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.

Contents

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 [1] or [2]) 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

The Real Thing

Chapter 3

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.

TODO we or passive voice?

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.

3.1 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 λ 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 λ N ~⟨⟨ N† , E ⟩⟩, where λ 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.

3.2 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

3.3 Fusion lemmas for the closure language λT

foo

3.4 Back to ~rename and ~subst

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

infix 3 _~σ_
record _~σ_ {Γ Δ : Context} (ρ : S.Subst Γ Δ) (ρ† : T.Subst Γ Δ) : Set where
  field ρ~ρ† : ∀ {σ} → (x : Var σ Γ) → lookup ρ x ~ lookup ρ† x
open _~σ_ public

~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†} ~ρ) (s x) = ~rename E.extend (ρ~ρ† ~ρ x)

~id-subst : ∀ {Γ} → S.id-subst {Γ} ~σ T.id-subst {Γ}
ρ~ρ† ~id-subst x = ~V

_~•_ : ∀ {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

~subst : ∀ {Γ Δ}
```

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

/V≡E•V† : ∀ {Γ Δ σ τ}
      {N : S.Lam τ (σ :: Γ)} {N† : T.Lam τ (σ :: Δ)} {E : T.Subst Δ Γ}
      {V : S.Lam σ Γ} {V† : T.Lam σ Γ}
    → N ~ T.subst (T.exts E) N†
    → V ~ V†
      --------------------------
    → N / V ~ T.subst (E • V†) N†
/V≡E•V† {N = N} {N†} {E} {VV} {V†} ~N ~VV
  rewrite cong (λ e → (N / VV) ~ e) (sym (TT.subst-E•V N† E V†))
  = ~subst (~id-subst ~• ~VV) ~N

data Leg {Γ σ} (M† : T.Lam σ Γ) (N : S.Lam σ Γ) : Set where

  leg : ∀ {N† : T.Lam σ Γ}
      → N ~ N†
      → M† T.—→ N†
        --------
      → Leg M† N

sim : ∀ {Γ σ} {M N : S.Lam σ Γ} {M† : T.Lam σ Γ}
    → M ~ M†
    → M S.—→ N
      ---------
    → Leg M† N
sim ~V ()
sim (~L ~N) ()
sim (~A ~M ~N) (S.ξ-A₁ M—→)
  with sim ~M M—→
... | leg ~M' M†—→ = leg (~A ~M' ~N) (T.ξ-A₁ M†—→)
sim (~A ~M ~N) (S.ξ-A₂ VV N—→)
  with sim ~N N—→
... | leg ~N′ N†—→ = leg (~A ~M ~N′) (T.ξ-A₂ (~val ~M VV) N†—→)
sim (~A (~L {N = N} {N†} ~N) ~VV) (S.β-L VV)
  = leg (/V≡E•V† {N = N} {N†} ~N ~VV) (T.β-L (~val ~VV VV))
```

Of course you may want to use several chapters and much more text than here.

Bibliography

[1] Hiroki Arimura. Learning acyclic first-order horn sentences from entailment. In *Proc. of the 8th Intl. Conf. on Algorithmic Learning Theory, ALT '97*, pages 432–445, 1997.

[2] Chen-Chung Chang and H. Jerome Keisler. *Model Theory*. North-Holland, third edition, 1990.