*Technical Report*

UCAM-CL-TR-370
ISSN 1476-2986

Number 370

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

# A package for non-primitive recursive function definitions in HOL

Sten Agerholm

July 1995

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500
*https://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*https://www.cl.cam.ac.uk/techreports/*

ISSN 1476-2986

# A Package for Non-primitive Recursive Function Definitions in HOL*

Sten Agerholm
University of Cambridge Computer Laboratory,
New Museums Site, Cambridge CB2 3QG, UK

**Abstract**

This paper provides an approach to the problem of introducing non-primitive recursive function definitions in the HOL system. A recursive specification is translated into a domain theory version, where the recursive calls are treated as potentially non-terminating. Once we have proved termination, the original specification can be derived easily. Automated tools implemented in HOL88 are provided to support the definition of both partial recursive functions and total recursive functions which have well-founded recursive specifications. There are constructions for building well-founded relations easily.

*This work was supported by an HCMP fellowship under the EuroForm network.

# Contents

# 1 Introduction

In order to introduce a recursive function in the HOL system, we are required to prove its existence as a total function in higher order logic (see [8] page 263). While this has been automated for certain primitive recursive functions in the type definition package [10], the HOL system does not support the definition of recursive functions which are not also primitive recursive. In particular, it does not support partial functions whose undefinedness is induced by non-terminating recursive calls.

Previous work [3, 4, 2] has shown that a formalization of domain theory in HOL provides some useful concepts and techniques for reasoning about both partial and total recursive functions in HOL. However, the formalization previously presented has been too complicated and difficult to use, for instance because it required the use of a dependent $\lambda$-abstraction instead of the standard HOL one and because all function constructions like the fixed point operator were parameterized with cpos. If we focus on just extending HOL with better support for recursive function definitions, then the formalization can be simplified considerably.

This paper presents a package for non-primitive recursive function definitions which is based on a much simplified (and weaker) formalization. This formalization is integrated more closely with the HOL logic in order to allow very smooth and easy transitions between higher order logic and domain theory. A main goal was to make the domain theory as invisible as possible whilst providing efficient automated support for difficult recursive function definitions at the same time. A similar methodology of having domain theory behind the scenes may be useful for other purposes.

In treating partial recursive functions, the user will be faced with only very few simple constructs of domain theory, which are needed to represent partiality. However, these have straightforward syntactic interpretations and are inserted automatically by the package, so no knowledge of domain theory is necessary. As an example consider the following interaction with HOL which defines a partial recursive function of type `"prf:(num)list#num->(num)lift"`, though the variable of the specification has type `"prf:(num)list#num->num"` (HOL sessions are framed):

```
#let prf_def = new_prec_definition `prf_def`
#  "prf(l,y) =
#   (NULL l => y | prf(APPEND(r(HD l,y))(TL l),s(HD l,y)))";;
prf_def =
|- !l y.
    prf(l,y) =
    (NULL l => lift y | prf(APPEND(r(HD l,y))(TL l),s(HD l,y)))
Run time: 1.7s
Intermediate theorems generated: 353
```

The definitions of the constants `r` and `s` are not important here; in fact we could have chosen to represent `r` and `s` as variables. The constant `prf` is a partial function since we do not know whether or not the recursive call terminates (unless `r` always returns the empty list from a certain point it will not terminate). where `r` and `s` are constants denoting arbitrary values. Partiality is represented by lifting types. The constant `lift`, which is inserted in the definition automatically, is a constructor of the abstract datatype

of syntax `(*)lift =. bot | lift *`, which is used to add a new element to types. A partial function is undefined if it equals the new element `bot`, and defined otherwise. The constant `prf` is introduced by a definition using the fixed point operator of domain theory; the above equation is derived from this definition. An advantage of domain theory is that it allows a (partial) recursive function to be defined and reasoned about directly, rather than having to first prove the existence of a total function satisfying a specification in HOL.

The paper also presents an automated tool for introducing a wide class of recursive functions whose termination can be proved by well-founded induction. The problem of proving the existence of a total function in higher order logic is approached by first defining a potentially partial recursive function in domain theory using the fixed point operator, and then proving this function terminates on all inputs by well-founded induction. The details of domain theory never appear to the user who just supplies a well-founded relation, a recursive specification, and a list of termination properties of the specification.

Let us consider a famous example of a well-founded recursive function: the (binary) Ackermann function. Often it is specified by a collection of recursion equations

$$\begin{aligned} A(0,y) &= y+1 \\ A(x+1,0) &= A(x,1) \\ A(x+1,y+1) &= A(x,A(x+1,y)), \end{aligned}$$

which are equivalent to the following conditional style of specification in HOL:

```
"ACK(m,n) =
 ((m = 0) => SUC n |
  (n = 0) => ACK(PRE m,1) | ACK(PRE m,ACK(m,PRE n)))".
```

In this term, called `ack_tm` below, `ACK` is a variable but we wish to obtain a constant `ACK` that satisfies the recursive specification. The Ackermann function is not primitive recursive since it contains recursive calls that roughly speaking do not decrease precisely one of its arguments. In other words, it cannot be defined using the syntax of primitive recursive specifications. However, its recursive specification can be proved to terminate by showing that in each recursive call the arguments are decreased with respect to some well-founded relation. A binary relation `"R:*->*->bool"` is called well-founded, written `"wf R"`, if it is does not allow any infinite decreasing sequences of values. Hence, recursions will terminate eventually.

In addition to the recursive specification, we must supply a well-founded relation and a list of termination properties of the specification. An ML function calculates the proof obligations for termination:

```
#calc_prf_obl ack_tm;;
["~(m = 0) /\ (n = 0) ==> R(PRE m,1)(m,n)";
 "~(m = 0) /\ ~(n = 0) ==> R(PRE m,k0)(m,n)";
 "~(m = 0) /\ ~(n = 0) ==> R(m,PRE n)(m,n)"]
: term list
```

These are constructed by looking at the arguments of each recursive call. Note that the function does not guess a well-founded relation but uses instead a variable `R`. We must

find a proper instantiation for R and prove each resulting term is a theorem. It is easy to see that a suitable well-founded relation in this example is a lexicographic combination of the less-than ordering on natural numbers with itself. Proving that this relation is well-founded is trivial since lexicographic combination and the less-than ordering are standard constructions on well-founded relations which are provided by the package (see Section 2):

```
#wf_less;;
|- wf $<

#wf_lex;;
|- !R. wf R ==> (!R'. wf R' ==> wf(lex(R,R')))

#let wf_ack = MATCH_MP (MATCH_MP wf_lex wf_less) wf_less;;
wf_ack = |- wf(lex($<,$<))
```

Hence, we substitute `lex($<,$<)` for the variable R in the proof obligations above (there is a separate tool for this). We shall omit the proofs here but assume the proven termination properties have been saved in the ML variable `obl_thl`.

The Ackermann function can now be defined automatically using a definition tool called `new_wfrec_definition`. This introduces a new constant `"ACK:num#num->num"` and proves that it satisfies the recursive specification presented above:

```
#let ACK_DEF = new_wfrec_definition 'ACK_DEF' wf_ack obl_thl ack_tm;;
ACK_DEF =
|- !m n.
    ACK(m,n) =
    ((m = 0) => SUC n |
     (n = 0) => ACK(PRE m,1) | ACK(PRE m,ACK(m,PRE n)))
Run time: 4.3s
Intermediate theorems generated: 788
```

If we wish we can now prove that ACK satisfies the recursion equations listed above. The run time is measured on a standard Sun Sparc ELC, and is acceptable, I believe. The present implementation of `new_wfrec_definition` is a prototype which I have made no serious attempts to optimize for efficiency.

For simplicity of implementation, the function definition package only accepts recursive specifications written in the conditional style:

$$f\,x = (b_1[f,x] \to h_1[f,x] \mid \ldots \mid b_n[f,x] \to h_n[f,x] \mid h_{n+1}[f,x]).$$

Recursion equation specifications can (probably) always be written equivalently in the conditional style, but eventually it would be nice to support these. Furthermore, the package expects the recursive function being specified to be uncurried as in $f(x,y)$, hence, $f\,x\,y$ is not allowed, and it does not allow recursive calls within the body of a $\lambda$-abstraction. However, it does support the use of `let`-terms where the left-hand sides are variables, for

instance:

$$
\begin{array}{l}
f\,x = \\
(b_1[f,x] \rightarrow h_1[f,x] \mid \\
\quad let\ y = g[f,x]\ in\ b_2[f,x,y] \rightarrow h_2[f,x,y] \mid h_3[f,x,y]).
\end{array}
$$

Finally, the package expects recursive occurrences of a function to be applied to an argument and does not allow recursive occurrences in the body of $\lambda$-abstractions[1].

The rest of the paper is organized as follows. A theory of well-founded relations is presented in Section 2 and a minimal theory of domain theory is presented in Section 3. The underlying algorithms of the package are presented in Section 4. Section 5 treats an example of reasoning about partial recursive functions and Section 6 provides two additional examples of well-founded recursive functions. Section 7 presents a non-trivial example of a well-founded recursive function: the unification algorithm. Section 8 contains the conclusions and related work. Appendix A shows a list of theorems and tools provided by the package and Appendix B presents the ML code of the Ackermann example treated above and of the example of Section 5.

# 2 Well-founded Relations

A binary relation is defined to be well-founded on some type if all non-empty subsets of the type have a minimal element with respect to the relation:

```
|- !R.
    wf R =
    (!A. ~(A = (\x. F)) ==> (?x. A x /\ ~(?y. A y /\ R y x))).
```

The HOL theory of well-founded relations presented here was obtained by developing a special case of the theory presented in [1], which was based on a chapter of the book by Dijkstra and Scholten [7].

In general, it can be non-trivial to prove a given relation is well-founded. It is therefore useful to have standard ways of combining well-founded relations to build new ones. The package provides the following standard constructions on well-founded relations, which can be used to prove very easily that relations are well-founded:

**Less-than on numbers:** ML name `wf_less`:

```
|- wf $<.
```

**Product:** ML name `wf_prod`:

```
|- !R. wf R ==> (!R'. wf R' ==> wf(prod(R,R'))).
```

Defined by

```
|- !R R' b c. prod(R,R')b c = R(FST b)(FST c) /\ R'(SND b)(SND c).
```

[1]Let-terms are represented using $\lambda$-abstractions but recursive calls in the body of such abstractions are supported.

**Lexicographic combination:** ML name `wf_lex`:

```
|- !R. wf R ==> (!R'. wf R' ==> wf(lex(R,R'))).
```

Defined by

```
|- !R R' b c.
    lex(R,R')b c =
    R(FST b)(FST c) \/ (FST b = FST c) /\ R'(SND b)(SND c).
```

**Inverse image:** ML name `wf_inv_gen`:

```
|- !R. wf R ==> (!R' f. (!x y. R' x y ==> R(f x)(f y)) ==> wf R').
```

A useful special case of the construction is (ML name `wf_inv`):

```
|- !R. wf R ==> (!f. wf(inv(R,f))).
```

Defined by

```
|- !R f. inv(R,f)x y = R(f x)(f y).
```

In the examples, we construct well-founded relations solely by instantiating these constructions.

When the built-in constructions do not suffice, a relation can be proved to be well-founded from the definition of `wf`, or, which is often more convenient, either from the theorem

```
|- !R. wf R = ~(?X. !n. R(X(SUC n))(X n)),
```

which states that a relation is well-founded if and only if there are no infinite decreasing sequences of values, or from the principle of well-founded induction:

```
|- !R. wf R = (!P. (!x. (!y. R y x ==> P y) ==> P x) ==> (!x. P x)),
```

which states that a relation is well-founded if and only if it admits mathematical induction. Note that this theorem can be used both to prove a relation is well-founded by proving it admits induction and to perform an induction with a relation which is known to be well-founded.

# 3 Domain Theory

In [3], I presented a formalization of basic concepts of domain theory in the HOL system. Using this extension of HOL, it was possible to reason about non-termination and general recursive functions defined as fixed points. The main challenge of the present work has been to simplify the previous formalization to be more useful for automation, in particular more concrete, and to automate the previously manual process of giving a domain theoretic (fixed point) definition of a partial recursive function, and possibly, if the function can be proved to be total, then deriving a pure HOL definition from this. The simplified formalization is presented in this section and the automation in the following section.

The basic concepts of domain theory can be formalized as follows. A partial order is a binary relation `"R:*->*->bool"` which is reflexive, transitive and antisymmetric:

```
|- !R.
    po R =
    (!x. R x x) /\
    (!x y z. R x y /\ R y z ==> R x z) /\
    (!x y. R x y /\ R y x ==> (x = y)).
```

A complete partial order is a partial order which contains the least upper bounds of all non-decreasing chains of values:

```
|- !R. cpo R = po R /\ (!X. chain R X ==> (?x. islub R X x)),
```

where we have defined

```
|- !R X x. isub R X x = (!n. R(X n)x)
|- !R X x. islub R X x = isub R X x /\ (!y. isub R X y ==> R x y)
|- !R X. chain R X = (!n. R(X n)(X(SUC n))).
```

Also essential to domain theory is the notion of continuous functions, which are monotonic functions that preserve least upper bounds of chains:

```
|- !f R R'.
    cont f(R,R') =
    (!x y. R x y ==> R'(f x)(f y)) /\
    (!X. chain R X ==> (f(lub R X) = lub R'(\n. f(X n))))
```

where

```
|- !R X. lub R X = (@x. islub R X x).
```

Compared to the more powerful formalization presented in [3], a main simplification above is the formalization of partial orders as just relations instead of pairs of sets and relations. This simplification is possible since it is not necessary to consider the cpo construction on continuous functions generally, which is also called the continuous function space, but only to consider one particular instance of this construction (called `frel` below). Thereby we in turn avoid the need for for a notion of partially specified functions, which are specified on cpo subsets only. In turn, a new $\lambda$-abstraction would have to be defined to make continuous functions determined by their action on the subsets, by ensuring that they yield a fixed arbitrary value outside the subsets. Otherwise, it is not possible to show that continuous functions constitute a cpo with the pointwise ordering.

We are able to manage the entire development with just two different cpo relations, defined as follows

```
|- !x y. lrel x y = (x = bot) \/ (x = y)
|- !f g. frel f g = (!x. lrel(f x)(g x)),
```

where `bot` is a constructor of a new datatype of syntax specified by

```
lift = bot | lift *.
```

Note that `lift` is the name of both the type being specified and of one of the two constructors. The relation `lrel` ensures that `bot` is the bottom element, i.e. a least value which can be used to represent undefinedness, and behaves as the discrete ordering on lifted values. The relation `frel` is the pointwise ordering on functions and works on functions with a lifted range type. The cpo `frel` also has a bottom element, which is the everywhere undefined function, i.e. the constant function that sends all values to `bot`.

The notions of cpos and continuous functions allow a fixed point operator to be defined for continuous functions on a cpo with a bottom element. However, we shall only wish to take the fixed points of continuous functionals on `frel` whose types are instances of the type `":(*->(**)lift)->(*->(**)lift)"`:

```
|- !f. cont f(frel,frel) ==> (f(fix f) = fix f).
```

This theorem is called the fixed point property, and is essential to the automation. Recursive functions are defined as fixed points of continuous functionals. The fixed point operator is defined by

```
|- !f. fix f = lub frel(\n. power n f)
```

where

```
|- (!f. power 0 f = (\x. bot)) /\
   (!n f. power(SUC n)f = f(power n f)).
```

The need for the lifted type constructor appears in the definition of the fixed point operator and in turn in the definition of `power`. Note that the function returned by `power` in the zero case is the bottom element of the cpo `frel`.

Note that a recursive function defined as a fixed point has a type of the form `":* -> (**)lift"`, where the range type is lifted. This means that recursive calls in its specification cannot be used directly with other HOL terms, which would expect an unlifted term of type `":**"`. In order to solve this problem, we introduce a construction `ext`, called function extension, which can be used to extend HOL functions in a strict way:

```
|- (!f. ext f bot = bot) /\ (!f x. ext f(lift x) = f x).
```

For instance, the term `"ext(\x. x+5)"` extends addition to a strict function in its first argument. In the automation presented below, we shall use function extension to isolate recursive calls from pure HOL terms.

It is easy to derive domain theoretic techniques for recursion such as Park induction, which is stated by the theorem

```
|- !f. cont f(frel,frel) ==> !x. frel(f x)x ==> frel(fix f)x,
```

and the principle of fixed point induction, which is stated by

```
|- !P f.
    cont f(frel,frel) ==>
    admiss P /\ P(\x. bot) /\ (!x. P x ==> P(f x)) ==>
    P(fix f)
```

where the notion of admissibility of a predicate for fixed point induction is defined by

```
|- !P.
    admiss P = (!X. chain frel X /\ (!n. P(X n)) ==> P(lub frel X)).
```

# 4 Automation

The purpose of the above formalization is to serve as a basis for defining recursive functions in HOL. A recursive function specification "g x = rhs[g,x]" must be given (x may be a pair), where rhs is specified using the conditional style mentioned in the introduction; we shall say that the conditionals constitute a "backbone" of the specification. The function g must be uncurried, i.e. it must have a type like "g:*1#...#*n->**". We make two assumptions about rhs[g,x]:

- All occurrences of g must be applied to a term, in order to avoid function types with a lifted range type in unexpected places—these are allowed in arguments of ext only.
- No occurrences of recursive calls appear in the body of a $\lambda$-abstraction (unless it is part of a let-term), again in order to avoid function types with a lifted range type.

The definition of a constant g that satisfies the specification can be automated in the following steps:

1. Generate a functional G in domain theory from the recursive specification. This functional has the form \g' x. rhs'[g',x], where g' is a variable like g but with a lifted range type and where rhs' is a lifted domain theory version of rhs. The type of G is ":(*1#...#*n->(**)lift)->(*1#...#*n->(**)lift)".
2. Prove G is continuous: |- cont G(frel,frel).

For a partial recursive function definition the final steps are:

3'. Define a constant g by |- g = fix G. Its type is "g:*1#...#*n->(**)lift".

4'. Derive the recursive definition |- !x. g x = rhs'[g,x], where rhs' is a domain theoretic version of rhs, corresponding to the body of G. The derivation exploits continuity and the fixed point property.

These are both trivial steps. For a well-founded recursive function definition a well-founded relation, gR say, and a number of termination properties of the form

```
|- ~b1 /\ ... /\ ~bi-1 /\ bi ==> gR y x,
```

where the b's are conditions in the conditional backbone of the rhs and y is the argument of a recursive call of the i'th branch, must be given in addition to the specification, and the final steps are:

3. Prove the statement |- ?g. !x. lift(g x) = fix G x, saying that the recursive function defined by "fix G" always terminates. The proof is conducted by well-founded induction (using the termination properties) and exploits continuity.
4. Define a constant g from this theorem using constant specification. The type of g is "g:*1#...#*n->**".
5. Finally, prove "fix G x" is equal to "G(fix G)x", by the fixed point theorem and continuity, in turn this is equal to "rhs'[g,x]" by definition of g and finally, this is equal to "rhs[g,x]" by straight-forward case analyses on the conditional structure of rhs. Hence, we have derived the desired specification |- !x. g x = rhs[g,x].

Each of the first three (non-trivial) steps are described in separate sections below.

## 4.1 Generating the Functional (Step 1)

As explained above the goal is to generate a domain theory version of the right-hand side `rhs[g,x]`. This is done by two recursive algorithms, one for the backbone conditionals (nested with let-terms) and one for branches and conditions. We imagine the backbone algorithm is called first with the right-hand side of a specification. In the description below, we use primes to indicate that a term has been transformed, and therefore has a lifted type. In particular, the function variable `"g:*1#...#*n->**"` is replaced by the primed variable `"g':*1#...#*n->(**)lift"` with a lifted range type.

### Algorithm for Backbone

The input is either a conditional, a let-term, or the last branch of the backbone conditional:

**Conditional:** The input term has the form $(b \rightarrow t_1 \mid t_2)$. The branch $t_2$, which may be a new condition or let-term in the backbone, is transformed recursively, and $t_1$ is transformed using the branch and condition algorithm described below. If the condition does not contain $g$ then the result is $(b \rightarrow t_1' \mid t_2')$. Otherwise, $b$ is transformed using the branch and condition algorithm and the result is $\mathtt{ext}(\lambda a.\ (a \rightarrow t_1' \mid t_2'))b'$, where the condition $b$ has been separated from the conditional using function extension.

**Let-term:** The input has the form `let` $a = t_1$ `in` $t_2$, which may use a list of bindings separated by and's. Transform $t_2$ recursively and use the branch algorithm on $t_1$. The result has the form $\mathtt{ext}(\lambda a.\ t_2')t_1'$. Lists of bindings are transformed into nested uses of function extension.

**Otherwise:** The term is considered to be the last branch of the backbone and therefore transformed using the branch algorithm.

### Algorithm for Branches and Conditions

The input has no particular form. The purpose of the algorithm is to lift terms that do not contain recursive calls and to isolate recursive calls using function extension in the terms that do.

**No recursive call:** If the variable $g$ does not appear in a free position in the input term $t$, then return `lift` $t$.

**Recursive call:** Assume the input term is a recursive call $g(t_1, \ldots, t_n)$. Each $t_i$ that contains $g$ must be transformed recursively. Separate these from the argument pair of $g$ using function extension and replace $g$ with $g'$. Assuming for illustration that $g$ takes four arguments of which the first and the third ones contain $g$, then the result has the form

$$\mathtt{ext}(\lambda a_1.\ \mathtt{ext}(\lambda a_3.\ g'(a_1, t_2, a_3, t_4))t_3')t_1' \ .$$

**Let-term:** The input has the form `let` $a = t_1$ `in` $t_2$, which may use a list of bindings separated by and's. Transform $t_1$ and $t_2$ recursively. The result has the form $\mathtt{ext}(\lambda a.\ t_2')t_1'$. Lists of bindings are transformed into nested uses of function extension.

**Combination:** The term has the form $t\ t_1\ \ldots\ t_n$, where $t$ is not a combination (or an abstraction containing $g$). Each argument of $t$ that contains $g$ is transformed recursively and these arguments are separated from the combination using nested function extensions. The combination in the body of the function extensions is lifted. Assuming for illustration that the input is $t\ t_1\ t_2\ t_3\ t_4$, and that $t_1$ and $t_3$ contain $g$, then the result has the form

$$\texttt{ext}(\lambda a_1.\ \texttt{ext}(\lambda a_3.\ \texttt{lift}(t\ a_1\ t_2\ a_3\ t_4))t_3')t_1' \ .$$

For a simple example consider the term $5 + g(2,3)$ which is transformed into the term $\texttt{ext}(\lambda a.\ \texttt{lift}(5 + a))(g'(2,3))$.

## 4.2 The Continuity Prover (Step 2)

The most complicated part of the automation is perhaps the continuity prover. Given the functional `G` constructed in the first step above, it must prove the continuity statement: `|- cont G(frel,frel).`

Recall that `G` is the abstraction `"\g' x. rhs'[g',x]"`. We first prove

```
|- !x. cont(\g'. rhs'[g',x])(frel,lrel)
```

and then establish the desired result using the continuity-abstraction theorem:

```
|- (!x. cont(\f. t f x)(frel,lrel)) ==> cont(\f x. t f x)(frel,frel).
```

To prove the first theorem, we let the conditional and `ext` term structure of `rhs'` guide our action in a recursive traversal. At each stage of the recursion, we have one of the following four cases (selected top-down):

**No recursive call:** The term does not contain any free occurrences of `g'`. The desired continuity theorem (upto $\alpha$-conversion) is obtained by instantiating

```
|- !t. cont(\f. t)(frel,lrel).
```

**Recursive call:** The term is a recursive call `"g'(t1,...,tn)"`. Instantiate the following theorem with `"(t1,...,tn)"` and do an $\alpha$-conversion:

```
|- !t. cont(\f. f t)(frel,lrel)
```

**Conditional:** The term is a conditional `"(b => t1 | t2)"`. Traverse the branches recursively, yielding

```
|- cont(\g'. t1)(frel,lrel)
|- cont(\g'. t2)(frel,lrel).
```

Note that the boolean guard `b` cannot depend on `g'` since such dependency would have been removed when the functional was generated. The desired result is obtained essentially by instantiating the following theorem (and using modus ponens):

```
|- !t1 t2.
    cont t1(frel,lrel) ==>
    cont t2(frel,lrel) ==>
    (!b. cont(\f. (b => t1 f | t2 f))(frel,lrel)).
```

**Function extension:** The term is an ext term "ext(\y. t1)t2". The terms t1 and t2 are traversed recursively, yielding

```
|- cont(\g'. t1)(frel,lrel)
|- cont(\g'. t2)(frel,lrel).
```

Next, the first of these and the continuity-abstraction theorem is used to deduce |- cont(\g' y. t1)(frel,frel). The desired result is obtained essentially by instantiating the following theorem:

```
|- !t1 t2.
    cont t1(frel,frel) ==>
    cont t2(frel,lrel) ==>
    cont(\f. ext(t1 f)(t2 f))(frel,lrel).
```

This completes the description of the continuity prover.

## 4.3 The Well-founded Induction (Step 3)

The goal of step 3 is to prove the statement "?g. !x. lift(g x) = fix G x" by well-founded induction. A user supplies a theorem stating some relation, gR say, is well-founded and a theorem list of termination properties of the original specification.

The principle of well-founded induction is stated as follows

```
|- !R. wf R = (!P. (!x. (!y. R y x ==> P y) ==> P x) ==> (!x. P x)).
```

Since our induction proofs always have the same structure, it is advantageous to derive the desired instance of this theorem once and for all:

```
|- !R.
    wf R ==>
    (!f.
      cont f(frel,frel) ==>
      (!x.
        (!x'. R x' x ==> (?y. fix f x' = lift y)) ==>
        (?y. f(fix f)x = lift y)) ==>
      (?g. !x. lift(g x) = fix f x))
```

Note that the conclusion matches the goal of step 3. The theorem is obtained by a few trivial manipulations. The induction predicate of the previous theorem is instantiated with "\x. ?y. fix f x = lift y". Then the consequent of the theorem is skolemized, which means that the existential ?y is moved outside the !x where it becomes ?g; note that y is a value while g is a function. Symmetry of equality is also used on the consequent.

Then the continuity assumption is used to obtain the term "?y. f(fix f)x = lift y" instead of "?y. fix f x = lift y" in the induction proof (i.e. the third antecedent); the fixed point property justifies this.

In step 3, the first two assumptions of the previous theorem are discharged by the user-supplied theorem |- wf gR and the continuity prover, respectively. The last assumption yields the induction proof:

```
"!x.
  (!x'. gR x' x ==> (?y. fix G x' = lift y)) ==>
  (?y. G(fix G)x = lift y)",
```

where the variables R and f have been instantiated. This proof is guided by the syntactic structure of the term "G(fix G)x", which by $\beta$-conversion is equal to "rhs'[fix G,x]". A case analysis is done for each conditional. For each recursive call, there must be a termination theorem in the user-supplied list of proof obligations. This allows us to use the induction hypothesis, i.e. the antecedent above. Hence, from the hypothesis and some proof obligation we derive that each recursive call terminates. In this way, we become able to reduce away all occurrences of ext and arrive at statements of the form "?y. lift t = lift y", which hold trivally.

# 5 Example: Partial Recursive Functions

Exercise 10.20 in Section 10.5 of Winskel's book [17] is a small but non-trivial exercise using various techniques for recursion to show the equality of two partial recursive functions on finite lists. Below, we present a solution to this exercise, the actual ML code is shown in Appendix B.

The two partial recursive functions are defined by:

```
#let f_def = new_prec_definition `f_def`
#  "f(l,y) =
#   (NULL l => y |
#    let x = HD l and l' = TL l in f(APPEND(r(x,y))l',s(x,y)))";;
f_def =
|- !l y.
    f(l,y) =
    (NULL l =>
     lift y |
     let x = HD l and l' = TL l in f(APPEND(r(x,y))l',s(x,y)))
Run time: 2.4s
Intermediate theorems generated: 532

#let g_def = new_prec_definition `g_def`
#  "g(l,y) =
#   (NULL l => y |
#    let x = HD l and l' = TL l in g(l',g(r(x,y),s(x,y))))";;
g_def =
|- !l y.
    g(l,y) =
    (NULL l =>
     lift y |
     let x = HD l and l' = TL l in ext(\v. g(l',v))(g(r(x,y),s(x,y))))
Run time: 3.0s
Intermediate theorems generated: 599
```

Note that the domain theory for representing partiality is introduced automatically. In particular, function extension is used at the nested recursive call since a recursive call may not terminate. The function `new_prec_definition` has various side effects on the theory file, apart from saving the returned theorem under the name supplied as an argument. It also saves the fixed point definition of the partial function (under the name *const*`_fix_def`, e.g. `f_fix_def` in the definition of `f` above), and defines a constant for the functional of this definition (*const*`_fun_def`). Finally, it saves a theorem stating the functional is continuous (*const*`_fun_cont`). These definitions and theorems are autoloaded, but the induction tactics reads the theory files directly.

The exercise is to prove the non-trivial statement: `"f = g"`. The proof starts by using antisymmetry of the cpo `frel` to reduce the statement to `"frel f g"` and `"frel g f"`. The first relation follows by proving `g` is a fixed point of the functional used in `f`'s fixed point definition, this functional is called `f_fun`, and exploiting that `f` is the least prefixed point of `f_fun`, by Park induction. The second relation is proved in a slightly different way since `f` is not a fixed point of `g_fun` directly. The proof also employs a Park induction and in addition a fixed point induction.

It is easy and convenient to prove the following recursion equations for `f`, `g`, and for the functionals used to define `f` and `g`:

```
f_EQS =
```

```
|- (!y. f([],y) = lift y) /\
   (!x xs y. f(CONS x xs,y) = f(APPEND(r(x,y))xs,s(x,y)))

g_EQS =
|- (!y. g([],y) = lift y) /\
   (!x xs y. g(CONS x xs,y) = ext(\v. g(xs,v))(g(r(x,y),s(x,y))))

f_fun_EQS =
|- (!y h. f_fun h([],y) = lift y) /\
   (!x xs y h. f_fun h(CONS x xs,y) = h(APPEND(r(x,y))xs,s(x,y)))

g_fun_EQS =
|- (!y h. g_fun h([],y) = lift y) /\
   (!x xs y h.
     g_fun h(CONS x xs,y) = ext(\v. h(xs,v))(h(r(x,y),s(x,y))))
```

Now, we can start an overview of the proof.

We first prove that g is a fixed point of f_fun, i.e. that the following statement holds:

```
|- f_fun g = g
```

From this, the desired relation "frel f g" follows by Park induction (see below for an example on using the Park induction tactic).

By function equality, we must prove "f_fun g(l,n) = g(l,n)" for all lists l and natural numbers n. From the equations for g and f_fun listed above it appears that the equality holds trivially when l is the empty list [] and when l is a list "CONS h t" it reduces to the goal

```
"g(APPEND(r(h,y))t,s(h,y)) = ext(\v. g(t,v))(g(r(h,y),s(h,y)))"
```

We prove the following slightly more general statement

```
"!xs l y. g(APPEND l xs,y) = ext(\v. g(xs,v))(g(l,y))",
```

using structural induction on the universally quantified list l. By definition of APPEND and by the equations for g we must prove the following two subgoals:

```
(step)
"!h y.
  ext(\v. g(APPEND l xs,v))(g(r(h,y),s(h,y))) =
  ext(\v. g(xs,v))(ext(\v. g(l,v))(g(r(h,y),s(h,y))))"
  1  ["!y. g(APPEND l xs,y) = ext(\v. g(xs,v))(g(l,y))" ]

(base)
"!y. g(xs,y) = ext(\v. g(xs,v))(lift y)"
```

The base case is first reduced using the equations for ext. There is a special tactic for this:

```
#e ext_REDUCE_TAC;;
OK..
"!y. g(xs,y) = g(xs,y)"
```

The resulting subgoal holds trivially. In the induction step, we first perform a case analysis on the argument of `ext`, i.e. on whether it equals `bot` or `lift`. Again, there is a special tactic for this purpose:

```
#E(GEN_TAC THEN GEN_TAC THEN ext_CASES_TAC "g(r(h,y),s(h,y))");;
OK..
2 subgoals
"g(APPEND l xs,x) = ext(\v. g(xs,v))(g(l,x))"
  2  ["!y. g(APPEND l xs,y) = ext(\v. g(xs,v))(g(l,y))" ]
  1  ["g(r(h,y),s(h,y)) = lift x" ]

"bot = bot"
  2  ["!y. g(APPEND l xs,y) = ext(\v. g(xs,v))(g(l,y))" ]
  1  ["g(r(h,y),s(h,y)) = bot" ]
```

The first subgoals holds by reflexivity and the seconds holds by the assumption. This completes the proof of `|- frel f g`.

Next, we consider the proof of the statement `"frel g f"`. The first step is a Park induction:

```
#e PARK_INDUCT_TAC;;
OK..
"frel(g_fun f)f"
```

The tactic automatically fetches the fixed point definition of g and the continuity theorem about g_fun from the theory file (and uses these theorems with the Park induction theorem of Section 3). To prove this goal, it is enough to prove `"lrel(g_fun f(l,y))(f(l,y))"` for any `(l,y)`, since the function ordering relation is defined pointwise. Doing a case split on the list `l` and rewriting with the equations for `f` and g_fun, we obtain the following two subgoals:

```
"lrel(ext(\v. f(t,v))(f(r(h,y),s(h,y))))(f(APPEND(r(h,y))t,s(h,y)))"
  1  ["l = CONS h t" ]

"lrel(lift y)(lift y)"
  1  ["l = []" ]
```

The first subgoal, where `l` is the empty list, is finished off using reflexivity of the cpo `lrel`, and in the proof of the second subgoal, where `l` is a non-empty list, we choose to abstract over the occurrences of `r` and `s` by proving the lemma:

```
"!xs l y. lrel(ext(\u. f(xs,u))(f(l,y)))(f(APPEND l xs,y))".
```

The first step is a fixed point induction on the second occurrence of f:

```
#e(FP_INDUCT_TAC `f` [2]);;
OK..
2 subgoals
"!xs l y. lrel(ext(\u. f(xs,u))(f_fun x''(l,y)))(f(APPEND l xs,y))"
  1  ["!xs l y. lrel(ext(\u. f(xs,u))(x''(l,y)))(f(APPEND l xs,y))" ]

"!xs l y. lrel(ext(\u. f(xs,u))bot)(f(APPEND l xs,y))"
```

This tactic automatically fetches the fixed point definition of f and the continuity of the functional f_fun from the theory file (see the fixed point induction theorem of Section 3). It also proves that the predicate admits induction, using some ad hoc syntactic checks (see Appendix A).

The first goal is the base case of the induction. It is proved by reducing ext, using the theorem |- !f. ext f bot = bot or the tactic used above, and using that bot is the least element with respect to lrel. In the induction step, we first do a case analysis on the list l and then uses the ext reduction tactic and various theorems (e.g. the definition and associativity of APPEND, the equations for f and f_fun and the reflexivity property of the cpo lrel). We shall not go into the details of the proof here. This completes the proof of |- frel g f.

To sum up, we have proved the equality |- f = g of two partial recursive functions f and g on finite lists. The equality was first split up into two (relation) cases, using the antisymmetry property of the cpo frel. Various techniques for recursion were then applied: structural induction on lists, Park induction and fixed point induction (which proved admissibility behind the scenes).

# 6 Examples: Well-founded Recursive Functions

This section provides another two examples of well-founded recursive function definitions. These were done previously by van der Voort in [16], which describes another approach to introducing well-founded recursive function definitions inspired by Boyer-Moore [5]. He avoids domain theory and does not treat partial functions. The examples below show that van der Voorts approach (or prototype implementation) is much less efficient than the present. Both examples yield less than half as many 'intermediate theorems generated' and executes 6 times faster using the present package (but of course this is machine dependent and so on). Van der Voort's approach is also discussed briefly in Section 8.

## 6.1 A Fast Exponential

We first consider a fast exponential:

```
#FASTEXP;;
"FASTEXP(m,n) =
 ((n = 0) =>
  1 |
  (EVEN n =>
   let x = FASTEXP(m,n DIV 2) in x * x |
   m * (FASTEXP(m,PRE n))))"
: term
```

When the exponent is even it is divided by two. In this way, the fast exponential roughly speaking halves the number of computation steps compared to the standard primitive recursive definition.

Once the recursive specification has been written we can calculate the proof obligations for the termination proof:

```
#calc_prf_obl FASTEXP;;
["~(n = 0) /\ EVEN n ==> R(m,n DIV 2)(m,n)";
 "~(n = 0) /\ ~EVEN n ==> R(m,PRE n)(m,n)"]
: term list
```

From these we read that the well-founded relation that relates the second components of pairs of natural numbers by the less-than ordering is suitable for this example. The well-founded relation is obtained easily using the inverse image construction:

```
#let wf_fastexp = ISPEC"SND:num#num->num"(MATCH_MP wf_inv wf_less);;
wf_fastexp = |- wf(inv($<,SND))
```

Next, the instantiated proof obligations are calculated:

```
#let obl_tml = get_prf_obl wf_fastexp FASTEXP;;
obl_tml =
["~(n = 0) /\ EVEN n ==> inv($<,SND)(m,n DIV 2)(m,n)";
 "~(n = 0) /\ ~EVEN n ==> inv($<,SND)(m,PRE n)(m,n)"]
: term list
```

Again we shall omit the simple proofs of these obligations, and just assume the desired theorems are bound to the ML variable `obl_thl`. The fast exponential is now defined automatically as follows:

```
#let FASTEXP_DEF =
#  new_wfrec_definition `FASTEXP_DEF` wf_fastexp obl_thl FASTEXP;;
FASTEXP_DEF =
|- !m n.
    FASTEXP(m,n) =
    ((n = 0) =>
     1 |
     (EVEN n =>
      let x = FASTEXP(m,n DIV 2) in x * x |
      m * (FASTEXP(m,PRE n))))
Run time: 4.8s
Intermediate theorems generated: 858
```

## 6.2 Quicksort

A specification of the well-known quicksort algorithm may be written as follows:

```
#QSORT;;
"QSORT l =
 ((l = []) =>
  [] |
  let x = HD l
  and xs = TL l
  in
   APPEND
   (APPEND(QSORT(FILTER(\y. y < x)xs))[x])
   (QSORT(FILTER(\y. ~y < x)xs)))"
: term
```

The proof obligations for termination are then calculated:

```
#calc_prf_obl QSORT;;
["~(l = []) ==> R(FILTER(\y. y < (HD l))(TL l))l";
 "~(l = []) ==> R(FILTER(\y. ~y < (HD l))(TL l))l"]
: term list
```

Here, we see that the length of the list argument is reduced in each recursive call. Hence, a suitable well-founded relation is obtained by the inverse image construction and the less-than ordering:

```
#let wf_qsort =
#  ISPEC"LENGTH:(num)list->num"(MATCH_MP wf_inv wf_less);;
wf_qsort = |- wf(inv($<,LENGTH))
```

Assume the proven proof obligations are bound to the ML variable `obl_thl`. The definition of quick sort is introduced by:

```
#let QSORT_DEF =
#  new_wfrec_definition 'QSORT_DEF' wf_qsort obl_thl QSORT;;
QSORT_DEF =
|- !l.
   QSORT l =
   ((l = []) =>
    [] |
    let x = HD l
    and xs = TL l
    in
     APPEND
     (APPEND(QSORT(FILTER(\y. y < x)xs))[x])
     (QSORT(FILTER(\y. ~y < x)xs)))
Run time: 4.3s
Intermediate theorems generated: 764
```

# 7 A Larger Example: The Unification Algorithm

A larger and non-trivial example, the unification algorithm, is presented in this section. The example is non-trivial for two reasons. The termination properties are very difficult to prove, and further, their proofs rely on the correctness of the algorithm. In other words, we cannot separate the proof of termination from the proof of correctness, as we did in the Ackermann example in Section 1. Still, a temporary well-founded definition can be specified in the conditional style and introduced by `new_wfrec_definition`. Once the correctness of the temporary definition has been established, we can deduce the desired definition. Alternatively, we could have defined a partial recursive function first and proved correctness and termination in the same well-founded induction (as in [3]). This would introduce lifting and function extension in many places and therefore be more inconvenient to work with (partiality is avoided in the approach below).

Unification is the problem of finding a substitution to yield a common instance of two expressions, if this is possible, and if not, to yield a failure. An expression, also called a term, can be a constant, a variable or a combination. Variables are regarded as place holders for which any expression may be substituted. The unification algorithm is a prescription for computing such a substitution, also called a unifier. Furthermore, the desired substitution must be the most general unifier in a certain sense.

I also considered the unification algorithm in details in [3] and partly in [4], exploting previous work by Manna and Waldinger [9] and Paulson [14]. In this paper, we shall therefore skip many details and concentrate on the well-founded definition.

A recursive type of terms can be defined using the type definition package with the following specification

```
term = Const name | Var name | Comb term term,
```

where name can be a type like num or string (or any type).

A central notion is substitution. The operation of applying a substitution to a term can be defined by primitive recursion as follows

```
|- (!c s. (Const c) subst s = Const c) /\
   (!v s. (Var v) subst s = lookup v s) /\
   (!t1 t2 s. (Comb t1 t2) subst s = Comb(t1 subst s)(t2 subst s))
```

where the constant lookup is defined by

```
|- (!v. lookup v[] = Var v) /\
   (!v x s. lookup v(CONS x s) = ((v = FST x) => SND x | lookup v s))
```

Hence, we have chosen to represent a substitution as a list of pairs of variables and terms. A more natural representation could be as a function from variables to terms (see [11]) since this representation avoids dealing with potentially multiple occurences of a variable as in a list representation. For instance, due to the potentially multiple occurences, we must define a special equality for substitution:

```
|- !r s. r == s = (!t. t subst r = t subst s).
```

Here, HOL equality does not work since it looks at lists syntactically. Note that lookup is defined such that it chooses the first occurence of a variable in a substitution (if there is more than one).

We wish the unification algorithm to yield a failure or a successful substitution. It is therefore convenient to introduce a type of attempts to represent the result of a unification:

```
attempt = Failure | Success (name#term)list
```

A first (and sensible) suggestion for a recursive specification of the unification algorithm could then be:

```
"UNIFY(t,u) =
 (is_Const t => unifyC(dest_Const t)u |
  is_Var t => (dest_Var t) assign u |
  is_Const u => Failure |
  is_Var u => (dest_Var u) assign t |
  let a = UNIFY(dest_Comb1 t,dest_Comb1 u)
  in
   (is_Failure a => Failure |
    let s = dest_Success a in
    let a' = UNIFY((dest_Comb2 t) subst s,(dest_Comb2 u) subst s)
    in
     (is_Failure a' => Failure |
      let s' = dest_Success a' in Success(s thens s'))))",
```

which relies on the following (unimportant) definitions

```
|- (!c c'. unifyC c(Const c') = ((c = c') => Success[] | Failure)) /\
   (!c v. unifyC c(Var v) = v assign (Const c)) /\
   (!c t u. unifyC c(Comb t u) = Failure)
```

```
|- (!s. [] thens s = s) /\
   (!x r s.
     (CONS x r) thens s = CONS(FST x,(SND x) subst s)(r thens s))
|- !v t. v assign t = ((Var v) << t => Failure | Success[v,t])
|- (!t c. t << (Const c) = F) /\
   (!t v. t << (Var v) = F) /\
   (!t t1 t2.
     t << (Comb t1 t2) = (t = t1) \/ (t = t2) \/ t << t1 \/ t << t2)
```

where << is an occurs-in relation and thens defines composition of substitutions. However, this recursive specification is not suitable for proving the proof obligations for termination.

In order to explain why, let us first consider the complicated well-founded relation for the termination proof. It can be defined using built-in constructions as follows:

```
|- unifyR =
   inv(lex($PSUBSET,$<<),(\(t,u). ((vars t) UNION (vars u),t)))
|- wf unifyR,
```

where vars gives the finite set of variables of a term:

```
|- (!c. vars(Const c) = {}) /\
   (!v. vars(Var v) = {v}) /\
   (!t1 t2. vars(Comb t1 t2) = (vars t1) UNION (vars t2))
```

Here, we use the library of finite sets [13]. The general inverse image theorem can be used to obtain that both PSUBSET and << are well-founded relations by using respectively the cardinality function on finite sets and a size function on terms. In this way, a proof of the well-foundedness of the two relations is reduced to the well-foundedness of the less-than ordering on natural numbers.

Given this well-founded relation, we can calculate the proof obligations for the above specification (using an ML function get_prf_obl):

```
["~is_Const t /\ ~is_Var t /\ ~is_Const u /\ ~is_Var u ==>
  unifyR(dest_Comb1 t,dest_Comb1 u)(t,u)";
 "~is_Const t /\ ~is_Var t /\
  ~is_Const u /\ ~is_Var u /\ ~is_Failure a ==>
  unifyR
  ((dest_Comb2 t) subst (dest_Success a),
   (dest_Comb2 u) subst (dest_Success a))
  (t,u)"]
```

The problem of the specification is that we cannot prove the second obligation, obtained from the second recursive call, without knowing that the variable a, which corresponds to the result of unifying the first components of two combinations, denotes a best unifier in a certain sense, i.e. without knowing that the recursive specification is correct. We cannot separate the termination proof from the correctness proof in this example.

Fortunately, there is a way of cheating. We can add a test in the recursive specification temporarily to ensure that the result of the unification is correct, otherwise let the algorithm yield an arbitrary value:

```
#unify_tm;;
"UNIFY(t,u) =
 (is_Const t => unifyC(dest_Const t)u |
  is_Var t => (dest_Var t) assign u |
  is_Const u => Failure |
  is_Var u => (dest_Var u) assign t |
  let a = UNIFY(dest_Combl t,dest_Combl u)
  in
   (is_Failure a => Failure |
    let s = dest_Success a
    in
     ((~best_unifier(s,dest_Combl t,dest_Combl u)) => ARB |
      let a' = UNIFY((dest_Comb2 t) subst s,(dest_Comb2 u) subst s)
      in
       (is_Failure a' => Failure |
        let s' = dest_Success a' in Success(s thens s')))))"
: term
```

The notion of best unifier is defined as follows, though the reader does not have to understand such details:

```
|- !s t u. unifier(s,t,u) = (t subst s = u subst s)
|- !s1 s2. more_gen(s1,s2) = (?r. s2 == (s1 thens r))
|- !s t u.
    most_gen_unifier(s,t,u) =
    unifier(s,t,u) /\ (!r. unifier(r,t,u) ==> more_gen(s,r))
|- !s t u.
    best_unifier(s,t,u) = most_gen_unifier(s,t,u) /\ (s thens s) == s
```

The proof obligations now becomes

```
#get_prf_obl wf_unifyR unify_tm;;
["~is_Const t /\ ~is_Var t /\ ~is_Const u /\ ~is_Var u ==>
  unifyR(dest_Combl t,dest_Combl u)(t,u)";
 "~is_Const t /\
  ~is_Var t /\
  ~is_Const u /\
  ~is_Var u /\
  ~is_Failure a /\
  ~~best_unifier(dest_Success a,dest_Combl t,dest_Combl u) ==>
  unifyR
  ((dest_Comb2 t) subst (dest_Success a),
   (dest_Comb2 u) subst (dest_Success a))
  (t,u)"]
: term list
```

which can be proved (though this is not straightforward).

Assume theorems for the proof obligations are in the list `obl_thl`. A well-founded recursive definition of the unification algorithm can now be introduced by:

```
#let UNIFY_DEF_TMP =
#  new_wfrec_definition `UNIFY_DEF_TMP` wf_unifyR obl_thl unify_tm;;
UNIFY_DEF_TMP =
|- !t u.
    UNIFY(t,u) =
    (is_Const t => . . . )
Run time: 11.8s
Garbage collection time: 3.7s
Intermediate theorems generated: 1705
```

Here, parts of the definition have been omitted to save space. This is only a temporary definition and not the desired one. However, once we have proved the correctness of `UNIFY`, stated by the theorem (which was also proved by well-founded induction)

```
|- !t u. best_unify_try(UNIFY(t,u),t,u)
```

where `best_unify_try` is defined by

```
|- !a t u.
    best_unify_try(a,t,u) =
    (a = Failure) /\ (!s. ~unifier(s,t,u)) \/
    (?s. (a = Success s) /\ best_unifier(s,t,u)),
```

we can get rid of the additional unifier test and easily derive the desired definition:

```
#UNIFY_DEF;;
|- !t u.
    UNIFY(t,u) =
    (is_Const t => unifyC(dest_Const t)u |
     is_Var t => (dest_Var t) assign u |
     is_Const u => Failure |
     is_Var u => (dest_Var u) assign t |
     let a = UNIFY(dest_Comb1 t,dest_Comb1 u)
     in
      (is_Failure a => Failure |
       let s = dest_Success a in
       let a' = UNIFY((dest_Comb2 t) subst s,(dest_Comb2 u) subst s)
       in
        (is_Failure a' => Failure |
         let s' = dest_Success a' in Success(s thens s'))))
```

# 8 Conclusions and Related Work

The paper has described a package for automating non-primitive recursive function definitions in HOL. Given a recursive function specification in higher order logic, the package works by constructing a potentially partial domain theory version of the function, which is defined using the fixed point operator. In case this function can be proved to be total, it further constructs a pure HOL recursive function which satisfies the original specification. The proof of termination is performed by well-founded induction automatically but the user of the package must supply a well-founded relation and a list of termination properties for the induction proof. The latter is calculated but not proved by the package. Furthermore, the package provides a number of constructions for well-founded relations, which make it essentially trivial to prove most relations that occur in practice are well-founded. Finally, the package also provides a number of theorems and tools to support the domain theoretic techniques for recursion, such as Park induction and fixed point induction (see Section 3 and Appendix A), which may be useful to reason about partial recursive functions (but not to reason about total well-founded recursive functions).

This work was motivated by previous work on formalizing domain theory in HOL [3, 4]. Some examples were also considered there, though in a much more complicated domain theoretic framework than the present one; further, both partial and total well-founded recursive functions were only treated manually. Since the package exploits domain theory in a very precise and concrete way, we have been able to instantiate the theory considerably. We have restricted ourselves to consider domain theory as a means to extend the support for recursive function definitions in HOL only, not for instance recursive functions on arbitrary domains with e.g. infinite values. We use just two different cpos and one kind of continuous functional for recursive definitions via the fixed point operator. Further, being able to exploit HOL functions directly, we avoid the use of a dependent $\lambda$-abstraction, which was a main reason for complication previously; it was used to make functions determined by their action, in turn this notion was required to prove that the continuous function space was a construction on cpos.

These simplifications and the design and engineering of proper tools have been the main challenges of this work. The goal was to make domain theory as invisible as possible by integrating the theory closely with higher order logic. Indeed, in defining well-founded recursive functions the user never sees any domain theory, and in defining partial recursive functions, the domain theory constructs needed to represent partiality are inserted automatically. No knowledge of domain theory is required to use the package.

There might be recursive definitions which cannot be introduced by the package. A main restriction might be that a recursive call is not allowed to appear in the body of a $\lambda$-abstraction (unless it is part of a let-expression). The problem occurs when the recursive call uses the variable of the abstraction, in other cases the call can be moved out of the body. It might be possible to implement support for abstractions, but this would complicate the domain theory and the implementation of the package considerably. Another associated restriction is that the function being defined must be applied to its argument at all occurences in the right-hand side of a definition[2]. Finally, functions must be specified using conditionals nested with let-expressions. This conditional style is fairly

[2]This requirement ensures that we can treat a recursive call (function plus argument) as a unit. In general, we want to avoid a function type where the right-most type is lifted. This is also the reason for abandoning recursive calls in the body of abstractions.

powerful but there is no evidence that it will always work.

Konrad Slind has developed a similar package for well-founded recursive function definitions (in HOL90), but this does not support other recursive functions. Its implementation is based on the well-founded recursion theorem, which gives a more direct and efficient implementation, since all domain theory is avoided. Further, the well-founded induction is performed once and for all in the proof of the well-founded recursion theorem, whereas in the present package an induction is performed for each definition. However, an advantage of domain theory is that it allows a recursive function to be defined directly without proving whether it is total. Sometimes, recursive functions are partial, or the proof of termination may depend on properties of the function (which is the case with the unification algorithm, see Section 7), in which case it may be advantageous that we can define a version of the function and reason about this before deriving the desired total one.

Mark van der Voort describes another approach to introducing well-founded recursive function definitions in [16], inspired by the one employed in Boyer-Moore [5]. Like Slind, he also avoids domain theory and does not treat partial functions. However, instead of using well-founded relations like Slind and we do, he supplies a natural number measure with each definition. A recursive call must reduce this measure with respect to the less-than ordering. It seems more direct to use well-founded relations rather than a measure which destroys the structure of data. Further, an induction principle must be derived with each recursive definition. It is difficult to compare the two packages directly, since van der Voort's is not available. Some unscientific experiments on defining van der Voort's examples using the present package (see Section 6) shows that his approach (or prototype implementation) is much less efficient. Both examples yield less than half as many 'intermediate theorems generated' and executes 6 times faster using the present package (but of course this is machine dependent).

Tom Melham's package for inductive relation definitions [12, 6] could be used to define many recursive functions as well. This would require a recursive specification to be transformed into a set of inference rules which gives an inductive definition of a relation representation of the function. The recursive function could then be extracted from the inductively defined relation by a uniqueness proof, showing that the relation specifies a (potentially partial) function, and a definedness proof, showing that the relation specifies a total function. It is difficult to say whether or not such an approach would be simpler than the present one, which works with HOL functions directly.

# References


[1] S. Agerholm, *Mechanizing Program Verification in HOL.* M.Sc. thesis, Aarhus University, Computer Science Department, Report IR-111, April 1992. See also: *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*, Davis California, 1991 (IEEE Computer Society Press, 1992.)

[2] S. Agerholm, 'Domain Theory in HOL'. In the *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.), Vancouver, B.C., Canada, 1993, LNCS 780, Springer-Verlag 1994.

[3] S. Agerholm, *A HOL Basis for Reasoning about Functional Programs.* Ph.D. Thesis, BRICS RS-94-44, University of Aarhus, Department of Computer Science, December 1994.

[4] S. Agerholm, 'LCF Examples in HOL'. The Computer Journal, Vol. 38, No. 2, 1995.

[5] R.S. Boyer and J.S. Moore, *A Computational Logic.* Academic Press, 1979.

[6] J. Camilleri and T.F. Melham, 'Reasoning with Inductively Defined Relations in the HOL Theorem Prover'. Technical Report No. 265, University of Cambridge Computer Laboratory, August 1992.

[7] E.W. Dijkstra and C. Scholten, *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[8] M.J.C. Gordon and T.F. Melham (Eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[9] Z. Manna and R. Waldinger, 'Deductive Synthesis of the Unification Algorithm'. *Science of Computer Programming*, Vol. 1, 1981, pp. 5–48.

[10] T.F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic'. In G. Birtwistle and P.A. Subrahmanyam (Eds.), *Current Trends in Hardware Verification and Theorem Proving*, Springer-Verlag, 1989.

[11] T.F. Melham, 'A Mechanized Theory of the $\pi$-calculus in HOL'. Technical Report No. 244, University of Cambridge Computer Laboratory, January 1992. To appear in the Nordic Journal of Computing.

[12] T. Melham, 'A Package for Inductive Relation Definitions in HOL'. In the *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and Its Applications*, Davis California, 1991 (IEEE Computer Society Press, 1992).

[13] T.F. Melham, 'The HOL finite_sets Library' (Draft). University of Cambridge, Computer Laboratory, February 1992. (Appears in [15].)

[14] L.C. Paulson, 'Verifying the Unification Algorithm in LCF'. *Science of Computer Programming*, Vol. 5, 1985, pp. 143–169.

[15] University of Cambridge Computer Laboratory, *The HOL System: Libraries.* Version 2 (for HOL88.2.02), March 1994. (This documentation is distributed with the HOL system.)

[16] M. van der Voort, 'Introducing Well-founded Function Definitions in HOL'. In *Higher Order Logic Theorem Proving and its Applications*, L.J.M. Claesen and M.J.C. Gordon (Eds.), IFIP Transactions A-20, North-Holland, 1993.

[17] G. Winskel, *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

# A Theorems and Tools

In this appendix, we list the ML names of some theorems and tools that may be useful for someone defining partial or well-founded recursive functions with the package. Examples of the use of tools are provided in the previous sections and in Appendix B.

## A.1 Partial Recursive Function Definitions

File: `prec.ml`.

A partial recursive function can be defined using the function

```
new_prec_definition: (string -> conv).
```

It takes two arguments: a string, which is the name under which the returned (derived) definition is saved, and a term, which is a recursive specification in higher order logic. The tool defines a partial function and the returned specification will contain domain theory constructs (see also Section 5).

## A.2 Well-founded Recursive Function Definitions

File: `wfrec.ml`.

Proof obligations for well-founded recursive function definitions are calculated by

```
calc_prf_obl: (term -> term list)
```

which takes a recursive specification and produces the proof obligations containing a variable relation. The function

```
inst_prf_obl: (thm -> term list -> term list)
```

can be used to instantiate the obligations according to a theorem stating a well-founded relation. The function

```
get_prf_obl: (thm -> term -> term list)
```

combines the previous two functions.

A well-founded recursive function definition can be introduced using:

```
new_wfrec_definition: (string -> thm -> thm list -> conv)
```

It takes four arguments: a string which is the name under which the derived definition is saved, a theorem which states a well-founded relation, a theorem list which provides the proven proof obligations, and a term which is the well-founded recursive specification. It returns the derived definition.

## A.3 Well-founded Relations

File: mk_wfrec.ml. Theory: wfrec.th.

### Well-foundedness

```
wf_def:
  |- !R.
      wf R = (!A. ~(A = empty) ==> (?x. A x /\ ~(?y. A y /\ R y x)))
wf_no_inf_decr_chain:
  |- !R. wf R = ~(?X. !n. R(X(SUC n))(X n))
wf_induct:
  |- !R.
      wf R = (!P. (!x. (!y. R y x ==> P y) ==> P x) ==> (!x. P x))
```

### Constructions

```
prod_def:
  |- !R R' b c. prod(R,R')b c = R(FST b)(FST c) /\ R'(SND b)(SND c)
lex_def:
  |- !R R' b c.
      lex(R,R')b c =
      R(FST b)(FST c) \/ (FST b = FST c) /\ R'(SND b)(SND c)
inv_def:
  |- !R f. inv(R,f)x y = R(f x)(f y)

wf_prod:
  |- !R. wf R ==> (!R'. wf R' ==> wf(prod(R,R')))
wf_lex:
  |- !R. wf R ==> (!R'. wf R' ==> wf(lex(R,R')))
wf_inv:
  |- !R. wf R ==> (!f. wf(inv(R,f)))
wf_inv_gen:
  |- !R. wf R ==> (!R' f. (!x y. R' x y ==> R(f x)(f y)) ==> wf R')
wf_less:
  |- wf $<
```

## A.4 Domain Theory

Files: mk_dom.ml and tools.ml. Theory: dom.th.

It can sometimes be useful to employ the following theorems and tools to reason about partial recursive functions.

### Partial orders

```
cpo_refl:
  |- cpo R ==> (!x. R x x)
```

```
cpo_trans:
  |- cpo R ==> (!x y z. R x y /\ R y z ==> R x z)
cpo_antisym:
  |- cpo R ==> (!x y. R x y /\ R y x ==> (x = y))
```

**Partiality** Type specification: `(*)lift = bot | lift *.`

```
lift_Axiom:
  |- !e f. ?! fn. (fn bot = e) /\ (!x. fn(lift x) = f x)

lift_CASES_THM:
  |- !l. (l = bot) \/ (?x. l = lift x)
```

There is a tactic to apply the last theorem:

```
lift_CASES_TAC: (term -> tactic)
```

The term argument is used to instantiate the theorem which is stripped apart and then the equalities are substituted in both the goal and the assumptions, and finally assumed.

### Concrete cpos

```
lrel:
  |- !x y. lrel x y = (x = bot) \/ (x = y)
frel:
  |- !f g. frel f g = (!x. lrel(f x)(g x))

lrel_bot:
  |- !x. lrel bot x
lrel_lift:
  |- !x y. lrel(lift x)(lift y) = (x = y)
frel_bot:
  |- !f. frel(\x. bot)f

cpo_lrel:
  |- cpo lrel
cpo_frel:
  |- cpo frel
```

### Function extension

```
ext:
  |- (!f. ext f bot = bot) /\ (!f x. ext f(lift x) = f x)
ext_bot:
  |- !f. ext f bot = bot
ext_lift:
  |- !f x. ext f(lift x) = f x
```

There is a tactic to reduce occurrences of ext that performs a $\beta$-conversion after using the last theorem:

```
ext_REDUCE_TAC: tactic
```

### Park induction

```
Park_induct:
  |- !f. cont f(frel,frel) ==> (!x. frel(f x)x ==> frel(fix f)x)
```

Continuity of functionals for recursive definitions is saved by `new_prec_definition` and loaded by the tactic:

```
PARK_INDUCT_TAC: tactic
```

Hence, this only produces one subgoal (corresponding to the second antecedent of the previous theorem). The tactic expects a constant instead of a fixed point term. It looks up the fixed point definition of the constant behind the scenes.

### Fixed point induction

```
fp_induct:
  |- !P f.
      cont f(frel,frel) ==>
      admiss P /\ P(\x. bot) /\ (!x. P x ==> P(f x)) ==>
      P(fix f)

admiss:
  |- !P.
      admiss P =
      (!X. chain frel X /\ (!n. P(X n)) ==> P(lub frel X))
```

The fixed point induction tactic loads continuity like the Park induction tactic and also expects a constant rather than a fixed point term:

```
FP_INDUCT_TAC: (string -> int list -> tactic)
```

The string argument is the name of the partial recursive function on which the induction must be performed, and the integer list argument specifies on which occurrences of the constant the induction must be performed.

The tactic produces two or three subgoals dependent on whether or not the built-in admissibility prover is able to prove the `admiss` condition automatically. Presently, it implements the following syntactic notation, assuming we wish to prove the statement `"admiss(\x.P[x])"`:

```
P ::= e = e' | lrel e e' | P1 /\ P2 | !y. P
```

where e and e' are continuous in x, i.e. the continuity prover must be able to prove e.g. `cont(\x.e)(frel,lrel)`. The admissibility prover implements a standard backwards chaining on the theorems (this strategy was used often in [3], see also the description of the continuity prover in Section 4):

```
|- !e e'.
    cont e(frel,lrel) /\ cont e'(frel,lrel) ==>
    admiss(\x. e x = e' x)
|- !e e'.
    cont e(frel,lrel) /\ cont e'(frel,lrel) ==>
    admiss(\x. lrel(e x)(e' x))
|- !P1 P2. admiss P1 /\ admiss P2 ==> admiss(\x. P1 x /\ P2 x)
|- !P. (!y. admiss(P y)) ==> admiss(\x. !y. P y x)
```

# B Examples

## B.1 A Well-founded Recursive Definition

File: mk_ex_ack.ml.

```
new_theory`ex_ack`;;

let ACK =
  "ACK(m,n) = (m = 0) => SUC n
          | (n = 0) => ACK(PRE m,1)
          | ACK(PRE m,ACK(m,PRE n))";;

calc_prf_obl ACK;;

let wf_ack = MATCH_MP (MATCH_MP wf_lex wf_less) wf_less;;

let obl = get_prf_obl wf_ack ACK;;

let not_zero = prove(
  "!n. ~(n=0) ==> ?k. n = SUC k",
  INDUCT_TAC THEN RT[] THEN DISCH_TAC THEN EXISTS_TAC"n:num" THEN REFL_TAC);;

let obl_proof = \tm.
  prove(tm,
    RT[FST;SND;lex_def] THEN STRIP_TAC
    THEN IMP_RES_THEN STRIP_SUBST1_TAC not_zero
    THEN RT[PRE;LESS_SUC_REFL]);;

let obl_thl = map obl_proof obl;;

timer true;;

let ACK_DEF = new_wfrec_definition `ACK_DEF` wf_ack obl_thl ACK;;
```

```
timer false;;
```

## B.2 Partial Recursive Functions and Domain Theoretic Reasoning

File: mk_ex_10_20.ml.

```
% This file gives a solution to exercise 10.20 [Winskel93, section 10.5]. %

loadf`extras`;;

new_theory`ex_10_20`;;

new_constant(`r`,":num#num->(num)list");;
new_constant(`s`,":num#num->num");;

timer true;;

% Would be one second faster without the use of let. %

let f_def = new_prec_definition `f_def`
  "f(l,y) =
   (NULL l => y |
    let x = HD l and l' = TL l in f(APPEND(r(x,y))l',s(x,y)))";;

let g_def = new_prec_definition `g_def`
  "g(l,y) =
   (NULL l => y |
    let x = HD l and l' = TL l in g(l',g(r(x,y),s(x,y))))";;

timer false;;



%
  We wish to prove "f = g", a non-trivial statement! Since frel
  is a cpo it is enough to prove "frel f g" and "frel g f",
  by antisymmetry. We first prove these two statements and then
  assemble them at the end.
%



% For convenience we derive recursion equations for functions. %

let f_fun_EQS = prove_thm
  (`f_fun_EQS`,
   "(!y h. f_fun h([],y) = lift y) /\
    (!x xs y h.
```

```
      f_fun h(CONS x xs,y) = h(APPEND(r(x,y))xs,s(x,y)))",
  PORT[f_fun_def] THEN PBETA_TAC
  THEN CONV_TAC(ONCE_DEPTH_CONV let_CONV)
  THEN RT[NULL;HD;TL]);;

let g_fun_EQS = prove_thm
  ('g_fun_EQS',
   "(!y h. g_fun h([],y) = lift y) /\
    (!x xs y h.
      g_fun h(CONS x xs,y) = ext(\v. h(xs,v))(h(r(x,y),s(x,y))))",
  PORT[g_fun_def] THEN PBETA_TAC
  THEN CONV_TAC(ONCE_DEPTH_CONV let_CONV)
  THEN RT[NULL;HD;TL]);;

let f_EQS = prove_thm
  ('f_EQS',
   "(!y. f([],y) = lift y) /\
    (!x xs y.
      f(CONS x xs,y) = f(APPEND(r(x,y))xs,s(x,y)))",
  REPEAT STRIP_TAC
  THEN CONV_TAC(RATOR_CONV(RAND_CONV(REWR_CONV f_def)))
  THEN CONV_TAC(ONCE_DEPTH_CONV let_CONV)
  THEN RT[NULL;HD;TL]);;

let g_EQS = prove_thm
  ('g_EQS',
   "(!y. g([],y) = lift y) /\
    (!x xs y.
      g(CONS x xs,y) = ext(\v. g(xs,v))(g(r(x,y),s(x,y))))",
  REPEAT STRIP_TAC
  THEN CONV_TAC(RATOR_CONV(RAND_CONV(REWR_CONV g_def)))
  THEN CONV_TAC(ONCE_DEPTH_CONV let_CONV)
  THEN RT[NULL;HD;TL]);;

% We prove the first hint in [Winskel93]: lemma1. %

let lemma1 = prove_thm % Lemma for g_fixp. %
  ('lemma1',
   "!xs l y. g(APPEND l xs,y) = ext(\v. g(xs,v))(g(l,y))",
   GEN_TAC THEN LIST_INDUCT_TAC THEN PRT[APPEND;g_EQS]
   THENL
   [ext_REDUCE_TAC THEN GEN_TAC THEN REFL_TAC
   ;GEN_TAC THEN GEN_TAC THEN ext_CASES_TAC "g(r(h,y),s(h,y))"
    THEN ART[]]);;

%
  Next the hint says that we must deduce |- frel f g. In order
  to prove this we prove g is a fixed point of f_fun. Since f
  is the least prefixed point (Park induction) the result follows.
%
```

```
let g_fixp = prove_thm
  (`g_fixp`,
   "f_fun g = g",
   CONV_TAC(FUN_EQ_CONV THENC RAND_CONV(PALPHA_CONV"(l,y):(num)list#num"))
   THEN PGEN_TAC
   THEN CASES_TAC(ISPEC"l:(num)list"list_CASES)
   THEN RT[g_EQS;lemma1;f_fun_EQS]
);;

let frel_f_g = prove_thm
  (`frel_f_g`,
   "frel f g",
   PARK_INDUCT_TAC THEN RT[g_fixp;MATCH_MP cpo_refl cpo_frel]);;

%
  This completes the first part of the proof. Next we wish to prove
  g is at least as defined as f. A hint says that we should
  prove the following lemma2 first.
%

let lemma2 = prove_thm
  (`lemma2`,
   "!xs l y. lrel(ext(\u. f(xs,u))(f(l,y)))(f(APPEND l xs,y))",
   FP_INDUCT_TAC `f` [2]
   THENL
   [RT[ext_bot;lrel_bot]
   ;REPEAT GEN_TAC THEN CASES_TAC(ISPEC "l:(num)list" list_CASES)
    THEN PRT[f_fun_EQS;APPEND;f_EQS]
    THENL
    [ext_REDUCE_TAC THEN RT[MATCH_MP cpo_refl cpo_lrel]
    ;PORT[APPEND_ASSOC] THEN FIRST_ASSUM MATCH_ACCEPT_TAC]]);;

let frel_g_f = prove_thm
  (`frel_g_f`,
   "frel g f",
   PARK_INDUCT_TAC THEN PORT[frel]
   THEN CONV_TAC(RAND_CONV(PALPHA_CONV"(l,y):(num)list#num"))
   THEN PGEN_TAC THEN CASES_TAC(ISPEC "l:(num)list" list_CASES)
   THEN RT[f_EQS;g_fun_EQS;MATCH_MP cpo_refl cpo_lrel]
   THEN MATCH_ACCEPT_TAC lemma2);;

let ex_10_20 = prove_thm
  (`ex_10_20`,
   "f = g",
   MATCH_MP_TAC(MATCH_MP cpo_antisym cpo_frel)
   THEN CONJ_TAC THENL[ACCEPT_TAC frel_f_g;ACCEPT_TAC frel_g_f]);;
```