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 [6].

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 $\mathcal{V}$ (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).

```
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)
```

Notice that those three 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:

```
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 (s <$> ρ • z) N)
rename ρ (A M N)     =   A (rename ρ M) (rename ρ N)
```

```
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 (rename (pack s) <$> σ • V z) 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 [5], 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.

To see the usefulness of simultaneous renaming and substitution, consider that once an identity substitution is defined (one which leaves its argument unchanged):

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

Then defining a single substitution is simple. (A single substitution replaces occurrences of the last-bound variable in the context, and it is useful for defining the beta reduction for abstractions).

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

3.5 Small-step operational semantics

The formalisation of small step semantics for a call-by-value lambda calculus is adapted from [6].

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' and 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. [6].

3.6 TODO ACMM and semantics

3.7 TODO ACMM and synchronisation lemmas

3.8 TODO ACMM and fusion lemmas

Chapter 4

TODO Target language

4.1 Renaming and substitution

Chapter 5

TODO Minimising closure conversion

Chapter 6

Bisimulation

In the previous chapters, we defined the source and target languages of the closure conversion, together with reduction rules for each, and a translation function from source to target.

Our implementation of closure conversion is type and scope-preserving by construction. The property of type preservation would be considered a strong indication of correctness in a real-world compiler, but in this theoretical development which deals with a small, toy language, we prove stronger correctness properties which speak about operation correctness.

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

6.1 Similarity relation

Before we can precisely define bisimulation, we need a definition of similarity between source and target terms of closure conversion.

```
infix   4 _~_
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

  ~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

  ~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†
```

Definition. Given a source language term M and a target language term M†, the similarity 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 S.· N ~M† T.· N†.
- (Abstraction) If N ~T.subst (T.exts E) N†, then ƛ N ~⟨⟨ N† , E ⟩⟩.

We unpack the definition here. Recall that in our definition of closure conversion, source and target languages share the same (meta) type of (object) types, contexts, and variables (proofs of context membership). In fact, similarity is only defined for source and target terms of the same type in the same context (this is explicit in the Agda definition).

Therefore, similarity of (syntactic) variables can be defined in terms of identity of proofs of membership.

Similarity of applications is defined by congruence.

Finally, the non-trivial case of abstractions. What are the necessary conditions for ƛ N ~⟨⟨ N† , E ⟩⟩, where ƛ N and ⟨⟨ N† , E ⟩⟩ are defined in context Γ? Clearly, we cannot require that N ~N†, as the context Δ in which the closure body is defined is existentially quantified. However, recall that the closure environment E is defined as a substitution from Δ to Γ. Applying this substitution to the closure body (T.subst (T.exts E) N†) results in a term in Γ which can be in a similarity relation with N, and this is precisely what we require in the definition.

(Note: the exts accounts for the fact that the closure body is defined in the context Δ extended by the variable bound by the abstraction, similarly to the lambda body.)

It is natural to ask what the relationship between a closure conversion function and the similarity relation. Is the similarity relation as graph relation of a closure conversion function? It is not. Recall that closure conversion can be implemented by any function which takes source terms to target terms and preserves the type and context. But an implementation has freedom in how it deals with closure environments; the meta language type of closures only requires that the environment is *some* substitution from the closure body context Δ to the outer context Γ.

For example, the closure conversion transformation we described in Chapter ??? had the property that closure environments were minimal: they only contained parts of context actually used by the closure body. In contrast, the simplest possible closure conversion could use identity environments:

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

We require that for every well-behaved closure conversion function c, any source term N is similar to the result of its translation with c: N ~c N. This is indeed the case for the convert function. The proof is by straightforward induction; in the abstraction case, we need to argue that applying an identity substitution leaves the term unchanged.

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

A similar result could be obtained for the closure conversion algorithm which minimises environments from Chapter ???, but the proof would be longer.

However, given M ~M†, it is not necessarily the case that M† ≡ c M for any *fixed* function c; instead, M† ≡ c M holds for *some* function c. Therefore, the similarity relation is not the graph relation of any *specific* conversion c.

Having defined the notion of similarity, we are in a position to define bisimulation.

6.2 Bisimulation

Bisimulation is a two-way property which is defined in terms of a simpler one-way property of simulation.
Definition. Given two languages S and T and a similarity relation _~_ between them, S and T are in **simulation** if and only if the following holds: Given source language terms M and N, and a target language term M† such that M reduces to N in a single step

(M —→ N) and M is similar to M† (M M̃†), there exists a target language term N† such that M† reduces to N† in some number of steps (M† —→* N†) and N is similar to N† (N Ñ†).

Definition. Given two languages S and T, S and T are in a **bisimulation** if and only if S is in a simulation with T and T is in a simulation with S.

The essence of simulation can be captured in a diagram.

TODO diagram here

TODO give names to the source and target langs

In fact, our source and target languages of closure conversion have a stronger property: *lock-step* bisimulation, which is defined in terms of *lock-step* simulations. A lock-step simulation is one where for each reduction step of the source term, there is exactly one corresponding reduction step in the target language. We illustrate this at another diagram:

TODO another diagram

Before we can prove that λ and λT are in simulation, we need three lemmas:

1. Values commute with similarity. If M ~M† and M is a value, then M† is also a value.
2. Renaming commutes with similarity. If ρ is a renaming from Γ to Δ, and M ~M† are similar terms in the context Γ, then the results of renaming M and M† with ρ are also similar: S.rename ρ M ~T.rename ρ M†.
3. Substitution commutes with similarity. 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 given similar terms M ~M† in Γ, the results of applying ρ to M and ρ† to M† are also similar: S.subst ρ M ~T.subst ρ† M†.

The proof that values commute with similarity is straightforward.

```
~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) ()
```

Before we will be able to prove the lemmas about renaming and substitution, we need an interlude where we discuss so-called fusion lemmas for the closure language λT.

TODO acknowledge PLFA here

6.3 Fusion lemmas for the closure language λT

When studying the meta-theory of a calculus, one systematically needs to prove fusion lemmas for various semantics (TODO wording) (recall that a semantic is a traversal of a term). A fusion lemma relates three semantics: the pair we sequence and their sequential composition (TODO wording). In our proof of bimisulation for closure conversion, we have interactions of two semantics: renaming and substitution. In fact, we need fusion lemmas for all four combinations of them:

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

As it turns out, the first composition (a renaming followed by a renaming) is equivalent to a renaming, and the other three compositions are equivalent to substitutions.

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

```
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
```

The Agda proofs of those four lemmas can be found in the appendix (TODO ref); here we outline the proof structure, analyse one of the four lemmas, and compare fusion lemmas for $\lambda^{cl}$with the corresponding lemmas for STLC.

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 logical relations (TODO explain) 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 (semantics) 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 λ^{cl}‘do not go under binders‘, as can be seen from their definitions:

```
Subst Γ Δ = (Γ –Env) Lam Δ

Syntactic : Context → Context → Set
Syntactic Γ Δ = ∀  → Lam σ Γ → Lam σ Δ


rename   : ∀ {Γ Δ}
           →   Thinning Γ Δ
               ---------------------------
           → Syntactic Γ 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}
  →   Subst Γ Δ
      ----------------
  → Syntactic Γ 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)
```

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 for a renaming followed by a subsitution:

```
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 congruence (TODO wording).
- In the closure case, the proof is also by congruence, but an equational proof is required to show that the LHS and RHS act the same of the environment E.

The equational proof for E 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 < $>-distr,
2. It uses the fact that when f and g are extensionally equal (∀ {x} → f x ≡ g x), then f <$> E ≡ g <$> E which is captured by the lemma <$>-fun,
3. <$>-fun is instantiated with the inductive hypothesis.

Unfortunately, Agda does not recognise our fusion lemmas as terminating, and we were unable to provide a termination proof. Still, we believe that the proof function is in fact terminating.

6.4 Back to ~rename and ~subst

Recall the two result which we said would be needed for showing bisimulation. We start with:

Renaming commutes with similarity. If ρ is a renaming from Γ to Δ, and M ~M† are similar terms in the context Γ, then the results of renaming M and M† with ρ are also similar: S.rename ρ M ~T.rename ρ M†.

```
~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 proof is by induction on the similarity relation. The variable and application cases are easy, but as ever, the abstraction case is worth looking at. Recall that a source abstraction is similar to the target closure S.L N ~T.L N† E when N ~T.subst (T.exts E) N† by the inductive constructor ~L.

In the abstraction case, we have that

S.L N ~T.L N† E (1)

We need to show that

S.rename ρ (S.L N) ~T.rename ρ (T.L N† E) (2)

but (2) simplifies to

S.L (S.rename (S.exts ρ) N) ~T.L N† (T.rename ρ <$> E) (3)

which holds by ~L when the following holds

S.rename (S.exts ρ) N ~T.subst (T.exts (T.rename ρ <$> E)) N† (4)

On the other hand, from (1) by ~L we have that

N ~T.subst (T.exts E) N† (5)

Applying the induction hypotesis to (5), we get

S.rename (S.exts ρ) N ~T.rename (S.exts ρ) (T.subst (T.exts E) N†) (6)

Thus we require (4) and have (6), so to complete the proof, we need to show that

T.subst (T.exts (T.rename ρ <$> E)) N† ≡ T.rename (S.exts ρ) (T.subst (T.exts E) N†)

or, in Agda

```
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
```

This indeed holds, and the proof uses the fusion lemma lemmas for renaming and substitution in several places.

The remaining result to prove is quite similar, but the concept of a pointwise similar substitution makes it worth analysing.

Substitution commutes with similarity. 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 given similar terms M ~M† in Γ, the results of applying ρ to M and ρ† to M† are also similar: S.subst ρ M ~T.subst ρ† M†.

The notion of pointwise-similar substitutions ρ and ρ† from Γ to Δ can be captured by a function which, for each variables x in Γ, produces a proof that that the corresponsing terms are similar: `lookup ρ x ~lookup ρ† x`. We encapsulate this function in an Agda record:

```
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
```

The notion of a pointwise relation between environments (like substitutions) is used by ACMM to prove synchronisation and fusion lemmas for STLC. Unlike for STLC, the closure language λ^{cl}does not require us to prove that pointwise similarity is preserved as a traversal goes under a binder.

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

```
~exts : ∀ {Γ Δ σ} {ρ   : S.Subst Γ Δ} {ρ† : T.Subst Γ Δ}
  →   ρ ~σ ρ†
      ------------------------------
  → S.exts {σ = 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}
  → {ρ   : S.Subst Γ Δ}
  → {ρ† : T.Subst Γ Δ}
  →   ρ ~σ ρ†
      ---------------------------------------------------------
  → (∀  {M : S.Lam τ Γ} {M† : T.Lam τ Γ} → 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
```

TODO finish

Chapter 7

Relationship to the UG4 project

This work is a natural continuation of the UG4 project, but it admittedly takes the project in a new direction. In terms of its goals, the UG4 project was concerned with program derivations. Derivations are distinct from program transformations or traversals, as found in compilers or in this UG5 project.

Program transformations like closure conversion, or continuation passing style (CPS) transformation, have two distinct characteristics.

7.1 Program transformation vs derivation

Firstly, the source and target languages can be different, e.g. the source language may have lambda abstractions with free variables, and the target language may have closures with environments. Of course, many transformations in compilers happen within the same language, e.g. constant expression folding.

Secondly, compiler transformations are usually characterised by replacing one program construct with another, in a one-way fashion, e.g. lambda abstractions with closures. It would be strange if changes happened both ways. One might imagine a language with both lambda abstractions and closures, and a transformation which replaces some closures with lambda abstractions, and some abstraction with closures, according to arbitrary rules. It is difficult to see how such a transformation would be useful in a compiler.

Of course, one might imagine a transformation which replaces some occurrences of constructs A and constructs B, and vice versa, in order to optimise the program. But any such optimising transformation is still guided by some measure of performance.

In contrast, program derivation consists of transforming a program in arbitrary places, and using arbitrary rules, with a specific goal of obtaining one program from another, so that the obtained program has some desirable features, like efficiency, or even faster asymptotic running time. Importantly, the derivation happens within a single language.

(TODO but specification can be in terms of relations, which are not part of the language).

7.2 Program derivations in the UG4 project

The UG4 project analyses two instances of program derivation in detail. The first one is a derivation 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 a derivation for 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.

7.3 Rewrite rules

The notion of a rewrite rule is central to program derivation. A classical example of a rewrite rule for a function program is one stating: "a composition of mappings is a mapping with a composition":

```
forall f g xs. map f (map g xs) $=\!=$ map (f . 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.

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)

In their simplest form, a program derivation is a sequence of intermediate forms of the program, intertwined with rewrite rules which justify each step of the derivation. For examples, the reader may consult the UG4 project [**?**].

7.4 Program derivations in compilers and their limitations

While we argued that compiler transformations and program derivations are distinct in their character, the lines may arguably be blurry at times. A good example of this is the support for rewrite rules in GHC, a Haskell compiler. A Haskell programmer may specify a rule like (*) 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). Such compiler pass may be considered an instance of program derivation, and it would go some way towards deriving the aforementioned efficient implementation for the maximum segment sum problem (MSS).

However, rewriting as implemented in compilers is too limited to carry out most derivations. To see one limitation, consider a derivation which needs to apply the rule (*) right to left: replace an occurrence of the RHS with the LHS. But clearly, unguided application of rewrite rules can only be one way, otherwise it would not terminate.

Another limitation is the fact that "the right" derivation can require that rules be applied in a specific order. Thus, rewriting a program becomes a search problem, where rewrites are applied until a program satisfying some (performance) objective is found. This is the approach taken by the Lift compiler [**?**]

Yet another limitation is that many derivations elude the notion of an objective function, thus rendering a search futile. It seems that human insight is required to guide a derivation.

A final limitation is that there are conceivable rewrite rules which are only applicable when certain conditions are met. Such conditions could range algebraic properties of operators to general predicates and relations on terms. And these are undecidable in general (TODO how to phrase this).

These considerations, taken together, mean that the technique of program derivation is more useful to the programmer than the compiler. 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 about correctness of an 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 the correctness of the "specification" program, and the correctness of each rewrite rule. Then we can obtain correctness of the "implementation" program.

7.5 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 parametricity. The language is point-free, that is, based on function composition rather than on lambda abstractions. This was 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.

Importantly, 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?)

7.6 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] Conor McBride. Type-preserving renaming and substitution. 2005.

[6] 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.