Functional Programming: Project Ideas

Thomas Dinsdale-Young

2017

1 Project Requirements

1.1 Coq Development

At the top of the Coq file, you should list in a comment the authors of the file. It is expected that each member of the group should submit the same Coq file.

You should take care to structure your development in a fashion that is logical and easy to follow. In particular, you should give definitions and theorems sensible and consistent names. Judicious use of comments is recommended. Definitions (of functions in particular) should be accompanied by unit tests (or better quickchick code). Where relevant, you may use Coq's `Section` and `Module` commands to help structure the code.

Consider how best to refactor your code. For long proofs, are there sensible ways of breaking them down into lemmas? Can a result be generalised, and does that make it simpler to prove? If a proof is repetitive, can parts of it be automated using Ltac?

Your Coq development should consist of a single file, with no dependencies except the Coq standard library (`https://coq.inria.fr/library/index.html`). You are encouraged to use the standard library wherever it is helpful to do so. In particular, if your development involves numbers in a non-trivial way then you should expect to `Require Import Arith` and use results from the library.

1.2 Report

You should submit an individual report on the project. The report should not be very long. It should describe the project in a sufficient level of detail that a competent reader could reproduce your results (without reference to your code). You should present your results in mathematical language and avoid references to Coq tactics.

Your report should comprise the following:

Introduction. This section should explain what the key results of your project are, what they mean, and how they fit into the bigger picture. (For this, you might need to look into some of the literature.)

For example, suppose that your project was on proving that the simply-typed lambda calculus is strongly normalizing. Your introduction might:

- Explain that the lambda calculus is a simple functional programming language based on abstraction and application, introduced by Church.

- Explain that computation is modelled by β-reduction, in which the application of an abstraction to a term is reduced by substitution.
- Explain what it means for a term to be normalising (has some terminating reduction path) and strongly normalising (all reduction paths are terminating).
- Explain that not all terms are strongly normalising.
- Explain that the simply-typed lambda calculus associates types with terms, to enforce that well-typed terms are well behaved.
- Explain that in particular that means that well-typed terms are strongly normalising, which is the key result of the project.
- Explain that (weak) normalisation was first proved by Turing, and later results proving strong normalisation from normalisation are due to Nederpelt and Gandy.
- Explain that you are following the approach presented in Barandregt, based on Tait and Girard.

(When referencing the literature you should include relevant citations.)

Main Results. The bulk of the report should consist of one or more sections explaining the main results from the Coq development. Your presentation should, where possible, present everything in mathematical language rather than the language of Coq. (For instance, typing rules should be presented as a deductive system, rather than as the definition of a Coq inductive predicate.) You should not give full proofs, but include enough detail about how the proof is carried out that a competent reader can check for themselves. Ideally, these sections should be structured in a similar manner to the Coq development and should flow logically.

Conclusions. This section should briefly summarise the results and point out the key insights used in the main results. You should also include a paragraph of *acknowledgements* listing your collaborators and anyone else who might have helped you (and in what capacity).

Bibliography. If you have cited literature, you should have a bibliography of references.

2 Projects

There are three suggestions of projects you may choose from: proving the confluence of the SKI-calculus; proving that reduction in the simply-typed lambda calculus preserves types; and implementing and verifying a 2–3 tree. These projects are described below. If you have a proposal for another project that you would like to do then I am open to suggestions.

2.1 Confluence of the SKI-Calculus

In the lecture notes, it was stated but not formally proved that the SKI-calculus is confluent. That is:

Theorem 2.1.1. *For all SKI-terms x, y, z, if $x \to^* y$ and $x \to^* z$ then there exists w such that $y \to^* w$ and $z \to^* w$.*

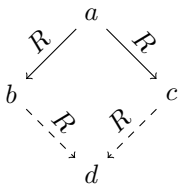
In this project, you will prove this theorem formally in Coq. To do so, you will have to:

- Formalise the terms of SKI with inductive types.
- Formalise the reduction relation of SKI with inductive predicates.
- Formulate and prove confluence.

2.1.1 Suggested Approach

We can characterise confluence as the reflexive-transitive closure of the reduction relation ($\to^*$) having the following strong diamond property.

Definition 2.1.2 (Strong Diamond Property)**.** A relation $R \subseteq A \times A$ has the *strong diamond property* if for all $a, b, c \in A$ with $a\ [R]\ b$ and $a\ [R]\ c$ there exists $d \in A$ with $b\ [R]\ d$ and $c\ [R]\ d$.



The following general result gives a way of proving that the reflexive-transitive closure of a relation has the strong diamond property:

Lemma 2.1.3. *Suppose that relation R has the strong diamond property. Then the reflexive-transitive closure R^* has the strong diamond property.*

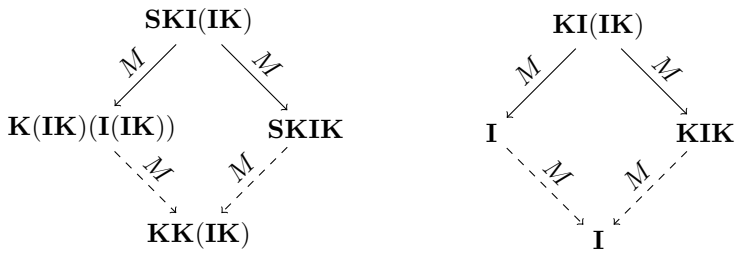
Unfortunately $\to$ itself does not have the strong diamond property (see §1.1 of the notes). The idea instead is to define a new relation $\xrightarrow{M}$ with the following properties:

Lemma 2.1.4. *The relation $\xrightarrow{M}$ has the strong diamond property.*

Lemma 2.1.5. *For all terms x, y, $x \to^* y$ if and only if $x \xrightarrow{M}{}^* y$.*

Theorem 2.1.1 then follows as a corollary of the above three lemmas.

A challenge of this project is to find an appropriate definition for $\xrightarrow{M}$. The definition should allow multiple simultaneous reductions in a term, so that, for instance:



2.1.2 Extensions

Once you have proved the main result, you should also consider some ways of extending the project. You are free to think up your own, but here are some suggestions:

- Extend the calculus with other combinators such as $\mathbf{B}, \mathbf{C}, \mathbf{W}$. (Easy)
- Can the calculus be extended in a modular way? That is, is there a property that can be checked on each combinator/rule separately that guarantees that the combined system will be confluent? (Tricky)
- Show the confluence of lambda calculus. (Tricky)

2.2 Type Preservation in the Simply-Typed Lambda Calculus

For this project, you should:

- Formalise a representation of the simply-typed lambda calculus in Coq.
- Formalise β-reduction.
- Formalise the typing judgement as an inductive proposition.
- Define a type-checking function that determines whether a term is well-typed in a given context and, if it is, what its type is.
- Prove the correctness of the type-checking function.
- Prove that β-reduction preserves the type of a term in a given context.

The type-checking function establishes decidability of type checking. This result is separate from (and not required for) type preservation. Both results are required for the project.

2.2.1 Suggested Approach

Binders and names are notoriously tricky to handle in proof assistants. To avoid the problems they cause, I suggest representing terms of the lambda calculus using de Bruijn indices. With de Bruijn indices, the syntax of simply-typed lambda terms may be defined as:

$$M, N ::= \underline{n} \mid MN \mid \lambda_\sigma M$$

where n ranges over natural numbers. A variable $\underline{n}$ refers to the binding λ that is n layers out.

names	de Bruijn indices
$\lambda x : \alpha . x$	$\lambda_\alpha \underline{0}$
$\lambda x : \beta \to \alpha, y : \beta . xy$	$\lambda_{\beta\to\alpha}\lambda_\beta \underline{1}\ \underline{0}$
$\lambda x : \alpha . (\lambda y : \alpha . y)x$	$\lambda_\alpha(\lambda_\alpha \underline{0})\ \underline{0}$
$\lambda x : \alpha . (\lambda y : \alpha . x)x$	$\lambda_\alpha(\lambda_\alpha \underline{1})\ \underline{0}$

Free variables will be mapped to indexes, and a typing context is effectively a list of types, with the position in the list determining the index corresponding to the free variable.

names	de Bruijn indices
$x : \alpha \vdash x : \alpha$	$\alpha \vdash \underline{0} : \alpha$
$x : \alpha, y : \beta \vdash x : \alpha$	$\alpha, \beta \vdash \underline{1} : \alpha$
$x : \alpha \vdash \lambda y : \beta . x : \beta \to \alpha$	$\alpha \vdash \lambda_\beta \underline{1} : \beta \to \alpha$

To define reduction on the simply-typed lambda calculus, you have to define capture-avoiding substitution. This is not trivial to define correctly. For example, we have

$$\begin{aligned}
\underline{1}\,[\underline{2}\,/\,\underline{0}] &= \underline{0} \\
\underline{1}\,[\underline{2}\,/\,\underline{1}] &= \underline{2} \\
\underline{1}\,[\underline{2}\,/\,\underline{2}] &= \underline{1} \\
(\lambda_\alpha \underline{1})[\underline{1}\,/\,\underline{0}] &= \lambda_\alpha(\underline{1}\,[\underline{2}\,/\,\underline{1}]) = \lambda_\alpha \underline{2} \\
(\lambda_\alpha \underline{1})[\lambda_\beta \underline{1}\,/\,\underline{0}] &= \lambda_\alpha(\underline{1}\,[\lambda_\beta \underline{2}\,/\,\underline{1}]) = \lambda_\alpha\lambda_\beta \underline{2} \\
(\lambda_\alpha \underline{1})[\lambda_\beta \underline{0}\,/\,\underline{0}] &= \lambda_\alpha(\underline{1}\,[\lambda_\beta \underline{0}\,/\,\underline{1}]) = \lambda_\alpha\lambda_\beta \underline{0}
\end{aligned}$$

The substitution lemma should be something like:

Lemma 2.2.1. *Suppose that*

$$\begin{aligned}
\Gamma, \Delta &\vdash M : \sigma \\
\Gamma, \sigma, \Delta &\vdash N : \rho
\end{aligned}$$

then

$$\Gamma, \Delta \vdash N[M/\,\underline{n}\,] : \rho$$

where $n = |\Delta|$ is the length of the context Δ.

Once you have proved the substitution lemma, it should be simple to establish the main theorem:

Theorem 2.2.2. *Suppose that $\Gamma \vdash M : \sigma$ and $M \to_\beta N$. Then $\Gamma \vdash N : \beta$.*

2.2.2 Extensions

Once you have proved the main results, you should also consider some ways of extending the project. You are free to think up your own, but here are some suggestions:

- Extend the calculus with some of the following, showing that the results (type checking and type preservation) still hold:
  - booleans and conditionals
  - pairs and projections
  - a fixpoint combinator
- Also show that η-reduction preserves types.

2.3 2–3 Trees

A 2–3 tree is a form of self-balancing search tree[1] in which each node has either two or three children (at a leaf node, these children are the empty tree). A 2-node has two children and one key; all keys in the left child must be lower than the key, and all keys in the right child must be higher than the key. A 3-node has three children and two keys; all keys in the first child must be lower than the first key; all keys in the second child must be between the first and second keys; and all keys in the third child must be above the second key.

To determine if a key is in the tree[2] we use the search tree property to check along a path in the tree that is guaranteed to have the key if it is in the tree at all.

To insert a key, we find the leaf where it would belong if it were in the tree. (If it is already in the tree then we need not do anything.) We then add the key to the leaf. If the leaf is a 2-node, it becomes a 3-node. If it is a 3-node then it is split into two 2-nodes with the lowest and greatest keys, and the middle key is promoted to the next level in the tree. At the upper level, this means replacing the child with a key and two children, which will either turn a 2-node into a 3-node or split a 3-node. This proceeds up the tree. If the root is split, then it is replaced with a new 2-node. Figure 1 illustrates the insertion process.

For this project, you should:

- Define an inductive data type to represent a 2–3 tree.
- Implement the search and insert functions on 2–3 trees.
- Test these functions on suitable inputs.
- Define a predicate that expresses when a key is in a 2–3 tree.
- Define well-formedness predicates on 2–3 trees that express the invariants that they are balanced and represent search trees.
- Show that search and insert behave correctly when the invariants hold.
- Show that insert preserves the invariants.

[1] A 2–3 tree is a specialised type of B-tree.

[2] For simplicity, we won't view the tree as mapping keys to values; instead we will treat it as representing a finite set of keys.

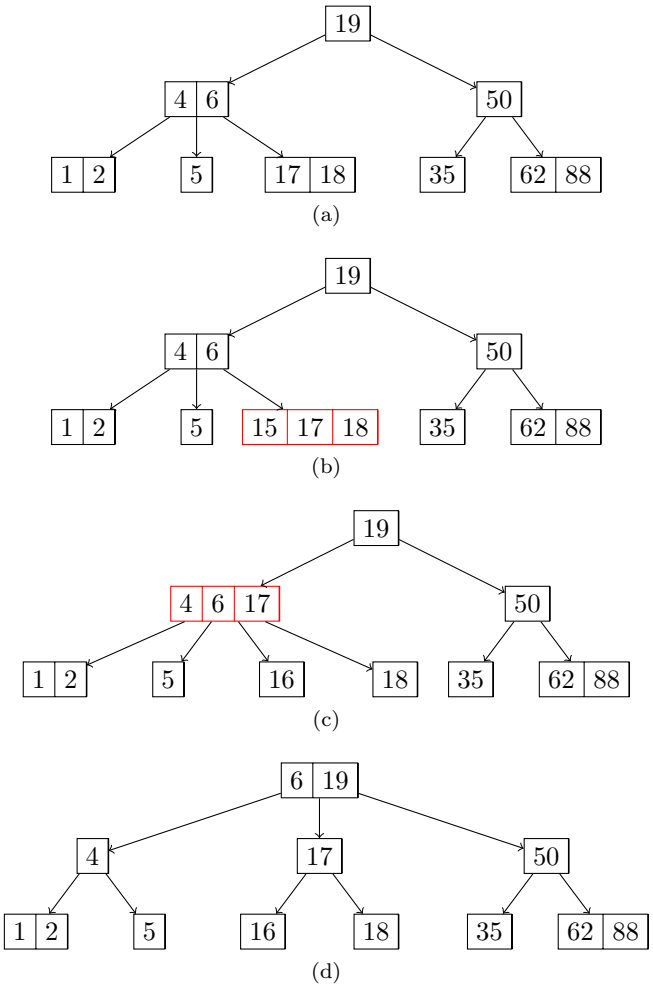


Figure 1: Inserting the value 15 into a 2–3 tree.

2.3.1 Suggested Approach

One of the tricky parts of this project is to define the insert function correctly. You may need to define additional inductive types and functions to help in the definition. Make use of the comparison operator `?=` from the standard library. You are likely to find that you see many similar cases in your proofs; consider using Ltac to automate the repetitive parts.

2.3.2 Extensions

Once you have proved the main results, you should also consider some ways of extending the project. You are free to think up your own, but here are some suggestions:

- Store values in the tree associated with keys, so that the tree represents a finite map, not just a finite set.

- Implement and verify deletion from the tree. (You can look up how deletion is performed on B-trees and adapt that.) (Tricky)
- Generalise the tree and proofs so that not only `nat` can be used as the type of keys, but any type for which we have a comparison operation with certain properties. (Tricky)