System F in Agda via Pitts

Charlotte Ausel

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

2024

Abstract

This skeleton demonstrates how to use the `infthesis` style for undergraduate dissertations in the School of Informatics. It also emphasises the page limit, and that you must not deviate from the required style. The file `skeleton.tex` generates this document and should be used as a starting point for your thesis. Replace this abstract text with a concise summary of your report.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Charlotte Ausel*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

1 Introduction **1**

2 Background **2**
2.1 Agda 2
2.2 The λ-Calculus and System F 3
2.3 De Bruijn indices and Locally Nameless Sets 5
2.4 Prior research 7

3 Conclusions **8**
3.1 Final Reminder 8

Bibliography **9**

A First appendix **11**
A.1 First section 11

B Participants' information sheet **12**

C Participants' consent form **13**

Chapter 1

Introduction

TODO

Chapter 2

Background

2.1 Agda

Agda is a dependently-typed functional programming language, which makes it suitable as a proof-assistant for intuitionistic logic (Norell, 2007), similar to other such proof-assistants like Coq or Lean. Its syntax is very similar to Haskell, and in fact, the two are closely related; the Agda compiler is a transpiler to Haskell, and the Haskell standard library can be used in Agda (Kusee, 2017). Yet, Agda has stricter limitations on recursive functions and some other such language features which, if included, would make it harder to reason about proofs (Bove et al., 2009).

Agda most commonly uses inductive definitions. For example, following the Peano axioms for the natural numbers $\mathbb{N}$ (Boolos, 1995), we may define them like so.

```
module dissertation where
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Taking advantage of the Curry-Howard correspondence, a proof in Agda is simply a function with an appropriate type signature and function body (Wadler, 2015). So, a proof that addition is associative would use recursion, which corresponds to induction, as shown below.

```
open import Relation.Binary.PropositionalEquality
  using (_≡_; refl; cong)
open import Data.Nat using (ℕ; zero; suc; _+_)

+-assoc : ∀ (m n p : ℕ) → (m + n) + p ≡ m + (n + p)
+-assoc zero    n p = refl
+-assoc (suc m) n p = cong suc (+-assoc m n p)
```

2.2 The λ-Calculus and System F

The λ-calculus. The λ-calculus (pronounced *lambda calculus*), is a theoretical model of computation developed by Alonzo Church in the 1930s (first described in Church (1932)) and is what System F is based upon. It looks and works similarly to familiar functional programming languages, yet its definition is as minimal as possible while still being Turing-complete[1]. As described in Pierce (2002), the most relevant results are summarised below.

We have the following familiar BNF grammar for any λ-calculus term t.

$$\begin{array}{rcll} t & ::= & x & \text{variables} \\ & | & \lambda x.t & (\lambda)\text{-abstractions} \\ & | & t_1 t_2 & \text{function application} \end{array}$$

Function application is left-associative (so for all abstractions f, $fab = (fa)b$), and λ-abstractions extend as far as possible (e.g. $\lambda f.fab = \lambda f.(fab)$).

The λ-calculus includes three reductions. α-conversion is the renaming of variables such that the semantics of the program are not changed; terms which are semantically identical but use different variable bindings are called α-equivalent, and α-equivalence is an equivalence-relation. β-reduction is how functions are applied; for any term t, we write a reduction as $t \mapsto_\beta t'$. For any function application $(\lambda x.t)u$, we replace all occurrences of x in t with u. β-reduction is a congruence, so if $t \mapsto_\beta t'$, then $st \mapsto_\beta st'$, and $ts \mapsto_\beta t's$.

Finally, we have η-reduction; for all λ-abstractions $\lambda x.fx$, we have that $\lambda x.fx \mapsto_\eta f$ (this is the idea of function extensionality).

If further β-reducations do not simplify a term, we say that it is in its *normal form*. We shall represent an arbitrary number of successive β-reductions as $\mapsto_{\beta^\star}$.

Lastly, the Church-Rosser theorem states that for any term t, if it β-reduces to two terms a and b, then there exists a common term t' which both a and b eventually β-reduce to (Church and Rosser, 1936).

Notably, some terms may not have a normal form. A famous example is *omega* (sometimes called the *omega combinator*) defined as $(\lambda x.(xx))(\lambda x.(xx))$, which when applied to itself doesn't reduce any further.

$$(\lambda x.(xx))(\lambda x.(xx)) \quad \mapsto_\beta \quad (\lambda x.(xx))(\lambda x.(xx))$$

Just the first $\lambda x.(xx)$ part on its own is called *little omega*. Another famous example is the fixpoint called the *y-combinator*[2]. Given any argument, it will β-reduce to the argument applied to itself.

We say that these terms *do not have a normal form*.

[1] Although Turing machines would be invented after the λ-calculus, 'turing-complete' has become a shorthand for 'universal method of computation'. Such a universal method was not Church's initial goal, but is why we're still interested in the λ-calculus.

[2] Defined as $\mathcal{Y} \triangleq (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$

The simply-typed λ-calculus. While not being Turing-complete, the simply-typed λ-calculus (STLC for short and sometimes given the symbol $\lambda^{\rightarrow}$) is an extension of the untyped λ-calculus described above that was developed in Church (1940) which requires each term to have a *type*. Terms which do not have a normal form cannot be given a type. As such, we only permit 'nice' expressions, that is, all expressions will (after successive β-reductions) result in an irreducible expression—their normal form. This is also called *strong normalisation*. (Pierce, 2002)

Since we can no longer represent all of the terms of the untyped λ-calculus, the STLC is not Turing-complete. Nevertheless, the STLC is still useful, since we can guarantee that any term which can be given a type will have a normal form. Having a normal form is the λ-calculus equivalent of a Turing machine encoding halting. The popular saying goes, 'well-typed programs can't go wrong!' (Milner, 1978)

The base types we will commonly be using are the booleans $\mathbb{B}$ and the naturals $\mathbb{N}$ (we could use just one, but using two will make the examples easier to understand)[3]. We shall call the set of base types T, so in our case, $T = \{\mathbb{B}, \mathbb{N}\}$. The set of variables will be V. Our *type context* (also called *type environment*) will be given the symbol Γ, which is a map from variables to types $\Gamma\colon V \to T$ (alternatively, we can see it as a sequence indexed by variables $(T_v)_{v \in V}$). We closely follow the syntax that is used by Pierce (2002).

In the STLC, we couldn't give a type to little omega, since we can't give a type to both the argument and the argument applied to itself ($\lambda x\colon ?.xx\colon ??$). If our context contained the mapping $x\colon \mathbb{N}$, then we could write an function which applies its argument to x like so,

$$x\colon \mathbb{N} \in \Gamma \vdash (\lambda f\colon \mathbb{N} \to \mathbb{N}.fx)\colon \mathbb{N}.$$

Crucially, if x had type $\mathbb{B}$, we would fail to determine the type of the function, and so we can guarantee that all permitted expressions 'can't go wrong' (Milner, 1978) (since untypable expresisons are disallowed).

This paper will build upon and borrow notation from the textbook *Programming Language Foundations in Agda* (Wadler et al., 2022), which includes an implementation of the STLC and also serves as further background.

System F. System F has been the formal background to what many modern programming languages call *generics*. For example, in Rust, we could write a function which checks if a list's length is two.

```
fn is_length_two<A>(slice: &[A]) -> bool {
    slice.len() == 2
}
```

[3] If we didn't include any base types, then our computational model becomes *degenerate* (that is, there are no terms). This is because everything has to have a type, and if there are no types, then nothing can exist.

This function will accept any kind of list. This is because we used a *type parameter* in the function's type signature, in this case called `A`.

System F is the STLC equipped with *polymorphic types*, another term for type parameters. It was independently discovered by Jean-Yves Girard (1972) and John Reynolds (1974) (who gave it the more straight-forward name, *the polymorphic* λ*-calulus*). We could write a function which applies a function twice to an argument,

$$\Lambda X.\lambda f\colon X \to X.\lambda x\colon X.f(fx)\colon \forall X.(X \to X) \to X \to X.$$

If we wanted to use this function, we would need to instantiate it with a specific type, notated using [square brackets]. For instance,

$$\begin{aligned} s\colon \mathbb{N} \to \mathbb{N}, z\colon \mathbb{N} \in \Gamma \vdash (\Lambda X.\lambda f\colon X \to X.\lambda x\colon X.f(fx))[\mathbb{N}]sz\colon \mathbb{N} \\ \mapsto_\beta (\lambda f\colon \mathbb{N} \to \mathbb{N}.\lambda x\colon \mathbb{N}.f(fx))sz\colon \mathbb{N} \\ \mapsto_{\beta^\star} ssz\colon \mathbb{N}. \end{aligned}$$

When trying to formalise System F, we will face a choice of using an intrinsically- or extrinsically-typed approach; although these were first described by Alonzo Church and Haskell Curry respectively, we will use the former two terms. The different approaches are described in detail in Reynolds (2003), but to summarise, an intrinsic approach will define the types of terms before the terms themselves, whereas an extrinsic approach will define terms first (without types), and only later justify that the type system is consistent.

Using the intrinsic approach will require less code and effort, so we shall proceed that way.

2.3 De Bruijn indices and Locally Nameless Sets

Suppose we had the following expression,

$$x\colon \mathbb{N} \in \Gamma \vdash \lambda y\colon \mathbb{N} \to \mathbb{N}.yx\colon \mathbb{N}.$$

This takes in a function y and applies it to the x that is in the context. Now suppose we move this expression into a context where we already have a bound y.

$$x\colon \mathbb{N}, y\colon \mathbb{N} \to \mathbb{N} \in \Gamma \vdash \lambda y\colon \mathbb{N} \to \mathbb{N}.yx\colon \mathbb{N}.$$

It's unclear whether we are referring to the local y or the previously bound y that is in our context (if we were to use actual functions on the naturals, then we could have a real problem if the outer y is the squaring function and the inner y the successor function, for instance). We can use an α-conversion and resolve this issue. We shall apply the conversion $y \mapsto_\alpha q$ to our inner expression,

$$x\colon \mathbb{N}, y\colon \mathbb{N} \to \mathbb{N} \in \Gamma \vdash \lambda q\colon \mathbb{N} \to \mathbb{N}.qx\colon \mathbb{N},$$

which solves our problem. We can add further assumptions to our context without affecting the semantics of the expression (for example, adding $k\colon \mathbb{N}$ to Γ won't change the semantics of the expression), this is called weakening-invariance (taken from proof theory, where extra assumptions make a theorem weaker).

Since we have these α-equivalent expressions, we can say that we have *quotient* inductive definitions (Aydemir et al., 2008), since we have both an inductive definition of the λ-calculus, but also (infinitely) many α-equivalence classes which we need to deal with[4]. This becomes difficult in Agda, since Agda primarily relies on inductive definitions (Pitts, 2023).

We can solve this by using *De Bruijn indices*. Using these ensures that no matter what variables we have in our context, we don't have any local variable names which could cause issues (since we only use indices that directly reference the scope). In our example,

$$x\colon \mathbb{N}, y\colon \mathbb{N} \to \mathbb{N} \in \Gamma \vdash \lambda \mathbb{N} \to \mathbb{N}.02\colon \mathbb{N}.$$

We can't use any α-conversions since each bound variable is indexed by a (unique) natural number. However, if we were to change the context, we would need to change the index too. So we have a purely inductive definition. For example, if we remove the y,

$$x\colon \mathbb{N}, \in \Gamma \vdash \lambda \mathbb{N} \to \mathbb{N}.01\colon \mathbb{N},$$

then we need to reindex the 2 to a 1. We have lost weakening invariance (Aydemir et al., 2008).

Using *locally nameless sets*, we can get both purely inductive definitions *and* weakening-invariance. Free variables will use variable names while bound variables will use indices. Our example becomes

$$x\colon \mathbb{N}, \in \Gamma \vdash \lambda \mathbb{N} \to \mathbb{N}.0x\colon \mathbb{N}, \quad \text{or with another variable in the context,}$$
$$x\colon \mathbb{N}, y\colon \mathbb{N} \to \mathbb{N} \in \Gamma \vdash \lambda \mathbb{N} \to \mathbb{N}.0x\colon \mathbb{N}.$$

This approach has been of research interest as of late (Aydemir et al., 2008) (Charguéraud, 2012). As part of using locally nameless terms, common operations (called 'infrastructure' by Aydemir et al. (2008)) need to be defined for the language used as part of the

[4]The name *quotient* is taken from other areas of mathematics where equivalence classes produce quotient objects. For example, in ring theory, for a ring R and ideal $I \subseteq R$, the quotient ring R/I is the set of equivalence classes where for all $a, b \in R$ we have the relation $a \sim b \iff a - b \in I$. If we let $R = \mathbb{Z}$ and $I = 2\mathbb{Z}$, the relation is $a \sim b \iff a - b \in 2\mathbb{Z} \iff$ they have the same parity. The quotient ring R/I is just the set $\{0, 1\}$. In our case, the relation is $a \sim b \iff a =_\alpha b$, and our quotient becomes the set of possible terms which are semantically distinct

metatheory. A recent article by Pitts (2023) explores locally nameless sets in Agda, and proves that this infrastructure can be defined in a syntax-agnostic way. This will form part of the basis of our approach to implementing System F in Agda.

2.4 Prior research

System F was previously formalised in Agda by Chapman et al. (2019). However, the authors of that paper formalised a variant of System F with language extensions known as *System* $F_{\omega\mu}$. They also used a different approach, opting to make use of De Bruijn indices. This paper will feature a novel approach using locally nameless sets.

Chapter 3

Conclusions

3.1 Final Reminder

The body of your dissertation, before the references and any appendices, *must* finish by page 40. The introduction, after preliminary material, should have started on page 1.

You may not change the dissertation format (e.g., reduce the font size, change the margins, or reduce the line spacing from the default single spacing). Be careful if you copy-paste packages into your document preamble from elsewhere. Some LaTeX packages, such as `fullpage` or `savetrees`, change the margins of your document. Do not include them!

Over-length or incorrectly-formatted dissertations will not be accepted and you would have to modify your dissertation and resubmit. You cannot assume we will check your submission before the final deadline and if it requires resubmission after the deadline to conform to the page and style requirements you will be subject to the usual late penalties based on your final submission time.

Bibliography

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 3–15, San Francisco California USA, January 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328443. URL https://dl.acm.org/doi/10.1145/1328438.1328443.

George Boolos. Frege's Theorem and the Peano Postulates. *Bulletin of Symbolic Logic*, 1(3):317–326, September 1995. ISSN 1079-8986, 1943-5894. doi: 10.2307/421158. URL https://www.cambridge.org/core/product/identifier/S1079898600008118/type/journal_article.

Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674, pages 73–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03358-2 978-3-642-03359-9. doi: 10.1007/978-3-642-03359-9_6. URL http://link.springer.com/10.1007/978-3-642-03359-9_6. Series Title: Lecture Notes in Computer Science.

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for Fun and Profit. In Graham Hutton, editor, *Mathematics of Program Construction*, volume 11825, pages 255–297. Springer International Publishing, Cham, 2019. ISBN 978-3-030-33635-6 978-3-030-33636-3. doi: 10.1007/978-3-030-33636-3_10. URL http://link.springer.com/10.1007/978-3-030-33636-3_10. Series Title: Lecture Notes in Computer Science.

Arthur Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-011-9225-2. URL http://link.springer.com/10.1007/s10817-011-9225-2.

Alonzo Church. A Set of Postulates for the Foundation of Logic. *The Annals of Mathematics*, 33(2):346, April 1932. ISSN 0003486X. doi: 10.2307/1968337. URL https://www.jstor.org/stable/1968337?origin=crossref.

Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940. ISSN 0022-4812, 1943-5886. doi: 10.2307/

2266170. URL https://www.cambridge.org/core/product/identifier/S0022481200108187/type/journal_article.

Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. ISSN 0002-9947, 1088-6850. doi: 10.1090/S0002-9947-1936-1501858-0. URL https://www.ams.org/tran/1936-039-03/S0002-9947-1936-1501858-0/.

J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'État, Univerité Paris VII, 1972.

WH Kusee. Compiling Agda to Haskell with fewer coercions. Master's thesis, University of Utrecht, Utrecht, December 2017. URL https://studenttheses.uu.nl/bitstream/handle/20.500.12932/28192/3800296.pdf.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. ISSN 00220000. doi: 10.1016/0022-0000(78)90014-4. URL https://linkinghub.elsevier.com/retrieve/pii/0022000078900144.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Massachusetts, 2002. ISBN 978-0-262-16209-8 978-0-262-25681-0.

Andrew M. Pitts. Locally Nameless Sets. *Proceedings of the ACM on Programming Languages*, 7(POPL):488–514, January 2023. ISSN 2475-1421. doi: 10.1145/3571210. URL https://dl.acm.org/doi/10.1145/3571210.

John C. Reynolds. Towards a theory of type structure. In G. Goos, J. Hartmanis, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, N. Wirth, and B. Robinet, editors, *Programming Symposium*, volume 19, pages 408–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 1974. ISBN 978-3-540-06859-4 978-3-540-37819-8. doi: 10.1007/3-540-06859-7_148. URL http://link.springer.com/10.1007/3-540-06859-7_148. Series Title: Lecture Notes in Computer Science.

John C. Reynolds. What do types mean? — From intrinsic to extrinsic semantics. In David Gries, Fred B. Schneider, Annabelle McIver, and Carroll Morgan, editors, *Programming Methodology*, pages 309–327. Springer New York, New York, NY, 2003. ISBN 978-1-4419-2964-8 978-0-387-21798-7. doi: 10.1007/978-0-387-21798-7_15. URL https://link.springer.com/10.1007/978-0-387-21798-7_15. Series Title: Monographs in Computer Science.

Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, November 2015. ISSN 0001-0782, 1557-7317. doi: 10.1145/2699407. URL https://dl.acm.org/doi/10.1145/2699407.

Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL https://plfa.inf.ed.ac.uk/22.08/.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration. This information is often a copy of a consent form.