Michael Rawson

Verified Metatheory and Type Inference for a Name-Carrying Simply-Typed λ-Calculus

Computer Science Tripos – Part II

Robinson College

April 27, 2023

Proforma

Name:	**Michael Rawson**
College:	**Robinson College**
Project Title:	**Verified Metatheory and Type Inference for a Name-Carrying Simply-Typed λ-Calculus**
Examination:	**Computer Science Tripos – Part II, July 2017**
Word Count:	11,976[1]
Project Originator:	Dr. Dominic Mulligan
Supervisors:	Dr. Dominic Mulligan and Dr. Victor Gomes

Original Aims of the Project

I aim to create a mechanisation in Isabelle of the simply-typed λ-calculus, together with a verified algorithm for type inference. Emphasis is placed on the treatment of binders in the calculus and the approach taken with representing α-equivalence.

Work Completed

This project meets all proposed success criteria, and adds two extensions. I have implemented the calculus, encoded the α-equivalence equivalence relation, and specified typing rules and a type inference algorithm for the calculus. A number of correctness proofs accompany this implementation. Further to this, I implemented extensions, one to add unit and pair values and corresponding types to the implementation, and one to show the confluence property of the calculus.

Special Difficulties

None.

[1] As computed by *texcount*

Declaration

I, Michael Rawson of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Acknowledgements

I wish to acknowledge the efforts of my supervisors in guiding me in this project. Both offered advice when I needed it, and let me get on when I didn't — an unusual and invaluable skill!

Chapter 1

Introduction

Lambda calculi express abstract computation, forming part of research into computability theory [1], programming languages [2], and type systems. My dissertation describes the implementation of machine-checked proofs in the proof assistant Isabelle concerning a typed λ-calculus, culminating with correctness properties of the calculus. I also produce executable Haskell code that implements a type inference algorithm for the calculus, extracting the code from the formal implementation, and supply a proof of correctness of the algorithm.

I draw on areas in theoretical Computer Science for my project: λ-calculi, types, formal logic, and verified reasoning. By verifying a *typed* calculus, the project required some type theory in addition to results about the λ-calculus, including properties such as the preservation of types under β-reduction. Verification itself requires knowledge (and precise application) of formal logic, as an intuitive argument will not satisfy the checker. Finally, the techniques used with informal proof versus verified proof must differ: while technology has improved so that an informal proof's structure remains in a verified proof, details that a human would discount as trivial are necessarily included.

1.1 Project summary

During the project, I implemented the following:

- Encoding the calculus in Isabelle.
- α-equivalence, with an unusual approach.
- A typing relation on the calculus.
- An executable type inference algorithm, shown to be correct against the typing relation.
- Extracted code for this inference algorithm.
- Safety properties of the calculus: progress, preservation, and safety.

For extension work, I implemented unit and pair terms and associated types, and showed that β-reduction is confluent, using a proof technique due to Tait, Martin-Löf, and Takahashi.

1.2 Previous work

The theory behind typed λ-calculi is well-known: Church's λ-calculus has a distinguished history [3], as do types since Russell's original *theory of types* [4]. My work used well-established knowledge, so there was little risk of attempting a mathematically-impossible project. Formal verification also has previous work that can be re-used: proof *assistants* now include automation tools, tactics and theorem provers, and libraries of formalised mathematics: the Mizar system [5] contains a library of over 50,000 proofs.

There is also more directly-related previous work. Several implementations of typed λ-calculus have been verified (see for instance the PoplMark challenge [6], a set of challenges designed to measure progress in mechanising programming language metatheory) using an assortment of techniques, all of which I can draw on for inspiration.

1.3 Completed work

I have met all criteria specified in the project proposal, and have added some extensions. I define and encode in Isabelle a simply-typed calculus, and show several results about α-equivalence in the calculus. Using this encoding, I then add a typing relation, type inference and β-reduction to the calculus, then show the main results: progress, type-preservation, and safety. Finishing, I show that the type inference algorithm is correct with respect to the type system, and hence also has safety properties.

Chapter 2

Preparation

After identifying the main goals, these coarse requirements are refined to be precise about objectives, and to drive development. This results in a set of *requirements* that can be analysed to predict problems and measure success. Once any problems are resolved, work can begin on implementation. Some mathematics is required to increase precision, and can be used to direct refinement of requirements. In this chapter I discuss the theory required to begin formalisation, briefly introduce Isabelle, and produce a list of requirements.

2.1 λ-Calculus

The λ-calculus [7] is a system of computation, represented by operations on *terms*.

Definition. *Terms M are inductively defined:*

1. *A variable, x, is always a term. These may be sub-categorised to be* bound *if some* binder *in an expression binds them, or* free*, if there is no such binder.*
2. *If M is a term, abstractions $\lambda x.M$ are also terms. This intuitively represents an anonymous function returning $M(x)$), and* binds *x in M.*
3. *If M and N are terms, applying M to N is also a term, $(M\ N)$.*

Terms can be represented with an algebraic datatype. For instance, in Standard ML:

```
datatype 'a trm =
    Var of 'a
  | Fn  of ('a * 'a trm)
  | App of ('a trm * 'a trm)
```

Computation here is performed by β-reduction: terms *reduce* to another term according to a series of rules, and hence computation occurs by sequential reductions.

Definition. *$M \to_\beta M'$ when one of the following holds:*

1. *If left or right subterms of an application reduce, then the application also reduces: if $M \to_\beta M'$, then $(M\ N) \to_\beta (M'\ N)$.*

2. *If $M \to_\beta M'$, then $\lambda x.M$ becomes $\lambda x.M'$.*

3. *If a term is of the form $((\lambda x.M)\ N)$, then it is a β-redex, and reduces to*

$$M[x := N]$$

*(*viz. *M with occurrences of x substituted in a capture-avoiding fashion for N).*

The order in which reduction steps occur in a computation is important for many applications of the λ-calculus, but I did not use this property in my dissertation. Substitution, used informally above, is defined recursively.

Definition. *Suppose N is substituted for x in M, and y is any name that is not x. Then the result, $M[x := N]$, is*

$$M[x := N] = \begin{cases} N & M = x \\ M & M = y \\ M & M = \lambda x.M' \\ \lambda y.\,(M'[x := N]) & M = \lambda y.M' \\ (M_1[x := N]\ M_2[x := N]) & M = (M_1\ M_2) \end{cases}$$

β-reduction has *confluence* property. Confluence states that if A reduces in many steps to B, and similarly on another path to C, there is a D such that B and C reduce to D, asserting that the order of reductions does not affect the final result. There are terms that cannot be further reduced, like x, $\lambda y.y$, or $(f\ x)$. These terms are considered to be values, or *in normal form.*

Definition. *Variables x are in normal form. Applications are in normal form if they are not a β-redex and both subterms are themselves in normal form. Binders are in normal form if their bound subterm is in normal form.*

2.2 Simple types

Untyped calculi have disadvantages. No types mean that unexpected constructions can occur, such as applying a non-function, which a type system generally prevents. "Programs" in the calculus may also fail to terminate: a sequence of reductions may not necessarily finish. Consider

$$\Omega = (\lambda x.\,(x\ x))\ \ (\lambda x.\,(x\ x))$$

Then the only possible reduction for Ω produces Ω, which may not terminate. The untyped calculus can be extended to include a type system while maintaining an approach to names: a *type* is simply added to each binder, so $\lambda x.M$ becomes $\lambda(x : T).M$, for an arbitrary T. Note that I use this, the Church style of typing, exclusively in my project.

Definition. *Simple types τ are either*

$$\text{VAR}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{FN}\ \frac{\Gamma\{x \mapsto \tau\} \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau).M : \tau \to \sigma} \qquad \text{APP}\ \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash (M\ N) : \sigma}$$

Figure 2.1: typing rules for the simply-typed calculus

1. *A base type, say ι.*
2. *An arrow type $\tau_1 \to \tau_2$ from one type to another.*

Adding simple types to the binders of the untyped calculus produces the *simply-typed* λ-calculus. The typing relation $\Gamma \vdash M : \tau$ is given inductively in Figure 2.1. Γ here is a typing context: a partial function from variables to types.

I show several correctness properties that are not possible in an untyped calculus: progress, type preservation (subject reduction), and safety. These capture semantics of Milner's [8] maxim "well-typed programs do not go wrong".

Definition. *The progress property asserts if $\Gamma \vdash M : \tau$, M is either in normal form or can be reduced further.*

Definition. *The preservation property holds if, assuming $\Gamma \vdash M : \tau$ and M reduces to M', $\Gamma \vdash M' : \tau$.*

Generally these are desirable properties: expressions should not "get stuck" computing non-values, or change type mid-reduction.

Definition. *A language has the safety property if, when reducing a well-typed term M by a number of steps resulting in M', M' is either in normal form, or can be reduced further.*

This property is the main goal of verification: it shows that if a term is well-typed, there is no scenario in which reduction fails — the term reduces, or computation has finished.

Type inference is the process of producing a type τ for M such that $\Gamma \vdash M : \tau$. One advantage of the simply-typed calculus is that type inference is decidable, and straightforward, with no unification steps (in the Church style) or complexity that more advanced typing systems encounter. The type inference algorithm $\text{infer}(\Gamma, M)$ can be described by

$$\text{infer}\,(\Gamma, M) = \begin{cases} \Gamma(x) & M = x \\ \tau \to \text{infer}\,(\Gamma\{x \mapsto \tau\}, N) & M = \lambda(x : \tau).N \\ \text{apply}\,(\text{infer}\,(\Gamma, A), \text{infer}\,(\Gamma, B)) & M = (A\ B) \end{cases}$$

where $\text{apply}\,(\tau \to \sigma, \tau)$ produces σ. All other input is undefined. I use an option type to propagate errors upwards, which the above omits for simplicity. Type inference can be shown correct with respect to a type system if it exclusively infers correct types (*soundness*), and infers all possible correct types (*completeness*). With these properties, the typing rules and the inference algorithm are equivalent.

2.3 The problem of α-equivalence

This representation, with names and binders, is insufficient. It is convenient to reason that e.g. $\lambda x.x$ and $\lambda y.y$ are equal: they produce identical results on all inputs. However, they differ structurally: x is not the same as y. This reasoning is called *α-equivalence*.

Definition. *The α-equivalence relation $\equiv_\alpha$ is the least congruence on terms such that*

$$\lambda x.M \equiv_\alpha \lambda y.M'$$

where y does not occur free in M, and M' is M with x substituted for y (avoiding captures).

Such an equivalence could be assumed whenever required. Nonetheless, it is cumbersome to carry such an assumption, and proof assistants often reason better about equality than equivalence relations, as automation tools are tuned for equality. This problem can be solved using *quotient types* [9].

Definition. *A quotient type Q is a base type R, an equivalence relation $\sim$ on R, and functions* Abs $: R \to Q$ *and* Rep $: Q \to R$. *Items $q_1 : Q$ and $q_2 : Q$ are equal iff Rep $q_1 \sim$ Rep q_2.*

I now define a quotient type for λ-terms modulo α-equivalence by encoding a datatype for pre-terms without equivalence as before, then the new type is a quotient over α-equivalence. While we now use equality directly, this equivalence relation is awkward to use. There are alternative ways of handling names, the most prominent de Bruijn indices [10], Higher-Order Abstract Syntax [11] (HOAS), and nominal techniques [12]. Ideally, these would allow the "Barendregt convention" for reasoning about names: for any term, assume that the bound variables are distinct, and fresh for a given set [7]. This simplifies proofs about name-carrying syntax, and is useful in informal reasoning about λ-calculus.

De Bruijn indices remove names altogether, and instead use natural numbers for bound variables to refer to the number of other binders between the variable and the respective binder.

$$\lambda x.\lambda y.x$$

becomes

$$\lambda.\lambda.1$$

using de Bruijn indices. α-equivalence is now equality, as all bound names have been removed. The downside is that using this representation for argument is unintuitive, using numbers rather than names. There are variations on these indices, including de Bruijn levels (counting binders from the start, not relative to the variable — the constant function becomes $\lambda.\lambda.0$), and conventions separating free and bound variables [13] syntactically, but the disadvantage stands: reasoning is complicated by arithmetic.

HOAS uses the host's (in this case Isabelle) own implementation of names to handle binding. The datatype presented earlier would then be

```
datatype trm =
    Fn  of (trm -> trm)
  | App of (trm * trm)
```

$\lambda x.\lambda y.x$ would be represented as `Fn (fn x => Fn (fn y => x))`. While this implementation avoids many issues of other approaches, it is not possible to show certain properties with this representation [11]. Manipulating terms also becomes difficult, as binders have to be applied to access their terms. Additionally, this representation is negative, so cannot be represented in many proof assistants, which require strictly-positive datatypes to avoid inconsistencies [14]. Theoretical issues arise from the ability to place arbitrary host terms under binders, some of which may non-terms, so the representation is too permissive and allows "junk terms".

Parametric HOAS [15] removes some problems by re-introducing explicit variables, and parameterising binders over a set of names. The approach reduces the effect of junk terms (since only terms depending on names are expressible), and the datatype is now strictly-positive:

```
datatype 'var trm =
    Var of 'var
  | Fn of ('var -> 'var trm)
  | App of ('var trm * 'var trm)
```

Finally, the ideas of "nominal techniques" [12] introduced by Gabbay and Pitts are a new approach, which retain the explicit representation of names, as in the naïve version. The technique uses a definition of α-equivalence based on *permuting* names in a given expression. I chose this approach, as it allowed a natural representation of the calculus, without compromising usability or theoretical properties, but allowing concise arguments.

Several further techniques exist, including recent research into viewing λ-terms as maps of occurrences of variables in a tree [16]. The area of binders is still active, with new approaches in continuous development.

2.4 Nominal techniques

I use the following simplified presentation of the nominal idea of α-equivalence in my project, but the theoretical background is more general. I present the simplified idea first, then an overview of the generality of nominal techniques.

Definition. *A swapping* $[x \leftrightarrow y]$ *is a pair of variables* x*,* y*.*

These represent changing instances of x to y and y to x within a structure, leaving other variables unchanged.

$$\frac{}{x \sim x}\ \text{VAR} \qquad \frac{A \sim C \qquad B \sim D}{(A\ B) \sim (C\ D)}\ \text{APP} \qquad \frac{[z \leftrightarrow x] \cdot M \sim [z \leftrightarrow y] \cdot N \qquad z\#M \qquad z\#N}{\lambda x.M \sim \lambda y.N}\ \text{FN}$$

Figure 2.2: an equivalence defined in terms of swappings

Definition. *The effect of a swapping,* $[x \leftrightarrow y] \cdot M$ *is defined as*

$$[x \leftrightarrow y] \cdot M = \begin{cases} y & M = x \\ x & M = y \\ z & M = z, z \notin \{x, y\} \\ \lambda([x \leftrightarrow y] \cdot z).\,([x \leftrightarrow y] \cdot N) & M = \lambda z.N \\ ([x \leftrightarrow y] \cdot A)\ \ ([x \leftrightarrow y] \cdot B) & M = (A\ B) \end{cases}$$

An equivalence $\sim$ can be defined using only this operation, as shown in Figure 2.2 — the preconditions $z\#M$ and $z\#N$ mean that "z is *fresh* for M and N". An element x is fresh in a set S iff $x \notin S$, and a variable x is fresh for a term M iff x is fresh for the free variables of M.

It can be shown [17] that $\sim$ is equivalent to $\equiv_\alpha$. Therefore, my approach to representing terms-modulo-α-equivalence will be to develop a theory of swappings, then use it to show $\sim$ is an equivalence relation, and finally produce a new type as a quotient of the concrete type with $\sim$. I also need a verified implementation of freshness.

These definitions suffice for my project. However, they are a simplification of more general theory, presented briefly here and in more detail elsewhere [12, 18, 19].

Consider a set $\mathbb{A}$ of *names*. Then, there is another set $\mathrm{Perm}(\mathbb{A})$, which is the set of finite permutations on $\mathbb{A}$. This set forms a group: the group operator is composition, and identity is the identity permutation ε. Each permutation has an inverse. Note here that every permutation can be decomposed into a sequence of swappings, permutations which only swap two variables, and hence composition is simply concatenating lists of swappings.

The *action* of a permutation π on a structure $x \in X$ which contains $\mathbb{A}$ — for instance, the set of λ-terms using $\mathbb{A}$ as variables — maps X onto itself, permuting the names in x.

Definition. *The action of* π *on* X*,* $\pi \cdot x$*, is a function that satisfies* $\pi_1 \cdot \pi_2 \cdot x = (\pi_1 \circ \pi_2) \cdot x$*, and is* x *when* π *is the identity permutation.*

Some set $\mathrm{supp}(x)$ of names *support* x: if a permutation does not change $\mathrm{supp}(x)$, the permutation action will not change x. In the λ-calculus under α-equivalence, $\mathrm{supp}(x) = \mathrm{fvs}(x)$, as permuting bound variables will not change the term under α-equivalence, but changing free variables does change x.

Definition. *A set* X *is a nominal set if for each* $x \in X$*,* $\mathrm{supp}(x)$ *is finite.*

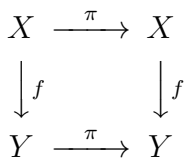


Figure 2.3: A commutative diagram showing the behaviour of an equivariant function.

Nominal sets form a category, where objects are nominal sets, and arrows are *equivariant* functions, functions f that satisfy

$$f(\pi \cdot x) = \pi \cdot f(x)$$

Each nominal set X yields another nominal set $[\mathbb{A}]X$, whose inhabitants are *name-abstractions* $\langle a \rangle x$, with an equivalence relation $\sim$. $\langle a \rangle x \sim \langle b \rangle y$ iff there is a new name c, fresh for a, b, x, y such that $[a \leftrightarrow c] \cdot x = [b \leftrightarrow c] \cdot y$. The permutation action on $[\mathbb{A}]X$ is defined to be such that $\pi \cdot (\langle a \rangle x) = \langle \pi \cdot a \rangle (\pi \cdot x)$. By using this definition on λ-terms, Gabbay arrives at the definition presented in Figure 2.2.

2.5 Isabelle

Isabelle [20] is a logical framework, supporting several object logics (I use Higher-Order Logic, the default), proof methods to remove tedious proof steps, and a human-readable proof language, Isar [21]. Proofs are checked in Isabelle by providing any definitions one wishes to make, then arguing theorems in this context. Since each step is checked, the theorems must be logically correct with respect to the definitions used. There are also other features Isabelle provides that I use in my project.

Quotient types are used heavily in my dissertation, for equivalence of permutations, and for α-equivalence. Isabelle provides this via the `quotient_datatype` command [22], which takes a base type R and an equivalence relation $\sim$ and produces a new quotient type Q, Abs and Rep. It also provides "lifting" and "transfer" operations to move between the types. Type classes [23] are another feature I used, providing a way for types to conform to an interface (for instance, all types that can be ordered might form an ordering typeclass). I used type classes to implement freshness polymorphically.

A major advantage of Isabelle over others was the straightforward support for quotient types. Isabelle's rivals Agda and Coq both suffer from "setoid hell", in which non-equality equivalence relations must be passed around to reason about propositions involving equivalence. There is no direct equivalent of `quotient_type`, and the theory becomes somewhat involved to implement such a construct [24]. Isabelle also has excellent tooling and a stable core, and these factors combined made it a good choice for my project.

2.6 Requirements analysis and engineering

Moving from coarse requirements to finer ones is now easier, as constraints are imposed by the mathematics. Implementation can now be broken into the following steps:

1. Develop work about freshness and swappings to support later developments.
2. Define a datatype for representing simple types.
3. Define *pre-terms*, the raw terms of the calculus.
4. Define the effect of swappings on pre-terms.
5. Define the α-equivalence relation.
6. Show it is an equivalence relation.
7. Define a type inference algorithm on pre-terms, show invariance under α-equivalence.
8. Produce a quotient type of *terms* from pre-terms under equivalence.
9. Lift required definitions and lemmas over the quotient.
10. Prove required theorems by reference to identical results on pre-terms.
11. Define a typing relation on terms.
12. Argue safety properties of this typing relation.
13. Show that the algorithm is sound and complete with respect to the typing relation.
14. Conclude that the implementation is verified, extract code.

The dependencies between steps are largely sequential, as shown in Figure 2.4, and each step can only be "complete", or "incomplete", considering the boolean nature of formal verification development. A waterfall model of development seems appropriate: topological sorting of Figure 2.4 would suffice to follow such a model. However, Isabelle allows admitting propositions as axioms, so to a limited extent each step can follow an iterative model where proofs are progressively made more explicit, with fewer admitted steps.

As far as possible, I followed software engineering good-practice. All project files were checked into version control (git), and shared (read-only) with supervisors using a well-known hosting website. Weekly progress meetings were observed to ensure the project did not get off-track.

2.7 Starting point

At the start of the project, I was familiar with the theory required for implementing λ-calculus and type theory. I was not familiar with approaches to binding described above, or with Isabelle itself. I am pleased to say I have now learned more about these topics.

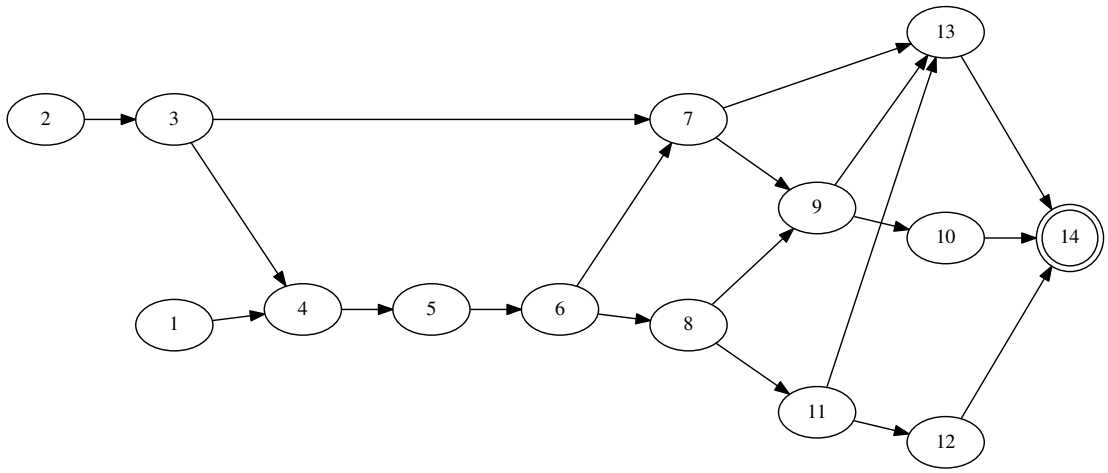


Figure 2.4: A directed graph showing which tasks must be completed to begin others. $A \to B$ shows that A must be complete before B may begin.

2.8 Summary

In this chapter I discussed work that took place before implementation began. I introduced background theory, covering untyped λ-calculus (§2.1), simple types (§2.2), α-equivalence and approaches to this problem (§2.3), and introduced nominal techniques (§2.4). I then described the Isabelle tooling (§2.5), preparing for practical and theoretical challenges later. Finally, I decomposed the project into steps (§2.6), and described engineering techniques used to tackle the problem.

Chapter 3

Implementation

In this chapter I discuss implementation details of the project. The work is split into four modules, what Isabelle calls *theories*. The first (*Fresh*, §3.1) deals with fresh names, the second (*Permutation*, §3.2) with permutations, and the third and fourth (*PreSimplyTyped*, §3.3 and *SimplyTyped*, §3.4) with λ-terms, before and after the quotient respectively. All theories build upon Isabelle's standard library, called *Main*. A dependency graph is shown in Figure 3.1. Results shown which have a corresponding theorem in the formalisation are marked as such, like this:

Theorem 0 (`example-in-isabelle`)**.** *The statement of the theorem.*

Proof. A proof of the theorem. □

3.1 Freshness

I develop a theory of freshness, used later to obtain a fresh name for a given binder. The implementation should accept a set of names S, and produce an element not in S. In order to implement this interface, I used type classes [23] to make a class for types that can produce a fresh element:

```
class fresh =
  fixes fresh_in :: "'a set ⇒ 'a"
  assumes "finite S ⟹ fresh_in S ∉ S"
```

Note the pre-condition of a *finite* S: otherwise, useful implementations such as numbers or strings cannot conform to this interface. To see this, consider (possibly-infinite) sets S of natural numbers. Since S can be infinite, choose S to be $\mathbb{N}$, the set of all natural numbers. Now, if x is fresh in S, x must be a natural number. But since $x \notin S$, x must also *not* be a natural number, since $S = \mathbb{N}$ — a contradiction.

To extract executable code, there must be at least one implementation of freshness. Unfortunately, not every implementation will suffice: for example, Isabelle allows the Hilbert indefinite description operator ϵ inside definitions. Hilbert's operator is a *choice*

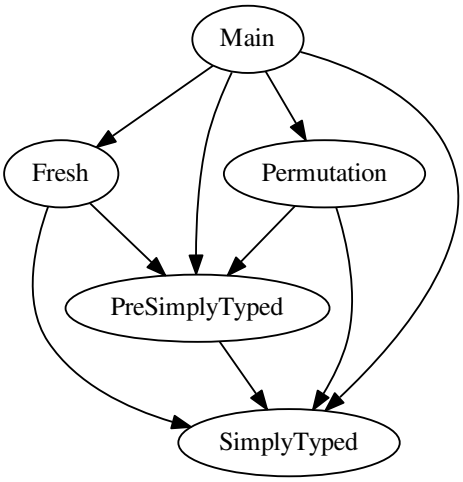


Figure 3.1: A directed graph showing which of the theories depend on each other. $A \to B$ shows that B depends upon results in A.

principle — the semantics of which are "assuming that there is at least one x such that $P(x)$, $\epsilon x.P(x)$ chooses one such x (by no specific means) and returns x" — so

$$\epsilon x.x \notin S$$

would be a definition of freshness if there is such an x — but this is not *executable*, and code cannot be extracted from it. Natural numbers are one possible implementation: to make a fresh natural number from a finite set S, take the largest element of the set (or 0), then add 1 to it.

```
instantiation nat :: fresh
begin
  definition fresh_in_nat :: "nat set ⇒ nat" where
    [code]: "fresh_in_nat S ≡ (if Set.is_empty S then 0 else Max S + 1)"
```

The `code` tag indicates code to be extracted later. The above also generates a proof obligation to show the implementation satisfies the freshness specification.

Lemma 1. *For any finite set S of natural numbers, the procedure given produces an n such that $n \notin S$.*

Proof. S is either the empty set, or it is not. If S is empty, 0 is produced as fresh in S, and clearly $0 \notin S$. If S is non-empty, then n' is produced such that n' is larger than the largest element of S, n. Suppose for contradiction that n' were in S. Then n' would be the largest element of S, not n. Hence n' must not be in S. □

3.2 Swappings and permutations

Finitely-supported permutations are used in the nominal definition (§2.4) of α-equivalence. I define permutations as sequences of swappings, permuting by swapping single variables at a time. Multiple sequences can correspond to the same permutation, so simple equality here will not suffice: permutations are "equal" iff they have the same effect on all inputs. This is another use for quotient types: by identifying equivalent permutations, a new type can be made that does not make this distinction. Therefore, I define *pre-permutations*, then *permutations*. Here, I use type synonyms for the definition of the permutation type:

```
type_synonym 'a swp = "'a × 'a"
type_synonym 'a preprm = "'a swp list"
```

Definition (`preprm-apply`)**.** *Application of pre-permutations to names, π `$` x, is defined recursively:*

1. ε `$` $x = x$
2. $((a, b) :: \pi') \cdot x = (a, b) \cdot \pi' \cdot x$, *since π' is applied first, then (a, b).*

```
fun swp_apply :: "'a swp ⇒ 'a ⇒ 'a" where
  "swp_apply (a, b) x = (if x = a then b else (if x = b then a else x))"

fun preprm_apply :: "'a preprm ⇒ 'a ⇒ 'a" where
  "preprm_apply [] x = x"
| "preprm_apply (s # ss) x = swp_apply s (preprm_apply ss x)"
```

The defined identity element is in fact identity:

Lemma 2 (`preprm-apply-id`)**.** ε *satisfies ε `$` $x = x$, for any x.*

Proof. By definition of application. □

In later proofs about α-equivalence, I require that $x = y \implies \pi\$x = \pi\$y$ (which follows from identity), its inverse, $x \neq y \implies \pi\$x \neq \pi\$y$, and the converse, $\pi\$x = \pi\$y \implies x = y$.

Lemma 3 (`preprm-apply-unequal`)**.** *If $x \neq y$, then π `$` $x \neq \pi$ `$` y*

Proof. By induction on π. The base case follows from Lemma 2. For the inductive step, suppose $x' \neq y'$. Then $[(a, b)]$ `$` $x' \neq [(a, b)]$ `$` y', by cases on x' and y'. □

Lemma 4 (`preprm-apply-injective`)**.** *Application of permutations π is injective.*

Proof. By induction on π. The base case follows by definition. Using the inductive hypothesis, obtain $[(a, b)]$ `$` $(\pi'$ `$` $x) = [(a, b)]$ `$` $(\pi'$ `$` $y)$. Then use the contrapositive of Lemma 3 to show that $x = y$ using the inductive hypothesis. □

Some operations on pre-permutations are defined on the pre-permutations, then lifted over the equivalence relation.

Definition (`preprm-compose`)**.** *The composite pre-permutation $\pi \diamond \sigma$ is σ appended to π.*

Lemma 5 (`preprm-apply-composition`)**.** *Application of $\pi \diamond \sigma$ is equivalent to applying first σ, then π.*

Proof. By induction on π, both cases by definition. □

Lemma 6 (`preprm-unit-involution`)**.** *Composition of $[(a, b)]$ with itself is equivalent to the identity element.*

Proof. Consider whether x, the variable the permutation is applied to, is a, b, or neither. If it is a or b, then swapping a to b, then b to a (or vice-versa) produces the same x. If it is not, then the swapping has no effect. □

Definition (`preprm-inv`)**.** *The inverse of a permutation π, π^{-1}, is defined to be the reverse of π.*

```
definition preprm_inv :: "'a preprm ⇒ 'a preprm" where
  "preprm_inv π ≡ rev π"
```

It can be shown that this is the inverse operation:

Lemma 7 (`preprm-inv-involution`)**.** *For any π and x, $\pi^{-1} \mathbin{\$} \pi \mathbin{\$} x = x$, and vice-versa, $\pi \mathbin{\$} \pi^{-1} \mathbin{\$} x = x$.*

Proof. By induction on π, with a trivial base case. For the inductive step, assume that $\pi'^{-1} \mathbin{\$} \pi' \mathbin{\$} x = x$ and try to show

$$([(a, b)] \diamond \pi')^{-1} \mathbin{\$} ([(a, b)] \diamond \pi') \mathbin{\$} x = x$$

Note that

$$([(a, b)] \diamond \pi')^{-1} = \pi'^{-1} \diamond [(a, b)]$$

by definition. Hence, using Lemmas 5 and 6 and the inductive hypothesis,

$$\begin{aligned}
([(a, b)] \diamond \pi')^{-1} \mathbin{\$} (([(a, b)] \diamond \pi') \mathbin{\$} x) &= \left(\pi'^{-1} \diamond [(a, b)]\right) \mathbin{\$} (([(a, b)] \diamond \pi') \mathbin{\$} x) \\
&= \pi'^{-1} \mathbin{\$} ([(a, b)] \diamond [(a, b)]) \mathbin{\$} \pi' \mathbin{\$} x \\
&= \pi'^{-1} \mathbin{\$} (\varepsilon) \mathbin{\$} \pi' \mathbin{\$} x \\
&= \pi'^{-1} \mathbin{\$} \pi' \mathbin{\$} x \\
&= x
\end{aligned}$$

as required. The alternative proposition follows from this and the fact that $(\pi^{-1})^{-1} = \pi$, given reversing a list is involutive. □

I define an extensional equivalence relation to relate equivalent permutations.

```
definition
  preprm_ext :: "'a preprm ⇒ 'a preprm ⇒ bool"
  where
  "π =p σ ≡ ∀x. preprm_apply π x = preprm_apply σ x"
```

i.e. that $\pi \equiv_p \sigma$ when for any x applying π and σ produces the same result. This relation is an equivalence relation:

Lemma 8. *$\equiv_p$ is an equivalence relation.*

Proof. Unfolding the definition of equivalence, $\equiv_p$ is reflexive, symmetric, and transitive, and hence an equivalence relation. □

Several properties behave under equivalence as they would under equality.

Lemma 9 (`preprm-ext-compose-left`)**.** *If $\sigma \equiv_p \tau$, then $\pi \diamond \sigma \equiv_p \pi \diamond \tau$.*

Lemma 10 (`preprm-ext-compose-right`)**.** *If $\sigma \equiv_p \tau$, then $\sigma \diamond \pi \equiv_p \tau \diamond \pi$.*

Proof. Both of these follow from Lemma 5. □

Lemma 11 (`preprm-ext-uncompose`)**.** *If $\pi \equiv_p \sigma$ and $\pi \diamond \tau \equiv_p \sigma \diamond \rho$, then $\tau \equiv_p \rho$.*

Proof. By assumption obtain $\pi \diamond \tau \equiv_p \pi \diamond \rho$, then using the injectivity of permutation application, arrive at the result. □

Lemma 12 (`preprm-inv-ext`)**.** *If $\pi \equiv_p \sigma$, then $\pi^{-1} \equiv_p \sigma^{-1}$.*

Proof. Note that $(\pi^{-1})^{-1} \diamond \pi^{-1} \equiv_p \varepsilon$ and $(\sigma^{-1})^{-1} \diamond \sigma^{-1} \equiv_p \varepsilon$, using Lemma 7. Hence $\pi \diamond \pi^{-1} \equiv_p \varepsilon$, and correspondingly for σ. From these obtain $\pi \diamond \pi^{-1} \equiv_p \sigma \diamond \sigma^{-1}$, since $\equiv_p$ is an equivalence relation. Finally, derive the result by applying Lemma 11 and the assumptions. □

It is also expected that composition with an inverse produces identity.

Lemma 13 (`preprm-inv-compose`)**.** *For any π, $\pi^{-1} \diamond \pi \equiv_p \varepsilon$.*

Proof. Using Lemmas 7 and 5, this follows directly. □

The preceding theory can now be lifted into an extensional context with a quotient type.

```
quotient_type 'a prm = "'a preprm" / preprm_ext
proof(rule equivpI)
  show "reflp preprm_ext" using preprm_ext_reflp.
  show "symp preprm_ext" using preprm_ext_symp.
  show "transp preprm_ext" using preprm_ext_transp.
qed
```

The quotient-type construction produces an obligation to show that $\equiv_p$ is an equivalence (Lemma 8). All work must now be lifted into the new equivalence. Definitions are lifted like so:

```
lift_definition prm_id :: "'a prm" ("ε") is preprm_id
```

Lemmas as follows:

```
lemma prm_apply_injective:
  shows "inj (prm_apply π)"
by(transfer, metis preprm_apply_injective)
```

The set of permutations over a base set S, P_S form a group $(P_S, \circ)$: each element has an inverse, and ε is the identity element. This is shown in the Isabelle source. Other operations could be defined directly on the quotient type.

Definition (`prm-set`)**.** *The image of a set S under a permutation π, π **{$}** S, is defined to be $\{\pi \$ x \mid x \in S\}$.*

This is the *pointwise action* of π on S, which allows for a permutation action on sets of names, and hence for a notion of equivariance on the free variables of a term.

```
definition prm_set :: "'a prm ⇒ 'a set ⇒ 'a set" where
  "prm_set π S ≡ image (prm_apply π) S"
```

3.3 Raw λ-terms

Simple types $\tau, \sigma, \ldots$ and λ-terms (uppercase variables) are defined as an Isabelle datatype, as described in §2.1. I use natural numbers for a concrete variable type as they implement freshness, but any other type that satisfies freshness can be used.

```
type_synonym tvar = nat

datatype type =
  TVar tvar
| TArr type type

datatype 'a ptrm =
  PVar 'a
| PApp "'a ptrm" "'a ptrm"
| PFn 'a type "'a ptrm"
```

The action of permutations on pre-terms, $\pi \bullet X$ commutes with the structure of the term until the base case (i.e. variables or the name in a binder), so is expressed recursively.

```
fun ptrm_apply_prm :: "'a prm ⇒ 'a ptrm ⇒ 'a ptrm"
  "ptrm_apply_prm π (PVar x) = PVar (π $ x)"
| "ptrm_apply_prm π (PApp A B) = PApp
    (ptrm_apply_prm π A)
    (ptrm_apply_prm π B)"
| "ptrm_apply_prm π (PFn x T A) = PFn (π $ x) T (ptrm_apply_prm π A)"
```

I re-define results from §3.2 in the context of pre-terms by transcribing them. For example, compare the following with corresponding lemmas about application of permutations.

```
lemma ptrm_prm_apply_id:
  shows "ε • X = X"
by(induction X, simp_all add: prm_apply_id)

lemma ptrm_prm_apply_compose:
  shows "π • σ • X = (π ◇ σ) • X"
by(induction X, simp_all add: prm_apply_composition)
```

Free variables of a term can also be defined recursively.

```
fun ptrm_fvs :: "'a ptrm ⇒ 'a set" where
  "ptrm_fvs (PVar x) = {x}"
| "ptrm_fvs (PApp A B) = ptrm_fvs A ∪ ptrm_fvs B"
| "ptrm_fvs (PFn x _ A) = ptrm_fvs A - {x}"
```

Later, a set of names is frequently required to be finite, so that a fresh name can be generated. To aid this, I show that the free variables of a pre-term is always a finite set.

Lemma 14 (`ptrm-fvs-finite`)**.** *The free variables of X form a finite set.*

Proof. By induction on X. □

This operation can also be shown equivariant using pointwise action, a property used later when working with the α-equivalence of λ-terms:

Lemma 15 (`ptrm-prm-fvs`)**.** *The free variables of $\pi \bullet X$ are the image of π in the free variables of X.*

Proof. By induction on X.

1. For a variable x, note that $\pi \bullet x = \pi \mathbin{\$} x$. Then the proposition follows directly.

2. Assume that $\text{fvs}\,(\pi \bullet A) = \pi \,\{\$\}\, \text{fvs}(A)$, and similarly for B. Then the lemma holds for $(A\ B)$:

$$\begin{aligned}\text{fvs}\,(\pi \bullet (A\ B)) &= \text{fvs}\,((\pi \bullet A)\ \ (\pi \bullet B)) \\ &= \text{fvs}\,(\pi \bullet A) \cup \text{fvs}\,(\pi \bullet A) \\ &= \pi \,\{\$\}\, \text{fvs}(A) \cup \pi \,\{\$\}\, \text{fvs}(B) \\ &= \pi \,\{\$\}\, (\text{fvs}(A) \cup \text{fvs}(B)) \\ &= \pi \,\{\$\}\, \text{fvs}\,(A\ B)\end{aligned}$$

3. Assume that $\text{fvs}\,(\pi \bullet A) = \pi \,\{\$\}\, \text{fvs}(A)$. Then finish as in the application case:

$$\begin{aligned}\text{fvs}\,(\pi \bullet \lambda\,(x : T)\,.A) &= \text{fvs}\,(\lambda\,(\pi \,\$\, x : T)\,.\,(\pi \bullet A)) \\ &= \text{fvs}\,(\pi \bullet A) - (\pi \,\$\, x) \\ &= \pi \,\{\$\}\, \text{fvs}(A) - (\pi \,\$\, x) \\ &= \pi \,\{\$\}\, (\text{fvs}(A) - x) \\ &= \pi \,\{\$\}\, (\text{fvs}\,(\lambda(x : T).A))\end{aligned}$$

□

Isabelle generates a function called `size` with every datatype which reports the number of nodes in a given instance — this corresponds to the formal definition of term size $|M|$ described in §4.3. To use the size of a term (e.g. for induction), it is useful to show that size is equivariant, that $|\pi \bullet M| = \pi \cdot |M| = |M|$.

Lemma 16 (`ptrm-size-prm`)**.** *The size of a pre-term X is the same as the size of X under a permutation, $\pi \bullet X$.*

Proof. By induction on the structure of X, then by definition of permutation application. □

3.3.1 α-equivalence

Isabelle provides inductive definitions, so the nominal definition of α-equivalence introduced in §2 is straightforward. I used a slightly different formulation in which there are two cases for functions: one where the swapping variables are the same, and one when they are not. This is equivalent, but makes for a shorter proof script.

```
inductive
  ptrm_alpha_equiv :: "'a ptrm ⇒ 'a ptrm ⇒ bool"
  where
  var: "(PVar x) ≈ (PVar x)"
| app: "⟦A ≈ B; C ≈ D⟧ ⟹ (PApp A C) ≈ (PApp B D)"
| fn1: "A ≈ B ⟹ (PFn x T A) ≈ (PFn x T B)"
| fn2: "⟦a ≠ b; A ≈ [a ↔ b] • B; a ∉ ptrm_fvs B⟧
       ⟹ (PFn a T A) ≈ (PFn b T B)"
```

However, Isabelle does not provide eliminators by default, and these must be added manually. Eliminators provide reasoning to "go backwards" from an inductive predicate, obtaining conditions which cause the truth of the predicate. If $(A\ C) \sim (B\ D)$, the only conclusion possible from the definition of $\sim$ is that $A \sim B$ and $C \sim D$.

```
inductive_cases varE:  "PVar x ≈ Y"
inductive_cases appE:  "PApp A B ≈ Y"
inductive_cases fnE:   "PFn x T A ≈ Y"
```

These provide this style of reasoning, bound to labels as a theorem. For instance *varE* is effectively

```
lemma varE
  fixes P
  assumes "PVar x ≈ Y"
  assumes "Y = PVar x ⟹ P"
  shows "P"
proof
  ...
qed
```

A test of this relation is to show that free variables and permutation application are preserved under it, a property of α-equivalence.

Lemma 17 (ptrm-alpha-equiv-fvs)**.** *Assume* $X \sim Y$*. Then* $\text{fvs}(X) = \text{fvs}(Y)$.

Proof. By induction on the derivation of $X \sim Y$. *var*, *app*, and *fn1* are easy. For *fn2*, assume:

- $a \neq b$
- $A \sim [a \leftrightarrow b] \bullet B$
- $a \notin \text{fvs}(B)$
- the induction hypothesis, $\text{fvs}(A) = \text{fvs}\left([a \leftrightarrow b] \bullet B\right)$

Then, try and show $\text{fvs}\left(\lambda(a : T).A\right) = \text{fvs}\left(\lambda(b : T).B\right)$.

$$\begin{aligned}\text{fvs}\left(\lambda(a : T).A\right) &= \text{fvs}(A) - a \\ &= \text{fvs}\left([a \leftrightarrow b] \bullet B\right) - a \\ &= \left([a \leftrightarrow b]\ \{\$\}\ \text{fvs}(B)\right) - a\end{aligned}$$

Now the proof is stuck — the proof will follow from arriving at the term $\text{fvs}(B) - b$, but simplifying will not arrive there. Consider whether b is in the free variables of B. If it is, then $[a \leftrightarrow b]\ \{\$\}\ \text{fvs}(B) = \text{fvs}(B) - b \cup a$, since $a \notin \text{fvs}(B)$, so

$$\begin{aligned}\left([a \leftrightarrow b]\ \{\$\}\ \text{fvs}(B)\right) - a &= \text{fvs}(B) - b \cup a - a \\ &= \text{fvs}(B) - b\end{aligned}$$

If it is not, then $[a \leftrightarrow b]$ has no effect, so

$$\begin{aligned}([a \leftrightarrow b] \,\{\$\}\, \mathrm{fvs}(B)) - a &= \mathrm{fvs}(B) - a \\ &= \mathrm{fvs}(B) = \mathrm{fvs}(B) - b\end{aligned}$$

Now

$$\begin{aligned}\mathrm{fvs}\,(\lambda(a:T).A) &= \mathrm{fvs}(B) - b \\ &= \mathrm{fvs}\,(\lambda(b:T).B)\end{aligned}$$

□

Lemma 18 (`ptrm-alpha-equiv-prm`)**.** *Assume $X \sim Y$. Then $\pi \bullet X \sim \pi \bullet Y$. This shows that α-equivalence is respected by permuting variable names.*

Proof. By induction on the derivation of $X \sim Y$. Again *var*, *app*, and *fn1* turn out to be easy. For *fn2*, assume as usual that $a \neq b$, $a \notin \mathrm{fvs}(B)$, and the induction hypothesis $\pi \bullet A \sim \pi \bullet [a \leftrightarrow b] \bullet B$. Then, deduce from these the preconditions for $\lambda(\pi \,\$\, a : T).\pi \bullet A \sim \lambda(\pi \,\$\, b : T).\pi \bullet B$, namely that

- $\pi \bullet a \neq \pi \bullet b$
- $\pi \,\$\, a \notin \mathrm{fvs}\,(\pi \bullet B)$
- $\pi \bullet A \sim [\pi \,\$\, a \leftrightarrow \pi \,\$\, b] \bullet \pi \bullet B$

Finally, see that both sides of the equivalence can be simplified to produce the result. □

I show that $\sim$ is an equivalence relation. Some specific results are needed first, which are presented informally:

Lemma 19 (`ptrm-swp-transfer`)**.** *$[a \leftrightarrow b] \bullet X \sim Y$ is equivalent to $X \sim [a \leftrightarrow b] \bullet Y$.*

Proof. This is shown in both directions by permuting both sides by $[a \leftrightarrow b]$, then using the fact that $[a \leftrightarrow b] \diamond [a \leftrightarrow b] = \varepsilon$. □

Lemma 20 (`ptrm-alpha-equiv-fvs-transfer`)**.** *If $a \notin \mathrm{fvs}(B)$, and $A \sim [a \leftrightarrow b] \bullet B$, then $b \notin \mathrm{fvs}(A)$.*

Proof. By a similar argument. □

These are used to argue reflexivity, symmetry, and transitivity.

Lemma 21 (`ptrm-alpha-equiv-reflexive`)**.** *$\sim$ is reflexive: that is, for all terms X, $X \sim X$.*

Proof. By induction on the structure of X. □

Lemma 22 (`ptrm-alpha-equiv-symmetric`)**.** *$\sim$ is symmetric, so for all terms X, Y, $X \sim Y \Longrightarrow Y \sim X$.*

Proof. By induction on the derivation of $X \sim Y$. As usual, *fn2* is the difficult case. It is given from the induction process that $a \neq b$, $A \sim [a \leftrightarrow b] \bullet B$, $a \notin \text{fvs}(B)$, and the inductive hypothesis, $[a \leftrightarrow b] \bullet B \sim A$. From these and Lemmas 19 and 20, deduce that $b \neq a$, $B \sim [b \leftrightarrow a] \bullet A$, and $b \notin \text{fvs}(A)$. These are the conditions for $\lambda(b : T).B \sim \lambda(a : T).A$, which is the required result. □

Lemma 23 (`ptrm-alpha-equiv-transitive`)**.** $\sim$ *is transitive. If* $X \sim Y$ *and* $Y \sim Z$*, then* $X \sim Z$*.*

Proof. By induction on the size of X, then by cases on X. This generates the inductive hypothesis

$$\forall KYZ.\, (|K| < |X|, K \sim Y, Y \sim Z) \Longrightarrow K \sim Z$$

If X is a variable, the result follows by deducing that Y and Z must also be variables (using the eliminators defined earlier), and that they must all be the same variable. If X is instead an application, the inductive hypothesis can be used to show that both components of the application in X and Z are equivalent, and hence that $X \sim Z$ by definition of $\sim$.

Finally, if X is an abstraction, there are four cases to consider, depending on whether *fn1* or *fn2* is used to get from X to Y, and again from Y to Z. The difficult case is when both relations are produced by *fn2*. This case can be further broken down when X and Z use the same variable in their binder — suppose $X = \lambda(x : T).A$, $Y = \lambda(y : T).B$, and $Z = \lambda(z : T).C$, then either $x = z$ or $x \neq z$. If $x = z$, then $A \sim C$; since $A \sim [x \leftrightarrow y] \bullet B$ and $B \sim [y \leftrightarrow x] \bullet C$, $A \sim [x \leftrightarrow y] \bullet [y \leftrightarrow x] \bullet C$, so $A \sim C$ since the swappings cancel out.

However, if $x \neq z$, $X \sim Z$ has to be shown by *fn2*. Since both derivations were by *fn2*, $A \sim [x \leftrightarrow y] \bullet [y \leftrightarrow z] \bullet C$, but since x, y, and z are all now distinct it is possible to show (by definition of permutations) that $[x \leftrightarrow y] \diamond [y \leftrightarrow z] = [x \leftrightarrow z]$, so $A \sim [x \leftrightarrow z] \bullet C$. Also, $x \notin \text{fvs}(C)$ holds by noting that $x \notin \text{fvs}([y \leftrightarrow z] \bullet C)$, then considering if z is in the free variables of C. If it is, then $x \notin \text{fvs}(C)$, since $x \neq y \neq z$ and hence swapping z for y has no effect on whether x is in the free variables of the term or not. If it is not, then the swapping has no effect, so $x \notin \text{fvs}(C)$ trivially. Concluding, $x \neq z$, $A \sim [x \leftrightarrow z] \bullet C$, and $x \notin \text{fvs}(C)$, so it follows that $\lambda(x : T).A \sim \lambda(z : T).C$. □

Theorem 1. $\sim$ *is an equivalence relation.*

Proof. Since $\sim$ is reflexive, symmetric, and transitive, it is an equivalence relation. □

3.3.2 Type inference algorithm

I implement a type inference algorithm on the concrete syntax, which can then be lifted to terms. The algorithm uses typing contexts, which are modelled as partial (i.e. total, but returning an option type) functions from names to types.

```
type_synonym 'a typing_ctx = "'a ⇀ type"

fun ptrm_infer_type :: "'a typing_ctx ⇒ 'a ptrm ⇒ type option" where
  "ptrm_infer_type Γ (PVar x) = Γ x"
| "ptrm_infer_type Γ (PApp A B) =
    (case (ptrm_infer_type Γ A, ptrm_infer_type Γ B) of
      (Some (TArr τ σ), Some τ') ⇒ (if τ = τ' then Some σ else None)
    | _ ⇒ None)"
| "ptrm_infer_type Γ (PFn x τ A) =
    (case ptrm_infer_type (Γ(x ↦ τ)) A of
      Some σ ⇒ Some (TArr τ σ)
    | None ⇒ None)"
```

To lift this definition, I show that it is invariant under α-equivalence. This is done with a lemma about swapping names in a typing context.

Lemma 24 (`ptrm-infer-type-swp`)**.** *Assume two names a and b are distinct, and $b \notin \text{fvs}(X)$. Then*

$$\text{infer}\,(\Gamma\{a \mapsto \tau\}, X) = \text{infer}\,(\Gamma\{b \mapsto \tau\}, [a \leftrightarrow b] \bullet X)$$

Proof. By induction on the structure of X. □

Theorem 2 (`ptrm-infer-type-alpha-equiv`)**.** *The inference algorithm is invariant under equivalence. If $X \sim Y$, then for any context Γ,*

$$\text{infer}\,(\Gamma, X) = \text{infer}\,(\Gamma, Y)$$

Proof. By induction on the derivation of $X \sim Y$. All cases other than *fn2* follow immediately by definition. For *fn2*, use the definition of the type inference algorithm and Lemma 24 to show that the function bodies have the same type σ, then note that both X and Y then have inferred type $\tau \to \sigma$. □

3.4 λ-terms with α-equivalence

As before with permutations, pre-terms are lifted to terms using the quotient-type machinery.

```
quotient_type 'a trm = "'a ptrm" / ptrm_alpha_equiv
proof(rule equivpI)
  show "reflp  ptrm_alpha_equiv" using ptrm_alpha_equiv_reflp.
  show "symp   ptrm_alpha_equiv" using ptrm_alpha_equiv_symp.
  show "transp ptrm_alpha_equiv" using ptrm_alpha_equiv_transp.
qed
```

There is extra boilerplate required by the quotient, as equality rules don't necessarily hold. First, every lifted constructor has to be shown to be not equal to every other constructor, of which the following is an indicative example.

```
lemma var_not_app:
  shows "Var x ≠ App A B"
proof(transfer)
  ...
qed
```

Then, the "obvious" injective conclusions from equality don't necessarily hold either, and must be shown manually. For example, suppose $(A\ B) = (C\ D)$. Conclude that $A = C$ and $B = D$ from this, but with the quotient type Isabelle cannot generate the result and it has to be shown manually. Isabelle does not generate an induction principle for the new terms automatically, but I produce one by deferring to the concrete induction principle.

Lemma 25 (`trm-induct`)**.** *Suppose P is a unary predicate on terms, and that for variables x, $P(x)$ holds, if $P(A)$ and $P(B)$ hold, then $P\,(A\ B)$ also holds, and finally that if $P(A)$ holds, then $P\,(\lambda(x:T).A)$ also holds. Then, for any M, $P(M)$.*

Proof. Transfer the hypotheses to work on the concrete level, then show $P\,(\mathrm{Abs}(X))$ by using the pre-term induction principle. The lemma then follows by lifting this result. □

Similar principles are shown for case-analysis and induction on the size of a term. *Strong induction* allows assuming that x in a binder is fresh for a given set S, which makes many proofs simpler and shorter — a formalised version of Barendregt's convention.

Lemma 26 (`trm-strong-induct`)**.** *Suppose now that P is a* binary *predicate on pairs consisting of a set of names and a term. The term is the object of induction, and S is a set of names to avoid when discussing binders. Suppose that*

1. *S is a finite set of names.*
2. *$P\,(S, x)$, for any variable x.*
3. *If $P\,(S, A)$ and $P\,(S, B)$, then $P\,(S, (A\ B))$.*
4. *If $x \notin S$ and $P\,(S, A)$, then $P\,(S, (\lambda(x:T).A))$.*

Now for any term M, $P\,(S, M)$.

Proof. First, show a more general property: that for any permutation π, $P\,(S, \pi \cdot M)$. This is argued by induction on M, using the simple induction rule proved earlier. The variable and application cases are by analogy with the simple induction, but the function binder needs special attention. In the binding case, assume that $P\,(S, \pi \cdot A)$ for any π, and show that $P\,(S, \pi \cdot \lambda(x:T).A)$. To prove this, produce a new variable y (using the freshness implementation) that is fresh with respect to the union of S, the free variables of $\pi \cdot A$, and $\pi\,\$\,z$. Now for any B, if $P\,(S, B)$, then $P\,(S, \lambda(y:T).B)$, using the assumptions — in particular, $P\,(S, \lambda(y:T).\,([y \leftrightarrow \pi\,\$\,x] \diamond \pi) \cdot A)$.

However, by using the *fn2* rule, it can be shown that

$$\begin{aligned}\lambda(y:T).\,([y \leftrightarrow \pi \mathbin{\$} x] \diamond \pi) \cdot A &= \lambda(\pi \mathbin{\$} x : T).\pi \cdot A \\ &= \pi \cdot \lambda(x:T).A\end{aligned}$$

And so $P\,(S, \pi \cdot \lambda(x:T).A)$ holds for any π. Now that the stronger result is established, the original result can be shown by setting $\pi = \varepsilon$. □

Further induction principles can be produced by combining the strong induction principle with proof by cases ("strong cases") and proof by induction on the size of a term ("strong depth induction"). Both follow immediately from the strong induction principle.

Lemma 27 (`trm-strong-cases`)**.** *Suppose P and S are defined as in Lemma 26, but now consider a term M and suppose*

1. *S is a finite set of names.*
2. *If $M = x$, or $M = (A\ B)$, then $P\,(S, M)$.*
3. *If $M = \lambda\,(x:T)\,.A$ and $x \notin S$, then $P\,(S, M)$.*

Then for any S and M, $P\,(S, M)$.

Lemma 28 (`trm-strong-depth-induct`)**.** *Suppose P and S are defined as above. Now, assume*

1. *S is a finite set of names.*
2. *$P\,(S, x)$ holds for all x.*
3. *If $P\,(S, K)$ holds for all K smaller than $(A\ B)$, $P\,(S, (A\ B))$.*
4. *If $P\,(S, K)$ holds for all K smaller than $\lambda\,(x:T)\,.A$, and $x \notin S$, $P\,(S, \lambda\,(x:T)\,.A)$.*

Then for any S and M, $P\,(S, M)$.

3.4.1 Typing judgements

An explicit typing relation is defined to ensure type inference is correct with respect to the relation. Additionally, it is easier to argue that inductive definitions respect certain properties (e.g. preservation), then show that the algorithm is correct, than it is to argue directly about the algorithm. The judgements presented in §2.2 are encoded in Isabelle:

```
inductive typing :: "'a typing_ctx ⇒ 'a trm ⇒ type ⇒ bool" where
  tvar:  "Γ x = Some τ ⟹ Γ ⊢ Var x : τ"
| tapp:  "⟦Γ ⊢ f : (TArr τ σ); Γ ⊢ x : τ⟧ ⟹ Γ ⊢ App f x : σ"
| tfn:   "Γ(x ↦ τ) ⊢ A : σ ⟹ Γ ⊢ Fn x τ A : (TArr τ σ)"
```

The usual eliminators then need to be proved manually, as the quotient type adds complexity that Isabelle's inductive-cases command cannot currently process. For example,

```
lemma typing_appE:
  assumes "Γ ⊢ App A B : σ"
  shows "∃τ. (Γ ⊢ A : (TArr τ σ)) ∧ (Γ ⊢ B : τ)"
```

provides a mechanism for reasoning about the typing judgement of an application in reverse. With these eliminators, I show the first property of the type system: weakening.

Theorem 3 (`typing-weaken`)**.** *Assume that* $\Gamma \vdash M : \tau$*, and pick* y *such that* $y \notin \mathrm{fvs}(M)$*. Then*

$$\Gamma\{y \mapsto \sigma\} \vdash M : \tau$$

for any σ *— weakening the context with another variable* y *derives the same type, providing that* y *is fresh.*

Proof. By induction on the derivation of the typing judgement. *tvar* and *tapp* follow directly, but the presence of a binder in *tfn* complicates matters. By assumption, note $y \notin \mathrm{fvs}\,(\lambda(x : \tau).A)$, so either $y = x$ or $y \notin \mathrm{fvs}(A)$. In the first case, adding the $x \mapsto \tau$ to the context overwrites the weakened context. For the second, $y \notin \mathrm{fvs}(A)$ can be combined with the induction hypothesis to show the result. □

3.4.2 Substitution and β-reduction

Some theorems about the type system of the calculus require β-reduction, which itself requires substitution to be defined first. I define a substitution relation inductively, then show that the relation is a function, then define β-reduction in terms of it. While it is possible to define substitution as a function directly, it has to be defined on pre-terms, then lifted, as Isabelle's function package does not support defining functions on equivalence classes.

```
inductive substitutes :: "'a trm ⇒ 'a ⇒ 'a trm ⇒ 'a trm ⇒ bool"
where
  var1: "x = y ⟹ substitutes (Var x) y M M"
| var2: "x ≠ y ⟹ substitutes (Var x) y M (Var x)"
| app:  "⟦substitutes A x M A'; substitutes B x M B'⟧
      ⟹ substitutes (App A B) x M (App A' B')"
| fn:   "⟦x ≠ y; y ∉ fvs M; substitutes A x M A'⟧
      ⟹ substitutes (Fn y T A) x M (Fn y T A')"
```

If a relation R is a function, it is both *total* and *single-valued*, so that R satisfies

$$\forall a.\exists b.R(a, b)$$

and

$$\forall a, b, c.R(a, b) \wedge R(a, c) \Longrightarrow b = c$$

Lemma 29 (`substitutes-total`)**.** *The substitution relation is total: for any term A, there is an A' which is the substitution of x for M in A.*

Proof. By strong induction on A, avoiding both x and the free variables of M. For the variable case, consider whether the variable is equal to x or not, and use either the *var1* or *var2* rules accordingly. Applications follow from the induction hypotheses. Finally, for functions $\lambda(y : T).B$, note that $y \neq x$ and $y \notin \text{fvs}(M)$, by the strong induction hypothesis. Hence the *fn* rule applies. □

Lemma 30 (`substitutes-unique`)**.** *The substitution relation is unique: if B and C are both substitutions of x for M in A, then $B = C$.*

Proof. By strong induction on A, avoiding x and $\text{fvs}(M)$, then directly using eliminators. □

Lemma 31 (`substitutes-function`)**.** *Substitution is a function.*

Proof. Using Lemmas 30 and 29, by definition. □

Isabelle can produce a function from this result and the relation. This is done using the definition description operator ι, which Isabelle calls "THE" — this is like the indefinite description operator, but returns the *only* such item. Then, a series of simplification lemmas are provided to allow reasoning about the function. While substitution is an executable algorithm, this definition is not exeexecutable as there is no direct definition. It may be possible to use Isabelle's code transformation with added user-defined properties to allow an extracted implementation (discussed later), but this is left as an extension.

```
definition subst :: "'a trm ⇒ 'a ⇒ 'a trm ⇒ 'a trm" ("_[_ ::= _]")
where
  "subst A x M ≡ (THE X. substitutes A x M X)"
```

I showcase the strength of the strong induction principle (and hence the whole nominal approach) by showing *Barendregt's substitution lemma* [7], which allows re-writing substitutions in a different order. It is used to show the confluence property.

Lemma 32 (`barendregt`)**.** *Assume that $y \neq z$, and that $y \notin \text{fvs}(L)$. Then*

$$M[y := N][z := L] = M[z := L][y := N[z := L]]$$

Proof. By strong induction on M, avoiding y, z, and the free variables of N and L. Normally, the proof proceeds by induction on M, deals easily with the variable and application cases, then becomes concerned with details of name clashing in the binder case. Consider $M = \lambda x.A$. At this point, substitution cannot necessarily be simplified as it may not be capture-avoiding. However, using the strong induction principle, it is provided that $x \neq y$, $x \neq z$, and x is fresh for both N and L. Therefore, the proof follows by simplification directly:

$$\begin{aligned}(\lambda x.A)\,[y := N][z := L] &= \lambda x.\,(A[y := N][z := L]) \\ &= \lambda x.\,(A[z := L][y := N[z := L]]) \\ &= (\lambda x.A)\,[z := L][y := N[z := L]]\end{aligned}$$

It can be seen here that by avoiding the names, and hence the problem, the proof is much simpler and the binder case follows directly. □

I show a result about the effect of substitution on typing, which is used while contracting a β-redex in the proof of type-preservation. This lemma also exercises the strong depth-induction principle, combining avoiding name conflicts in the proof and also assuming that the result holds for any term smaller than the term currently under consideration.

Lemma 33 (`typing-subst`)**.** *Assume that* $\Gamma \vdash N : \tau$*, and also that* $\Gamma\{z \mapsto \tau\} \vdash M : \sigma$*. Then* $\Gamma \vdash M[z := N] : \sigma$*.*

Proof. By strong induction on the size of M, avoiding z and $\mathrm{fvs}(N)$. The case of variables x follows by considering $x = z$, and the application case follows from the inductive hypothesis. For binders, the weakening result about the type system can be used to show that $\Gamma\{x \mapsto T\} \vdash N : \tau$. By assumption, $\Gamma\{z \mapsto \tau\} \vdash \lambda(x : T).A : \sigma$, and hence there is a type σ' such that $\sigma = T \to \sigma'$, and

$$\Gamma\{z \mapsto \tau, x \mapsto T\} \vdash A : \sigma'$$

Hence by the inductive hypotheses, $\Gamma\{x \mapsto T\} \vdash A[z := N] : \sigma'$, then

$$\Gamma \vdash \lambda(x : T).A[z := N] : \sigma$$

The result follows by simplification. □

I used substitution to define single-step β-reduction:

```
inductive beta_reduction :: "'a trm ⇒ 'a trm ⇒ bool" where
  beta:  "(App (Fn x T A) M) →β (A[x ::= M])"
| app1:  "A →β A' ⟹ (App A B) →β (App A' B)"
| app2:  "B →β B' ⟹ (App A B) →β (App A B')"
| fn:    "A →β A' ⟹ (Fn x T A) →β (Fn x T A')"
```

I show that types are preserved under single-step reduction, aiming towards the subject reduction property.

Lemma 34 (`preservation'`)**.** *Assume that* $\Gamma \vdash M : \tau$*, and that* $M \to_\beta M'$*. Then* $\Gamma \vdash M' : \tau$*.*

Proof. Induction on the derivation of $\Gamma \vdash M : \tau$. There are no possible beta-reductions for variables, and only one for functions. However, there are three possible ways an application can reduce: the left-hand case, the right-hand case, or the redex case. By considering these three cases and using the inductive hypotheses (Lemma 33 for the redex case), the result follows immediately in every case. □

3.4.3 Normal forms and the progress property

I define a normal-form predicate, then use it to show the progress property, a safety theorem of the type system originally presented by Milner [8], which captures the idea that well-typed terms cannot get "stuck": they are either normalised, or may be reduced further.

```
inductive NF :: "'a trm ⇒ bool" where
  var: "NF (Var x)"
| app: "⟦A ≠ Fn x T A'; NF A; NF B⟧ ⟹ NF (App A B)"
| fn: "NF A ⟹ NF (Fn x T A)"
```

Theorem 4 (`progress`)**.** *Assume* $\Gamma \vdash M : \tau$*. Then* M *is either in normal form, or there is an* M' *such that* $M \to_\beta M'$*.*

Proof. By induction on M. Variables are always in normal form, and the binder case follows directly from the induction hypothesis. In the application case, if either of the application's subterms can reduce, then the application can reduce. Alternatively, if the application forms a redex, it can also reduce. However, if none of these conditions hold, then it is in normal form, so the result holds. □

3.4.4 Many-step reduction

Many-step reduction is the reflexive, transitive closure of single-step reduction.

```
inductive beta_reduces :: "'a trm ⇒ 'a trm ⇒ bool" where
  reflexive:  "M →β* M"
| transitive: "⟦M →β* M'; M' →β M''⟧ ⟹ M →β* M''"
```

I prove the subject-reduction and safety properties, using Lemma 34 and the progress property shown in Theorem 4.

Theorem 5 (`preservation`)**.** *Assume that* $\Gamma \vdash M : \tau$*, and that* $M \to_\beta^* M'$*. Then* $\Gamma \vdash M' : \tau$*.*

Proof. By induction on the reduction $M \to_\beta^* M'$. The reflexive case follows immediately, and the transitive case from the induction hypothesis and Lemma 34. □

Theorem 6 (`safety`)**.** *Assume again that* M *is well-typed and* M *reduces in many steps to* M'*. Then* M *is either in normal form, or there is an* M'' *such that* M' *reduces to* M'' *in exactly one step.*

Proof. By induction on the many-step reduction. In the reflexive case, the results follows immediately from the progress property. For the transitive case, show that all terms in the transitive relation remain well-typed using the subject-reduction property. Then apply the progress property. □

I am now finished with the properties I stated in my project proposal. What remains is to show that the inference algorithm is correct with respect to the typing judgements, and hence that the type inference algorithm also has these properties.

3.4.5 Inference correctness

To show that the inference algorithm is correct, it has to be both *sound* and *complete*. The algorithm is sound if for any inference it makes, the same judgement can be made in the typing rules. Conversely, the algorithm is complete if for any judgement that can be made through the typing rules, it can also infer the same type.

Lemma 35 (`infer-sound`)**.** *Assume that* $\text{infer}\,(\Gamma, M) = \tau$*. Then* $\Gamma \vdash M : \tau$*.*

Proof. By induction on M. In each case, consider the pre-conditions required for the algorithm to produce τ, then use these to argue that $\Gamma \vdash M : \tau$. For example, if $M = x$, M has an inferred type iff $x \in \text{dom}(\Gamma)$. But then the typing rule for variables can also deduce this type, so the result holds for variables. Other cases are similar. □

Lemma 36 (`infer-complete`)**.** *Assume that* $\Gamma \vdash M : \tau$*. Then* $\text{infer}\,(\Gamma, M) = \tau$*.*

Proof. By induction on the typing derivation. All cases then follow by using the inductive hypothesis and the simplification rules for inference. □

Theorem 7 (`infer-valid`)**.** *The type inference algorithm is correct.*

Proof. Since it is sound and complete, the algorithm is correct, at least by reference to the typing relation. □

3.5 Extensions

3.5.1 Unit and pair terms

I added the unit value and pairs to the project, with suitable extensions to the type system. To do this, I extended the term datatype to include a unit value, a pair constructor, and projection functions for either side of a pair.

```
datatype 'a ptrm =
  PUnit
| PVar 'a
| PApp "'a ptrm" "'a ptrm"
| PFn 'a type "'a ptrm"
| PPair "'a ptrm" "'a ptrm"
| PFst "'a ptrm"
| PSnd "'a ptrm"
```

The definition of types was updated to include unit and pair types.

```
datatype type =
  TUnit
| TVar tvar
| TArr type type
| TPair type type
```

The main task for this extension was updating all the lemmas and definitions to include the new constructions. In general this was easier than writing the project over again as the semantics for the new constructions are simpler than the application and binder terms. The most interesting adaptation was new typing rules, which notably required that a projection function was only well-typed if applied to a term that was a pair type.

One major problem this extension highlighted with the approach I took with the project was the trivia that had to be written to determine that any datatype constructor was not equal to a distinct datatype constructor: for example, no pair is equal to a binder.

3.5.2 Confluence

I showed the *confluence* property. The confluence property for a reduction system such as this one states that "if A reduces in many steps to B and also to C, then there is a D such that B and C both reduce in many steps to D". There are several techniques to show this property. I took the approach taken by Takahashi [25] in his work on the λ-calculus which defines two new reduction relations, parallel reduction and complete development. I followed Pollack's overview [26] of the technique for a simple untyped calculus and extended it to my calculus.

First, I define parallel reduction (written $A \gg B$). Intuitively, this relation is β-reduction, but at any step where reductions on subterms as well as the main term is possible, it performs all at once.

```
inductive parallel_reduction :: "'a trm ⇒ 'a trm ⇒ bool" where
  refl: "A >> A"
| beta: "⟦A >> A'; B >> B'⟧ ⟹ (App (Fn x T A) B) >> (A'[x ::= B'])"
| eta:  "A >> A' ⟹ (Fn x T A) >> (Fn x T A')"
| app:  "⟦A >> A'; B >> B'⟧ ⟹ (App A B) >> (App A' B')"
| pair: "⟦A >> A'; B >> B'⟧ ⟹ (Pair A B) >> (Pair A' B')"
| fst1: "P >> P' ⟹ (Fst P) >> (Fst P')"
| fst2: "A >> A' ⟹ (Fst (Pair A B)) >> A'"
| snd1: "P >> P' ⟹ (Snd P) >> (Snd P')"
| snd2: "B >> B' ⟹ (Snd (Pair A B)) >> B'"
```

Where the relation needed extending, I kept with this intuition and reduce sub-terms in rules wherever possible — note *fst2* and *snd2*.

Next, the complete development relation (written $A \ggg B$) is similar to parallel reduction, but written in such a way as to remove any ambiguity as to which rule applies. Consider that $A \gg A'$, by some rule. However, it can also reduce by *refl* to itself: complete developments remove this ambiguity.

```
inductive complete_development :: "'a trm ⇒ 'a trm ⇒ bool" where
  unit: "Unit >>> Unit"
| var:  "(Var x) >>> (Var x)"
| beta: "⟦A >>> A'; B >>> B'⟧ ⟹ (App (Fn x T A) B) >>> (A'[x ::= B'])"
| eta:  "A >>> A' ⟹ (Fn x T A) >>> (Fn x T A')"
| app:  "⟦A >>> A'; B >>> B'; (⋀x T M. A ≠ Fn x T M)⟧
      ⟹ (App A B) >>> (App A' B')"
| pair: "⟦A >>> A'; B >>> B'⟧ ⟹ (Pair A B) >>> (Pair A' B')"
| fst1: "⟦P >>> P'; (⋀A B. P ≠ Pair A B)⟧ ⟹ (Fst P) >>> (Fst P')"
| fst2: "A >>> A' ⟹ (Fst (Pair A B)) >>> A'"
| snd1: "⟦P >>> P'; (⋀A B. P ≠ Pair A B)⟧ ⟹ (Snd P) >>> (Snd P')"
| snd2: "B >>> B' ⟹ (Snd (Pair A B)) >>> B'"
```

It is always the case that a term can be reduced by the complete development relation; this holds by *refl* for parallel reduction, but is also true for complete developments.

Lemma 37 (`complete-development-exists`)**.** *For any term A, there is an A' such that $A \ggg A'$.*

Proof. By induction on the structure of A. In each case, choose a rule based on the structure, then obtain A' by means of that rule and the induction hypothesis. □

To show the confluence property, I need the *diamond property* for parallel reduction, an auxiliary lemma is required first about decomposing a complete development into two parallel reductions.

Lemma 38 (`complete-development-triangle`)**.** *Suppose $A \ggg D$ and $A \gg B$. Then $B \gg D$.*

Proof. By induction on the derivation of $A \ggg D$. For each case, consider how A might have reduced under parallel reduction to B, obtain a value for B, then show that $B \gg D$ by any relevant rule. □

It can now be shown that parallel reduction has the diamond property: that diverging reductions can always be re-united by another step.

Lemma 39 (`parallel-reduction-diamond`)**.** *Assume that $A \gg B$ and $A \gg C$. Then there is a D such that $B \gg D$ and $C \gg D$.*

Proof. Obtain a D such that $A \ggg D$, since this always exists. Hence by the previous lemma, both $B \gg D$ and $C \gg D$. Thus the diamond property holds for $(\gg)$. □

I define $(\gg^*)$ as the reflexive-transitive closure of $(\gg)$. It is shown by a diagram chase (Figure 3.2) that $(\gg^*)$ has the diamond property, so what remains is to show that $(\gg^*) = \left(\rightarrow_\beta^*\right)$. This is done by exhibiting a bidirectional conversion:

Lemma 40 (`parallel-reduction-is-beta-reduction`)**.** *If $A \gg B$, then $A \rightarrow_\beta^* B$.*

Proof. By induction on the derivation of $A \gg B$. Each case can be converted straightforwardly to zero or more β-reductions. □

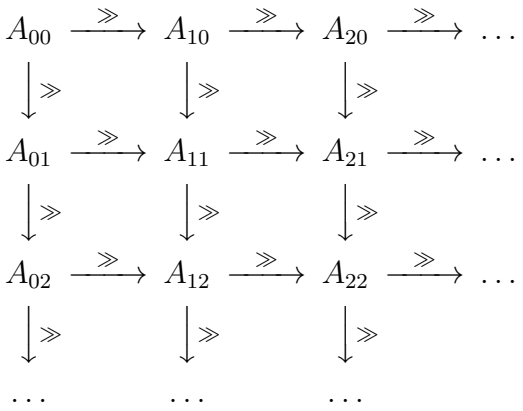


Figure 3.2: The diamond property of $(\gg^*)$ can be shown using the diamond property of $(\gg)$ by induction on the (conceptual) rows of this commutative diagram, then by another induction on the columns.

Lemma 41 (`beta-reduction-is-parallel-reduction`)**.** *If $A \to_\beta B$, then $A \gg B$.*

Proof. By induction on the derivation of the β-reduction. Each case forms part of the corresponding parallel reduction, and *refl* allows reducing subterms that the β-reduction does not reduce. □

Lemma 42 (`parallel-reduction-beta-reduces-equivalent`)**.** $\left(\to_\beta^*\right)$ *and* $(\gg^*)$ *are equivalent.*

Proof. Directly from the previous pair of lemmas. □

The confluence property follows from this and the diamond property of the closure of parallel reduction.

Theorem 8 (`confluence`)**.** *Suppose $A \to_\beta^* B$ and $A \to_\beta^* C$. Then there exists a D such that $B \to_\beta^* D$ and $C \to_\beta^* D$.*

Proof. By assumption, $A \gg^* B$ and $A \gg^* C$, since the two relations are equivalent by Lemma 42. Then obtain a D such that $B \gg^* D$ and $C \gg^* D$, using the diamond property of $(\gg^*)$. Finally, note that $B \to_\beta^* D$ and $C \to_\beta^* D$, again using the equivalence. □

3.6 Summary

In this chapter I implemented the formal parts of my project. I provided implementations of freshness (§3.1) and permutations (§3.2) to use for subsequent stages in the project. Then, I used these to define operations on concrete λ-terms (§3.3), and produced a new term, quotiented by α-equivalence (§3.4), and typing judgements on these terms. I then implemented a type inference algorithm (§3.3.2), lifted it to the quotient terms, and proved it correct. I also added some extensions to the original task, namely unit and pair types (§3.5.1), and a proof of confluence (§3.5.2).

Chapter 4

Evaluation

While my implementation is *verified*, there are several places in which it can still fail. The project as a whole rests on the correctness of Isabelle's logical kernel, and on the correctness of its code extraction mechanism. If either of these (although the code extraction mechanism is far more likely) were to be shown incorrect, my project is also potentially incorrect. In addition, the definitions encoded within Isabelle may not be definitions that match the conventional definitions — they could be technically incorrect, surprising, or vacuous. However, there are other metrics to the project's success other than its relative truth:

- Practical examples: while the implementation is theoretically correct with respect to a set of definitions, these definitions must correspond to an intuitive understanding of what the implementation should do. Therefore, manually-written tests should check the expected properties of the calculus to avoid vacuous or controversial definitions.
- Performance: extracted algorithms should perform well, at least asymptotically. It is difficult to argue this directly in a proof assistant, so it is easier to produce empirical data and show that statistically the algorithm works as expected.
- Comparison to other work: there is existing academic work in this general area, and comparisons must be drawn to highlight successes and failures in my approach. The choice of representation especially in the project is somewhat unusual, and comes with advantages and disadvantages compared to other approaches.

The majority of the practical evaluation was directed at the extracted type inference algorithm, but there is also some testing of freshened variables and the representation of the calculus itself.

4.1 Framework for evaluation

Isabelle supports four programming languages in its code-extraction machinery [27]: Standard ML, OCaml, Haskell, and Scala. Haskell was the chosen language for evaluating the type inference algorithm: it offered better library and runtime support for testing

and benchmarking than Standard ML and OCaml, but remained closer to the algorithm expressed in Isabelle than Scala (which has a more complex type system, and is object-oriented). To build and evaluate the extracted code, I used Stack [28], a tool for Haskell that provides sandboxing, compiler and package isolation, build systems, and support for running tests and benchmarks, in combination with GHC [29], the *de facto* standard Haskell compiler.

Isabelle's code extraction mechanism immediately produced correct code that resembled the formalisation very well without further tweaking, but there were several problems that needed to be fixed without touching (and thereby invalidating) the extracted code. I added a new module *Utils* to the extracted code which provided some utilities for testing and benchmarking and fixed the problems:

- The extracted code initially wouldn't compile, as it produced invalid Haskell type signatures that are neither Haskell '98 nor Haskell 2010. The type signatures added explicit universal quantification to type variables, where usually the compiler would infer that all type variables were implicitly universally quantified. For example, the polymorphic identity function in standard Haskell

  ```
  id :: a -> a
  id x = x
  ```

  became

  ```
  id :: forall a. a -> a
  id x = x
  ```

  However, this could be made to build using the *ExplicitForAll* GHC extension, which allows this syntax (albeit only to facilitate more useful extensions).

- Extracted code often implemented its own version of standard Haskell typeclasses, (such as that for partial orders), implemented but did not instantiate a typeclass, or did not implement/export useful functions (such as serialisation functions). In all cases typeclass instances could be provided easily: standard Haskell typeclasses can be implemented in terms of the custom typeclasses,

  ```
  instance Ord Nat where
    (<=) = Orderings.less_eq
  ```

  instances can be provided *ex post facto* in Haskell,

  ```
  instance Eq Nat where
    (==) = equal_nat
  ```

and trivial typeclass instances can be implemented manually, or, more efficiently, using the *StandaloneDeriving* extension.

```
deriving instance Show Nat
deriving instance Read Nat
```

- In a similar vein, despite extracting both a typeclass for fresh variables and an implementation of the typeclass for natural numbers, the code extraction failed to join the two, so

```
instance Fresh Nat where
  fresh_in = fresh_in_nat
```

had to be added manually to use the freshness implementation.

4.2 Practical examples and property testing

A common method of testing software is unit testing, in which code is subjected to single pass/fail test units. This works well for most cases, but the mathematical nature of the domain means that lots of test cases ought to be universally quantified over their inputs, rather than a fixed constant as in unit testing. Property testing, on the other hand, generates random test data and checks a property holds for all of them. This technique is better suited to the problem: it approximates universal quantification, and the random inputs find edge-cases quickly.

To implement property testing, I used HSpec [30], a library for structuring Haskell tests, combined with QuickCheck [31], a library for property testing. A test program using these libraries looks like Figure 4.1. HSpec and QuickCheck together provide some useful functions which I use to write my tests:

`hspec` — run a suite of tests

`describe` — group a set of tests by which "feature" they belong to

`it` — describe a single test

`property` — test a QuickCheck property in HSpec

`forAll` — check that a property holds when quantified over a given generator

One problem with property testing à-la-QuickCheck is that random generators for all tested types have to be written. QuickCheck provides generators for all basic types, and some derived ones via a typeclass mechanism (e.g. if you have generators for types t_1 and t_2, you also have a generator for $t_1 \times t_2$), but any new types introduced have to have custom generators written. I wrote generators for natural numbers, types, typing contexts, and pre-terms, and also for ill-typed and well-typed terms.

```
main :: IO ()
main = hspec $ do
  describe "unit inference" $ do
    it "infers unit values to have unit type" $ do
      property $
        forAll contexts $ \c ->
        infer' c PUnit `shouldBe` pure TUnit
```

Figure 4.1: an example test program: this asserts that, for all typing contexts c, the inferred type of a unit value is the unit type

With these generators, I then wrote tests that asserted, in the context of various categories, the following propositions — note the similarity in structure to the inductive typing judgements. Write $\text{infer}(\Gamma, M)$ for the result of running the inference algorithm with context Γ on term M, and use $\bot$ to indicate a type error. Consider all free variables universally quantified (i.e. randomly-generated in the test).

- fresh variables:
  - a fresh x generated with respect to S has the property $x \notin S$
- unit inference:
  - $\text{infer}(\Gamma, \text{unit}) = 0$
- variable inference:
  - any variable x satisfies $\text{infer}(\Gamma, x) = \bot$
  - if x is fresh in the domain of Γ, $\text{infer}(\Gamma, x) = \bot$
  - if there is a τ such that $\Gamma(x) = \tau$, then $\text{infer}(\emptyset, x) = \tau$
- λ-inference:
  - if $\text{infer}(\Gamma\{x \mapsto \tau\}, M) = \bot$, then $\text{infer}(\Gamma, \lambda(x : \tau).M) = \bot$
  - for all variables x, y, $x \neq y$ implies that y is not bound in the expression $\lambda(x : \tau).y$
  - if $\text{infer}(\Gamma\{x \mapsto \tau\}, M) = \sigma$, then $\text{infer}(\Gamma, \lambda(x : \tau).M) = \tau \to \sigma$
- application inference:
  - if $\text{infer}(\Gamma, M) = \bot$, $\text{infer}(\Gamma, (M\ N)) = \bot$
  - also, if $\text{infer}(\Gamma, N) = \bot$, $\text{infer}(\Gamma, (M\ N)) = \bot$
  - if $\text{infer}(\Gamma, M) = T$ and T is not of the form $\tau \to \sigma$, then $\text{infer}(\Gamma, (M\ N)) = \bot$
  - if T *is* of the form $\tau \to \sigma$, but $\text{infer}(\Gamma, N) \neq \tau$, then $\text{infer}(\Gamma, (M\ N)) = \bot$
  - finally, if $\text{infer}(\Gamma, N) = \tau$, then $\text{infer}(\Gamma, (M\ N)) = \sigma$

- pair inference:
  - if $\text{infer}(\Gamma, M) = \bot$, then $\text{infer}(\Gamma, (M, N)) = \bot$
  - also, if $\text{infer}(\Gamma, N) = \bot$, then $\text{infer}(\Gamma, (M, N)) = \bot$
  - if $\text{infer}(\Gamma, M) = \tau_1$ and $\text{infer}(\Gamma, N) = \tau_2$, then $\text{infer}(\Gamma, (M, N)) = \tau_1 \times \tau_2$
- projection inference:
  - if $\text{infer}(\Gamma, M) = \bot$, then $\text{infer}(\Gamma, \pi_1(M)) = \bot$ and $\text{infer}(\Gamma, \pi_2(M)) = \bot$
  - if $\text{infer}(\Gamma, M) = T$, but T is not of the form $\tau_1 \times \tau_2$, then $\text{infer}(\Gamma, \pi_1(M)) = \bot$ and $\text{infer}(\Gamma, \pi_2(M)) = \bot$
  - if T is of the form $\tau_1 \times \tau_2$, then $\text{infer}(\Gamma, \pi_1(M)) = \tau_1$ and $\text{infer}(\Gamma, \pi_2(M)) = \tau_2$
- testing assumptions
  - if a term M is generated from the ill-typed pool, $\text{infer}(\Gamma, M) = \bot$
  - on the other hand, if M is generated from the well-typed pool, $\text{infer}(\Gamma, M) = \tau$ for some τ

All tests pass, producing output similar to that shown in Figure 4.2. While property tests can never provide absolute confidence in the project (in this case the parameter space is infinite), they do produce empirical evidence to *suggest* correctness.

4.3 Benchmarking and performance

Inputs to the type inference algorithm can be given a notion of size by counting the number of nodes in the syntax tree of the input. For instance, the term $\pi_1((\lambda(x : 0).(x, \text{unit}))\ \text{unit})$ is shown in Figure 4.3 and has 7 syntactic nodes. Formally, the size of a term M, $|M|$, can be given recursively by

$$|M| = \begin{cases} 1 & M = \text{unit} \\ 1 & M = x \\ 1 + |M'| & M = \lambda(x : \tau).M' \\ 1 + |A| + |B| & M = (A\ B) \\ 1 + |A| + |B| & M = (A, B) \\ 1 + |P| & M = \pi_1(P) \\ 1 + |P| & M = \pi_2(P) \end{cases}$$

Given this, the formalised type inference algorithm can be seen (by induction) to have time complexity $O(|M|)$ with respect to the input M, assuming that finite map update/lookup operations are $O(1)$. However, it is difficult to see how to show this formally in Isabelle, and there is no reason to worry about performance until the code is extracted anyway — at which point the code may have drastically different performance characteristics than the idealised formal specification. This is because Isabelle does not

```
fresh variables
  fresh in a set
unit inference
  infers unit values to have unit type
inference of variables
  undefined in the empty context
  undefined if fresh in a context
  infers the correct type if in a context
inference of lambdas
  propagates type errors
  doesn't bind extraneous variables
  infers a correct type
inference of applications
  propagates type errors on the left
  propagates type errors on the right
  undefined for non-arrow application
  undefined for type mismatch
  infers correct type
inference of pairs
  propagates type errors on the left
  propagates type errors on the right
  infers correct type
inference of projection
  propagates type errors
  undefined for non-pair application
  infers correct type
testing assumptions
  ill-typed terms are ill-typed
  well-typed terms are well-typed

Finished in 48.3063 seconds
21 examples, 0 failures
```

Figure 4.2: output from property testing

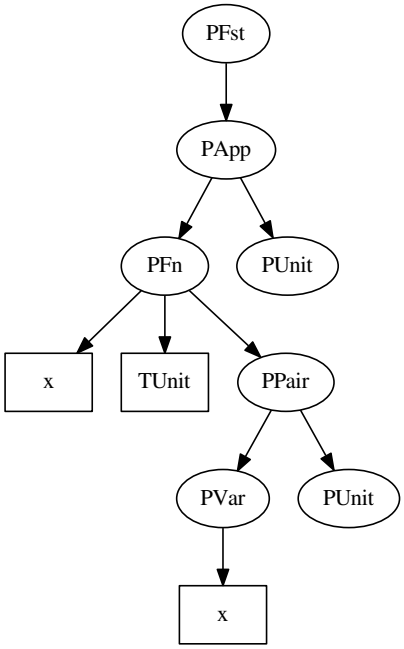


Figure 4.3: an example syntax tree: syntactic nodes are shown by ellipses, with incidental data in boxes

directly "project out" the function definition (as type-theoretic proof assistants such as Coq [32] often do with proof terms), but instead performs a series of translations. The steps [33] are as follows:

1. Re-write the function definition using a set of rules encoded in a higher-order rewrite system — a rewrite system on typed λ-terms — these may be automatically generated by e.g. the function definition, or manually by proving a rewriting to hold in Isabelle.

2. Once the re-writing has taken place, the result is encoded in an intermediate language similar to Haskell, known as Mini-Haskell.

3. If the target language is one without typeclasses (i.e. not Haskell), then the typeclasses in Isabelle are translated into a dictionary structure using a translation [34] originally used to specify Haskell's typeclasses.

4. Finally, the resulting intermediate language statements are translated directly into functional programming languages which allow for pure code: otherwise, the translations would not be valid.

Any of these steps may cause unpredictable performance characteristics. Therefore, we are left to try and establish performance characteristics of the extracted code by the scientific method, rather than mathematically.

4.3.1 Experimental method

There is a problem in that asymptotic complexity cannot be determined by any performance measurements on finite inputs. However, since type inference is only likely to be run on terms of a reasonable size, say, no more than 10^5, we can state that it has a certain complexity within a given range of sizes, then conjecture that this extends to an asymptotic result.

The experimental approach, is as follows:

1. Generate representative test data of several different orders of magnitude in the reasonable range.

2. Benchmark the algorithm on each of these test inputs and record performance.

3. Correlate (logarithmic) runtime against (logarithmic) input size to determine an asymptotic performance bound. A log/log plot is used since the range of values is too large to plot reasonably, but I do not wish to disturb the relationship between the data points. Plotting both logarithmically allows us to display the relationship between inputs and runtime of several different orders of magnitude.

Ideally the data generated would be (pseudo-) random, in order to prevent any patterns introduced in the test data from affecting the benchmark: the generators used to produce test data could then be re-purposed to benchmark data. Another constraint is that the expression must be well-typed in general, or the error propagating through will remove most of the processing from the benchmark.

Unfortunately, QuickCheck's random generation procedures turned out to be too slow to produce large randomised inputs efficiently. QuickCheck uses the standard random package for generic applications which is not optimised for speed, and QuickCheck itself is not designed to produce large inputs, as its main use case is finding edge-cases (which tend to be smaller). Benchmarking showed that random number generation was the bottleneck, but perhaps this would be different if QuickCheck were designed differently. Instead, I used deterministic algorithms to produce large inputs, according to a variety of patterns:

1. well-typed terms, using all datatype constructors

2. pairs, with each sub-term a pair

3. projection functions applied to pairs repeatedly

4. applications, with each sub-term of equal size

5. applications, with unbalanced sub-terms

6. function binders in a chain

7. a sequence of binders, followed by bound variables

```
benchProjections :: Int -> Term
benchProjections n
  | (n <= 0) = PUnit
  | (n `mod` 2 == 0) = PSnd . PPair PUnit $ benchProjections (n - 2)
  | otherwise = PFst . flip PPair PUnit $ benchProjections (n - 2)
```

Figure 4.4: The Haskell function generating a test of projection performance based on a size parameter.

These are designed to test general performance on varied inputs, on specific inputs, and finally to stress the typing context operations. I then used the patterns to generate data, sized from 1, increasing by a factor of 10 to 100,000, and wrote the data to file for reproducibility (available under `infer/samples` in the source tree submitted with this dissertation).

Producing well-typed terms was quite tricky, even for this simple type system. The general approach was for each generator to assume that any recursive calls generated well-typed terms, and maintain the invariant that the result it generates is also well-typed. Then, the matter of choosing which sort of node was to be produced at any step in the recursive algorithm was done by taking the remainder of the size required and producing the corresponding type of data, which produced a well-distributed tree for a deterministic algorithm. For instance, the algorithm for testing projections is shown in Figure 4.4.

Benchmarking algorithms on small inputs ("micro-benchmarking") has the problem of jitter: outside factors such as cache effects, branch prediction and kernel scheduling can all cause the same algorithm with the same input to run at different rates.

Haskell's laziness also adds another problem — since the parameter passed to the function is not strictly-evaluated, the first evaluation of the function can take significantly longer as in this case the input needs to be loaded from disk and parsed before being processed.

Both these problems were solved by using the Criterion micro-benchmarking library [35]. This solution deals with jitter by running the algorithm repeatedly and differing numbers of times to generate enough samples for a statistically significant result (generally an R^2 value exceeding 0.99), and with laziness by evaluating the argument to a normal form first.

4.3.2 Results

Results were extremely positive for the $O(|M|)$ hypothesis. Figure 4.5 shows a strong linear correlation for the general-performance dataset, while Figure 4.6 supports this further, showing that every sort of structure produces linear performance on the target range. The high levels of correlation on every graph lend extra credit to the hypothesis of a linear-time algorithm. While it is possible that the algorithm is, in fact, super-linear (or sub-linear!), *on the input range* the algorithm correlates well to a linear trend.

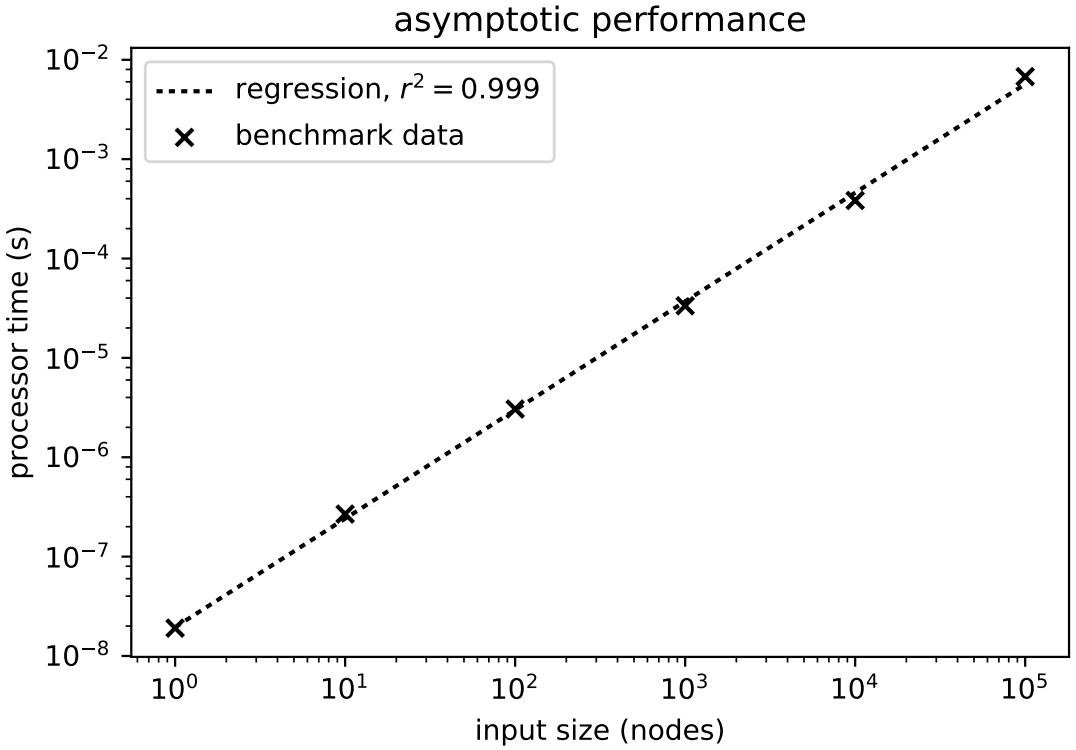
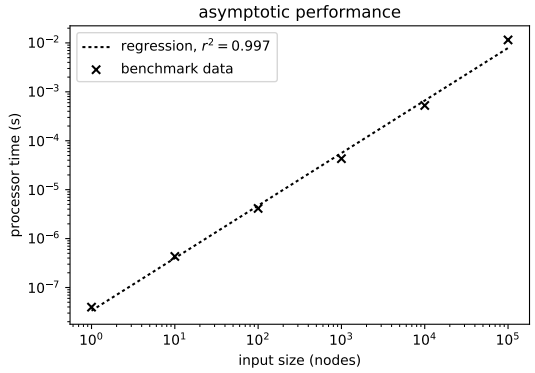


Figure 4.5: log-log plot of benchmark data size vs processor time, with a linear regression fitted

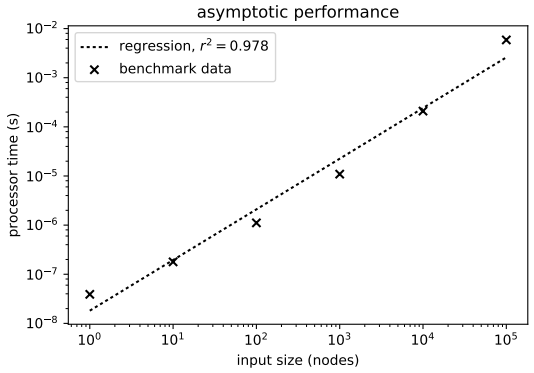
Of particular interest is the context stress-test: this was included as the extracted code uses a method of representing finite maps using closures that should be extremely inefficient. It implements the lookup and update operations as follows:

```
type Context k a = k -> Maybe a

lookup :: Context k a -> k -> Maybe a
lookup c k = c k

update :: Eq k => Context k a -> k -> a -> Context k a
update c k a = \x -> if x == k then Just a else c x

empty :: Context k a
empty = \x -> Nothing
```

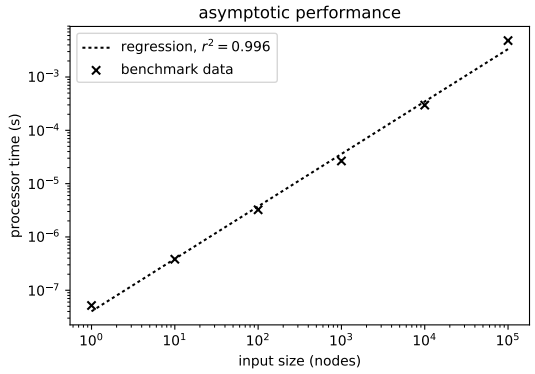
In the case of the extracted inference algorithm, this implementation of map should result in quadratic slowdown when applied to binders followed by variable lookup, as the closure representing the finite map grows in size. Unexpectedly, in the typing-context dataset, this does not appear to be the case: even profiling the executable does not show significant time spent in the closure, or in the equality implementation. It is possible that the compiler is able to spot the idiom of using a closure as a dictionary and optimise this



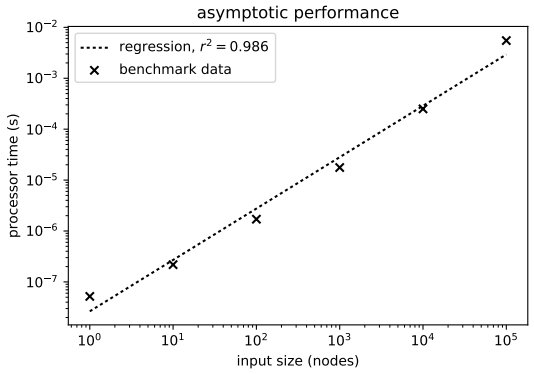
(a) pairs



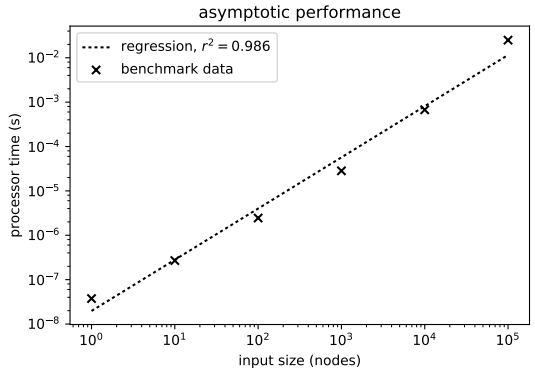
(b) projections



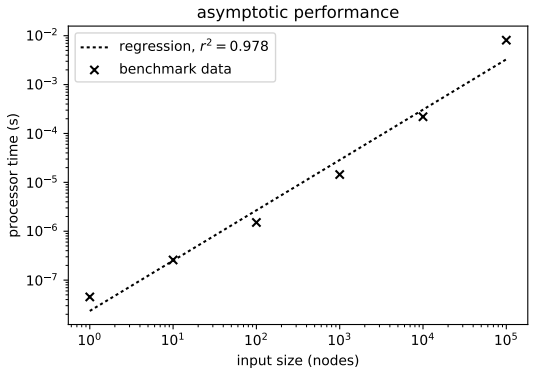
(c) balanced applications



(d) applications, biased to one side

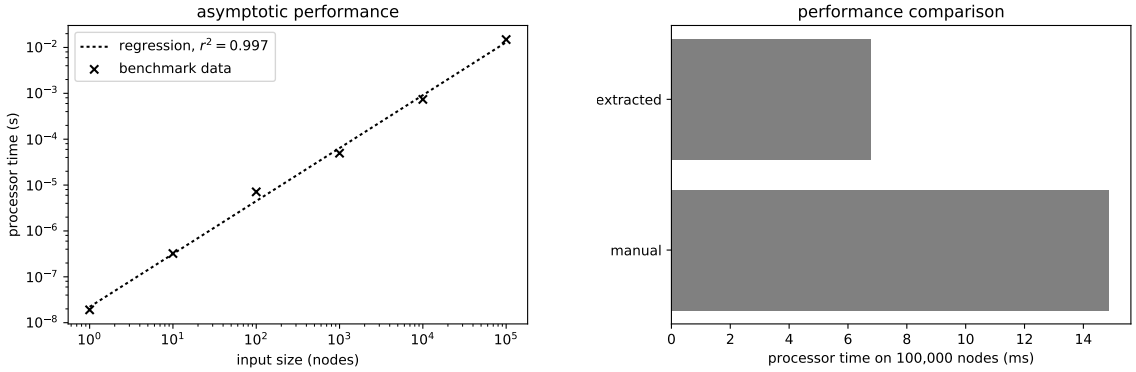


(e) chain of function binders



(f) typing context stress

Figure 4.6: log-log plots (similar to Figure 4.5), showing performance on specialised inputs



(a) asymptotic performance

(b) comparison against extracted code

Figure 4.7: plots showing the performance of the manual implementation

away to a lookup table, or at least mitigate the poor performance of repeatedly branching to a new closure while looking up a key. However, this seems very unlikely given the current level of optimisations possible. GHC does use a somewhat unusual execution model [36] as a compilation step, which may offer a performance improvement in this case, compared to the naïve idea of how such an algorithm is compiled to a physical architecture. More likely is that the input size is simply not large enough yet to cause the linear-time lookup of the context to become significant compared to other (constant-time) parts of the algorithm.

However, I would still expect a reasonable implementation with e.g. an ordered map to be more performant: Figure 4.7 shows that a hand-implemented version using Haskell's *containers* ordered-map implementation still matches a linear trend in the size of the input (as expected, since context lookup was not significant with the original implementation), but the implementation is considerably slower than the naïve version. It seems that context lookup may well be inefficient, but on reasonable input sizes it does not take enough time to become significant.

4.4 Comparison to previous work

There have been other Isabelle formalisations of typed λ-calculi. Isabelle has its own implementation (in `src/HOL/Proofs/Lambda/` in the Isabelle source tree), which I will use as a context in which to evaluate my project — using another implementation written with different proof tools would confuse the issue, as different tools produce different approaches to play to their strengths and weaknesses.

Despite using the same tooling, the two implementations are still very different: Isabelle's implementation uses chained, tactic-style reasoning almost exclusively, introduces types separately to terms, and uses a nameless representation of bound variables, whereas my project uses explicit reasoning steps almost exclusively, attaches an explicit type to every binder, and uses a named representation, combined with the use of a quotient type to represent α-equivalence.

4.4.1 Chained tactics vs. Isar

Using only tactics does tend to make the proof shorter (as can be seen from the proof lengths for comparable lemmas in both implementations: subject reduction is shown in approximately 15 lines in an apply-style proof, but my more explicitly-reasoned proof takes over 90), but can also (albeit subjectively) decrease readability of the generated proof [21]. It should be noted that line count is not in general a good metric for proof length: the use of previous results and different proof approaches, proof steps, and tools can affect line count significantly. Another advantage of the Isar approach is ease of learning: if I could not get a tactic to show what I wanted, it was easier to break down the proof statement into smaller steps using Isar and invoke a prover on each than it was to learn what sequence of tactic applications would produce the desired result.

4.4.2 Church- vs. Curry-style types

The Isabelle implementation actually formalises the untyped calculus, before adding simple types afterwards. This approach allows for multiple type systems to be implemented on top of a base that deals only with untyped operations, a great improvement on my project, which would need to be re-written totally in order to add a significant change to the type system.

However, not adding explicit types to binders does produce a marked difference in what the theory shows: this is now Curry-style typing, where the question for type schemes given a term M is no longer the Church-style "what is the type of M?", but "what is the set of possible types for M?". Of course, this is intentional — but if a Church-style typing system is desired, then this approach is not possible.

One way to achieve this modularity in a Church-style context would be to parameterise the datatype for representing terms over all possible datatypes for representing types, as well as variables. With this modification, adding a new type system would involve only defining a new datatype representing the new types, and building the new typing system around the term parameterised by these new types.

Even more generally, most of the work with binders and substitution could be done on an abstract notion of terms-with-names. Consider defining *abstract terms* $\mathcal{A}$ to be either

1. a variable (bound or free) x
2. a pair $(\mathcal{A}, \mathcal{A})$
3. a binder $x_C.\mathcal{A}$, binding a variable in an abstract term with some "context" C

This is sufficient to define α-equivalence, substitution and so forth. Then define a bijection f between abstract terms and the concrete terms of the simply-typed calculus as follows:

$$\begin{aligned} f(x) &= x \\ f((M\ N)) &= (f(M), f(N)) \\ f(\lambda(x : C).M) &= x_C.f(M) \end{aligned}$$

and the obvious definition of f^{-1}. All reasoning purely about operations involving names can now be done on the abstract terms, then added to the concrete terms by defining the concrete operation in terms of the abstract operation and f.

By way of example, suppose the notion of substitution on abstract terms has been defined, and that we have proven Barendregt's substitution lemma — that is, if $x \neq y$ and x is not free in $\mathcal{L}$, then

$$\mathcal{M}[x ::= \mathcal{N}][y ::= \mathcal{L}] = \mathcal{M}[y ::= \mathcal{L}][x ::= \mathcal{N}[y ::= \mathcal{L}]]$$

It can now be shown without proving the lemma again that this property holds for the concrete terms as well. Define substitution on concrete terms (using a different notation for clarity) by reference to substitution on abstract terms, using f as follows:

$$M[N/x] \equiv f^{-1}\left(f(M)[x ::= f(N)]\right)$$

Then the lemma holds easily, since

$$\begin{aligned}
M[N/x][L/y] &\equiv \left(f^{-1}\left(f(M)[x ::= f(N)]\right)\right)[y ::= L] \\
&\equiv f^{-1}\left(f\left(f^{-1}\left(f(M)[x ::= f(N)]\right)\right)[y ::= f(L)]\right) \\
&\equiv f^{-1}\left(f(M)[x ::= f(N)][y ::= f(L)]\right) \\
&\equiv f^{-1}\left(f(M)[y ::= f(L)][x ::= f(N)[y ::= f(L)]]\right) \\
&\equiv f^{-1}\left(f\left(M[L/y]\right)[x ::= f(N)[y ::= f(L)]]\right) \\
&\equiv f^{-1}\left(f\left(M[L/y]\right)[x ::= f\left(N[L/y]\right)]\right) \\
&\equiv M[L/y][N[L/y]/x]
\end{aligned}$$

These terms also form a nominal set with similar structure to the original calculus, so all results still hold with little modification. In this way a level of modularity and re-usability can be achieved, without sacrificing Church-style typing. Currently, the amount of project-specific Isabelle stands at around 4000 lines, with around 500 lines of reusable theories. With this modification, the majority of work would be re-usable, and the only project-specific work to deal with names would be defining a bijection with these abstract terms and any language with similar semantics about binders. While this idea is my own, it was previously found in a more general form by Gabbay *et al* [37]. Implementing this approach as a library in Isabelle is left as future work.

4.4.3 Approaches to binders

The treatment of binders is a difficult part of the formalisation of any system that uses them. The authors of the PoplMark challenge [6] specifically mention this topic, and note that, of the solutions they have seen, there was no clear winner.

It can be seen from the extremely-small Isabelle implementation of de Bruijn indices that a nameless representation (where α-equivalence is simply equality) is much easier to formalise initially than the approach taken in my project — no permutation lemmas, no α-equivalence arguments, no quotient types and so on. However, the subsequent effort required to argue simple theorems such as the substitution lemma described above with de

Bruijn indices is significant. Berghofer and Urban argue [38] that a nominal representation has many advantages over indices, particularly in the context of the PoplMark challenge, once the initial infrastructure has been set up. They mention that de Bruijn indices are convenient to define, but become tedious after a while (at least in the context of PoplMark), while nominal techniques have a significant setup cost but are convenient thereafter in general. However, the paper does not draw a significant conclusion as the scope of usage is limited to only a few problems.

Assuming that Berghofer and Urban's conclusions and my own are correct, then, nameless representations are a good choice for (shorter) implementations that do not use the binding aspects of λ-calculus much, whereas the initial effort of nominal methods produce easier results when applied to more difficult theorems about binding. Additionally, the named representation is arguably more human-readable and presents less notational impedance than the nameless version, promoting later re-use.

4.4.4 Nominal implementation

An important comparison to draw is that of a manual approach to a nominal implementation, with that of Nominal Isabelle. I chose to perform things manually, as neither Nominal Isabelle, nor its successor Nominal 2 support code generation at the time of writing, and code generation was a required part of the project.

Unfortunately, this led to a lot of effort, that while educational, could have been avoided with use of the automation provided by Nominal Isabelle. I had to manually implement:

- A theory of permutations. While a theory was added to Isabelle's library after I began my project, initially there was no implementation of permutations in the standard library.
- (strong) Induction principles for α-equivalent terms.
- A quotient type.
- Inequality rules for these terms, such as $\forall A, B.\ (A, B) \neq \text{unit}$ — these are particularly unfortunate as the number of lemmas required grow quadratically with the number of different varieties of term.
- Structural equality rules, such as $\forall A, B.\ (\pi_1(A) = \pi_1(B) \implies A = B)$.
- Proofs showing that functions on pre-quotient terms can also be functions on α-equivalence classes, like the type inference function, then lifting them over the quotient. This property is not true in general (consider the function that returns the bound variables of a term), but Nominal Isabelle provides a couple of ways to define functions which generate a proof obligation to show this property [39].

Nominal Isabelle would have saved a considerable amount of effort, and made several tedious aspects of the project significantly shorter. However, the ability to extract code allowed the project to be more practically useful, and testing this allowed for another dimension that would not have been available with Nominal Isabelle.

4.5 Lessons learned

I learned several lessons during the course of the project. The first and most important, was that the treatment of names in any language featuring binders is a large aspect of formalising the language. Considerable thought must be dedicated to correctly-formalising the precise semantics of an area that is traditionally avoided as mathematically uninteresting.

Secondly, there is a tradeoff between preparatory work and ease of use: nameless representations of the λ-calculus allow for an easy definition of both the calculus and α-equivalence, but become tedious to use later on. Nominal techniques require a lot of initial work to set up, but make subsequent work easier by comparison to nameless representations.

More practically, extracting code from formalised implementations produces correct code, but it may have unusual performance characteristics that do not match the idealised versions.

4.6 Summary

In this chapter, I prepared the project for evaluation (§4.1), ran some property tests to add confidence of the project's correctness (§4.2). Moving on to benchmarking, I produced some benchmark data, then ran benchmarks to determine both asymptotic and relative performance of the inference algorithm (§4.3). Finally, I evaluated techniques for implementing this project's brief against similar work (comparing proof styles, church and curry-style typing, approaches to binders, and automatic versus manual nominal implementation) in Isabelle (§4.4), before finishing with the lessons I learned from the project (§4.5).

Chapter 5

Conclusion

I have shown the development of a formalised implementation of a typed λ-calculus in the proof assistant Isabelle, complete with correctness properties about the type system and verified, extracted, code for type inference. All success criteria have been met, and some extensions have been made, augmenting the core calculus and showing the confluence property. This work differs from a typical implementation in its use of nominal techniques that have several advantages over other methods of name binding.

5.1 List of results

Taking success criteria from my project proposal, they have all been met:

1. I have now learned sufficient theory to understand, implement, and justify my approach to the problem.
2. I gained sufficient practical experience before and during my project about the Isabelle proof assistant to efficiently implement the project.
3. The representation of the calculus I chose has been sufficient to produce the rest of my dissertation with.
4. I have proven the progress, preservation, and safety properties of the type system.
5. The implementation of type inference has been verified by showing it equivalent to the inductive typing rules.
6. The extracted Standard ML code does compile and run as expected. Although Haskell was the language I eventually used for testing, I don't consider that this change of decision disqualifies this success criterion.
7. The dissertation is complete.

5.2 Further work

There is scope for further work in this area, and any one of several areas could be pursued. Improving the nominal approach is one possible extension, perhaps adding some automation to remove some of the painful points. Nominal Isabelle could be improved/extended so that code extraction is possible, or alternatively the abstract-term approach could be implemented so that only relevant theory is implemented for any given system and issues of names can be avoided altogether. Alternatively, I could extend the project to more interesting calculi, like System F. System F adds binders at the type level, so this would be a good test of the nominal approach when multiple binders are present. Additionally, the type system is more powerful, allowing for more practical programming, while type inference remains reasonable [40].

Improving performance of the extracted code is certainly possible. While performance of the extracted code (surprisingly) is good without modification, I do not yet know *why* this is the case. Investigating this will lead to some optimisations. Combined with a more powerful type system, there are plenty of opportunities for efficiency improvements.

Further properties of the calculus can be shown, such as the strong normalisation property. Strong normalisation is an important property of the simply-typed calculus from the perspective of computability theory, as it shows that the calculus is not Turing-complete (Turing machines may run forever).

5.3 Closing remarks

λ-calculus produces a model of computation by manipulating terms involving bound and unbound names. By investigating a variety of approaches to binding names, and implementing the most theoretically-pleasing approach, I have arrived at a verified representation of the λ-calculus, as used informally in mathematical arguments. Further, I have shown that representing λ-terms by means of an explicit quotient with a nominal equivalence relation over the concrete syntax is laborious, but feasible as an approach, and comes with many advantages.

Bibliography

[1] Alonzo Church. "An unsolvable problem of elementary number theory". In: *American journal of mathematics* 58.2 (1936), pp. 345–363.

[2] Guy Lewis Steele and Gerald Jay Sussman. "Lambda: The Ultimate Imperative". 1976.

[3] J Barkley Rosser. "Highlights of the history of the lambda-calculus". In: *Annals of the History of Computing* 6.4 (1984), pp. 337–349.

[4] Bertrand Russell. "Mathematical logic as based on the theory of types". In: *American journal of mathematics* 30.3 (1908), pp. 222–262.

[5] Andrzej Trybulec and Howard A Blair. "Computer Assisted Reasoning with MIZAR." In: *IJCAI*. Vol. 85. Citeseer. 1985, pp. 26–28.

[6] Brian E Aydemir et al. "Mechanized metatheory for the masses: The POPLmark challenge". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 50–65.

[7] Hendrik Pieter Barendregt et al. *The lambda calculus*. Vol. 3. 1984.

[8] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375.

[9] Martin Hofmann. "A simple model for quotient types". In: *Typed lambda calculi and applications* (1995), pp. 216–234.

[10] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[11] Furio Honsell, Marino Miculan, and Ivan Scagnetto. "π-calculus in (co) inductive-type theory". In: *Theoretical computer science* 253.2 (2001), pp. 239–285.

[12] Andrew M Pitts. "Nominal logic, a first order theory of names and binding". In: *Information and computation* 186.2 (2003), pp. 165–193.

[13] Conor McBride and James McKinna. "Functional pearl: I am not a number — I am a free variable". In: *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*. ACM. 2004, pp. 1–9.

[14] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *COLOG-88*. Springer. 1990, pp. 50–66.

[15] Adam Chlipala. "Parametric higher-order abstract syntax for mechanized semantics". In: *ACM Sigplan Notices*. Vol. 43. 9. ACM. 2008, pp. 143–156.

[16] Masahiko Sato et al. "Viewing λ-terms through Maps". In: *Indagationes Mathematicae* 24.4 (2013), pp. 1073–1104.

[17] Murdoch Gabbay and Andrew Pitts. "A NEW approach to abstract syntax involving binders". In: *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*. IEEE. 1999, pp. 214–224.

[18] Andrew Pitts. *Nominal Sets and their Applications*. Talk. 2011. URL: `https://www.cl.cam.ac.uk/~amp12/talks/MGS2011_nominal_sets_slides.pdf`.

[19] Murdoch J. Gabbay. "A Theory of Inductive Definitions with alpha-Equivalence". PhD thesis. University of Cambridge, 2001.

[20] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[21] Markus Wenzel et al. "Isabelle/Isar — a versatile environment for human-readable formal proof documents". PhD thesis. Institut für Informatik, Technische Universität München, 2002.

[22] Brian Huffman and Ondřej Kunčar. "Lifting and Transfer: A modular design for quotients in Isabelle/HOL". In: *International Conference on Certified Programs and Proofs*. Springer. 2013, pp. 131–146.

[23] Florian Haftmann and Makarius Wenzel. "Constructive type classes in Isabelle". In: *International Workshop on Types for Proofs and Programs*. Springer. 2006, pp. 160–174.

[24] Laurent Chicli, Loïc Pottier, and Carlos Simpson. "Mathematical quotients and quotient types in Coq". In: *International Workshop on Types for Proofs and Programs*. Springer. 2002, pp. 95–107.

[25] Masako Takahashi. "Parallel reductions in λ-calculus". In: *Information and computation* 118.1 (1995), pp. 120–127.

[26] Robert Pollack. "Polishing up the Tait-Martin-Löf proof of the Church-Rosser theorem". In: (1995).

[27] Florian Haftmann and Lukas Bulwahn. *Code generation from Isabelle/HOL theories*. 2016.

[28] The Commercial Haskell Group. *The Haskell Tool Stack*. 2017. URL: `https://haskellstack.org` (visited on 04/07/2017).

[29] The GHC Team. *The Glasgow Haskell Compiler*. 2017. URL: `https://www.haskell.org/ghc` (visited on 04/07/2017).

[30] Simon Hengel et al. *HSpec: A testing framework for Haskell*. 2017. URL: `https://hspec.github.io` (visited on 03/14/2017).

[31] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266.

[32] Pierre Letouzey. "A new extraction for Coq". In: *International Workshop on Types for Proofs and Programs*. Springer. 2002, pp. 200–219.

[33] Florian Haftmann and Tobias Nipkow. "Code generation via higher-order rewrite systems". In: *International Symposium on Functional and Logic Programming*. Springer. 2010, pp. 103–117.

[34] Cordelia V Hall et al. "Type classes in Haskell". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), pp. 109–138.

[35] Bryan O'Sullivan. *Criterion: a Haskell micro-benchmarking library*. 2017. URL: https://www.serpentine.com/criterion (visited on 04/07/2017).

[36] Simon L Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of functional programming* 2.02 (1992), pp. 127–202.

[37] Murdoch J Gabbay and Aad Mathijssen. "Capture-avoiding substitution as a nominal algebra". In: *Formal Aspects of Computing* 20.4 (2008), pp. 451–479.

[38] Stefan Berghofer and Christian Urban. "A head-to-head comparison of de Bruijn indices and names". In: *Proc. Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*. 2006, pp. 46–59.

[39] Cezary Kaliszyk and Henk Barendregt. "Reasoning about constants in Nominal Isabelle, or: how to formalize the second fixed point theorem". In: *International Conference on Certified Programs and Proofs*. Springer. 2011, pp. 87–102.

[40] Luis Damas and Robin Milner. "Principal type-schemes for functional programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1982, pp. 207–212.

[41] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[42] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Vol. 149. 2006.

[43] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. 2002.

[44] *Isabelle installation page*. URL: http://isabelle.in.tum.de/installation.html (visited on 07/10/2016).

[45] Murdoch J Gabbay and Andrew M Pitts. "A new approach to abstract syntax with variable binding". In: *Formal aspects of computing* 13.3-5 (2002), pp. 341–363.

Appendix A

Project Proposal

Michael Rawson
Robinson College
mr644

Part II Project Proposal, Computer Science Tripos

Verified Metatheory and Type Inference for a Name-Carrying Simply-Typed λ-Calculus

April 27, 2023

Project Originator: Dr. Dominic Mulligan

Resources Required: None

Project Supervisors: Dr. Dominic Mulligan and Dr. Victor Gomes

Signatures:

Director of Studies: Dr. Alastair Beresford

Signature:

Overseers: Dr. Ian Wassell and Prof. Lawrence Paulson

Signatures:

Introduction

λ-calculus (see [7] for an overview) is a formal system of terms, often used in computability theory, but also more recently as a base system for various *type theories*.

The calculus expresses computation as a series of *abstractions* (anonymous first-class functions) and *applications* (function application), with reduction relations between them. For example, the identity function

$$\lambda x.x$$

applied to some term, say T, is clearly T: thus, the term

$$(\lambda x.x)\ T$$

reduces to T.

This calculus, the "untyped" λ-calculus, clearly lacks any sort of type system. Adding types to the calculus allows for various *typed* λ-calculi: these add many useful properties, including strong normalisation for some calculi [41], even allowing for mathematical theorems to be expressed under the "propositions-as-types" slogan [42]. The simplest of these is the *simply-typed* calculus $\lambda_{\to}$, consisting only of base names $A, B, \ldots$ and a function-arrow type constructor, e.g. $A \to B$.

Advantages of *formalising* such a calculus in a computerised proof assistant are twofold: first, theorems and proofs about the calculus can be expressed, and therefore automated, in the assistant. Secondly, it allows the generation of formally-verified code from the assistant to a target language more suited for execution, thus achieving the holy grail of bug-free software development.

Therefore, I propose that I use Isabelle, a mature and versatile proof assistant [43] to formalise $\lambda_{\to}$ and prove some metatheory about the encoded calculus. In order to achieve this, and by means of innovation, I will also attempt to use an unusual method for encoding the calculus' bound variables. I hypothesise that Isabelle's quotient datatype implementation is sufficiently mature to work with an explicit quotient over α-equivalence without any external tooling support, such as Nominal Isabelle. Testing this hypothesis will provide opportunity for evaluation, but to better judge my work, I will use my formalisation to implement a type inference algorithm for $\lambda_{\to}$, prove it correct, then extract executable code for it.

Required Resources

No extra resources other than the Isabelle software package is required for this project. Isabelle's 2016 release is freely available online [44], so this should not present a problem.

Starting Point

I'm familiar with types and the λ-calculus, both together and separately, through extra-curricular study and through the various theory courses present in Part I of the tripos.

However, I'm a novice user of the Isabelle proof assistant: I have been given a crash course in the assistant by my supervisor over the course of the summer break in the form of exercises, reading, and advice, but my Isabelle could still use some improvement.

Structure of the Project

The aim of this project is to verify some metatheory about the calculus, but I will use the goal of producing a verified type inference implementation to focus this process. Additionally, this type inference goal allows me to evaluate the project more readily than as a collection of theorems. A number of design choices have already been made in order to make a concrete plan.

1. The λ-calculus contains *binding* terms, the abstractions, which it uses to bind variables "under" the term. The subject of binders is surprisingly complex, with many possible representations, including:

   - a concrete representation involving explicitly-bound variables
   - de Bruijn indices, in which variables are referred to by the number of binders from the point of reference
   - higher-order abstract syntax, which embeds the binding in the implementation language's binders
   - more complex approaches involving *permutations* of variables, such as in Nominal Logic [45]

   I have chosen the permutation approach, as it is the most mathematically interesting and offers some advantages over other approaches (as argued in [45]). A further advantage of this explicit approach to name-carrying over existing implementations, such as Nominal Isabelle, is the ability to extract executable code from the verification.

2. In a similar vein, the notion of α-equivalence — that is, two terms are equivalent iff they are the same except for their use of different bound variable names: $f(x) = x^2$ and $f(y) = y^2$ would be said to be α-equivalent — needs to be expressed in the statement of lemmas in order to be fully general. This can either be interjected wherever necessary, or equality of terms can be redefined using Isabelle's *quotient type* implementation. The latter simplifies later lemma definitions in exchange for implementation difficulty, so I have chosen the quotient-type option for greater elegance.

3. Several properties of the implementation can be used to check its correctness. Specifically, I will prove the progress, type-preservation, and safety properties (as seen in the Part IB semantics course) for my implementation. Additionally, I will show that the type-checking procedure always produces the same results as the type inference rules expressed inductively. This should suffice to "verify" the implementation has been formalised as expected.

The structure of the project is as follows, with several main sections:

1. An in-depth study of the simply-typed calculus and varieties (see above), to ensure I have details correct before starting work.
2. Any necessary research in order to operate the Isabelle package effectively for this task.
3. Development of the representation and operations of the calculus in Isabelle, allowing expression of more complex theorems.
4. Proof of the progress, preservation, and safety properties of the calculus, following on from the representational work.
5. Implementing and verifying the type inference algorithm.
6. Extracting Standard ML code for the algorithm.
7. Producing the dissertation.

Success Criteria

Each section from the project has its own success criterion:

1. Do I have sufficient *theoretical* knowledge to implement the project confidently?
2. Do I have sufficient *practical* knowledge to implement the project confidently?
3. Can I use my representation as a typed calculus successfully?
4. Have I proven the progress, preservation, and safety properties for the encoded calculus?
5. Have I verified my type inference algorithm is equivalent to using the typing judgements inductively?
6. Does the Standard ML code compile/work as expected?
7. Is the dissertation complete?

Since the project is formally-verified, there is an overall success criterion: will Isabelle's proof checker pass all my proofs as valid?

In order to properly evaluate the project, some other metrics of success might be employed:

- speed of generated code
- fuzz-testing generated code, as a sanity check
- quality/complexity of the formalisation: compare my implementation to prior art for code quality or complexity

- compare and contrast my unusual approach of using Isabelle's quotient types directly for name binding with other approaches

In case of finishing early, I have also planned some extensions.

Extensions

- extend the implementation to more advanced calculi, like System F or $\lambda\Pi$
- prove more properties about the calculus, like the congruence property — almost any metatheoretical result about the calculus is relevant here
- experiment with different formulations of the type inference algorithm and observe the effects on generated code and its performance

Timetable

In 10 fortnightly steps, the proposed timetable for this project is as follows:

1. Michaelmas, 24/10–7/11. Preparatory academic work: survey the current state of the art, particularly in the areas of name-binding and typed calculi. I should be able to implement and explain ideas from relevant papers in order to complete my project.
2. Michaelmas, 7/11–21/11. Preparatory practical work: experiment and practise with the Isabelle proof assistant. I should be able to express relevant ideas and theorems more easily in the software.
3. Michaelmas, 21/11–5/12. Encode the calculus in Isabelle and start work on the permutation approach to α-equivalence.
4. Michaelmas, 5/12–19/12. Finish work on the permutation theory and encode equivalence with a quotient type. I will already have proven some vital properties of the calculus at this point to show that equivalence of terms is an equivalence relation.
5. Lent, 9/1–23/1. Start proving properties about the calculus. I will aim to finish at least a few to show for the progress report and presentation.
6. Lent, 23/1–6/2. Complete as many properties as possible before moving on to the type inference. Implement the type inference algorithm and extract executable code for it.
7. Lent, 6/2–20/2. Verify the type inference algorithm behaves correctly and finish any remaining tasks for the practical work.
8. Lent, 20/2–6/3. Start writing the dissertation. The Introduction and Preparation chapters should be complete, with the Implementation chapter started.

9. Lent, 6/3–20/3. Finish writing the dissertation. All chapters should be complete as far as possible at this point.

10. Lent and Easter, 20/3–19/5. Review, proof-read and make any required changes to the dissertation. Reserve space for submission and emergencies, but also for examination preparation.

This timetable assumes I complete no work at all outside of Full Term. Obviously, I plan to work outside of term additionally, but this buffer allows me a safety margin to ensure I complete the project.