Partial Recursive Functions in Higher-Order Logic

Alexander Krauss

Technische Universität München, Institut für Informatik
http://www4.in.tum.de/~krauss

Abstract. Based on inductive definitions, we develop an automated tool for defining partial recursive functions in Higher-Order Logic and providing appropriate reasoning tools for them. Our method expresses termination in a uniform manner and includes a very general form of pattern matching, where patterns can be arbitrary expressions. Termination proofs can be deferred, restricted to subsets of arguments and are interchangeable with other proofs about the function. We show that this approach can also facilitate termination arguments for total functions, in particular for nested recursions. We implemented our tool as a definitional specification mechanism for Isabelle/HOL.

1 Introduction

Advanced specification mechanisms that introduce definitions in a natural way are essential for the practical usability of proof assistants.

In a logic of total functions, notably Higher-Order Logic (HOL), recursive function definitions usually require a termination proof. On the other hand, many interesting algorithms do not always terminate: Some examples are search in an infinite search space, semi-decision procedures or the evaluation of programs.

There are several ways to express partiality in a logic of total functions [14], but defining a function from a set of non-terminating equations is generally difficult, especially if it is not clear when the recursion terminates, or if the termination proof is nontrivial. Thus, modelling non-terminating algorithms as logic functions often requires artificial manual workarounds, which can complicate subsequent reasoning.

In order to improve this situation, we describe a general function definition principle for Isabelle/HOL [15], which is not limited to terminating recursions. From a set of recursive equations, our package defines a partial function (modeled as an underspecified total function), together with a set describing the function's domain. On the domain, the defined function coincides with the specification. The provided tools allow to reason about such a partial function as conveniently as it is common for total functions. Our package has the following key properties:

Definitional. Every definition is reduced to a simpler form that can be processed by existing means. The original specification is then *derived* from that definition by an automated proof procedure. Since all reasoning is performed within the theorem prover, this approach offers a maximum of safety without having to rely on an external soundess proof.

Generalized Pattern matching. Functions may be specified by pattern matching. Patterns are not restricted to datatype constructors, may contain guards and overlap, but must be proved to be compatible.

Reasoning Principles. For each recursive function f, an induction principle (which we call f*-induction*) is proved, which corresponds to the recursive structure of the definition of f, and is the main tool for reasoning about it.

Deferred Termination. Termination proofs are strictly separated from function definitions. At definition time, no other input than the specification is needed. Properties of the function can be proved before its termination is established. This particularly simplifies the treatment of nested recursions.

1.1 Motivation

Partiality As an example for partiality, we define an interpreter for a minimalistic imperative language. Such an interpreter must be partial, since the interpreted program might loop and this non-termination cannot be detected. However we would expect to be able to prove termination for certain classes of programs, for example the class of all programs without while loops.

The language is straightforward and we present it directly in Isabelle/HOL notation. For simplicitly, a shallow embedding is used for expressions, instead of modeling their syntax. The notation $f(x := y)$ denotes function update, and *iter* denotes function exponentiation.

types *var* = *nat*
val = *nat*
env = *var* ⇒ *val*
exp = *env* ⇒ *val*

consts
exec :: *com* ⇒ *env* ⇒ *env*

datatype *com* =
ASSIGN *var exp*
| SEQ *com com*
| IF *exp com com*
| WHILE *exp com*
| FOR *exp com*

function
exec (ASSIGN v *exp*) e = e(v:=*exp* e)
exec (SEQ c_1 c_2) e = exec c_2 (exec c_1 e)
exec (IF *exp* c_1 c_2) e = if *exp* $e \neq 0$ then exec c_1 e else exec c_2 e
exec (FOR *exp* c) e = iter (*exp* e) (exec c) e
exec (WHILE *exp* c) e = if *exp* $e \neq 0$
then exec (WHILE *exp* c) (exec c e) else e

Current tools in Isabelle/HOL cannot handle the definition of *exec*. The attempt would lead to an unsolvable termination proof obligation.

As a workaround, we can always extend a partial function to a total one: If we know that the function terminates under certain conditions, this check can be added to the function body, returning a dummy value if the check fails:

$$\mathrm{f}\ x = \text{if } \langle \text{guard} \rangle\ x \text{ then } \langle \text{body}_\mathrm{f} \rangle\ x \text{ else dummy}$$

Then f can be defined as a total function. But this is unsatisfactory as a general method for two reasons: First, the termination guard must be known at definition time. If it turns out later that this condition was too restrictive, one must change the definition of the function. Second, the workaround changes the body of the function. Since the

termination guard is alien to the functional specification, this is inelegant and may cause difficulties when executable code is to be extracted from the definition at a later stage. Restricting our interpreter to programs without while loops is certainly inadequate. We would have to find a condition that covers all possible terminating programs, which is not obvious.

Our package allows to define *exec* as a partial function and later prove its termination on the sets we need.

Generalized Pattern matching Function definitions by pattern matching are convenient in functional programming, and the same is true for logic.

In functional languages, patterns consist only of variables and datatype constructors, so that they can be compiled into efficient tests. Some languages also allow simple invertible arithmetic expressions like $n + 2$.

Such restrictions seem inappropriate for an extensible logical framework like Isabelle/HOL. We can for example define a type for α-equated lambda terms, using a package of Urban and Berghofer (see [20]), and the need arises to define functions on such terms as well. For example, the following equations describe capture-avoiding substitution $[. ::= .]$:

$$\begin{aligned} (\text{Var } a)[b::=u] &= \textsf{if } a{=}b \textsf{ then } u \textsf{ else } \text{Var } a \\ (\text{App } t_1\ t_2)[b::=u] &= \text{App } (t_1[b::=u])\ (t_2[b::=u]) \\ a\sharp(b,u) \Longrightarrow (\text{Lam } [a].t)[b::=u] &= \text{Lam } [a].(t[b::=u]) \end{aligned}$$

This is obviously a form of pattern matching, but different from patterns used in functional programming: First, due to α-equivalence, the constructor in the lambda case is not injective (eg. Lam $[a]$.(Var a) = Lam$[b]$.(Var b)), and second, the lambda case is conditional: It requires that a is fresh in b and u.

To be able to support such constructions, we adopt a more general notion of pattern matching, which is purely logical. Our patterns may essentially be arbitrary expressions, but we require two properties to ensure that the patterns really form a function definition:

1. Every possible argument is matched by at least one pattern. (*Completeness*)
2. If more than one pattern matches an argument, the associated right hand sides are equal. (*Compatibility*)

Pattern completeness will be needed to generate an induction rule and does not stand in the way of partial definitions, since missing patterns can easily be added by introducing trivial equations $\mathrm{f}\ p = \mathrm{f}\ p$. Compatibility ensurtes that the whole specification is consistent. Note that in contrast to functional programming, the equations do not have any associated order. Every equation will become a simplification rule on its own.

These two conditions are very natural (though undecidable), and sometimes used to justify function definitions in pen-and-paper theories. As expected, a definition with our package requires a proof of these conditions.

1.2 An overview of our method

Starting from the specification of a function, our package inductively defines its graph G and its smallest recursion relation R, which captures the recursive structure of the definition. Using the definite description operator, the graph is turned into a total function f, which models the specified partial function. Its domain $\mathrm{dom_f}$ is defined as the accessible part of R.

Then, pattern completeness and compatibility must be proved by the user or automated tools. Our package builds on these facts to prove that G actually describes a function on $\mathrm{dom_f}$. Then it automatically derives the original recursion equations and an induction rule. The rules are constrained by preconditions of the form $t \in \mathrm{dom_f}$, that is, they describe the function's behaviour *on its domain*. Despite these constraints, they allow convenient reasoning about the function, before its termination is established. To support natural termination proofs, the package provides introduction rules for $\mathrm{dom_f}$ and a special termination rule.

The rest of this paper is organized as follows: In §2 we introduce some logical concepts required by our package. In §3, we describe the automated definition process. In §4, we discuss how our package supports termination proofs. In §5, we show how our particular method can improve the treatment of nested recursive definitions. In §6, we briefly describe case studies, and §7 discusses related work.

2 Logical Preliminaries

We work in classical Higher-Order Logic. Derivations are expressed in the natural deduction framework Isabelle/Pure, using universal quantification $\bigwedge$ and implication $\Longrightarrow$.

2.1 Recursion Relations and Termination Conditions

The extraction of termination conditions was first introduced by Boyer and More [5] and is an important component in every implementation of general recursion in theorem provers, definitional or not.

Given a function definition, we call a relation R on the function's argument type a *recursion relation*, if the function argument decreases wrt. R in every recursive call. Consider the following function definition:

$$\begin{aligned}
\mathrm{gcd}(x, 0) &= x \\
\mathrm{gcd}(0, y) &= y \\
\mathrm{gcd}(x+1, y+1) &= \mathsf{if}\ x < y\ \mathsf{then}\ \mathrm{gcd}(x+1, y-x) \\
&\qquad\qquad \mathsf{else}\ \mathrm{gcd}(x-y, y+1)
\end{aligned}$$

Then the relation $\{((x_1, y_1), (x_2, y_2)) \mid x_1 + y_1 < x_2 + y_2\}$ is a recursion relation. Another one is the lexicographic ordering, and a third one is the universal relation where everything is smaller than everything else. Note that we do not require the relation to be wellfounded.

From a definition, we can automatically extract a set of *termination conditions*, which a recursion relation must satisfy. In our example, these conditions are:

$$x < y \Longrightarrow (x+1, y-x) <_R (x+1, y+1)$$
$$\neg x < y \Longrightarrow (x-y, y+1) <_R (x+1, y+1)$$

The details of the extraction in HOL are described in Slind's thesis [18]. Abstractly, the extraction is a syntax directed search for recursive calls in the function body. For each recursive call, a condition is generated, stating that the argument in the recursive call is smaller than the original argument.

Note that this extraction must be context-aware and take the positions of recursive calls into account. In the above example, these occur inside an if-expression, which is reflected in additional premises. In general, a context consists of bound variables and premises, and is denoted by $\mathbf{\Gamma}$. Termination conditions have the general form[1]:

$$\mathbf{\Gamma}_k \Longrightarrow \mathbf{r}_k <_R \mathbf{lhs}$$

This general notion of context can express Higher-Order recursion: Consider a datatype for trees and a function which maps its argument f over all leaves:

$$\begin{array}{ll} \text{treemap } f \text{ (Leaf } x) & = \text{Leaf } (f\ x) \\ \text{treemap } f \text{ (Branch } ts) & = \text{Branch (map (treemap } f) \ ts) \end{array}$$

The termination condition reflects the fact that recursive calls occur only on elements of ts:

$$\bigwedge x.\ x \in \text{set}\, ts \Longrightarrow x <_R \text{Branch}\ ts$$

Since the extraction process requires some knowledge about the contexts resulting for terms occuring at certain positions, the algorithm is parametrized with a set of *congruence rules*, which express this knowledge. The shape of congruence rules is not relevant for us, and can be found in Slind's thesis [18].

2.2 Accessible Part

We adopt the notion of the *accessible part* (or wellfounded part) of a relation, denoted by $\text{acc}\, R$. The accessible part consists of just the elements which do not occur in an infinitely descending R-chain. It is inductively defined by the following rule:

$$\frac{\forall y <_R x.\, y \in \text{acc}\, R}{x \in \text{acc}\, R}\ (\textsc{Acc-Intro})$$

If R is wellfounded, then $\text{acc}\, R$ is just the universal set. In other cases, it might still contain interesting subsets. Using this notion instead of wellfoundedness is crucial to be able to support non-termination.

[1] To distinguish them from logical symbols, meta-variables like contexts are printed in bold. Also note that $\mathbf{\Gamma} \Longrightarrow \phi$ is slight abuse of notation: If $\mathbf{\Gamma}$ binds variables, they are quantified over the whole implication: $\bigwedge v_1 \ldots v_n.\, \mathbf{\Gamma} \Longrightarrow \phi$

The accessible part comes with a very general induction principle, which we call acc-induction:

$$\frac{\forall x \in \operatorname{acc} R.\, (\forall y <_R x.\, P\, y) \longrightarrow P\, x}{\forall x \in \operatorname{acc} R.\, P\, x} \text{ (ACC-INDUCT)}$$

This rule works like wellfounded induction, but with arbitrary relations. Consequently, the inductive result is only proved for the elements of $\operatorname{acc} R$.

2.3 Inductive Definitions

To define inductive relations, our tool uses an existing package [16] from Isabelle/HOL, which introduces inductive sets as least fixed-points by means of the Knaster-Tarski fixed-point theorem. Given some introduction rules for a set S, the package defines the set and returns the introduction rules as theorems. An elimination rule, expressing that S is in fact the smallest such set, is also generated automatically. The package also provides an induction rule, but we will not use it, since we rely on acc-induction.

3 The process of definition

We start with the recursive equations given as input by the user and which he would like to get back as simplification rules in the end:

$$\begin{aligned} \mathbf{C}_1 &\Longrightarrow \mathrm{f}\ \mathbf{lhs}_1 = \mathbf{rhs}_1 \\ &\vdots \\ \mathbf{C}_n &\Longrightarrow \mathrm{f}\ \mathbf{lhs}_n = \mathbf{rhs}_n \end{aligned}$$

Each equation may contain free variables, which we collectively denote by $\mathbf{v}_i^*$.

3.1 Defining the graph and the recursion relation

We transform the equations into an inductive definition of a relation G, representing the graph of the function. The first step is the termination condition extraction from §2.1, which extracts from each equation a list of m_i recursive calls and their contexts: $(\mathbf{\Gamma}_{i1}, \mathbf{r}_{i1}), \ldots, (\mathbf{\Gamma}_{im_i}, \mathbf{r}_{im_i})$. From the i-th equation we now build the following introduction rule for G:

$$\frac{\mathbf{C}_i \quad (\mathbf{\Gamma}_{i1}^{[h/\mathrm{f}]} \Longrightarrow (\mathbf{r}_{i1}^{[h/\mathrm{f}]}, h(\mathbf{r}_{i1}^{[h/\mathrm{f}]})) \in G) \ldots \quad (\mathbf{\Gamma}_{im_i}^{[h/\mathrm{f}]} \Longrightarrow (\mathbf{r}_{im_i}^{[h/\mathrm{f}]}, h(\mathbf{r}_{im_i}^{[h/\mathrm{f}]})) \in G)}{(\mathbf{lhs}_i, \mathbf{rhs}_i^{[h/\mathrm{f}]}) \in G} \text{ (GINTRO}_i\text{)}$$

In $\mathbf{\Gamma}^{[h/\mathrm{f}]}$ etc, the function variable h is substituted for the function symbol f.

Compared to a naive relational description, which would invent a new variable for the result of each recursive call[2], we use a single function variable h, which is constrained to the graph on all recursive calls.

[2] Inventing separate variables for the recursive calls would require additional bookkeeping and lead to problems with Higher-Order recursion.

The introduction rules form an inductive definition of the relation G, and we can introduce G, using the package for inductive definitions. Next, we can already define the function f, using HOL's definite description operator:

$$\mathrm{f} \stackrel{\mathrm{def}}{=} \lambda x.\, \mathtt{THE}\, y.\, (x, y) \in G \qquad \text{(F-DEF)}$$

The basic reasoning tool for our new function is acc-induction on a recursion relation. While other packages require the user to supply such a relation, this is not necessary, since it can be defined automatically.

The essential observation is that the termination conditions extracted from a definition (see §2.1) always have the form of introduction rules for R:

$$\begin{array}{l} \mathbf{C}_i \Longrightarrow \mathbf{\Gamma}_{i,1} \quad \Longrightarrow \mathbf{r}_{i,1} <_R \mathbf{lhs}_i \\ \qquad\qquad \vdots \\ \mathbf{C}_i \Longrightarrow \mathbf{\Gamma}_{i,m_i} \Longrightarrow \mathbf{r}_{i,m_i} <_R \mathbf{lhs}_i \end{array} \qquad (\text{RINTROS}_i)$$

Collecting all the rules, one for each recursive call, we can regard them as a specification of the recursion relation, and thus define a relation R from them. For convenience, we use the package for inductive definitions again, but in fact the specification is not truly inductive: R appears only on the right hand sides.

Note that from such a definition we get the *smallest* recursion relation for our function and that smaller relations have bigger accessible parts. Since the accessible part of this relation will be the domain of our function, the smallest recursion relation is the best we can get.

3.2 Completeness and Compatibility

The following proof requires *completeness* and *compatibility* of the patterns, which were already mentioned in §1.1.

Completeness ensures that every possible input value is matched by one of the patterns. It has the following form[3]:

$$\frac{\bigwedge \mathbf{v}_1^*.\, \mathbf{C}_1 \Longrightarrow x = \mathbf{lhs}_1 \Longrightarrow P \quad \ldots \quad \bigwedge \mathbf{v}_n^*.\, \mathbf{C}_n \Longrightarrow x = \mathbf{lhs}_n \Longrightarrow P}{P} \qquad (\text{COMPLETE})$$

Compatibility states that the set of equations are not contradictory, in the sense that they assign different values to the same argument. This is expressed by special elimination rules for G, one for each equation:

$$\frac{\begin{array}{ll} & \bigwedge h.\ (\mathbf{\Gamma}_{i1} \Longrightarrow (\mathbf{r}_{i1}, h(\mathbf{r}_{i1})) \in G) \Longrightarrow \ldots \\ & \qquad \Longrightarrow (\mathbf{\Gamma}_{im_i} \Longrightarrow (\mathbf{r}_{im_i}, h(\mathbf{r}_{im_i})) \in G) \\ \mathbf{C}_i \quad (\mathbf{lhs}_i, y) \in G & \qquad \Longrightarrow y = \mathbf{rhs}_i[h/\mathrm{f}] \Longrightarrow P \end{array}}{P} \qquad (\text{COMPAT}_i)$$

[3] Completeness and compatibility are existential statements, which are expressed in Isabelle/Pure by elimination rules.

In many cases, compatibility can easily be proved using the definition of G. In particular, disjoint patterns, where each argument is matched by at most one pattern, are trivially compatible.

Our package generates proof obligations for completeness and compatibility which must be solved to complete the definition. For the case of disjoint constructor patterns, we provide an automated proof tactic which solves these goals. Automation of completeness proofs works as described by Slind [18], and compatibility uses the definition of G and simplification.

3.3 The relation is a function

We are now prepared to prove that the relation G describes a function on $\operatorname{acc} R$:

$$\forall x \in \operatorname{acc} R.\ \exists! y.(x, y) \in G$$

The proof of this property is performed automatically by our package, when a definition is made, using only primitive inference steps and rewriting. The following informal but detailed proof sketch illustrates the structure of the derivation:

Proof. We use acc-induction on the recursion relation R. For the induction step, fix an $x \in \operatorname{acc} R$. As induction hypothesis, we can assume the property for all $z <_R x$. Splitting into existence and uniqueness, and using the fact that the unique value is denoted by f, this yields:

$$\bigwedge z.\ \ z <_R x \Longrightarrow (z, \mathrm{f}\ z) \in G \qquad \text{(IH-EXIST)}$$
$$\bigwedge y\, z.\ \ z <_R x \Longrightarrow (z, y) \in G \Longrightarrow y = \mathrm{f}\ z \qquad \text{(IH-UNIQUE)}$$

To complete the induction step, we have to prove $\exists! y.\ (x, y) \in G$. We distinguish cases and look at each of the defining equations in turn, proving existence and uniqueness separately: For the i-th equation, assume $\mathbf{C}_i$ and $x = \mathbf{lhs}_i$.

Existence: From RINTROS$_i$ and IH-EXIST get $\mathbf{\Gamma}_{ij} \Longrightarrow (\mathbf{r}_{ij}, \mathrm{f}\ \mathbf{r}_{ij}) \in G$ for all $j \leq m_i$. Applying GINTRO$_i$ yields $(\mathbf{lhs}_i, \mathbf{rhs}_i) \in G$.

Uniqueness: Assume $(\mathbf{lhs}_i, y) \in G$ for some y. Instantiate COMPAT$_i$ with $P := (y = \mathbf{rhs}_i)$ and apply the assumption. It remains to prove the second premise of COMPAT$_i$. For this, assume $\mathbf{\Gamma}_{ij} \Longrightarrow (\mathbf{r}_{ij}, h\ \mathbf{r}_{ij}) \in G$ for all $j \leq m_i$ and $y = \mathbf{rhs}_i[h/\mathrm{f}]$ for some function h. From RINTROS$_i$ and IH-UNIQUE, get $\mathbf{\Gamma}_{ij} \Longrightarrow h\ \mathbf{r}_{ij} = \mathrm{f}\ \mathbf{r}_{ij}$. Use these equations as contextual rewrite rules to show $\mathbf{rhs}_i[h/\mathrm{f}] = \mathbf{rhs}_i$. Thus $y = \mathbf{rhs}_i$.

Combining existence and uniqueness, we have for each equation $\mathbf{C}_i \Longrightarrow x = \mathbf{lhs}_i \Longrightarrow \exists! y.(x, y) \in G$. Using COMPLETE, these results can be combined into one, which completes the induction step. □

3.4 Deriving partial simplification and induction rules

Having established that function values exist and are unique on $\operatorname{acc} R$, we introduce the abbreviation $\operatorname{dom}_{\mathrm{f}} \equiv \operatorname{acc} R$ and prove the original recursion equations and an induction rule. The equations are guarded by termination assumptions:

$$\begin{aligned} \mathbf{lhs}_1 \in \operatorname{dom}_{\mathrm{f}} &\Longrightarrow \mathbf{C}_1 \Longrightarrow \mathrm{f}\ \mathbf{lhs}_1 = \mathbf{rhs}_1 \\ &\dots \\ \mathbf{lhs}_n \in \operatorname{dom}_{\mathrm{f}} &\Longrightarrow \mathbf{C}_n \Longrightarrow \mathrm{f}\ \mathbf{lhs}_n = \mathbf{rhs}_n \end{aligned}$$

Deriving the recursion equations is simple: From uniqueness we know that $(x, y) \in G$ implies $\mathrm{f}\ x = y$, and we have already proved the required relations in the existence part of the previous proof. We can reuse them after lifting them out of the induction context, which is technical but straightforward.

The partial induction rule follows the structure of the recursion: In each case, the property may be assumed on the arguments of the recursive calls, but the inductive result is restricted to $\operatorname{dom}_{\mathrm{f}}$. The rule is a simple consequence of acc-induction, the definition of R and pattern completeness:

$$\frac{\begin{array}{l} \bigwedge \mathbf{v}_1^*.\ \mathbf{lhs}_1 \in \operatorname{dom}_{\mathrm{f}} \Longrightarrow \mathbf{C}_1 \\ \qquad \Longrightarrow (\mathbf{\Gamma}_{11} \Longrightarrow P\,\mathbf{r}_{11}) \Longrightarrow \dots \Longrightarrow (\mathbf{\Gamma}_{1m_1} \Longrightarrow P\,\mathbf{r}_{1m_1}) \\ \qquad \Longrightarrow P\,\mathbf{lhs}_1 \\ \vdots \\ \bigwedge \mathbf{v}_n^*.\ \mathbf{lhs}_n \in \operatorname{dom}_{\mathrm{f}} \Longrightarrow \mathbf{C}_n \\ \qquad \Longrightarrow (\mathbf{\Gamma}_{n1} \Longrightarrow P\,\mathbf{r}_{n1}) \Longrightarrow \dots \Longrightarrow (\mathbf{\Gamma}_{nm_n} \Longrightarrow P\,\mathbf{r}_{nm3_n}) \\ \qquad \Longrightarrow P\,\mathbf{lhs}_n \end{array}}{a \in \operatorname{dom}_{\mathrm{f}} \Longrightarrow P\,a}\ (\text{F-PINDUCT})$$

With the proof of the partial simplification and induction rules, the actual definition process is finished: The rules provide adequate means for reasoning about the function. In particular, we can now establish the properties we might need for a termination proof. We will see in §5 that this is extremely useful when dealing with nested recursion.

3.5 A simple example

Let us define the Fibonacci function:

$$\begin{aligned} \mathrm{fib}\ 0 &= 1 \\ \mathrm{fib}\ 1 &= 1 \\ \mathrm{fib}\ (n+2) &= \mathrm{fib}\ n + \mathrm{fib}\ (n+1) \end{aligned}$$

The graph of the function, G_{fib}, is defined by

$$(0,1) \in G_{\mathrm{fib}} \qquad (1,1) \in G_{\mathrm{fib}} \qquad \frac{(n, h(n)) \in G_{\mathrm{fib}} \quad (n+1, h(n+1)) \in G_{\mathrm{fib}}}{(n+2, h(n) + h(n+1)) \in G_{\mathrm{fib}}}$$

The termination conditions (and definition of R_{fib}) are:

$$n <_{R_{\mathrm{fib}}} n+2 \qquad\qquad n+1 <_{R_{\mathrm{fib}}} n+2$$

The proof obligation for completeness is a simple property of natural numbers, and compatibility is trivial, since the patterns are disjoint. We get the following simplification and induction rules:

$$\begin{aligned} 0 \in \mathrm{dom}_{\mathrm{fib}} &\Longrightarrow \mathrm{fib}\,0 &&= 1 \\ 1 \in \mathrm{dom}_{\mathrm{fib}} &\Longrightarrow \mathrm{fib}\,1 &&= 1 \\ n+2 \in \mathrm{dom}_{\mathrm{fib}} &\Longrightarrow \mathrm{fib}\,(n+2) &&= \mathrm{fib}\,n + \mathrm{fib}\,(n+1) \end{aligned}$$

$$\frac{\begin{array}{l} 0 \in \mathrm{dom}_{\mathrm{fib}} \Longrightarrow P\,0 \\ 1 \in \mathrm{dom}_{\mathrm{fib}} \Longrightarrow P\,1 \\ \bigwedge n.\, n+2 \in \mathrm{dom}_{\mathrm{fib}} \Longrightarrow P\,n \Longrightarrow P\,(n+1) \Longrightarrow P\,(n+2) \end{array}}{a \in \mathrm{dom}_{\mathrm{fib}} \Longrightarrow P\,a}$$

4 Termination proofs

All results obtained from the partial simplification and induction rules will contain termination assumptions of the form $t \in \mathrm{dom}_{\mathrm{f}}$. Thus, it is desirable to know more about $\mathrm{dom}_{\mathrm{f}}$, which is the objective of a termination proof. Often, our goal will be to show that a function is total and any value is element of $\mathrm{dom}_{\mathrm{f}}$. For partial functions, we will usually be interested in a certain subset.

While the definition process was fully automated and worked for any function definition[4], we cannot expect such automation for termination proofs. But there are powerful methods for proving termination [8, 12, 21], and we plan to integrate them in subsequent work. Here we concentrate on the fundamental tools for conducting interactive termination proofs in a natural and simple way. This is particularly important for the difficult cases, where automated methods fail

4.1 Domain introduction rules

To show that some value belongs to the domain of a function, one can use the rule ACC-INTRO (cf. §2.2), and the definition of R. From this, general introduction rules for the domain can be derived. For fib, these rules are:

$$0 \in \mathrm{dom}_{\mathrm{fib}} \qquad 1 \in \mathrm{dom}_{\mathrm{fib}} \qquad \frac{n \in \mathrm{dom}_{\mathrm{fib}} \quad (n+1) \in \mathrm{dom}_{\mathrm{fib}}}{(n+2) \in \mathrm{dom}_{\mathrm{fib}}}$$

Domain introduction rules are a natural description of the termination behaviour of the function, and termination proofs with these rules are straightforward: We can show that

[4] It is instructive to see that for the non-terminating definition $f\ x = f\ x + 1$, our algorithm defines the Graph G as the empty set and the recursion relation R as the diagonal, where each element is smaller than itself. Then the accessible part of R is the empty set, which means that the derived partial simplification and induction rules are of little use: They are instances of *ex falso quodlibet*.

fib is total, i.e. that $\forall x.\, x \in \mathrm{dom}_{\mathrm{fib}}$, by simple mathematical induction on naturals, using the domain introduction rules.

Our package proves domain introduction rules automatically. Note however that in some cases of pathological pattern overlaps, they can be weaker then one first expects, since recursive calls can be "hidden" in other equations. Consider the definition

$$\begin{aligned} \mathrm{f}\ 0 &= 0 \\ \mathrm{f}\ (x+1) &= \mathrm{f}\ x \\ \mathrm{f}\ (x+2) &= \mathrm{f}\ x \end{aligned}$$

Here, the termination of $\mathrm{f}\ x+2$ case not only depends on $\mathrm{f}\ x$, but also on $\mathrm{f}\ (x+1)$, since the second equation is also applicable. However, in practice such cases do not appear very frequently, and the generated domain introduction rules are generally useful.

4.2 Wellfounded recursion relations

Often, a function is proved total by providing a wellfounded relation and showing that it is a recursion relation. Although we do not use user-specified recursion relations for the definition itself, we can still support this approach to termination proofs: Since we used the smallest recursion relation, if any other recursion relation is wellfounded, then so is R. This argument is reflected by the following rule (again, for our fib-example).

$$\frac{\mathrm{wf}\ R' \qquad n <_{R'} n+2 \qquad n+1 <_{R'} n+2}{\forall x.x \in \mathrm{dom}_{\mathrm{fib}}}$$

Our package automatically provides this rule, which allows existing termination proofs to be easily "ported" to our package.

4.3 Simplification and Induction rules revisited

For total functions, the termination assumptions in the simplification and induction rules are actually unnecessary, and can be easily removed after the termination proof, to get unconstrained simplification and induction rules familiar from total function definitions.

For partial functions, we can replace the abstract domain $\mathrm{dom}_{\mathrm{f}}$ by a concrete set D, for which we have proved termination. Note that in order to replace $\mathrm{dom}_{\mathrm{f}}$ in the *premises* of the induction rule, we must also show that D is downward closed under R, since the induction principle is only valid, if calls on elements of D only recurse on elements of D[5]. In practice, this is often simple. Our package supports the removal of termination assumptions by setting up the required proof obligations, and modifying the rules after the proof.

[5] For example, we cannot use the set of even numbers in our example fib. Although the function does terminate on all even numbers, the modified induction principle would be.

5 Nested Recursion

Functions with nested recursive calls are notoriously difficult to define and reason about. The central problem is that the termination conditions resulting from a nested recursion contain references to the function that is to be defined. As a very simple (and prominent) example, consider the following definition of the constant zero function:

$$\begin{gathered} \mathrm{f}\ 0 = 0 \\ \mathrm{f}\ (n+1) = \mathrm{f}\ (\mathrm{f}\ n) \end{gathered}$$

As a termination condition one would have to prove that $\mathrm{f}\ n < n + 1$. Since the function is always zero, this is certainly true, but seems difficult to prove, before the function f is "properly" defined. We can identify two problems here:

(1) If the system requires the termination proof to be conducted before the function symbol f is even introduced in the logic, it is difficult to support nested recursion, since the termination goal can not even be stated. Definitional packages like ours do not have this problem, since definitions are transformed into a non-recursive form and can be introduced into the logic immediately. Observe that nested recursions do not make our definitions circular, although the definition of R may refer to f.

(2) After stating the termination goal, we need to prove it, and this requires reasoning principles for the function. But the main tool, namely f-induction, is usually not available at that point, since it depends on f's termination.

Slind's recursion package TFL [17] provides a "provisional induction rule" [19] to solve nested termination goals. This rule is basically a severely mangled f-induction rule, where the unsolved termination conditions become part of the function body. With TFL's second definition principle, relationless definition, this becomes even more difficult. The provisional induction rule can help with termination proofs, but this is often quite inelegant due to the structure of the rule. Slind already observes these shortcomings in the conclusion of [19]:

> "We regard our results on relationless definition of nested recursion as only partly satisfactory. The specified recursion equations and induction theorems are automatically derived, which is good; however, the termination proof using the provisional induction theorem and recursion equations for the auxiliary function is usually clumsy and hard to explain."

As an alternative approach, Krstić and Matthews [11] proposed the notion of *inductive invariants* to describe properties of a function f in terms of an input-output relation, without the need to explicitly mention f. They show how such an inductive invariant can be used to prove f's termination.

But this comes at a high cost, since establishing an inductive invariant is comparatively hard: The proof of an inductive invariant corresponds to a wellfounded induction, and to be able to apply the induction hypothesis, one has to show that the arguments in the inner recursive calls are decreasing. This means that one has to anticipate parts of the termination proof to establish the inductive invariant.

Instead, we would like to be able to use f-induction, which is generally simpler[6]. Giesl [7] shows that this approach is sound: We may prove lemmas by f-induction and then use them (in a certain way) in the termination proof of f. His argument is that anything proved by f-induction is "partially true", i.e. it holds for all x where f terminates. Then, a close look reveals that at the positions where the lemmas are needed, one can assume this, since the inner recursive calls are proved first. But since Giesl's informal proofs include statements like *"P holds for all x where f terminates"*, it was not clear how to formalize them in a logic like HOL, where "termination" has no direct correspondence.

Fortunately, our framework provides adequate tools to express such notions, since termination is modeled by membership in $\mathrm{dom_f}$. We can state and prove that f returns zero, whenever it terminates:

$$x \in \mathrm{dom_f} \Longrightarrow \mathrm{f}\ x = 0$$

The proof is just as simple as if we already knew that the function is total: Induction and simplification, but using the partial induction and simplification rules. Then termination of f is equally simple, using structural induction and the domain introduction rules, making use of the lemma to show termination of the outer recursive call.

6 Case studies

To test our method and implementation, we conducted several small case studies, which show that our tool is practically useful.

We defined the *partial interpreter* we presented in the motivation. It is trivial to show by structural induction that it terminates on programs without while loops. It is also simple to prove termination of a bigger class of programs,

We were able to define the *substitution function for α-equated lambda-terms*, which makes extensive use of the new pattern matching features. Showing pattern compatibility turned out to be the main challenge for this.

For nested recursion, we adopted an example from Slind [19]: *first order unification*. Before proving termination, we use the partial induction principle to prove two lemmas about the substitutions returned by the algorithm. These properties are needed in the termination proof. Compared to Slind's quite technical proof, this approach avoids a large amount of "formal noise".

For space reasons, we cannot give a presentation of the theories. Formal proof documents in human-readable Isabelle/Isar notation are available as an electronic appendix to this paper[7].

[6] To compare wellfounded induction with f-induction, it is an interesting exercise to add even more nesting to the nested-zero example by changing the second equation to $f\ (n+1) = f\ (f\ (f\ (f\ n)))$, and then trying to prove the lemma $\forall n.f\ n = 0$ once by *nat*-induction, where the property can be assumed on smaller arguments and once by f-induction, where the property can be assumed on the arguments of all recursive calls.

[7] `http://www4.in.tum.de/~krauss/partial/`

7 Related Work

Both Isabelle 2005 and HOL4 [9] include (different versions of) the recursion package TFL, a work by Slind [17, 18]. In TFL, a definition is transformed into a functional, and a specialized fixed-point combinator is used to define the function. The user must specify a wellfounded recursion relation. Optionally, TFL allows deferred termination arguments, where the wellfounded relation is not given at definition time, but in later proofs. Proving termination amounts to showing that the relation is indeed a recursion relation and wellfounded. TFL generates an induction principle for each definition. Since it is based on wellfounded relations, TFL can only define total functions.

TFL allows pattern matching in the style of functional programming. Patterns are compiled to a conditional expression in a preprocessing step, while completing them and removing overlaps. This compilation is inherently limited to datatype patterns. Due to the preprocessing, the returned equations can differ from the original specification.

HOL Light [10] provides a similar mechanism, also based on a fixed-point combinator. Patterns are similar to ours, but since they are allowed to be incomplete, no induction principle can be provided. There is no general support for Higher-Order recursion.

Another approach to define certain partial functions is by tail recursion (see [13]). Since tail recursions always have a total model, they are immediately admissible without a termination proof. However, since no induction rule can be provided, the lack of reasoning principles often makes this approach harder than it sounds.

The idea of generating an explicit description of a function's domain was first presented by Dubois and Donzeau-Gouge [6] in a type theoretic setting, and later used by Bove and Capretta [4] to develop a definition principle for general recursion in type theory. In Coq [3], a recent package for general recursion [1] allows (non-nested) definitions in a manner similar to TFL.

A different approach for dealing with non-termination is to work in domain theory, where any computable function can easily be defined. But domain theory comes with a certain overhead which usually results in cumbersome reasoning about the functions later.

8 Conclusion

We have presented a method for recursive function definitions, based on an inductive definition of the function's graph, together with its domain as the accessible part of its recursion relation. Compared to existing approaches, we have been able to increase both the expressive power and the convenience in formal reasoning. In the future, we hope to use this as a basis for the principal tool for defining functions in Isabelle/HOL, subsuming both TFL [17] and the package for primitive recursion on datatypes [2]. The latter is just a special case of general recursion, where termination is immediate, but the user is forced into the recursion scheme of the datatype specification.

To complement this work, we intend to adapt existing techniques to automate termination proofs. The clear separation of definition and termination proofs in our design allows an easy integration of such external reasoning components.

References

1. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and Logic Programming (FLOPS'06)*, LNCS 3945. Springer, 2006. To Appear.
2. S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, TPHOLs '99*, LNCS 1690, pages 19–36. Springer, 1999.
3. Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical comp. science. Springer, 2004.
4. A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.
5. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
6. C. Dubois and V. Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. In *CADE-15 Workshop on mechanization of partial functions*, 1998.
7. J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, Aug. 1997.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In B. Gramlich, editor, *FroCos*, LNCS 3717, pages 216–231. Springer, 2005.
9. M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
10. J. Harrison. The HOL Light theorem prover. http://www.cl.cam.ac.uk/users/jrh/hol-light.
11. S. Krstić and J. Matthews. Inductive invariants for nested recursion. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2003*, LNCS 2758, pages 253–269. Springer, 2003.
12. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2001.
13. P. Manolios and J. S. Moore. Partial functions in ACL2. *J. Autom. Reasoning*, 31(2):107–127, 2003.
14. O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
16. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction - CADE-12*, LNCS 814, pages 148–161. Springer, 1994.
17. K. Slind. Function definition in Higher-Order Logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs '96*, LNCS 1125, pages 381–397. Springer, 1996.
18. K. Slind. *Reasoning About Terminating Functional Programs*. PhD thesis, Institut für Informatik, TU München, 1999.
19. K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLS 2000*, LNCS 1869, pages 498–518. Springer, 2000.
20. C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. In R. Nieuwenhuis, editor, *Automated Deduction - CADE-20*, LNCS 3632, pages 38–53. Springer, 2005.
21. C. Walther. On proving the termination of algorithms by machine. *Artif. Intell.*, 71(1):101–157, 1994.