Mechanised Computability Theory

Michael Norrish

Canberra Research Lab., NICTA;
also, Australian National University

Abstract. This paper presents a mechanisation of some basic computability theory. The mechanisation uses two models: the recursive functions and the λ-calculus, and shows that they have equivalent computational power. Results proved include the Recursion Theorem, an instance of the *s-m-n* theorem, the existence of a universal machine, Rice's Theorem, and closure facts about the recursive and recursively enumerable sets. The mechanisation was performed in the HOL4 system and is available online.

1 Introduction

This paper describes mechanisation work in one of computer science's foundational areas: computability theory. This is the theory of what can and cannot be computed by abstract computing machines, using models such as Turing machines, register machines, the λ-calculus and the recursive functions. This paper's focus is on the last two of these models, mainly because of their simplicity (in the case of the recursive functions), and because an existing background theory was available (in the case of the λ-calculus).

By showing the computational equivalence of the two models, we gain additional assurance that their mechanisations are correct. The other standard results, showing what the models are and are not capable of, further validate the work.

Mechanisation in an area such as this is intellectually satisfying in itself. Additionally, the development should provide the wherewithal to mechanise computability arguments where this has not been possible before. For example, Urban, Cheney and Berghofer's impressive paper, *Mechanising the Metatheory of LF* [10] includes an argument to the effect that the algorithm they have formalised (and shown correct) is indeed computable. In the absence of a theory of computability, the argument that the rules of the algorithm are computable is by a combination of careful discussion, suggestive theorems, and (necessarily un-formalised) human inspection.

Contributions

- The first mechanisation of the λ-calculus as a model of computation, including standard auxiliary notions such as the Church numerals.
- A mechanised proof of computational equivalence between two different models: the λ-calculus and the recursive functions.
- Mechanised proofs of a number of standard results from computability theory.
- Discussion of the results and theorem-proving techniques that made the above possible, including use of: the isomorphism between de Bruijn terms and quotiented λ-terms, the standardisation theorem, simplification with pre-orders, and bracket abstraction.

HOL4 Notation and Theorems All statements appearing with a turnstile ($\vdash$), or as natural deduction style rules, are HOL4 theorems, automatically pretty-printed to LaTeX from the relevant theory in the HOL4 development. Notation specific to this paper is explained as it is introduced. Otherwise, HOL4 supports a notation that is a generally pleasant combination of quantifiers ($\forall$, $\exists$) and functional programming (λ for function abstraction, juxtaposition for function application). Hilbert choice is available with the syntax $(\varepsilon x.\ P\ x)$, meaning "the x such that P holds". Such a term has an unspecified value if there is no such x.

The paper also uses the polymorphic option type (α `option`), with possible values `SOME` x and `NONE`. The `THE` function maps `SOME` x to x, and is unspecified on `NONE`. The term `OPTION_MAP` $f\ x$ returns `SOME` $(f\ y)$ when x is `SOME` y, and `NONE` when x is `NONE`.

Lists are constructed with the infix "cons" function ::. The length of a list ℓ is written $|\ell|$. Lists support other standard operations such as `MAP`.

Availability The sources for the mechanisation described in this paper are available as part of the standard HOL4 distribution ("Kananaskis-6" release), available from `hol.sourceforge.net`.

2 The λ-Calculus: First Steps

This work would not have been possible without earlier mechanisation effort targetting relevant aspects of the λ-calculus. In particular, it relies on my earlier proof of the standardisation theorem [3], and Vestergaard's and my proof [4] that the de Bruijn terms and their associated notion of β-reduction are isomorphic to the λ-terms (quotiented name-carrying syntax) with *their* own notion of β-reduction.

The λ-terms used in this earlier work, and thus in this paper also, are either variables ($\underline{v}$), (left-associating) applications ($M \cdot N$) or abstractions $(\boldsymbol{\lambda} v.\ M)$. These are terms of the object language: the bold lambda is a constructor (which takes a string and a λ-term as arguments) creating a value of type `term` within the higher-order logic. In contrast, the normal lambda of the meta-language creates values in the logic's function spaces. Similarly, the variable constructor, denoted with underlining, injects values from the HOL type of strings into the term type.

These terms are quotiented, and so have equality results such as

$$(\boldsymbol{\lambda} v.\ \underline{v}) = (\boldsymbol{\lambda} u.\ \underline{u})$$

The free variable function over terms is written `FV`, and the substitution notation is $M[v := N]$, meaning the term resulting from substituting term N for the (free) variable with name v throughout term M.

2.1 Normal Order Reduction

Definition 1. *To guarantee that λ-evaluations find normal forms, we use normal order reduction:*

$$\frac{}{(\lambda v.\ M) \diamond N \to_n M[v := N]} \qquad \frac{M_1 \to_n M_2 \quad \neg\mathtt{is\_abs}\ M_1}{M_1 \diamond N \to_n M_2 \diamond N}$$

$$\frac{M_1 \to_n M_2}{(\lambda v.\ M_1) \to_n (\lambda v.\ M_2)} \qquad \frac{N_1 \to_n N_2 \quad \mathtt{bnf}\ M \quad \neg\mathtt{is\_abs}\ M}{M \diamond N_1 \to_n M \diamond N_2}$$

where the predicate `is_abs` *is true of a term if it is an abstraction.*

We are then able to prove that if a term can β-reduce to a β-normal form, then a (necessarily deterministic) normal reduction will eventually arrive at the same place:

$$\vdash M \to_\beta^* N \land \mathtt{bnf}\ N \Rightarrow M \to_n^* N$$

The proof is as per Barendregt [1, §13.2]: in essence, a standard reduction (in the sense of the standardisation theorem) that reaches a β-normal form must also be normal-order as such a reduction can't have ignored a potential redex in its sweep across a term (outermost, left-to-right). By the standardisation theorem, all β-reductions can be emulated by a standard reduction, and so all β-reductions to normal forms can be emulated by normal order reduction.

2.2 Rewriting with β-Equivalence; Bracket Abstraction

In developing the λ-calculus implementations of types such as numbers and de Bruijn terms, it is critical to be able to prove facts of the form $M \equiv_\beta M'$, stating that M is β-equivalent to M'. (The β-equivalence relation is the symmetric, reflexive, transitive closure of the relation that reduces one β-redex.)

The HOL4 simplifier supports rewriting with arbitrary pre-orders, and rewriting with an equivalence (where we additionally have symmetry) is generally quite pleasant. One has to provide introduction rules such as

$$\frac{M_1 \equiv_\beta M_2 \quad N_1 \equiv_\beta N_2}{M_1 \equiv_\beta N_1 \iff M_2 \equiv_\beta N_2}$$

which switches the simplifier from rewriting an equality (boolean equivalence in this case) to β-equivalence. In addition, one can use the following rewrites

$$\vdash \mathtt{bnf}\ N \Rightarrow (M \to_n^* N \iff M \to_\beta^* N)$$
$$\vdash \mathtt{bnf}\ N \Rightarrow (M \to_\beta^* N \iff M \equiv_\beta N)$$

to move to rewriting with $\equiv_\beta$ from goals mentioning $\to_n^*$ and $\to_{\beta}^*$.

It's very important to be able to rewrite with theorems already proved, results such as (see Section 2.3 below for more on Church numerals and arithmetic)

$$\vdash \mathtt{cplus} \diamond \mathtt{church}\ m \diamond \mathtt{church}\ n \to_n^* \mathtt{church}\ (m + n)$$

This theorem is a statement about normal order reduction, not β-equivalence, but the simplifier is primed by the inclusion theorem:

$$\vdash M \rightarrow_n^* N \Rightarrow M \equiv_\beta N$$

and is able to use the above as a rewrite. Because β-equivalence is a congruence, such rewrites can be applied at any point within a term.

The basic rule governing β-redexes is present too:

$$\frac{}{(\lambda x.\ M) \diamond N \equiv_\beta M[x := N]}$$

but use of this rule is best avoided because of the possibility that bound variables will need to be renamed.

One might initially hope not to have to deal with variable renaming in a setting where the terms are already quotiented with respect to α-equivalence. Indeed, there is no *semantic* problem, but rather a pragmatic problem to do with making simplification as smooth as possible. The problem stems from the abstraction clause of the substitution rewrite:

$$\vdash v \neq u \land v \notin \texttt{FV}\ N \Rightarrow (\lambda v.\ M)[u := N] = (\lambda v.\ M[u := N])$$

It is not necessarily the case that the bound v will always be fresh with respect to the particular N. In that situation, the desired rewrite could be made to go through by first proving

$$(\lambda v.\ M) = (\lambda w.\ M[v := \underline{w}])$$

where w was chosen to be suitably fresh. Then this equality could be used to substitute "equals-for-equals", and the simplifier would end up simplifying the term

$$M[v := \underline{w}][u := N]$$

before proceeding further.

Implementing this is certainly possible, but would involve writing special-purpose code in ML that the simplifier could call out to as it traversed a term. It seems cleaner to use a technique that can work with the simplifier "as is". Our approach is a procedure inspired by bracket abstraction. The core theorems are shown in Figure 1. These can be applied automatically by the simplifier to prove β-equivalence results between terms with abstractions and terms without.

For example, the original definition of addition on Church numerals is[1]

$$\vdash \texttt{cplus} = (\lambda \texttt{"m"}\ \texttt{"n"}.\ \underline{\texttt{"m"}} \diamond \underline{\texttt{"n"}} \diamond \texttt{csuc})$$

but the application of the rewrites above returns the point-free characterisation:

[1] Note how we have to pick concrete names, "x" and "y", for the variables that are "bound" at the object-level. Though $x \neq y \Rightarrow \texttt{cplus} = (\lambda x\ y.\ \underline{x} \diamond \underline{y} \diamond \texttt{csuc})$ is true, any attempt to define cplus this way would stumble on the requirement to keep x and y apart, and on the fact that the definition would have (from HOL's perspective) free variables (x and y) on its RHS.

$\vdash\ v \notin \texttt{FV}\ M \land v \in \texttt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \texttt{B} \diamond M \diamond (\lambda v.\ N)$
$\vdash\ (\lambda v.\ \texttt{B} \diamond \underline{v}) \equiv_\beta \texttt{B}$
$\vdash\ v \in \texttt{FV}\ M \land v \notin \texttt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \texttt{C} \diamond (\lambda v.\ M) \diamond N$
$\vdash\ v \notin \texttt{FV}\ M \Rightarrow (\lambda v.\ M) \equiv_\beta \texttt{K} \diamond M$
$\vdash\ v \in \texttt{FV}\ M \land v \in \texttt{FV}\ N \Rightarrow (\lambda v.\ M \diamond N) \equiv_\beta \texttt{S} \diamond (\lambda v.\ M) \diamond (\lambda v.\ N)$
$\vdash\ (\lambda x.\ \underline{x}) = \texttt{I}$

Fig. 1. β-Equivalence rewrites implementing bracket abstraction. As this is β-equivalence, rather than $\beta\eta$-equivalence, we cannot η-contract freely. However, some η-contractions (as in the second rewrite above) are β-valid because the other half of the body is really an abstraction.

$\vdash\ \texttt{cplus} \equiv_\beta \texttt{C} \diamond (\texttt{B} \diamond \texttt{C} \diamond (\texttt{C} \diamond \texttt{B} \diamond \texttt{I})) \diamond \texttt{csuc}$

The combinator terms `B`, `C`, `I`, `K` and `S` are defined as abstractions, but these definitions are not unfolded by the simplifier. Instead, the combinatory characterisations are used as rewrites:

$\vdash\ \texttt{B} \diamond f \diamond g \diamond x \equiv_\beta f \diamond (g \diamond x)$
$\vdash\ \texttt{C} \diamond f \diamond x \diamond y \equiv_\beta f \diamond y \diamond x$
$\vdash\ \texttt{I} \diamond x \equiv_\beta x$
$\vdash\ \texttt{K} \diamond x \diamond y \equiv_\beta x$
$\vdash\ \texttt{S} \diamond f \diamond g \diamond x \equiv_\beta f \diamond x \diamond (g \diamond x)$

2.3 Church Arithmetic

Definition 2. *It is straightforward to encode numbers and other algebraic types within the λ-calculus using the method due to Church. For example, the natural numbers can be injected with the function* `church`*, of type* `num` $\rightarrow$ `term`*, which is defined:*

$\vdash\ \texttt{church}\ n = (\lambda\texttt{"z"}\ \texttt{"s"}.\ \texttt{FUNPOW}\ (\texttt{APP}\ \underline{\texttt{"s"}})\ n\ \underline{\texttt{"z"}})$

The form `APP` *M is a partial application of the constructor for application terms, and* `FUNPOW` *f n x applies the function f to the x argument n times.*

Thus, `church 3` expands to

$$(\lambda\texttt{"z"}\ \texttt{"s"}.\ \underline{\texttt{"s"}} \diamond (\underline{\texttt{"s"}} \diamond (\underline{\texttt{"s"}} \diamond \underline{\texttt{"z"}})))$$

Definition 3. *Having used the same approach to model pairs (with constructor* `cpair` *and projections* `cfst` *and* `csnd`*), one can then define a recursion combinator:*

$\vdash\ \texttt{natrec} =$
$\quad(\lambda\texttt{"z"}\ \texttt{"f"}\ \texttt{"n"}.$
$\quad\ \ \texttt{csnd}$
$\quad\ \ \ \diamond (\underline{\texttt{"n"}} \diamond (\texttt{cpair} \diamond \texttt{church}\ 0 \diamond \underline{\texttt{"z"}})$
$\quad\quad\quad \diamond (\lambda\texttt{"r"}.$
$\quad\quad\quad\quad \texttt{cpair} \diamond (\texttt{csuc} \diamond (\texttt{cfst} \diamond \underline{\texttt{"r"}}))$
$\quad\quad\quad\quad\ \ \diamond (\underline{\texttt{"f"}} \diamond (\texttt{cfst} \diamond \underline{\texttt{"r"}}) \diamond (\texttt{csnd} \diamond \underline{\texttt{"r"}})))))$

$\vdash$ $\texttt{cplus} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m + n)$
$\vdash$ $\texttt{cminus} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m - n)$
$\vdash$ $\texttt{cmult} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{church}\ (m \times n)$
$\vdash$ $0 < q \Rightarrow \texttt{cdiv} \diamond \texttt{church}\ p \diamond \texttt{church}\ q \to_n^* \texttt{church}\ (p\ \texttt{DIV}\ q)$

$\vdash$ $\texttt{ceqnat} \diamond \texttt{church}\ n \diamond \texttt{church}\ m \to_n^* \texttt{cB}\ (n = m)$
$\vdash$ $\texttt{cless} \diamond \texttt{church}\ m \diamond \texttt{church}\ n \to_n^* \texttt{cB}\ (m < n)$

$\vdash$ $\texttt{cfst} \diamond (\texttt{cpair} \diamond M \diamond N) \to_n^* M$
$\vdash$ $\texttt{csnd} \diamond (\texttt{cpair} \diamond M \diamond N) \to_n^* N$

Fig. 2. Theorems specifying the correctness of some of the various Church arithmetic and pair operations. The cB function takes a HOL boolean and returns the corresponding λ-term (either $(\lambda\texttt{"x"}\ \texttt{"y"}.\ \underline{\texttt{"x"}})$ for true, or $(\lambda\texttt{"x"}\ \texttt{"y"}.\ \underline{\texttt{"y"}})$ for false).

The underlying recursion returns a pair of the original number and the actual desired result.

The characterising theorems are quite readable:

$\vdash$ $\texttt{natrec} \diamond z \diamond f \diamond \texttt{church}\ 0 \equiv_\beta z$
$\vdash$ $\texttt{natrec} \diamond z \diamond f \diamond \texttt{church}\ (\texttt{SUC}\ n) \equiv_\beta$
$\quad f \diamond \texttt{church}\ n \diamond (\texttt{natrec} \diamond z \diamond f \diamond \texttt{church}\ n)$

With a combinator of this sort, subtraction can be defined (and verified!) easily. (The traditional Church definition, which doesn't use pairing, is much harder to deal with.)

As well as the standard arithmetic operations (see Figure 2), we also need to define the minimisation operator, here called cfindleast. This is the only place where unbounded recursion, in the form of the Y combinator, is required. The introduction rule for a successful call is:

$\vdash$ $(\forall n.\ \exists b.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB}\ b) \wedge P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB T} \Rightarrow$
$\texttt{cfindleast} \diamond P \diamond k \equiv_\beta$
$\quad k \diamond \texttt{church}\ (\texttt{LEAST}\ n.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB T})$

The preconditions require that

- the predicate P is total on numeric arguments, and also guaranteed to return a boolean on all such arguments; and
- the predicate P does indeed return true for at least one number.

The LEAST binder is the HOL analogue of cfindleast.

The k parameter to cfindleast is a continuation that is handed the result of a successful search for a number satisfying P. Using a continuation is a method for making functions that use minimisation *strict*. In other words, we want to be able to construct terms including minimisation, and to be sure that if the minimisation loops, then the whole term will have no β-normal form. The use of a continuation is the standard way to emulate call-by-value in a normal order setting. This insistence on strictness is consistent with the way we will handle the minimisation operator for recursive functions.

There is also an elimination rule for successful (terminating) `cfindleast` searches:

$$\begin{array}{l}
\vdash\ (\forall n.\ \exists b.\ P \diamond \texttt{church}\ n \equiv_\beta \texttt{cB}\ b) \wedge \texttt{cfindleast} \diamond P \diamond k \equiv_\beta r \wedge \\
\quad \texttt{bnf}\ r \Rightarrow \\
\quad \exists m. \\
\qquad r \equiv_\beta k \diamond \texttt{church}\ m \wedge P \diamond \texttt{church}\ m \equiv_\beta \texttt{cB T} \wedge \\
\qquad \forall m_0.\ m_0 < m \Rightarrow P \diamond \texttt{church}\ m_0 \equiv_\beta \texttt{cB F}
\end{array}$$

The proof is by complete induction on the number of steps taken to reach the result r.

3 Reflection and the Universal Machine in the λ-Calculus

An important precursor to our computability results is the demonstration that the λ-calculus can implement itself.

3.1 Church de Bruijn Terms

The Church-style encoding of algebraic types is also possible for the algebraic type that encodes the "pure" de Bruijn terms (`pdb`): the type with three constructors `dV`, `dAPP` and `dABS`, of types `num` $\rightarrow$ `pdb`, `pdb` $\rightarrow$ `pdb` $\rightarrow$ `pdb` and `pdb` $\rightarrow$ `pdb` respectively. As noted above, we already know that the de Bruijn notion of β-reduction is isomorphic to that of the λ-calculus.

So we begin by defining an injection function from de Bruijn terms into λ-terms (`cDB`), along with "constructors" `cdV`, `cdAPP` and `cdABS`. We derive the following characterisations:

$$\begin{array}{l}
\vdash\ \texttt{cdV} \diamond \texttt{church}\ n \rightarrow_n^* \texttt{cDB (dV}\ n\texttt{)} \\
\vdash\ \texttt{cdAPP} \diamond \texttt{cDB}\ M \diamond \texttt{cDB}\ N \rightarrow_n^* \texttt{cDB (dAPP}\ M\ N\texttt{)} \\
\vdash\ \texttt{cdABS} \diamond \texttt{cDB}\ M \rightarrow_n^* \texttt{cDB (dABS}\ M\texttt{)}
\end{array}$$

It is vital to be able to interpret de Bruijn terms at this point, rather than some sort of name-carrying syntax: with de Bruijn terms one does not have to implement variable-renaming when performing substitutions. Given the baggage of the Church encoding, the functions and terms developed here are already quite complicated enough without having to worry about some sort of `gensym` technology.

By analogy with `natrec` above, it is now possible to write a `termrec` recursion combinator for de Bruijn terms, with the following characterisation:

$$\begin{array}{l}
\vdash\ \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB (dV}\ i\texttt{)} \equiv_\beta v \diamond \texttt{church}\ i \\
\vdash\ \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB (dAPP}\ t\ u\texttt{)} \equiv_\beta \\
\qquad c \diamond \texttt{cDB}\ t \diamond \texttt{cDB}\ u \diamond \texttt{(termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ t\texttt{)} \\
\qquad\ \diamond \texttt{(termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ u\texttt{)} \\
\vdash\ \texttt{termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB (dABS}\ t\texttt{)} \equiv_\beta \\
\qquad a \diamond \texttt{cDB}\ t \diamond \texttt{(termrec} \diamond v \diamond c \diamond a \diamond \texttt{cDB}\ t\texttt{)}
\end{array}$$

With `termrec` defined, it is straightforward to define a function to implement normal-order reduction, and another to perform n steps of normal order reduction. With the minimisation operator, one can then define the function which finds the least n such that n steps of normal order reduction results in a term in β-normal form. Thus, we have a computable (and partial!) function for computing β-normal forms, which we call `cbnf_ofk`. As with `cfindleast`, the `cbnf_ofk` function takes a continuation parameter to help with strictness. We derive the following characterising theorems:

$\vdash$ $\texttt{bnf\_of}\ M\ \texttt{=}\ \texttt{NONE}\ \Rightarrow$
  $\texttt{bnf\_of}\ (\texttt{cbnf\_ofk} \diamond k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ M))\ \texttt{=}\ \texttt{NONE}$
$\vdash$ $\texttt{bnf\_of}\ M\ \texttt{=}\ \texttt{SOME}\ N\ \Rightarrow$
  $\texttt{cbnf\_ofk} \diamond k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ M) \equiv_\beta k \diamond \texttt{cDB}\ (\texttt{fromTerm}\ N)$
$\vdash$ $\texttt{cbnf\_ofk} \diamond k \diamond \texttt{cDB}\ M \to_n^* t' \land \texttt{bnf}\ t' \Rightarrow$
  $\exists M'.$
    $\texttt{bnf\_of}\ (\texttt{toTerm}\ M)\ \texttt{=}\ \texttt{SOME}\ (\texttt{toTerm}\ M') \land k \diamond \texttt{cDB}\ M' \to_n^* t'$

The `bnf_of` function is the (uncomputable) function in the logic which, using an option type to encode partiality, returns a term's β-normal form if it has one. The `fromTerm` and `toTerm` functions are mutual inverses mapping from the λ-terms to the de Bruijn terms and *vice versa*.

3.2 The Universal Machine

In order to compare the λ-calculus's capabilities to what is done in other computational models, we restrict our attention to functions on natural numbers only. We also index the computable functions with natural numbers, so that we can define

$$\Phi : \texttt{num} \to \texttt{num} \to \texttt{num option}$$

taking parameters specifying the computable function to run, and the argument to run it on. The restriction to a single parameter for the given function is not significant because of the existence of standard encodings for lists and pairs of numbers.

The first parameter to Φ requires a bijection between the natural numbers and the de Bruijn terms. The HOL function `dBnum` is defined:

$\vdash$ $\texttt{dBnum}\ (\texttt{dV}\ i)\ \texttt{=}\ 3 \times i$
$\vdash$ $\texttt{dBnum}\ (\texttt{dAPP}\ M\ N)\ \texttt{=}\ 3 \times (\texttt{dBnum}\ M \otimes \texttt{dBnum}\ N) + 1$
$\vdash$ $\texttt{dBnum}\ (\texttt{dABS}\ M)\ \texttt{=}\ 3 \times \texttt{dBnum}\ M + 2$

(where $x \otimes y$ is a bijective pairing function on natural numbers). Its inverse, `numdB`, is defined by recursion on $\mathbb{N}$.

Definition 4. *The Φ function is defined:*

$\vdash$ $\Phi\ m\ n\ \texttt{=}$
    $\texttt{OPTION\_MAP force\_num}$
      $(\texttt{bnf\_of}\ (\texttt{toTerm}\ (\texttt{numdB}\ m) \diamond \texttt{church}\ n))$

The `force_num` *function takes a λ-term and returns n if it is an instance of* `church` n*, and* `0` *otherwise.*

The Φ function gives us a purely HOL-level picture of the λ-calculus's computational capabilities, expressed in terms of functions on natural numbers. It will be our target when we investigate the capabilities of the recursive functions in Section 4 below.

Theorem 1. *There exists a λ-term that computes Φ. It is called* `UM`*, with characterising theorems:*

$$\vdash \Phi\ m\ n = \mathtt{NONE} \iff \mathtt{bnf\_of}\ (\mathtt{UM} \diamond \mathtt{church}\ (m \otimes n)) = \mathtt{NONE}$$
$$\vdash \Phi\ m\ n = \mathtt{SOME}\ p \iff$$
$$\mathtt{bnf\_of}\ (\mathtt{UM} \diamond \mathtt{church}\ (m \otimes n)) = \mathtt{SOME}\ (\mathtt{church}\ p)$$

4 The Recursive Functions

The first issue to resolve when modelling the recursive functions is whether to treat them "shallowly" or "deeply". This is not an issue that arises with the λ-calculus where it is natural to want to model the syntax of the calculus, and to then ascribe meaning to that syntax (a deep embedding). By way of contrast, with the recursive functions it seems equally natural to want to use the existing functions that exist in HOL, to identify a subset of those as primitive recursive, to then extend that subset with minimisation and thereby gain the recursive functions. Unfortunately, one then has to deal with the fact that the recursive functions are of variable arity, which is difficult to model in HOL's unsophisticated type system.

Rather than force the burden onto the type system, we use the type

$$\mathtt{num\ list} \rightarrow \mathtt{num}$$

for the primitive recursive functions, and add arity information to the inductive definition which identifies them. This approach doesn't treat application of a function to the wrong number of arguments (a list of the wrong length) as a type-error, but expects the sanity checking to be enforced through appropriate `primrec` assumptions (see Figure 3). The interesting auxiliary constants from that definition are for function composition (`Cn`) and primitive recursion (`Pr`), with characterising theorems:

$$\vdash \mathtt{Cn}\ f\ gs\ \ell = f\ (\mathtt{MAP}\ (\lambda g.\ g\ \ell)\ gs)$$
$$\vdash \mathtt{Pr}\ b\ r\ (\mathtt{0}::t) = b\ t$$
$$\vdash \mathtt{Pr}\ b\ r\ (\mathtt{SUC}\ m::t) = r\ (m::\mathtt{Pr}\ b\ r\ (m::t)::t)$$

Apart from all the standard arithmetic that can be shown to be primitive recursive, we gain confidence in this definition by also proving the famous result about Ackermann's function.

Theorem 2. *For any primitive recursive function f, there is an index J such that for all possible arguments xs, $f(xs)$ is always less than the Ackermann function applied to J and the sum of the values in xs:*

$$\frac{}{\texttt{primrec zerof 1}} \qquad \frac{}{\texttt{primrec succ 1}}$$

$$\frac{i < n}{\texttt{primrec (proj } i\texttt{) } n}$$

$$\frac{\texttt{primrec } f\ |gs| \quad \texttt{EVERY } (\lambda g.\ \texttt{primrec } g\ m)\ gs}{\texttt{primrec (Cn } f\ gs\texttt{) } m}$$

$$\frac{\texttt{primrec } b\ n \quad \texttt{primrec } r\ (n + 2)}{\texttt{primrec (Pr } b\ r\texttt{) } (n + 1)}$$

Fig. 3. The primitive recursive functions. The relation `primrec` $f\ n$ is true if f is primitive recursive and behaves “sensibly” on arguments of length n (because HOL functions are total, f will have a value on lists of other lengths too). The auxiliaries are as follows: `zerof` is the constant function returning 0; `succ` returns the successor of the head of a list; `proj` is the projection function on lists; `Cn` (composition) and `Pr` (primitive recursion) are described in the main text. The `EVERY` auxiliary is from HOL’s theory of lists and checks a predicate holds of every element in a list.

$$\begin{array}{l} \vdash \texttt{primrec } f\ k \Rightarrow \\ \quad \exists J.\ \forall xs.\ |xs| = k \Rightarrow f\ xs < \texttt{Ackermann } J\ (\texttt{SUM } xs) \end{array}$$

The proof closely follows the version of this result in the Isabelle/HOL sources, which is in turn based on Szasz [8].[2]

Recursive Functions Adding minimisation to the primitive recursive functions forces the use of the option type to correctly model partiality. Thus the recursive functions are all of type

$$\texttt{num list} \to \texttt{num option}$$

Both the minimisation operation and the composition operator for recursive functions have rather ugly definitions (see Figure 4). The term `minimise` $f\ \ell$ is `NONE` if there is no value x such that $f\ (x::\ell) = \texttt{SOME } 0$, or if there is some $y < x$ such that $f\ (y::\ell) = \texttt{NONE}$. Function composition is strict: if any of the functions in the list gs fails on the provided argument, so too does the composition. Similarly, primitive recursion: if a recursive call fails on $n < m$, then the recursive call on m must be held to fail as well.

The analogue of the `primrec` constant is `recfn`. It is an easy induction on the rules governing `primrec` to show

$$\vdash \texttt{primrec } f\ n \Rightarrow \texttt{recfn (SOME} \circ f\texttt{) } n$$

In the proofs to come, minimisation is only used once. All the other necessary operations were shown to be primitive recursive. This is implicitly a proof of Kleene’s

[2] My original proof, in the Kananaskis-6 release, follows Taylor’s more complicated argument [9]. Thanks to the anonymous referees for the pointer to the proofs in Isabelle/HOL and by Szasz.

```
⊢ recCn $f$ $gs$ $\ell$ =
    (let $results$ = MAP ($\lambda g.\ g\ \ell$) $gs$
     in
       if EVERY ($\lambda r.\ r \neq$ NONE) $results$ then
         $f$ (MAP THE $results$)
       else
         NONE)

⊢ minimise $f$ $\ell$ =
    if
      $\exists n.$
        $f$ ($n$::$\ell$) = SOME 0 $\land$
        $\forall i.\ i < n \Rightarrow \exists m.$ 0 $< m \land f$ ($i$::$\ell$) = SOME $m$
    then
      SOME
        ($\varepsilon n.$
           $f$ ($n$::$\ell$) = SOME 0 $\land$
           $\forall i.\ i < n \Rightarrow \exists m.$ 0 $< m \land f$ ($i$::$\ell$) = SOME $m$)
    else
      NONE
```

Fig. 4. The composition and minimisation operations for the recursive functions. As with `primrec`, these definitions are used in an inductive definition that specifies valid arities.

Normal Form theorem, stating that all recursive functions can be expressed as a composition of a primitive recursive function, minimisation and one other primitive recursive function.

5 Computational Equivalence

The "Easy" Direction Given the existence of `cfindleast`, and the general machinery of the Church numbers, one might imagine it straightforward to prove that the λ-calculus can implement the recursive functions. However, the journey is beset by a number of annoyances. First: how to represent the list of arguments the recursive functions expect? Our answer is to use the `nlist_of` function, which bijectively encodes a list of natural numbers as a single natural number.

Theorem 3. $\vdash$ `recfn` $f\ n \Rightarrow \exists i.\ \forall \ell.\ \Phi\ i$ (`nlist_of` ℓ) = $f\ \ell$

(The fact that the theorem quantifies over all ℓ *(rather than just those of length n) is a consequence of the fact that the definitions of the (primitive) recursive operators (*`Pr`, `Cn` *etc) actually give them reasonable values on lists of the wrong size. This can be emulated in the λ-calculus too.)*

Proof. The big issue in this proof is the accurate modelling of partiality. For example, consider the primitive recursion case. By our inductive hypothesis, we have an i and j which are the indexes of the 0-case function and `SUC`-case function respectively. If the argument on which the function is recursing is n, it is necessary to set up a stack of n pending computations, linked together with continuation arguments. Thus, machine i is run first, and if it terminates, its result is passed to machine j with varying argument 0. This instance will have a continuation that passes the result onto machine j with varying argument 1, and so on, all the way up to one final computation: machine j with arguments $n - 1$, the result of the previous computation, and a continuation which is the identity function. With this nesting structure, the constructed λ-term is guaranteed to loop (fail) if and only if there is a failure in the calls made by the recursive function.

The Hard Direction In the other direction, it is necessary to model the de Bruijn terms as numbers, and to perform all of the appropriate operations (*e.g.*, finding a redex, performing a substitution) purely arithmetically. Moreover, these operations are all shown to be primitive recursive, further increasing the complexity of the proofs and definitions.

As an example, the following theorems are the key facts about the form of substitution on de Bruijn terms (called `nsub` here) that simultaneously adjusts indices to reflect the disappearance of an outer abstraction (as happens in β-reduction). The first theorem states that the new constant `pr_nsub` really does the right thing with suitably encoded terms; the second that the constant really is primitive recursive:

```
⊢ pr_nsub [s; k; t] = dBnum (nsub (numdB s) k (numdB t))
⊢ primrec pr_nsub 3
```

The complexity in these proofs stem from the fact that we need to perform recursions that are not obviously primitive recursive. Firstly, when recursing over an encoded term, sub-terms have encodings that are numbers (much) smaller than the enclosing term, not just one less. Secondly, one also needs to be able to vary the accompanying parameters, as happens to k in the `dABS` clause of the definition of the `lift` function:

```
⊢ lift (dABS s) k = dABS (lift s (k + 1))
```

It is folklore that both of these variations do not require anything more than primitive recursion. Actually achieving them requires the use of primitive recursive functions that return large lists (encoded as numbers!) of results rather than single numbers. At the call-site, the calling function can then pick out the result it is really interested in, and then calculate an even larger list to be its own result.

When all this work within the primitive recursive functions has been done, the minimisation operation can be used to define the recursive "β normal form of" function, with definition:

```
⊢ recbnf_of =
    recCn (SOME ∘ pr_steps)
      [minimise (SOME ∘ pr_steps_pred); SOME ∘ proj 0]
```

The (primitive recursive) `pr_steps` function takes parameters n and t and performs n normal order reduction steps on t. The (primitive recursive) `pr_steps_pred` function takes parameters n and t and returns 0 if n steps of normal order reduction on t produces a term in β-normal form.

We are thus able to characterise `recbnf_of`:

$\vdash$ `recfn recbnf_of 1`
$\vdash$ `recbnf_of` $[t]$ `=`
  `OPTION_MAP (dBnum` $\circ$ `fromTerm) (bnf_of (toTerm (numdB` t`)))`

This leads to

Theorem 4. *There exists a recursive function* `recPhi` *of type*

`num list` $\rightarrow$ `num option`

which emulates Φ*:*

$\vdash$ `recfn recPhi 2`
$\vdash$ `recPhi` $[i;\ n]$ `=` $\Phi\ i\ n$

6 Computability Theorems

Here we list a number of standard results that can be derived on top of the framework that has been established. The most complicated proofs are those to do with the recursively enumerable sets, where care is often required to handle computations that may not terminate.

Definition 5. *A* recursive *set (of natural numbers) is one that a computable function decides:*

$\vdash$ `recursive` $s \iff$
  $\exists m.\ \forall e.\ \Phi\ m\ e$ `= SOME (`**if** $e \in s$ **then** `1` **else** `0)`

Theorem 5. *The empty, finite and universal sets are recursive; recursive sets are closed under union, intersection and complement.*

$\vdash$ `recursive` $\emptyset$
$\vdash$ `recursive` $\mathcal{U}$`(:num)`
$\vdash$ `FINITE` $s \Rightarrow$ `recursive` s
$\vdash$ `recursive` $s_1 \land$ `recursive` $s_2 \Rightarrow$ `recursive` $(s_1 \cup s_2)$
$\vdash$ `recursive` $s_1 \land$ `recursive` $s_2 \Rightarrow$ `recursive` $(s_1 \cap s_2)$
$\vdash$ `recursive (COMPL` s`)` $\iff$ `recursive` s

where $\mathcal{U}$`(:num)` *denotes the universal set of natural numbers, and where* `COMPL` s *is the complement of set s.*

Definition 6. *A* recursively enumerable *(r.e.) set is one that is the range of a computable function*

$\vdash$ `re` $s \iff \exists M_i.\ \forall e.\ e \in s \iff \exists j.\ \Phi\ M_i\ j$ `= SOME` e

Theorem 6. *Alternatively, the r.e. sets are those that are the domains of computable functions:*

$\vdash$ `re` $s \iff \exists N.\ \forall e.\ e \in s \iff \exists m.\ \Phi\ N\ e$ `= SOME` m

This result requires an implementation of dove-tailing, whereby the machine M_i is run on arguments $0..n-1$ for n steps, and the results examined for β-normal forms. If the argument e is not among them, then the process is repeated with parameter $n+1$.

Theorem 7. *All recursive sets are r.e. The r.e. sets are closed under union and intersection. If a set and its complement are r.e., then they are both recursive.*

$\vdash$ `recursive` $s \Rightarrow$ `re` s
$\vdash$ `re` $s \land$ `re` $t \Rightarrow$ `re` $(s \cap t)$
$\vdash$ `re` $s \land$ `re` $t \Rightarrow$ `re` $(s \cup t)$
$\vdash$ `re` $s \land$ `re (COMPL` s`)` $\Rightarrow$ `recursive` s

Theorem 8. The Halting Problem. *Let $\mathcal{K}$ be defined as follows ("the machines that halt on their own indices"):*

$\vdash \mathcal{K} = \{M_i \mid \exists z.\ \Phi\ M_i\ M_i = \texttt{SOME}\ z\}$

Then $\mathcal{K}$ is r.e. but not recursive. Its complement is not even r.e.

$\vdash \neg$`recursive` $\mathcal{K}$
$\vdash$ `re` $\mathcal{K}$
$\vdash \neg$`re (COMPL` $\mathcal{K}$`)`

Theorem 9. The "s-1-1" theorem. *There exists a computable function with index* `s11` *that, when given an encoded pair $x \otimes y$, returns the index of a function that computes the function $\lambda z.\ \Phi\ x\ (y \otimes z)$. In other words, x is the index of the function to be partially evaluated with parameter y provided in advance.*

$\vdash \forall x\ y.\ \exists f_i.\ \Phi\ \texttt{s11}\ (x \otimes y) = \texttt{SOME}\ f_i \land \forall z.\ \Phi\ f_i\ z = \Phi\ x\ (y \otimes z)$

Theorem 10. The Recursion Theorem. *If f_i is the index of a total function (understood to be computing indices of other functions), then it has a fix-point e such that the functions with indices $f(e)$ and e are extensionally equal.*

$\vdash (\forall n.\ \exists r.\ \Phi\ f_i\ n = \texttt{SOME}\ r) \Rightarrow \exists e.\ \Phi\ (\texttt{THE}\ (\Phi\ f_i\ e)) = \Phi\ e$

(With the λ-calculus to hand, directly using the Y combinator is a much more pleasant prospect than the route *via* this theorem, with all its confusions of terms and indices encoding terms.)

Theorem 11. Rice's Theorem. *Let P be a predicate on functions. The predicate P is of type* `(num` $\rightarrow$ `num option)` $\rightarrow$ `bool` *and thus considers just the functions' extensional behaviour. Let* `indices` P *be the set of indices of computable functions satisfying P. Then, if* `indices` P *is recursive, that set is either the empty set, or the set of all numbers.*

$\vdash$ `recursive (indices` P`)` $\Rightarrow$ `indices` $P = \emptyset \lor$ `indices` $P = \mathcal{U}$`(:num)`

7 Related Work

Zammit [11, §3] describes a HOL mechanisation of register machines, and shows that they can compute the recursive functions. He does not show the converse result. He also develops a Coq mechanisation of the recursive functions, and shows the *s*-*m*-*n* theorem in that model.

Computable functions of some form are necessarily a part of formalisations of Gödel's incompleteness theorems, and so mechanisations of that result by Shankar [7] and O'Connor [5] include approaches to computability. O'Connor uses the primitive recursive functions; Shankar uses a 'pure' subset of Lisp. Both are concerned with using their computational models to show that various formula manipulations are computable; neither is (directly) concerned with the limits of what is generally computable. Similarly, John Harrison's proof of Gödel's incompleteness theorem in the HOL Light system [2] focuses on showing the representability of primitive recursion in the embedded logic.

The Isabelle system comes with a mechanisation of the primitive recursive functions and a proof that Ackermann's function is not one of them. The ZF mechanisation is described in Paulson [6], who followed Szasz [8].

8 Conclusion

There is always more to do. Clearly, it would be appealing to mechanise the more operational models of computation: Turing and register machines. For the latter, the work by Zammit may be a good starting point. If register machines are unappealing because of their general fiddliness, Turing machines are an even more daunting prospect. Nonetheless, the completist would clearly want to include both these models.

It would also be fun to attack further results in computability theory. For example, the theory of Turing degrees includes a number of classic results, with fascinating proofs. Alternatively, there is always basic complexity theory...

We cannot yet provide an easy route to proofs of computability for complicated systems with their own elaborate data types (as would be required for the introduction's motivating example). Nonetheless, the work done to date has demonstrated that the λ-calculus provides a good environment for working with rich types (such as the de Bruijn terms), and for manipulating them in ways known to be computable.

Acknowledgements NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, revised edition, 1984.
2. John Harrison. The HOL Light system. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

3. Michael Norrish. Mechanising λ-calculus using a classical first order theory of terms with permutations. *Higher Order and Symbolic Computation*, 19:169–195, 2006.
4. Michael Norrish and René Vestergaard. Proof pearl: de Bruijn terms really do work. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference*, volume 4732 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.
5. Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In Joe Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference*, volume 3603 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2005.
6. Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *CADE-12: 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 1994.
7. N. Shankar. *Metamathematics, Machines and Gödel's Proof.* Number 38 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1994.
8. Nora Szasz. A machine checked proof that Ackermann's function is not primitive recursive. In *Papers Presented at the Second Annual Workshop on Logical Environments*, pages 317–338, New York, NY, USA, 1993. Cambridge University Press.
9. R. Gregory Taylor. Ackermann's function is not primitive recursive. From `http://home.manhattan.edu/~gregory.taylor/thcomp/pdf-files/ackerman.pdf`. Most recently accessed on 16 February 2011.
10. Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of LF. *ACM Transactions on Computational Logic*, 12:1–42, January 2011.
11. Vincent Zammit. *On the Readability of Machine Checkable Formal Proofs*. PhD thesis, University of Kent at Canterbury, March 1999.