BRICS RS-95-36 S. Agerholm: Non-primitive Recursive Function Definitions

BRICS

Basic Research in Computer Science

Non-primitive Recursive Function Definitions

Sten Agerholm

BRICS Report Series RS-95-36

ISSN 0909-0878 July 1995

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Non-primitive Recursive Function Definitions

Sten Agerholm

University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge CB2 3QG, UK

Abstract. This paper presents an approach to the problem of introducing non-primitive recursive function definitions in higher order logic. 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. A collection of algorithms are presented which hide the domain theory from a user. Hence, the derivation of a domain theory specification has been automated completely, and for well-founded recursive function specifications the process of deriving the original specification from the domain theory one has been automated as well, though a user must supply a well-founded relation and prove certain termination properties of the specification. There are constructions for building well-founded relations easily.

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 [9], 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.

This paper presents an approach to this problem via a simple formalization of basic concepts of domain theory in higher order logic. The overall idea is to take a recursive specification, stated in higher order logic and provided by a user, and translate it automatically into a domain theory version where the recursive function is defined by the fixed point operator. This recursive specification differs from the original one since it contains constructs of domain theory to handle the potentially non-terminating recursive calls. This kind of undefinedness is represented by adding a new undefined value to types, a standard construction of domain theory (called lifting).

The original function specification can be derived by eliminating undefinedness, i.e. by proving all recursive calls terminate. One approach is to prove that the argument in each recursive call decreases with respect to some well-founded relation. A binary relation is well-founded if it does not allow any infinite decreasing sequences of values. Hence, by well-founded induction, the recursive calls will terminate eventually, and the specified function is total.

The definition of well-founded recursive functions has been automated such that the details of domain theory and the well-founded induction never appear to the user; she just supplies a recursive specification, a well-founded relation, and a theorem list of termination properties of the specification. In addition, most well-founded relations that occur in practice can be proved very easily using a number of pre-proven constructions.

Our approach is rather pragmatic. While we on one hand want to be able to treat as many recursive specifications as possible, we also want the formalization of domain theory and the automated algorithms to be as simple and efficient as possible. We have therefore identified a number of restrictions of syntactic form which both simplify the formalization and the algorithms. The formalization is designed to allow very smooth and easy transitions between higher order logic and domain theory such that fairly simple tool support can make the domain theory essentially invisible. A similar methodology of having domain theory behind the scenes may be useful for other purposes.

The rest of the paper is organized as follows. The formalization of domain theory is introduced in Section 2. Automation for recursive function definitions in domain theory is presented in Section 3. Section 4 shows how well-founded recursive functions can be obtained from their domain theory versions. Section 6 treats a well-known example and finally, Section 7 contains the conclusions and related work.

Note This paper is a condensed version of [5], which contains more examples and a full list of theorems and tools provided by an implementation in HOL88.

2 Domain Theory

The basic concepts of domain theory (see e.g. [13]) can be formalized in higher order logic as follows. A partial order is a binary relation $R : \alpha \rightarrow \alpha \rightarrow \mathit{bool}$ which is reflexive, transitive and antisymmetric:

$$\begin{aligned}&\texttt{po}\ R \stackrel{\text{def}}{=}\\&(\forall x.\ R\ x\ x) \wedge\\&(\forall xyz.\ R\ x\ y \wedge R\ y\ z \Rightarrow R\ x\ z) \wedge (\forall xy.\ R\ x\ y \wedge R\ y\ x \Rightarrow (x = y))\ .\end{aligned}$$

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

$$\texttt{cpo}\ R \stackrel{\text{def}}{=} \texttt{po}\ R \wedge (\forall X.\ \texttt{chain}\ R\ X \Rightarrow (\exists x.\ \texttt{islub}\ R\ X\ x))\ ,$$

where

$$\begin{aligned}&\texttt{isub}\ R\ X\ x \stackrel{\text{def}}{=} (\forall n.\ R(X\ n)x)\\&\texttt{islub}\ R\ X\ x \stackrel{\text{def}}{=} \texttt{isub}\ R\ X\ x \wedge (\forall y.\ \texttt{isub}\ R\ X\ y \Rightarrow R\ x\ y)\\&\texttt{chain}\ R\ X \stackrel{\text{def}}{=} (\forall n.\ R(X\ n)(X(\texttt{SUC}\ n)))\ .\end{aligned}$$

Note that we do not require that cpos have a least value, but the concrete cpos we use later always have one.

Also essential to domain theory is the notion of continuous function, which is a monotonic function that preserves least upper bounds of chains:

$$\begin{array}{l}\texttt{cont}\ f(R,R') \stackrel{\text{def}}{=} \\ (\forall xy.\ R\ x\ y \Rightarrow R'(f\ x)(f\ y)) \wedge \\ (\forall X.\ \texttt{chain}\ R\ X \Rightarrow (f(\texttt{lub}\ R\ X) = \texttt{lub}\ R'(\lambda n.\ f(X\ n))))\ \ ,\end{array}$$

where `lub` is defined using the choice operator:

$$\texttt{lub}\ R\ X \stackrel{\text{def}}{=} (\epsilon x.\ \texttt{islub}\ R\ X\ x)\ \ .$$

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 we restrict ourselves to consider only one special case of the cpo construction on continuous functions, also called the continuous function space. This is important since the general version of this construction would force us to consider subsets of HOL types; not all HOL functions are continuous on arbitrary cpos. As it appeared in [3], this in turn induces the need for partially specified functions, which are specified on such subsets only. In turn, a new λ-abstraction must be defined to make these partially specified 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, called `lrel` and `frel`, which both have simple definitions:

$$\begin{array}{l}\texttt{lrel}\ x\ y \stackrel{\text{def}}{=} (x = \texttt{bot}) \vee (x = y) \\ \texttt{frel}\ f\ g \stackrel{\text{def}}{=} (\forall a.\ \texttt{lrel}(f\ a)(g\ a))\ \ .\end{array}$$

Here `bot` is a constructor of a new datatype of syntax, written $(\alpha)\mathit{lift}$, which may be specified as follows:

$$v : (\alpha)\mathit{lift}\ \texttt{::=}\ \texttt{bot}\ |\ \texttt{lift}\ (a : \alpha)\ \ .$$

Note that 'lift' is the name of both the type being specified and of one of the two constructors, though we use different fonts. The relation `lrel` ensures that `bot` is a bottom element, i.e. a least value which can be used to represent undefinedness in partial functions, 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 (and an unlifted domain type). It defines a cpo whose bottom element is the everywhere undefined function, i.e. the constant function that sends all values to `bot`.

Recursive functions can be defined as fixed points of continuous functionals. In the present approach, we restrict ourselves to consider only functions that can be defined as fixed points of continuous functionals on `frel`, which is the only instance of the continuous function space that we shall use here. Therefore, the fixed point operator is defined in the following special case

$$\texttt{fix}\ f \stackrel{\text{def}}{=} \texttt{lub}\ \texttt{frel}(\lambda n.\ \texttt{power}\ n\ f)\ ,$$

which is not parameterized over a cpo as it generally would be. The variable f has type $(\alpha \to (\beta)\mathit{lift}) \to (\alpha \to (\beta)\mathit{lift})$ and `power` is defined by

$$\begin{aligned}&\texttt{power}\ 0\ f \stackrel{\text{def}}{=} (\lambda x.\ \texttt{bot})\\&\texttt{power}(\texttt{SUC}\ n) f \stackrel{\text{def}}{=} f(\texttt{power}\ n\ f)\ .\end{aligned}$$

The fixed point theorem is also stated in a special case:

$$\vdash \forall f.\ \texttt{cont}\ f(\texttt{frel}, \texttt{frel}) \Rightarrow (\texttt{fix}\ f = f(\texttt{fix}\ f))\ .$$

This theorem is essential to the automation in Section 3 where it allows a recursive specification to be derived from the fixed point definition of a function.

Note that a recursive function defined as a fixed point has a type of the form $\alpha \to (\beta)\mathit{lift}$, 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 β. 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:

$$\begin{aligned}&\texttt{ext}\ f\ \texttt{bot} \stackrel{\text{def}}{=} \texttt{bot}\\&\texttt{ext}\ f(\texttt{lift}\ x) \stackrel{\text{def}}{=} f\ x\ .\end{aligned}$$

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

3 Automation for Fixed Point Definitions

The purpose of the above formalization is to serve as a basis for defining recursive functions in higher order logic. Given a recursive specification $g\ x = \mathit{rhs}[g, x]$, where g has a type of the form $\alpha_1 \times \ldots \times \alpha_n \to \beta$ and therefore is a paired (uncurried) function, we translate the *rhs* (right-hand side) into a domain theory functional G, which has type $(\alpha_1 \times \ldots \times \alpha_n \to (\beta)\mathit{lift}) \to (\alpha_1 \times \ldots \times \alpha_n \to (\beta)\mathit{lift})$ and is a function on a variant of g with a lifted range type and on x. More precisely, G stands for $\lambda g' x.\ \mathit{rhs}'[g', x]$, where g' has type $\alpha_1 \times \ldots \times \alpha_n \to (\beta)\mathit{lift}$ and rhs' stands for a (lifted) domain theory version of *rhs*. It is constructed by introducing function extension to separate recursive calls (of lifted type) from real HOL terms (of unlifted type). Once we have proved that G is continuous, we can then use the fixed point theorem to obtain $\vdash \texttt{fix}\ G = G(\texttt{fix}\ G)$. From this, a domain theory version of the recursive specification is obtained immediately:

$$\vdash \texttt{fix}\ G\ x = \mathit{rhs}'[\texttt{fix}\ G, x]\ .$$

A domain theory version of g can be defined as **fix** G, but we usually do not do that.

This section describes how we can automate these steps for a large and widely used class of recursive specifications that are written as a list of conditionals:

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

The use of let-terms nested with the conditionals is also supported, for instance:

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

The choice of this kind of a "backbone" of conditionals (possibly mixed with let-terms) is a pragmatic one and not necessary from the viewpoint of domain theory or the automation of this section (though it is exploited). Most recursive programs can be conveniently written using conditional control. In the next section, the conditions are also used as "context" information to prove termination of recursive calls.

Not all HOL terms of the above form are allowed. We also require:

- Recursive occurrences of g in *rhs* must be recursive calls, i.e. g must always be applied to an argument.
- No recursive call is allowed to appear in the body of a λ-abstraction (unless it is part of a let-term[1]).

Both restrictions are introduced to avoid the need for more complex cpos of continuous functions than those supported by **frel**, and to allow recursive occurrences of a function to be separated from real HOL terms easily. Due to the restrictions, we avoid function types with a lifted range type in unexpected places (allowed in arguments of **ext** only). A functional which take a function of this type as an argument is not necessarily continuous.

In the rest of this section we first describes an algorithm for generating the domain theory functional and then an algorithm for proving continuity of the functional; this property is necessary to exploit the fixed point theorem.

3.1 Generating the Functional

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 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 $g : \alpha_1 \times \ldots \times \alpha_n \rightarrow \beta$ is replaced by the primed variable $g' : \alpha_1 \times \ldots \times \alpha_n \rightarrow (\beta)\mathit{lift}$ with a lifted range type. Once *rhs* has been transformed to rhs' the desired functional called G is obtained by abstracting over g' and x.

[1] Recall that the let-term $\mathtt{let}\ a = t\ \mathtt{in}\ t'[a]$ parses to the internal syntax $\mathtt{LET}(\lambda a.\ t'[a])t$.

In order to be able to follow the descriptions below more easily, the reader may wish to try them on the following specification of the Ackermann function (see also Section 6):

$$
\begin{aligned}
&ACK(m,n) = \\
&((m=0) \rightarrow \texttt{SUC}\ n \mid \\
&\ (n=0) \rightarrow ACK(\texttt{PRE}\ m, 1) \mid ACK(\texttt{PRE}\ m, ACK(m, \texttt{PRE}\ n))) \ ,
\end{aligned}
$$

which yields the following translated term:

$$
\begin{aligned}
&((m=0) \rightarrow \texttt{lift}(\texttt{SUC}\ n) \mid \\
&\ (n=0) \rightarrow ACK'(\texttt{PRE}\ m, 1) \mid \\
&\ \texttt{ext}(\lambda a.\ ACK'(\texttt{PRE}\ m, a))(ACK'(m, \texttt{PRE}\ n))) \ .
\end{aligned}
$$

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 $\texttt{ext}(\lambda a.\ (a \rightarrow t_1' \mid t_2'))b'$, where the condition b has been separated from the conditional using function extension.

Let-term: The input has the form $\texttt{let}\ a = t_1\ \texttt{in}\ t_2$, which may use a list of bindings separated by $\texttt{and}$'s. Transform t_2 recursively and use the branch algorithm on t_1. The result has the form $\texttt{ext}(\lambda a.\ t_2')t_1'$. 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 $\texttt{lift}\ t$.

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 $g' : \alpha_1 \times \ldots \times \alpha_n \rightarrow (\beta)\mathit{lift}$. Assuming for illustration that g takes four arguments of which the first and the third ones contain g, then the result has the form

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

Let-term: The input has the form $\texttt{let}\ a = t_1\ \texttt{in}\ t_2$, which may use a list of bindings separated by $\texttt{and}$'s. Transform t_1 and t_2 recursively. The result has the form $\texttt{ext}(\lambda a.\ t_2')t_1'$. 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

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

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

3.2 The Continuity Prover

The most complicated part of the automation is perhaps the continuity prover. Given the functional G constructed above, it must prove the continuity statement: $\mathtt{cont}\ G(\mathtt{frel},\mathtt{frel})$.

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

$$\vdash \forall x.\ \mathtt{cont}(\lambda g'.\ rhs'[g',x])(\mathtt{frel},\mathtt{lrel})$$

and then establish the desired result using the continuity-abstraction theorem, which is stated as follows:

$$\vdash \forall h.\ (\forall x.\ \mathtt{cont}(\lambda f.\ h\ f\ x)(\mathtt{frel},\mathtt{lrel})) \Rightarrow \mathtt{cont}(\lambda f x.\ h\ f\ x)(\mathtt{frel},\mathtt{frel}) \ .$$

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 function call: The term does not contain any free occurrences of g'. The desired continuity theorem (up to α-conversion) is obtained by instantiating

$$\vdash \forall t.\ \mathtt{cont}(\lambda f.\ t)(\mathtt{frel},\mathtt{lrel}) \ .$$

Function call: The term is a function application $g'(t_1,\ldots,t_n)$. Instantiate the following theorem with $(t_1,\ldots,t_n)$ and do an α-conversion:

$$\vdash \forall t.\ \mathtt{cont}(\lambda f.\ f\ t)(\mathtt{frel},\mathtt{lrel}) \ .$$

Conditional: The term is a conditional $(b \rightarrow t_1 \mid t_2)$. Traverse the branches recursively, yielding

$$\vdash \mathtt{cont}(\lambda g'.\ t_1)(\mathtt{frel},\mathtt{lrel})$$
$$\vdash \mathtt{cont}(\lambda g'.\ t_2)(\mathtt{frel},\mathtt{lrel}) \ .$$

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):

$$\begin{aligned}&\vdash \forall f_1 f_2.\\ &\quad \mathtt{cont}\ f_1(\mathtt{frel},\mathtt{lrel}) \Rightarrow \mathtt{cont}\ f_2(\mathtt{frel},\mathtt{lrel}) \Rightarrow\\ &\quad (\forall b.\ \mathtt{cont}(\lambda f.\ (b \rightarrow f_1\ f \mid f_2\ f))(\mathtt{frel},\mathtt{lrel})) \ .\end{aligned}$$

Function extension: The term is an `ext` term of the form $\texttt{ext}(\lambda a.\ t_1)t_2$. The terms t_1 and t_2 are traversed recursively, yielding

$$\vdash \texttt{cont}(\lambda g'.\ t_1)(\texttt{frel}, \texttt{lrel})$$
$$\vdash \texttt{cont}(\lambda g'.\ t_2)(\texttt{frel}, \texttt{lrel})\ .$$

Next, generalizing over a, the first of these and the continuity-abstraction theorem can be used to deduce $\vdash \texttt{cont}(\lambda g' a.\ t_1)(\texttt{frel}, \texttt{frel})$. The desired result is obtained essentially by instantiating the following theorem:

$$\begin{array}{l} \vdash \forall f_1 f_2. \\ \quad \texttt{cont}\ f_1(\texttt{frel}, \texttt{frel}) \Rightarrow \texttt{cont}\ f_2(\texttt{frel}, \texttt{lrel}) \Rightarrow \\ \quad \texttt{cont}(\lambda f.\ \texttt{ext}(f_1\ f)(f_2\ f))(\texttt{frel}, \texttt{lrel})\ . \end{array}$$

This completes the description of the continuity prover.

4 Automation for Well-founded Recursive Definitions

In the previous section, it was shown how we can translate a recursive specification in higher order logic to a version where domain theory constructs appear to separate the potentially non-terminating recursive calls from real HOL terms. We must prove recursive calls terminate to eliminate the domain theory. Both systematic and ad hoc approaches to such termination proofs can be employed. This section shows how to automate one of the more powerful systematic ones, which is based on well-founded induction.

A large class of total recursive functions have well-founded recursive specifications. This means that the argument in each recursive call decreases with respect to some well-founded relation. A relation is well-founded if it does not allow any infinite decreasing sequences of values. Hence, by well-founded induction, recursive calls will terminate eventually, and the specified function is total.

A user must supply a well-founded relation and prove the proof obligations for termination, which are the statements saying that arguments in recursive calls decrease. The well-founded induction and the derivation of the original specification from the domain theory one can be automated as described in this section. Well-founded relations are introduced thoroughly in Section 5 below. In particular, a number of constructions are presented to make the proofs of well-foundedness essentially trivial for most relations that appear in practice.

4.1 Deriving the Specification

Recall that we derived the domain theory specification

$$\vdash \texttt{fix}\ G\ x = rhs'[\texttt{fix}\ G, x]$$

in the previous section, where G is a meta-variable which stands for the functional $\lambda g' x.\ rhs'[g', x]$ and where rhs' is a domain theory version of the right-hand side of the original higher order logic specification $g\ x = rhs[g, x]$.

In order to derive the original specification, we first prove

$$\vdash \exists g.\ \forall x.\ \texttt{lift}(g\ x) = \texttt{fix}\ G\ x$$

by well-founded induction; this proof is described in Section 4.3. Constant specification then yields a constant $\texttt{g}$ that satisfies $\vdash \forall x.\ \texttt{lift}(\texttt{g}\ x) = \texttt{fix}\ G\ x$. Rewriting the right-hand side with the domain theory specification above, we obtain $\vdash \forall x.\ \texttt{lift}(\texttt{g}\ x) = \mathit{rhs}'[\texttt{fix}\ G, x]$. We then prove $\vdash \mathit{rhs}'[\texttt{fix}\ G, x] = \texttt{lift}(\mathit{rhs}[\texttt{g}, x])$, by straight-forward case analyzes on the conditional backbone of rhs, and by exploiting the definition of $\texttt{g}$. Finally, using that the constructor $\texttt{lift}$ is one-one, we arrive at the original and desired specification:

$$\vdash \forall x.\ \texttt{g}\ x = \mathit{rhs}[\texttt{g}, x]\ .$$

4.2 Generating Proof Obligations for Termination

It is easy to generate the proof obligations for termination by traversing the conditional structure of the specification. If a recursive call $g\ y$ appears in the i'th branch and the conditions are labeled $p_1, \ldots, p_i$, then the proof obligation computed for that recursive call is $\neg p_1 \wedge \ldots \wedge \neg p_{i-1} \wedge p_i \Rightarrow R\ y\ x$. The first and last branches are obvious special cases. Nested recursive calls are treated by replacing each nested call by a new variable[2].

4.3 The Well-founded Induction

We wish to prove the statement $\exists g.\ \forall x.\ \texttt{lift}(g\ x) = \texttt{fix}\ G\ x$ 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 (see Section 5):

$$\vdash \forall R.\ \texttt{wf}\ R = (\forall P.\ (\forall x.\ (\forall y.\ R\ y\ x \Rightarrow P\ y) \Rightarrow P\ x) \Rightarrow (\forall x.\ P\ x))\ .$$

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

$$
\begin{array}{l}
\vdash \forall R. \\
\quad \texttt{wf}\ R \Rightarrow \\
\quad (\forall f. \\
\quad\quad \texttt{cont}\ f(\texttt{frel}, \texttt{frel}) \Rightarrow \\
\quad\quad (\forall x. \\
\quad\quad\quad (\forall x'.\ R\ x'\ x \Rightarrow (\exists y.\ \texttt{fix}\ f\ x' = \texttt{lift}\ y)) \Rightarrow \\
\quad\quad\quad (\exists y.\ f(\texttt{fix}\ f)x = \texttt{lift}\ y)) \Rightarrow \\
\quad\quad (\exists g.\ \forall x.\ \texttt{lift}(g\ x) = \texttt{fix}\ f\ x))\ .
\end{array}
$$

This is obtained by a few trivial manipulations. The induction predicate of the previous theorem is instantiated with $\lambda x.\ \exists y.\ \texttt{fix}\ f\ x = \texttt{lift}\ y$. Then the

[2] It may be necessary to restrict the range of the variable in order to prove the proof obligation. This can be done by introducing a condition in the backbone to express some property of the nested recursive call, and hence the variable (see [5]).

consequent of the theorem is skolemized, which means that the existential $\exists y$ is moved outside the $\forall x$ where it becomes $\exists g$; 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 $\exists y.\ f(\texttt{fix}\ f)x = \texttt{lift}\ y$ instead of $\exists y.\ \texttt{fix}\ f\ x = \texttt{lift}\ y$ in the induction proof (i.e. the third antecedent); the fixed point theorem justifies this.

Returning to the (high-level) example specification of g, the first two assumptions of the previous theorem are discharged by the user-supplied theorem $\vdash \texttt{wf}\ gR$ and the continuity prover (see Section 3.2), respectively. The last assumption yields the induction proof:

$$\begin{array}{l}\forall x. \\ \quad (\forall x'.\ gR\ x'\ x \Rightarrow (\exists y.\ \texttt{fix}\ G\ x' = \texttt{lift}\ y)) \Rightarrow \\ \quad (\exists y.\ G(\texttt{fix}\ G)x = \texttt{lift}\ y)\ .\end{array}$$

The proof of this is guided by the syntactic structure of the term $G(\texttt{fix}\ G)x$, which by β-conversion is equal to $rhs'[\texttt{fix}\ G, x]$. A case analysis is done for each condition of the conditional backbone. For each recursive call, there must be a termination theorem in the user-supplied theorem list. 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, which is the same as saying that it is equal to some lifted value. In this way, we become able to reduce away all occurrences of `ext`, due to the way it behaves on lifted values, and arrive at statements of the form $\exists y.\ \texttt{lift}\ t = \texttt{lift}\ y$, which hold trivially.

5 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:

$$\texttt{wf}\ R \stackrel{\text{def}}{=} (\forall A.\ A \neq (\lambda x.\ \texttt{F}) \Rightarrow (\exists x.\ A\ x \wedge \neg(\exists y.\ A\ y \wedge R\ y\ x)))\ .$$

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 [8].

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 easily. The theory provides the following standard constructions on well-founded relations:

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

$$\vdash \texttt{wf}\ \$\!<\ .$$

Product: ML name `wf_prod`:

$$\vdash \forall R.\ \texttt{wf}\ R \Rightarrow (\forall R'.\ \texttt{wf}\ R' \Rightarrow \texttt{wf}(\texttt{prod}(R, R')))\ .$$

Defined by

$$\texttt{prod}(R, R')b\ c \stackrel{\text{def}}{=} R(\texttt{FST}\ b)(\texttt{FST}\ c) \wedge R'(\texttt{SND}\ b)(\texttt{SND}\ c)\ .$$

Lexicographic combination: ML name `wf_lex`:

$$\vdash \forall R.\ \texttt{wf}\ R \Rightarrow (\forall R'.\ \texttt{wf}\ R' \Rightarrow \texttt{wf}(\texttt{lex}(R, R')))\ .$$

Defined by

$$\begin{array}{l}\texttt{lex}(R, R')b\ c \stackrel{\text{def}}{=} \\ R(\texttt{FST}\ b)(\texttt{FST}\ c) \vee (\texttt{FST}\ b = \texttt{FST}\ c) \wedge R'(\texttt{SND}\ b)(\texttt{SND}\ c)\ .\end{array}$$

Inverse image: ML name `wf_inv_gen`:

$$\vdash \forall R.\ \texttt{wf}\ R \Rightarrow (\forall R' f.\ (\forall xy.\ R'\ x\ y \Rightarrow R(f\ x)(f\ y)) \Rightarrow \texttt{wf}\ R')\ .$$

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

$$\vdash \forall R.\ \texttt{wf}\ R \Rightarrow (\forall f.\ \texttt{wf}(\texttt{inv}(R, f)))\ .$$

Defined by

$$\texttt{inv}(R, f)x\ y \stackrel{\text{def}}{=} R(f\ x)(f\ y)\ .$$

Most well-founded relations that appear in practice can be obtained easily by instantiating these constructions.

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

$$\vdash \forall R.\ \texttt{wf}\ R = \neg(\exists X.\ \forall n.\ R(X(\texttt{SUC}\ n))(X\ n))\ ,$$

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:

$$\vdash \forall R.\ \texttt{wf}\ R = (\forall P.\ (\forall x.\ (\forall y.\ R\ y\ x \Rightarrow P\ y) \Rightarrow P\ x) \Rightarrow (\forall x.\ P\ x))\ ,$$

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.

6 Example

The theories and algorithms presented above have been implemented in HOL88. In this section, we illustrate the use of the implemented tools on a famous example of a well-founded recursive function: the (binary) Ackermann function. The theorems and tools of the implementation are described in greater detail in [5]. In particular, this section only illustrates the automation for well-founded recursive function definitions. It does not mention the support for generating the intermediate domain theory specification from which one may then proceed towards the original specification by any method of proof at hand.

Often the Ackermann function is specified by a collection of recursion equations like

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

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. Note that the Ackermann function is not primitive recursive since it cannot be defined using the syntax of primitive recursive specifications.

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 as described in Section 4.2. The variable `k0` is introduced due to the nested recursive call in the last branch. Note that the ML 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 (see Section 5):

```
#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 derived definition tool called `new_wfrec_definition`. This introduces a new constant `ACK` 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)))
```

7 Conclusions and Related Work

This work was motivated by previous work on formalizing domain theory in HOL [3]. The Ackermann example was also considered there, though it was only treated manually and in a much more complicated domain theoretic framework than here. Since the present approach exploits domain theory in a very precise and concrete way, we have been able to instantiate the theory considerably. By tying domain theory up closely with higher order logic, we become able to restrict our use to involve only two different cpos and one kind of continuous functional for recursive definitions via the fixed point operator. We also avoid the need for a dependent λ-abstraction for writing functions, which was a main reason for complication in [3]. These simplifications and the design and engineering of proper tools were the main challenges of this work.

A goal was to make domain theory as invisible as possible. Indeed, in defining well-founded recursive functions the user never sees any domain theory, and in other cases the domain theory constructs are introduced automatically and have a very simple form. Their purpose is to separate the potentially non-terminating recursive calls from real HOL terms, which do not support a notion of undefinedness directly.

There might be recursive definitions that cannot be introduced by the present approach. A main restriction might be that a recursive call is not allowed to appear in the body of a λ-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 outside the body. It might be possible to implement support for abstractions, but this would probably complicate the domain theory and the associated algorithms considerably. Another potential restriction is that all recursive occurrences of a function in the right-hand side of a specification must be applied to arguments, i.e. we do not support unapplied occurrences of functions. Finally, functions must be specified using conditionals, possibly nested with let-expressions. This conditional style is fairly powerful but we have no evidence that it will work for all applications in practice.

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 approach an induction is performed for each definition. However, the advantage of domain theory is that it allows a version of a recursive function to be defined directly without proving whether it is total or not. Sometimes, recursive functions are undefined for some arguments due to non-terminating recursive calls, or the proof of termination may depend on correctness properties of the function; in both cases it is advantageous that we can define a version of the function and reason about this before deriving the desired function.

Mark van der Voort describes another approach to introducing well-founded recursive function definitions in [12], inspired by the one employed in the Boyer-Moore prover [6]. Like Slind, he also avoids domain theory and does not treat more general recursive functions. Though he supports well-founded recursive functions, he supplies a natural number measure with each definition instead of a well-founded relation (following Boyer-Moore). 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, a consequence is that an induction principle must be derived with each recursive definition.

Tom Melham's package for inductive relation definitions [11, 7] could be used to define many recursive functions as well. This would require a recursive specification to be translated into a set of inference rules that 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.

Acknowledgements

The research described here was supported by an HCMP fellowship under the EuroForm network, and partly supported by BRICS[3]. I would like to thank the anonymous referees for many useful suggestions for improvements.

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*, 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.), LNCS 780, Springer-Verlag, 1994.

[3] Basic Research in Computer Science, Centre of the Danish National Research Foundation.

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. S. Agerholm, 'A Package for Non-primitive Recursive Function Definitions in HOL'. Technical Report No. 370, University of Cambridge Computer Laboratory, 1995.
6. R.S. Boyer and J.S. Moore, *A Computational Logic.* Academic Press, 1979.
7. 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.
8. E.W. Dijkstra and C. Scholten, *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.
9. M.J.C. Gordon and T.F. Melham (Eds.), *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
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. 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*, IEEE Computer Society Press, 1992.
12. M. van der Voort, 'Introducing Well-founded Function Definitions in HOL'. In *Higher Order Logic Theorem Proving and its Applications* (HOL workshop proceedings 1992), L.J.M. Claesen and M.J.C. Gordon (Eds.), IFIP Transactions A-20, North-Holland, 1993.
13. G. Winskel, *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

This article was processed using the LaTeX macro package with LLNCS style

Recent Publications in the BRICS Report Series

RS-95-36 Sten Agerholm. *Non-primitive Recursive Function Definitions*. July 1995. 15 pp. To appear in *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, LNCS, 1995.

RS-95-35 Mayer Goldberg. *Constructing Fixed-Point Combinators Using Application Survival*. June 1995. 14 pp.

RS-95-34 Jens Palsberg. *Type Inference with Selftype*. June 1995. 22 pp.

RS-95-33 Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. *Type Inference with Non-structural Subtyping*. June 1995. 22 pp.

RS-95-32 Jens Palsberg. *Efficient Inference of Object Types*. June 1995. 32 pp. To appear in *Information and Computation*. Preliminary version appears in *Ninth Annual IEEE Symposium on Logic in Computer Science*, LICS '94 Proceedings, pages 186–195.

RS-95-31 Jens Palsberg and Peter Ørbæk. *Trust in the λ-calculus*. June 1995. 32 pp. To appear in *Static Analysis: 2nd International Symposium*, SAS '95 Proceedings, 1995.

RS-95-30 Franck van Breugel. *From Branching to Linear Metric Domains (and back)*. June 1995. 30 pp. Abstract appeared in Engberg, Larsen, and Mosses, editors, *6th Nordic Workshop on Programming Theory*, NWPT '6 Proceedings, 1994, pages 444-447.

RS-95-29 Nils Klarlund. *An $n \log n$ Algorithm for Online BDD Refinement*. May 1995. 20 pp.

RS-95-28 Luca Aceto and Jan Friso Groote. *A Complete Equational Axiomatization for MPA with String Iteration*. May 1995. 39 pp.

RS-95-27 David Janin and Igor Walukiewicz. *Automata for the μ-calculus and Related Results*. May 1995. 11 pp. To appear in *Mathematical Foundations of Computer Science: 20th Int. Symposium*, MFCS '95 Proceedings, LNCS, 1995.