Partial and Nested Recursive Function Definitions in Higher-Order Logic

Alexander Krauss

Received: 11 November 2008 / Accepted: 16 November 2009

Abstract Based on inductive definitions, we develop a tool that automates the definition of partial recursive functions in higher-order logic (HOL) and provides appropriate proof rules for reasoning about them. Termination is modeled by an inductive domain predicate which follows the structure of the recursion. Since a partial induction rule is available immediately, partial correctness properties can be proved before termination is established. It turns out that this modularity also facilitates termination arguments for total functions, in particular for nested recursions.

Our tool is implemented as a definitional package extending Isabelle/HOL. Various extensions provide convenience to the user: pattern matching, default values, tail recursion, mutual recursion and currying.

Contents

1 Introduction

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

Technische Universität München, Institut für Informatik
http://www.in.tum.de/~krauss
This is the author's version of the paper.
The official publication is available at http://dx.doi.org/10.1007/s10817-009-9157-2

beyond what the underlying logic natively supports. For example, the Isabelle/HOL [39] system provides packages for defining inductive datatypes [7] or inductive predicates [43]. All these tools are implemented as *definitional extensions*, which introduce a definition by reducing it to a simpler form that can be processed by existing means. The original specification is then *derived* from the primitive definition by an automated proof procedure. Since all reasoning is performed within the theorem prover, definitional extensions are conservative by construction, and thus offer a maximum of safety. At the same time they are convenient, as the internal constructions are transparent to the user.

Konrad Slind's TFL package [44,45] is such a definitional extension that can introduce total recursive functions using a special well-founded recursion operator. It is implemented both in HOL4 and Isabelle/HOL. At definition time, the user must prove the termination of the function, which ensures that the definition does not introduce inconsistencies. However, many interesting algorithms do not always terminate: search in an infinite search space, semi-decision procedures or the evaluation of programs are just a few examples.

Our aim is to provide a definitional principle that is not a priori limited to terminating recursions, but also works on functions which need not terminate on all input values. There are several known ways of expressing partiality in a logic of total functions (e.g., *option* types, underspecification, and using relations; see also [37]), but defining a function from a set of non-terminating equations is nontrivial, especially when the domain of the function is not obvious. Thus, modeling such functions often requires artificial manual workarounds, which can complicate subsequent reasoning.

This paper describes the foundations and mechanics of a definition principle for partial recursive functions in Isabelle/HOL. 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 proof rules allow convenient reasoning about such partial functions, as is common practice for total functions.

As a pleasant side effect of handling partiality, our approach naturally supports nested recursive definitions, which have posed technical problems for a long time. Most of these difficulties disappear entirely in our setting, since the explicit domain cleanly separates partial correctness and termination properties.

This paper is an extended and improved version of an earlier paper [30], with various simplifications in the presentation, more examples and a discussion of several extensions of the tool, and its limitations.

1.1 Motivation

1.1.1 Partiality

As an example of a partial recursive function, consider 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 below. For simplicity, a shallow embedding is used for expressions, instead of modeling their syntax. The notation $f(x := y)$ denotes function update, and *fun-pow* denotes function exponentiation.

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

datatype com =
Assign var exp
| Seq com com
| If exp com com
| For exp com
| While exp com

function *exec* :: *com* ⇒ *env* ⇒ *env*
where

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* ≠ *0* then *exec* c_1 *e* else *exec* c_2 *e*
exec (*For exp c*) *e*	=	*fun-pow* (*exp e*) (*exec c*) *e*
exec (*While exp c*) *e*	=	if *exp e* ≠ *0* then *exec* (*While exp c*) (*exec c e*) else *e*

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

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

$$f\,x = (\text{if } \langle guard \rangle \text{ then } \langle body \rangle \text{ else } dummy)$$

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, the definition must be changed. 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. So we would have to find a condition that covers all possible terminating programs, which is at least not obvious, and certainly not always executable.

In contrast, our package allows to define *exec* as a partial function and later prove its termination on the values needed.

1.1.2 Nested recursion

Functions with nested recursive calls are notoriously difficult to define and reason about. The central problem is that the termination proof for such functions requires some reasoning about partial correctness properties beforehand.

As a classic example, consider the following definition of the constant zero function on natural numbers:

$$Z\,n = (\text{if } n = 0 \text{ then } 0 \text{ else } Z\,(Z\,(n-1)))$$

For the termination of the outer call, one would usually prove that $n \neq 0 \Longrightarrow Z\,(n-1) < n$. Since the function always returns zero, this is certainly true, but seems difficult to prove before the function is "properly" defined. We can identify two problems here:

1. If the system requires the termination proof to be conducted before the function symbol Z is even introduced in the logic, it is difficult to support nested recursion, since the termination goal cannot 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 without proof. The challenge is then to derive the desired properties afterwards.
2. After stating the termination goal, we need to prove it, and this requires reasoning principles for the function. But the main tool, namely functional induction, is usually not available at that point, since it depends on the termination of the function. We solve this by making a restricted version of functional induction available from the very beginning. Using that rule, proofs for partial correctness properties are simple and natural.

1.2 An overview of our method

Starting from the specification of a function f, our package inductively defines its graph G_f and its domain dom_f, following 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 on the domain.

Then the package proves that G_f actually describes a function on dom_f, i.e. that function values exist and are unique. Then it automatically derives the original recursion equations and an induction rule. The rules are constrained by preconditions of the form $t \in dom_f$, that is, they describe the function's behaviour on its domain only. Despite these constraints, they allow convenient reasoning about the function, even before its termination is established. To support natural termination proofs, the package provides a special termination rule, in addition to the domain rules.

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 termination proofs are expressed. In §5, we show how our particular method improves the treatment of nested recursive definitions, which is further illustrated by the examples in §6. The algorithm for extracting recursive calls, which was omitted from §3 for brevity, is described in §7. In §§8–10, we discuss extensions to the core system, the limitations of the approach, and 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 $\Longrightarrow$. To make theorems more readable, we sometimes present them in inference rule notation using horizontal lines. Outermost universal quantifiers are often omitted.

In recent versions of Isabelle/HOL, the set type α *set* is just a synonym for $\alpha \Rightarrow$ *bool*. We often prefer set notation $x \in A$ over the synonymous predicate notation $A\ x$. We write $[]$ for the empty list and $\#$ for the cons operator. The function *set* $::$ α *list* $\Rightarrow$ α *set* forms the set of the elements of a list.

We will frequently use the notion of *context*. For us, a context is a set of variables and assumptions that are present locally, and captures how structured statements are decomposed. For example, the proposition $\bigwedge a\ b\ c.\ P \Longrightarrow Q \Longrightarrow R$ can be seen as a context $\Gamma = \bigwedge a\ b\ c.\ P; Q$ and a conclusion R. Conversely, we write $\Gamma \Longrightarrow R$ for the above proposition. By abuse of notation, Γ may bind variables in R.

For some of our examples we will use the declarative Isar proof language [49] to present proofs. Since Isar was designed to be human-readable, we hope that readers unfamiliar with it are nevertheless able to read the proofs and follow their structure.

To define inductive relations, we build on the existing inductive package [43] from Isabelle/HOL, which introduces inductive sets or predicates as least fixed-points by means of the Knaster-Tarski fixed-point theorem. Given some introduction rules for a set (or predicate) S, the package defines the set and returns the introduction rules as theorems. It also generates an induction rule expressing that S is in fact the smallest such set.

3 The Process of Definition

In this section, we describe the definitional core of the package. In this presentation, we ignore extra features like pattern matching, currying and mutual recursion and restrict our attention to the essential ingredients. We will discuss these extensions in §8.

We start with the recursive specification of the function, which is given by the user as a fixed point equation of some functional F.

$$f\ x = F\ f\ x$$

3.1 Recursive calls in a higher-order setting

First of all, we need to analyze the definition and extract the recursive calls.

A recursive call can be written $[\Gamma \rightsquigarrow r]$, where r is the argument of the call, and Γ is a context that specifies when the call occurs. In general, Γ can contain both bound variables and assumptions.

For example, the definition

$$f\,n = (\text{if } n = 0 \text{ then } 0 \text{ else } f\ (n - 1))$$

has a recursive call $[n \neq 0 \rightsquigarrow n - 1]$.

If we have higher-order recursion, the case becomes more complicated. Consider a datatype of n-ary trees:

$$\textbf{datatype}\ \alpha\ tree = Node\ \alpha\ (\alpha\ tree\ list)$$

We can define a function $mirror :: \alpha\ tree \Rightarrow \alpha\ tree$ as follows:

$$mirror\ (Node\ a\ ts) = Node\ a\ (map\ mirror\ (rev\ ts))$$

Here, *mirror* is passed as an argument to *map*, so we do not see immediately what the argument of the recursive call is. The answer is that the call can be described by $[\bigwedge x.\ x \in set\ (rev\ ts) \rightsquigarrow x]$. This means that calls may occur at any element of the list *rev ts*. Of course, some knowledge about *map* is required to come up with this description of the recursive call.

For the extraction of recursive calls, we use a procedure due to Slind [44,45], which can be configured by congruence rules and deals with certain forms of higher-order recursion. However, the details of the extraction process are not relevant for the understanding of our definitional principles, so we defer the description of the algorithm to §7. For the moment, we care only about the property that the recursive calls must satisfy:

Definition 1 (Congruence Condition) For a functional F and the recursive calls $[\Gamma_1 \rightsquigarrow r_1], \ldots, [\Gamma_k \rightsquigarrow r_k]$, the *congruence condition* is the implication

$$(\Gamma_1 \Longrightarrow f\,r_1 = f'\,r_1) \Longrightarrow \ldots \Longrightarrow (\Gamma_k \Longrightarrow f\,r_k = f'\,r_k) \Longrightarrow F\,f\,x = F\,f'\,x\,.$$

Intuitively, this condition states that it is enough to know how a function f behaves at the recursive calls in order to compute $F\,f\,x$. The value of f on any other input does not influence the result. So we assume for now that we have a procedure to extract calls from the recursive equation.

Note however that the congruence condition is always trivially satisfied by the single recursive call $[\bigwedge x.\ \rightsquigarrow x]$, where it degenerates into the vacuous

$$(\bigwedge x.\, f\,x = f'\,x) \Longrightarrow F\,f\,x = F\,f'\,x.$$

We could actually define any function from this, but it would be of little use: As we will see, the set of recursive calls also determines the definition of the *domain* of the function. In this case, the domain would always be empty.

3.2 Defining the graph, the function, and the domain

We transform the functional specification into an inductive definition of a relation G_f, which represents the graph of the function. Suppose we have extracted the recursive calls $[\Gamma_1 \rightsquigarrow r_1], \ldots, [\Gamma_k \rightsquigarrow r_k]$. The relation G_f is then defined inductively by the following rule:

$$\frac{(\Gamma_1^{[h/f]} \Longrightarrow (r_1^{[h/f]}, h(r_1^{[h/f]})) \in G_f) \ \ldots\ (\Gamma_k^{[h/f]} \Longrightarrow (r_k^{[h/f]}, h(r_k^{[h/f]})) \in G_f)}{(x, F\,h\,x) \in G_f}(GI)$$

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

Intuitively, we add the pair $(x, F\,h\,x)$ to G_f for a fresh function variable h. The premises state that h coincides with G_f on all recursive calls. Note that x and h are universally quantified at the level of the rule.

Compared with a naive relational description, which would invent a new variable for the result of each recursive call[1], we use a single function variable h, which is constrained to the graph on all recursive calls.

After introducing G_f using the package for inductive definitions, we can define the function f itself, using HOL's definite description operator:[2]

$$f = (\lambda x.\ \mathit{THE}\ y.\ (x, y) \in G_f)$$

Hence the function is defined to take the value given by the graph, whenever that value exists and is unique. Otherwise the value of f is unspecified, in the sense that we cannot derive anything about it. In other words, f is total in the usual sense, but possibly underspecified.

The function is now defined, but it is not yet usable. We need to prove that it actually satisfies the specification. An important reasoning tool will be the *domain* of the function. It is defined inductively, too:

[1] Inventing separate variables for the recursive calls would require additional bookkeeping and lead to problems with higher-order recursion.

[2] The description operator *THE* belongs to the axiomatic basis of HOL and is axiomatized as $(\mathit{THE}\,x.\ x = a) = a$. It is basically the ι operator used by Andrews [1].

$$\frac{\Gamma_1 \Longrightarrow r_1 \in dom_f \qquad \ldots \qquad \Gamma_k \Longrightarrow r_k \in dom_f}{x \in dom_f}\ (dom_f\text{-}intro)$$

This definition is structurally similar to the definition of G_f, but it is simpler, since it only talks about the function arguments, not the values. Also note that the function variable h is not used here, since f is already defined at this point. (Recall that in the case of nested recursion, some of the recursive calls may mention f.)

3.3 The relation is a function

We must now show that the relation G_f really describes a function on dom_f, which gives an a posteriori justification of our use of the description operator:

$$x \in dom_f \Longrightarrow \exists! y.\ (x, y) \in G_f$$

The proof of this property is performed automatically by our package for each definition. The following proof sketch illustrates the structure of the derivation:

The proof is by induction on dom_f. For some fixed $x \in dom_f$, we can assume that the property holds on all recursive calls. Splitting into existence and uniqueness, and using the fact that the unique value is denoted by f, we get for each recursive call $[\Gamma_i \rightsquigarrow r_i]$:

$$\begin{array}{ll} \Gamma_i \Longrightarrow (r_i, f\, r_i) \in G_f & (ihyp\text{-}ex_i) \\ \bigwedge z.\ \Gamma_i \Longrightarrow (r_i, z) \in G_f \Longrightarrow z = f\, r_i & (ihyp\text{-}un_i) \end{array}$$

Now, the $ihyp\text{-}ex_i$ are exactly the premises of the introduction rule for G. Hence we get the existence part $(x, F\, f\, x) \in G$.

For the uniqueness part, we must show that this is the only possible value of the function. Assume another y with $(x, y) \in G$. By inversion on G it follows that $y = F\, h\, x$ for some h, and that h follows G on the recursive calls. Hence $\Gamma_i \Longrightarrow h\, r_i = f\, r_i$ for each recursive call. It remains to apply the congruence property and conclude that $F\, f\, x = F\, h\, x$, which proves uniqueness.

3.4 Deriving simplification and induction rules

Having established that function values exist and are unique on the domain, we prove the original recursion equation and an induction rule. The equation is just as given in the original specification, but guarded by a domain condition:

$$x \in dom_f \Longrightarrow f\, x = F\, f\, x$$

Deriving the recursion equation is now simple: From uniqueness it follows that $(x, y) \in G_f$ implies $f\, x = y$, and we have already proved the required relation in the existence part of the previous proof. It can be reused after lifting it 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 final inductive result is restricted to dom_f:

$$\frac{\bigwedge x.\ x \in dom_f \Longrightarrow (\Gamma_1 \Longrightarrow P\, r_1) \Longrightarrow \ldots \Longrightarrow (\Gamma_k \Longrightarrow P\, r_k) \Longrightarrow P\, x}{a \in dom_f \Longrightarrow P\, a}\ (pinduct_f)$$

This rule is very similar to the induction principle that comes with dom_f, which looks like this:

$$\frac{\begin{array}{l}\bigwedge x.\ (\Gamma_1 \Longrightarrow r_1 \in dom_f) \Longrightarrow (\Gamma_1 \Longrightarrow P\ r_1) \Longrightarrow \ldots \\ \quad \Longrightarrow (\Gamma_k \Longrightarrow r_k \in dom_f) \Longrightarrow (\Gamma_k \Longrightarrow P\ r_k) \Longrightarrow P\ x\end{array}}{a \in dom_f \Longrightarrow P\ a}\ (dom_f\text{-}induct)$$

The first rule is easily derived from the second, but it is better suited for automation, since the premise $x \in dom_f$ is useful to unfold a call $f\,x$, which would normally occur in an actual induction.

The induction rule typically does not mention the function f. In a concrete induction, the function usually occurs in the instantiation of P. However, when the recursion is nested, then the function already appears in the induction rule, in some of the Γ_i or r_i.

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. The rest of the construction is considered internal, and users need not know about it.

3.5 Simple Examples

The following simple examples illustrate the behaviour of the package. For each function, we give the extracted recursive calls, the definitions of the graph and the domain, and the generated simplification and induction rules.

3.5.1 The Fibonacci function

Recursive equation:

$$fib\ n = (\text{if } n \leq 1 \text{ then } n \text{ else } fib\ (n-1) + fib\ (n-2))$$

Extracted calls:

$$[\neg\ n \leq 1 \rightsquigarrow n-1],\ [\neg\ n \leq 1 \rightsquigarrow n-2]$$

Graph:

$$\frac{\neg\ n \leq 1 \Longrightarrow (n-1, h\ (n-1)) \in G_{fib} \qquad \neg\ n \leq 1 \Longrightarrow (n-2, h\ (n-2)) \in G_{fib}}{(n, \text{if } n \leq 1 \text{ then } n \text{ else } h\ (n-1) + h\ (n-2)) \in G_{fib}}$$

Domain:

$$\frac{\neg\ n \leq 1 \Longrightarrow n-1 \in dom_{fib} \qquad \neg\ n \leq 1 \Longrightarrow n-2 \in dom_{fib}}{n \in dom_{fib}}$$

Simplification and induction rules:

$$n \in dom_{fib} \Longrightarrow fib\ n = (\text{if } n \leq 1 \text{ then } n \text{ else } fib\ (n-1) + fib\ (n-2))$$

$$\frac{\bigwedge n.\ n \in dom_{fib} \Longrightarrow (\neg\ n \leq 1 \Longrightarrow P\ (n-1)) \Longrightarrow (\neg\ n \leq 1 \Longrightarrow P\ (n-2)) \Longrightarrow P\ n}{a \in dom_{fib} \Longrightarrow P\ a}$$

3.5.2 Nested zero

We now define the nested zero function from §1.1.2. Observe that the nested recursion does not make our definitions circular. The definition of dom_Z below may refer to Z, which is already defined.

Recursive equation:

$$Z\,n = (\text{if } n = 0 \text{ then } 0 \text{ else } Z\,(Z\,(n-1)))$$

Extracted calls:

$$[n \neq 0 \rightsquigarrow n-1],\ [n \neq 0 \rightsquigarrow Z\,(n-1)]$$

Graph:

$$\frac{n \neq 0 \Longrightarrow (n-1, h\,(n-1)) \in G_Z \qquad n \neq 0 \Longrightarrow (h\,(n-1), h\,(h\,(n-1))) \in G_Z}{(n, \text{if } n = 0 \text{ then } 0 \text{ else } h\,(h\,(n-1))) \in G_Z}$$

Domain:

$$\frac{n \neq 0 \Longrightarrow n-1 \in dom_Z \qquad n \neq 0 \Longrightarrow Z\,(n-1) \in dom_Z}{n \in dom_Z}$$

Simplification and induction rules:

$$n \in dom_Z \Longrightarrow Z\,n = (\text{if } n = 0 \text{ then } 0 \text{ else } Z\,(Z\,(n-1)))$$

$$\frac{\bigwedge n.\ n \in dom_Z \Longrightarrow (n \neq 0 \Longrightarrow P\,(n-1)) \Longrightarrow (n \neq 0 \Longrightarrow P\,(Z\,(n-1))) \Longrightarrow P\,n}{a \in dom_Z \Longrightarrow P\,a}$$

3.5.3 The partial function findzero

The function *findzero* (f, n) returns the smallest value $n' \geq n$ such that $f\,n' = 0$. If no such value exists, the function diverges.

Recursive equation:

$$\mathit{findzero}\,(f, n) = (\text{if } f\,n = 0 \text{ then } n \text{ else } \mathit{findzero}\,(f, \mathit{Suc}\,n))$$

Extracted calls:

$$[f\,n \neq 0 \rightsquigarrow (f, \mathit{Suc}\,n)]$$

Graph:

$$\frac{f\,n \neq 0 \Longrightarrow ((f, \mathit{Suc}\,n), h\,(f, \mathit{Suc}\,n)) \in G_{fz}}{((f, n), \text{if } f\,n = 0 \text{ then } n \text{ else } h\,(f, \mathit{Suc}\,n)) \in G_{fz}}$$

Domain:

$$\frac{f\,n \neq 0 \Longrightarrow (f, \mathit{Suc}\,n) \in dom_{fz}}{(f, n) \in dom_{fz}}$$

Simplification and induction rules:

$$(f, n) \in dom_{fz} \Longrightarrow \mathit{findzero}\,(f, n) = (\text{if } f\,n = 0 \text{ then } n \text{ else } \mathit{findzero}\,(f, \mathit{Suc}\,n))$$

$$\frac{\bigwedge f\,n.\ (f, n) \in dom_{fz} \Longrightarrow (f\,n \neq 0 \Longrightarrow P\,(f, \mathit{Suc}\,n)) \Longrightarrow P\,(f, n)}{(f', n') \in dom_{fz} \Longrightarrow P\,(f', n')}$$

Using the partial induction rule, it is now straightforward to prove properties about the function, e.g., $(f, n) \in dom_{fz} \Longrightarrow \forall x \in \{n..<\mathit{findzero}\,(f, n)\}.\ f\,x \neq 0$.

3.5.4 The function that is always undefined

The package even lets us define the function that never terminates. However, the results are not very interesting:

Recursive equation:

$$U\,x = U\,x + 1$$

Extracted calls:

$$[\rightsquigarrow x]$$

Graph:

$$\frac{(x, h\,x) \in G_U}{(x, h\,x + 1) \in G_U}$$

Domain:

$$\frac{n \in dom_U}{n \in dom_U}$$

Simplification and induction rules:

$$x \in dom_U \Longrightarrow U\,x = U\,x + 1$$

$$\frac{\bigwedge x.\ x \in dom_U \Longrightarrow P\,x \Longrightarrow P\,x}{a \in dom_U \Longrightarrow P\,a}$$

Both the graph and the domain are just the empty set (which can easily be proved by induction), and hence the simplification and induction rules are just instances of *ex falso quodlibet*.

4 Termination Proofs

All results obtained from the partial simplification and induction rules will contain termination assumptions of the form $t \in dom_f$. It is desirable to know more about dom_f, which is the objective of a termination proof. Often, our goal will be to show that a function is total, that is, $\forall x.\ x \in dom_f$. For partial functions, we usually want to prove $S \subseteq dom_f$ for some interesting subset S.

While the definition process we saw above is fully automated and works for any function definition, we cannot expect such complete automation for termination proofs. However, there is a lot of research in automated termination proofs, and powerful methods exist, e.g., [2,33,48].

Our primary goal is to provide a clear and simple interface for integrating automated tools without requiring a deep knowledge of the inner workings of the function package. Moreover, since automated tools can always fail, we also need to provide a simple interface to the user, who may need to perform the termination proof interactively.

4.1 Elementary proofs, using the definition of the domain

The definition of the domain that we gave in the previous section is already a very natural description of the termination behaviour of the function. It is not hard to do a termination proof with just the domain introduction rule and a suitable induction principle.

For example, we can show $\forall n.\ n \in dom_{fib}$ by course-of-values induction: Under the assumption $\neg\ n \leq 1$, we know that both $n - 1$ and $n - 2$ are strictly less than n. Thus by induction hypothesis we get $\neg\ n \leq 1 \Longrightarrow n - 1 \in dom_{fib}$ and $\neg\ n \leq 1 \Longrightarrow n - 2 \in dom_{fib}$. By the introduction rule for the domain (cf. §3.5) we conclude $n \in dom_{fib}$.

4.2 Termination proofs using relations

Instead of doing a manual induction, the standard technique for proving that a function is total is to provide a well-founded relation R and show that all recursive calls are decreasing w.r.t. R. Although our definition principle does not require user-specified relations, we can still support them as a way to prove termination. This is accomplished by the following rule, provided by our package for *fib*:

$$\frac{wf\ R \qquad \bigwedge n.\ \neg\ n \leq 1 \Longrightarrow (n-1, n) \in R \qquad \bigwedge n.\ \neg\ n \leq 1 \Longrightarrow (n-2, n) \in R}{\forall x.\ x \in dom_{fib}}$$

With this rule, a termination proof can be done by giving a well-founded relation and a proof that every recursive call is decreasing. Of course, this is just what is going on in the induction proof above, but here the induction is implicit in the relation R. This makes it easier to automate the proof, as the decrease conditions are often inequalities that are relatively easy to show once a suitable relation is found.

Here is the general form of this rule, which we call the *termination rule*:

$$\frac{wf\ R \qquad \bigwedge x.\ \Gamma_1 \Longrightarrow (r_1, x) \in R \qquad \ldots \qquad \bigwedge x.\ \Gamma_k \Longrightarrow (r_k, x) \in R}{\forall x.\ x \in dom_f}$$

The proof of the termination rule is just a well-founded induction over R. The termination rule is proved automatically for each function definition.

4.3 Simplification and induction rules revisited

When we have proved that the function is total, the domain conditions in the simplification and induction rules become obsolete. It is now easy to project them away, and we obtain what we call the *total* simplification rule and the *total* induction rule. These rules are then the primary tools for reasoning about the function.

For partial functions, we can replace the abstract domain dom_f by a concrete set $D \subseteq dom_f$. Note that in order to replace dom_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. For example, we cannot use the set of even numbers in our Fibonacci example. Although the function does terminate on all even numbers, the modified induction principle would be invalid. In practice, proving this downward closure property is often simple.

4.4 Integration of automated tools

Since proving termination of a function is just like proving a lemma, there is a clear interface for integrating automated termination provers. What we require is just a tactic that is able to solve goals of a certain form. This form is given in the premises of the termination rule. In separate work [17,31], we have shown that it is possible to implement nontrivial termination provers in this way. There is ongoing work on integrating more termination proof techniques.

5 Nested Recursion

Nested recursive definitions have recursive calls that depend on the results of other recursive calls. Thus, we usually need to prove some property about the behaviour of the function before we can establish its termination. Here we are facing an apparent circularity, since the induction rule that we would like to use depends on termination.

Slind's recursion package TFL [44] provides a "provisional induction rule" [46] to solve nested termination goals. This rule is basically a modified functional 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 [46]:

> 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 [32] 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 well-founded induction, and to be able to apply the induction hypothesis, we must show that the arguments in the inner recursive calls are decreasing. This means that we must anticipate parts of the termination proof to establish the inductive invariant.

Instead, we would like to be able to use functional induction, which is generally simpler[3]. Giesl [22] shows that this approach is sound: We may prove lemmas by functional induction and then use them (in a certain way) in the termination proof of the same function. The argument is that anything proved by functional induction is "partially true", i.e., it holds for all values for which the function terminates. Then, a close look reveals that at the

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

positions where the lemmas are needed, we know that this condition holds, 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 previously 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 the domain. Given the nested zero function *Z* from §1.1.2, we can state and prove that *Z* returns zero whenever it terminates:

lemma Z-zero: $x \in dom_Z \Longrightarrow Z\ x = 0$
by (induct rule: $pinduct_Z$) auto

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 *Z* is equally simple, using induction and the domain introduction rules, making use of the lemma to show termination of the outer recursive call:

lemma Z-terminates: $x \in dom_Z$
proof (induct x rule: less-induct)
 fix x **assume** IH: $\bigwedge y.\ y < x \Longrightarrow y \in dom_Z$
 show $x \in dom_Z$
 proof cases
  assume $x = 0$ **thus** $x \in dom_Z$ **by** (auto intro: dom_Z-intro)
 next
  assume $x \neq 0$
  with IH **have** $(x - 1) \in dom_Z$ **by** auto
  hence $Z\ (x - 1) = 0$ **by** (rule Z-zero)
  hence $Z\ (x - 1) \in dom_Z$ **by** (auto intro: dom_Z-intro)
  from ‹$(x - 1) \in dom_Z$› **and** ‹$Z\ (x - 1) \in dom_Z$›
  show $x \in dom_Z$ **by** (auto intro: dom_Z-intro)
 qed
qed

This shows that our approach separates the partial correctness proof (*Z* returns zero if it terminates) and the termination proof (*Z* terminates), which makes reasoning very natural. In §6, we give more examples of nested recursions.

5.1 A termination rule for nested recursion

In order to formally justify the informal reasoning skeched in the beginning of this section, we have another look at the termination rule from §4.2. For the nested-zero function *Z*, the following rule would be generated:

$$\frac{wf\ R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (n-1, n) \in R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (Z\ (n-1), n) \in R}{\forall x.\ x \in dom_Z}$$

Now the second termination condition involves the function *Z*, which requires the lemma $x \in dom_Z \Longrightarrow Z\ x = 0$ to prove that the outer call is decreasing. But with this rule this is impossible, since we cannot assume termination of the inner call. In the previous induction proof, this was solved simply by treating the inner call first.

The following variant, which we call the *nested termination rule* solves the problem:

$$\frac{\mathit{wf}\ R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (n-1, n) \in R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow \boxed{n-1 \in \mathit{dom}_Z} \Longrightarrow (Z\ (n-1), n) \in R}{\forall x.\ x \in \mathit{dom}_Z}$$

The new assumption, marked with grey background, states that the inner call terminates, when proving that the outer call is decreasing. This is just enough to apply the lemma and conclude the proof, since we are just left with the trivial goal $n \neq 0 \Longrightarrow 0 < n$.

5.2 Proving the nested termination rule

The system proves the nested termination rule automatically by well-founded induction on the relation R. In the body of the induction we need to prove $x \in \mathit{dom}_f$ under the assumption $\bigwedge z.\ (z, x) \in R \Longrightarrow z \in \mathit{dom}_f$. Using the introduction rule for dom_f, it is sufficient to show $\Gamma_i \Longrightarrow r_i \in \mathit{dom}_f$ for each recursive call. Now we proceed from the innermost call to the outer calls. By assumption, the innermost call satisfies $\Gamma_1 \Longrightarrow (r_1, x) \in R$ and thus $\Gamma_1 \Longrightarrow r_1 \in \mathit{dom}_f$ holds by induction hypothesis. This fact is used to strengthen the assumptions for the outer recursive calls, which are then treated in the same way. The outermost calls come last, and for them we can assume termination of all inner calls.

6 Further Examples and Case Studies

In this section, we present some more examples that demonstrate how the partial induction rule simplifies reasoning about nested recursive and partial functions.

6.1 Nested list reversal

The following function defines list reversal in an uncommon way that does not need any auxiliary function or additional parameter. It was given to the author as a challenge problem by Konrad Slind, and it is probably more of a puzzle than a sensible implementation.

$$\begin{array}{lcl} \mathit{Rev}\ [] & = & [] \\ \mathit{Rev}\ (x \# xs) & = & \mathsf{case}\ \mathit{Rev}\ xs\ \mathsf{of}\ [] \Rightarrow [x] \mid y \# ys \Rightarrow y \# \mathit{Rev}\ (x \# \mathit{Rev}\ ys) \end{array}$$

In fact, proving that *Rev* is equivalent to Isabelle's built-in function *rev* is just a straightforward functional induction. But the nesting made this function hard to define and reason about with previous tools. Using the partial induction rule, we can jump forward and prove that *Rev* is equal to *rev* on its domain:

lemma Rev-eq-rev[simp]: xs $\in$ $\mathrm{dom}_{Rev} \Longrightarrow$ Rev xs = rev xs
by (induct xs rule: $\mathrm{pinduct}_{Rev}$) (auto split: list.splits)

This is so easy because the induction rule is tailored to the recursive structure of the function. Now, using this lemma, termination is not hard to prove, since much about *rev* is already known (in particular, $\mathit{length}\ (\mathit{rev}\ xs) = \mathit{length}\ xs$). The termination order is found automatically by Isabelle's termination prover [17].

6.2 McCarthy's 91 function

The ninety-one function is a well-known challenge problem due to John McCarthy:

$$f91\ n = (\text{if } 100 < n \text{ then } n - 10 \text{ else } f91\ (f91\ (n + 11)))$$

The termination argument relies on the following lemma:

lemma f91-estimate:
n ∈ dom_{f91} ⟹ n < f91 n + 11
by (induct rule: pinduct_{f91}) auto

Now we can proceed with a manual termination proof, which we give in Isar notation. Note how the assumption $n + 11 \in dom_{f91}$ is used for the outer call to discharge the hypothesis of the lemma:

lemma f91-terminates: ∀ x. x ∈ dom_{f91}
proof (rule f91.termination)
let ?R = measure (λx. 101 − x) — The termination relation used
show wf ?R **..**
fix n :: nat **assume** ¬ 100 < n — Assumptions for both calls
thus (n + 11, n) ∈ ?R **by** simp — Inner call
assume inner-trm: (n + 11) ∈ dom_{f91} — Outer call
with f91-estimate **have** n + 11 < f91 (n + 11) + 11 **.**
with ‹¬ 100 < n› **show** (f91 (n + 11), n) ∈ ?R **by** simp
qed

6.3 First order unification

A standard example of nested recursion is a unification algorithm on a simple first-order term language. This example was already presented by Slind [46], who in turn adapted it from Manna and Waldinger [34]. There are many other formalizations of unification (e.g., [42,9,36]), and we do not want to go into the details of this specific development, but merely show that our package makes reasoning about the unification function more direct than before, which leads to natural and straightforward proofs. The following description is focused on this aspect, but the interested reader can find the complete theory in the Isabelle 2009 distribution.

In our formalization, terms can be variables, constants and applications of one term to another:

datatype α *trm* =
Var α
| *Const* α
| *App* (α *trm*) (α *trm*) (**infix** · *60*)

Substitutions are modeled as association lists mapping variables to terms. The (parallel) application of a substitution σ to a term t is written $\sigma \triangleright t$, and we omit its straightforward definition. Composition of substitutions is written $\sigma_2 \circ \sigma_1$.

The unification function has the following definition:

$$
\begin{array}{lll}
\mathit{unify}\ (\mathit{Const}\ c)\ (M \cdot N) & = & \mathit{None} \\
\mathit{unify}\ (M \cdot N)\ (\mathit{Const}\ c) & = & \mathit{None} \\
\mathit{unify}\ (\mathit{Const}\ c)\ (\mathit{Var}\ v) & = & \mathit{Some}\ [(v, \mathit{Const}\ c)] \\
\mathit{unify}\ (M \cdot N)\ (\mathit{Var}\ v) & = & \text{if } \mathit{occ\text{-}check}\ (\mathit{Var}\ v)\ (M \cdot N) \text{ then } \mathit{None} \\
 & & \quad \text{else } \mathit{Some}\ [(v, M \cdot N)] \\
\mathit{unify}\ (\mathit{Var}\ v)\ M & = & \text{if } \mathit{occ\text{-}check}\ (\mathit{Var}\ v)\ M \text{ then } \mathit{None} \text{ else } \mathit{Some}\ [(v, M)] \\
\mathit{unify}\ (\mathit{Const}\ c)\ (\mathit{Const}\ d) & = & \text{if } c = d \text{ then } \mathit{Some}\ [] \text{ else } \mathit{None} \\
\mathit{unify}\ (M \cdot N)\ (M' \cdot N') & = & \mathsf{case}\ \mathit{unify}\ M\ M'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
 & & \quad \mid \mathit{Some}\ \vartheta \Rightarrow \\
 & & \quad\quad \mathsf{case}\ \mathit{unify}\ (\vartheta \triangleright N)\ (\vartheta \triangleright N')\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
 & & \quad\quad \mid \mathit{Some}\ \sigma \Rightarrow \mathit{Some}\ (\sigma \circ \vartheta) \\
\mathit{occ\text{-}check}\ u\ (\mathit{Var}\ v) & = & \mathit{False} \\
\mathit{occ\text{-}check}\ u\ (\mathit{Const}\ c) & = & \mathit{False} \\
\mathit{occ\text{-}check}\ u\ (M \cdot N) & = & (u = M \vee u = N \vee \mathit{occ\text{-}check}\ u\ M \vee \mathit{occ\text{-}check}\ u\ N)
\end{array}
$$

Note that the nesting of the recursion is not directly of the form $\mathit{unify}\ (\ldots\ \mathit{unify}(\ldots)\ \ldots)$. Instead the calls are connected via their contexts: The assumption $\mathit{unify}\ M\ M' = \mathit{Some}\ \vartheta$ occurs in the context of the second call.

To demonstrate how the partial induction rule simplifies reasoning, we prove partial correctness of the unification algorithm first (we omit the definition of *MGU*):

$$\frac{(M, N) \in \mathit{dom}_{\mathit{unify}} \qquad \mathit{unify}\ M\ N = \mathit{Some}\ \sigma}{\mathit{MGU}\ \sigma\ M\ N}$$

The proof is by induction using the partial induction rule $\mathit{pinduct}_{\mathit{unify}}$. Figure 1 shows the full proof.

In order to prove termination of the function, we need to establish two properties which have to do with occurrences of variables. First, substitutions produced by *unify* never introduce new variables (we omit the trivial definition of *vars-of*):

$$\frac{(M, N) \in \mathit{dom}_{\mathit{unify}} \qquad \mathit{unify}\ M\ N = \mathit{Some}\ \sigma}{\mathit{vars\text{-}of}\ (\sigma \triangleright t) \subseteq \mathit{vars\text{-}of}\ M \cup \mathit{vars\text{-}of}\ N \cup \mathit{vars\text{-}of}\ t}$$

Second, if *unify* returns a substitution σ, then σ is either the identity substitution or it eliminates a variable, which means that for any term t, $\sigma \triangleright t$ no longer contains the variable:

$$\frac{(M, N) \in \mathit{dom}_{\mathit{unify}} \qquad \mathit{unify}\ M\ N = \mathit{Some}\ \sigma}{(\exists\, v \in \mathit{vars\text{-}of}\ M \cup \mathit{vars\text{-}of}\ N.\ \forall\, t.\ v \notin \mathit{vars\text{-}of}\ (\sigma \triangleright t)) \vee \sigma =_s []}$$

These lemmas are again proved by partial induction, where the recursive case is the interesting one. Again, the partiality has no influence on the structure of the induction proof.

The termination proof then merely puts these results together, using the lexicographic combination of the measures $\lambda(M, N).\ \mathit{card}\ (\mathit{vars\text{-}of}\ M \cup \mathit{vars\text{-}of}\ N)$ and $\lambda(M, N).\ \mathit{size}\ M$, where *card* denotes the cardinality of a finite set. We then get total correctness as a corollary, since we now know that $(M, N) \in \mathit{dom}_{\mathit{unify}}$ always holds.

6.4 Depth-first search

Berghofer and Reiter [6] formalized various constructions on finite automata. Their theories use an abstract specification of depth-first search in directed graphs, which is developed in

```
lemma unify-partial-correctness:
  (M, N) ∈ dom_unify ⟹ unify M N = Some σ ⟹ MGU σ M N
proof (induct M N arbitrary: σ rule: pinduct_unify)
  case (7 M N M' N' σ) — The interesting case follows; the other cases are fully automatic (see last line)

  then obtain ϑ1 ϑ2
    where unify M M' = Some ϑ1 and unify (ϑ1 ▷ N) (ϑ1 ▷ N') = Some ϑ2
    and [simp]: σ = ϑ2 ∘ ϑ1
    and MGU-inner: MGU ϑ1 M M'
    and MGU-outer: MGU ϑ2 (ϑ1 ▷ N) (ϑ1 ▷ N')
    by (auto split: option.split-asm)

  show MGU σ (M · N) (M' · N')
  proof — We have a unifier:
    from MGU-inner and MGU-outer
    have ϑ1 ▷ M = ϑ1 ▷ M' and ϑ2 ▷ ϑ1 ▷ N = ϑ2 ▷ ϑ1 ▷ N'
      by (auto simp: MGU-def Unifier-def)
    thus σ ▷ M · N = σ ▷ M' · N' by simp
  next — The unifier is most general:
    fix σ' assume σ' ▷ M · N = σ' ▷ M' · N'
    hence σ' ▷ M = σ' ▷ M' and Ns: σ' ▷ N = σ' ▷ N' by auto

    with MGU-inner obtain δ where eqv: σ' =_s δ ∘ ϑ1
      by (auto simp: MGU-def Unifier-def)

    from Ns have δ ▷ ϑ1 ▷ N = δ ▷ ϑ1 ▷ N'
      by (simp add: eqv-dest[OF eqv])

    with MGU-outer obtain ϱ where eqv2: δ =_s ϱ ∘ ϑ2
      by (auto simp: MGU-def Unifier-def)

    have σ' =_s ϱ ∘ σ
      by (rule eqv-intro, auto simp: eqv-dest[OF eqv] eqv-dest[OF eqv2])
    thus ∃γ. σ' =_s γ ∘ σ ..
  qed
qed (auto split: split-if-asm) — Solve the remaining cases automatically
```

Fig. 1 Partial correctness of *unify*

an axiomatic context (using Isabelle's locale mechanism [3]), such that it can later be instantiated to different representations of the graph that is searched and the data structure that collects the results. The depth-first search function is loosely based on a previous formalization by Nishihara and Minamide [40]. A similar formalization in HOL4 is given by Owens and Slind [41].

The graph is modelled abstractly by a type *node* and the functions

succs	:: *node* ⇒ *node list*	(successor nodes)
is-node	:: *node* ⇒ *bool*	(wellformedness predicate)

Not every value of type *node* is necessarily a node in the graph, and the predicate *is-node* models the set of valid nodes, which must be finite. For a valid node, the function *succs* returns the list of successors.

During the traversal, nodes are collected in a data structure of another abstract type *C*, which behaves like a set of nodes:

$$\textit{is-node } x \Longrightarrow \textit{is-node } y \Longrightarrow \textit{invariant } S \Longrightarrow \neg\ \textit{memb } y\ S \Longrightarrow \textit{memb } x\ (\textit{ins } y\ S) = (x = y \vee \textit{memb } x\ S)$$

$$\textit{is-node } x \Longrightarrow \neg\ \textit{memb } x\ \textit{empt}$$

$$\textit{is-node } x \Longrightarrow \forall\, y{\in}\textit{set}\ (\textit{succs } x).\ \textit{is-node } y$$

$$\textit{invariant empt}$$

$$\textit{is-node } x \Longrightarrow \textit{invariant } S \Longrightarrow \neg\ \textit{memb } x\ S \Longrightarrow \textit{invariant}\ (\textit{ins } x\ S)$$

$$\textit{finite is-node}$$

Fig. 2 Axiomatic context for the depth-first search algorithm

empt	$:: C$	(empty collection)
ins	$:: \textit{node} \Rightarrow C \Rightarrow C$	(insert operation)
memb	$:: \textit{node} \Rightarrow C \Rightarrow \textit{bool}$	(membership test)
invariant	$:: C \Rightarrow \textit{bool}$	(collection invariant)

The axioms that describe these operations are given in Fig. 2. The advantage of working with an abstract specification of graphs and collections is that the algorithm can be instantiated later with concrete data structures suited for the particular applications, e.g., matrices and BDDs.

The depth-first search function is defined as follows:

$$\textit{dfs} :: C \Rightarrow \textit{node list} \Rightarrow C$$

$$\textit{dfs } S\ [] = S$$
$$\textit{dfs } S\ (x \# xs) =$$
$$(\text{if } \textit{memb } x\ S \text{ then } \textit{dfs } S\ xs \text{ else } \textit{dfs}\ (\textit{ins } x\ S)\ (\textit{succs } x\ @\ xs))$$

Even though the axiomatization ensures that the graph is finite, *dfs* is a partial function, since the behaviour of *ins* and *succs* is only specified when the invariants *is-node* and *invariant* are satisfied. Otherwise they may return a value that leads to nontermination of *dfs*.

However, termination of *dfs S xs* can be proved under these invariants, which is expressed by the theorem $\textit{invariant } S \Longrightarrow (\forall\, x{\in}xs.\ \textit{is-node } x) \Longrightarrow (S,xs) \in \textit{dom}_{\textit{dfs}}$. Moreover, as we shall see in §8.2, tail-recursion ensures that unconditional equations can be generated.

7 Extraction of Recursive Calls and Congruence Rules

The definition process described in §3 assumed an algorithm to extract the recursive calls from the right-hand side of an equation. In this section, we describe the extraction process, which takes a term and produces a set of recursive calls $[\Gamma_1 \rightsquigarrow r_1], \ldots, [\Gamma_k \rightsquigarrow r_k]$ such that the associated congruence condition is provable.

For the first-order case, such an extraction was already given by Boyer and Moore [15], but, as we have seen, higher-order recursion produces some difficulties, which were solved by Slind [45], who invented a generic extraction procedure that is parameterized by congruence rules. Our extraction is essentially the same, and our main motivation for presenting it here is to make this paper more self-contained.

Note that the extraction of calls critically influences the definition of the domain of the function and hence the termination proof obligations and the induction principle that the definition produces. Hence it is sometimes important for users to develop a basic understanding of this process.

$$\frac{xs = ys \qquad \bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x}{map\ f\ xs = map\ g\ ys}\ (map\text{-}cong) \qquad \frac{M = N \qquad \bigwedge x.\ x = N \Longrightarrow f\ x = g\ x}{Let\ M\ f = Let\ N\ g}\ (let\text{-}cong)$$

$$\frac{A = B \qquad \bigwedge x.\ x \in B \Longrightarrow P\ x = Q\ x}{(\forall x{\in}A.\ P\ x) = (\forall x{\in}B.\ Q\ x)}\ (Ball\text{-}cong) \qquad \frac{P = P' \qquad P' \Longrightarrow Q = Q'}{(P \wedge Q) = (P' \wedge Q')}\ (conj\text{-}cong)$$

$$\frac{f = g \qquad x = y}{f\ x = g\ y}\ (app\text{-}cong) \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ (lam\text{-}cong)$$

$$\frac{x = y \qquad y = 0 \Longrightarrow a = b \qquad \bigwedge n.\ y = Suc\ n \Longrightarrow f\ n = g\ n}{(\mathsf{case}\ x\ \mathsf{of}\ 0 \Rightarrow a \mid Suc\ n \Rightarrow f\ n) = (\mathsf{case}\ y\ \mathsf{of}\ 0 \Rightarrow b \mid Suc\ n \Rightarrow g\ n)}\ (nat\text{-}case\text{-}cong)$$

Fig. 3 Congruence rules for commonly used constants

7.1 Congruence rules

Congruence rules are used in contextual rewriting to accumulate context from the structure of the term. For example, when rewriting the then-part of an if-then-else expression, we may use the condition as a local assumption. In the *else* part, we may assume its negation.

This knowledge, which exploits a property of *if*, is not hard-coded in the rewriter, but expressed by a congruence rule:

$$\frac{c = c' \qquad c' \Longrightarrow t = t' \qquad \neg\, c' \Longrightarrow e = e'}{(\text{if } c \text{ then } t \text{ else } e) = (\text{if } c' \text{ then } t' \text{ else } e')}\ (\textit{if-cong})$$

Due to its characteristic form, the congruence rule can be interpreted as a recipe for rewriting an expression of the form if c then t else e: First, rewrite the condition to some c', then rewrite the *then* and *else* part, under the assumption c' or $\neg\, c'$, respectively.

Similar congruence rules exist for control structures like *if*, *case* or *let*, for higher-order combinators like *map* and *filter*, and also for bounded quantifiers like $\forall \cdot \in \cdot$. The congruence rules tell the contextual rewriter how to extend the context when traversing the term. Figure 3 shows congruence rules for other commonly used constructs. It is worth noting that some of the rules not only introduce new assumptions in the context but also bind new variables. For example, the rule for *map* introduces a variable x, which is constrained to be an element of the list.

7.2 Extracting calls

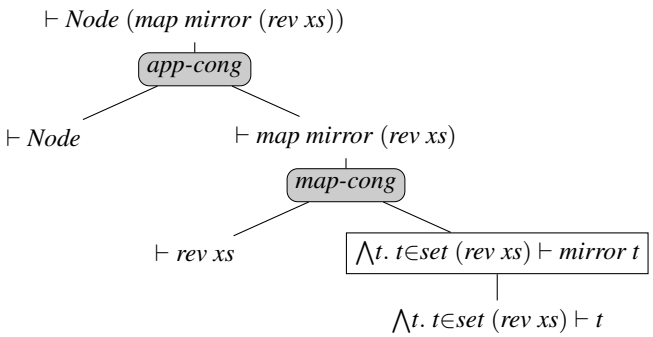
The extraction can now be described as a recursive algorithm that traverses a term t and incrementally builds a context Γ, starting from the empty context:

1. If t does not contain f, then there are no more calls and we can stop.
2. If t has the form $f\ r$, then we have the recursive call $[\Gamma \rightsquigarrow r]$ where Γ is the current context. We continue with the extraction on subterm r, in the same context.
3. Otherwise, we try the congruence rules in order, until the left-hand side of the conclusion of one rule matches t. Then, for each premise $\bigwedge xs.\ As \Longrightarrow C = C'$ of the instantiated congruence rule, extend the current context with the bound variables xs and assumptions As, and continue the extraction on C. The instantiated congruence rule is stored, as it is needed in later proofs.

The last two congruence rules that are tried are always *app-cong* and *lam-cong*. This ensures that the last case always works: Any application is just split into two parts by *app-cong*, and if a term still contains *f*, but is not an application, then we can apply *lam-cong*, possibly after eta-expansion.

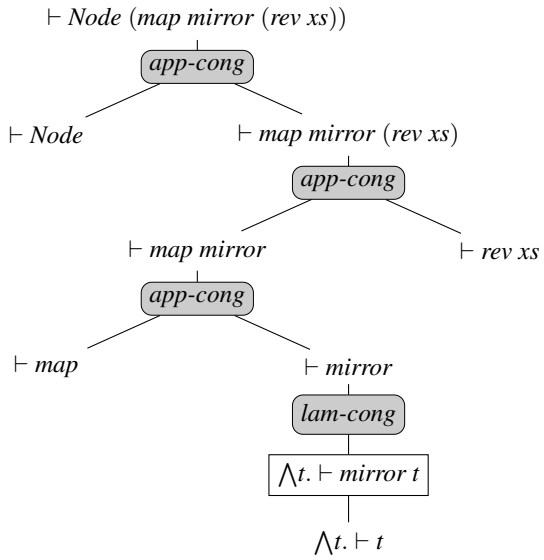
The extraction algorithm can be seen as a process of contextual rewriting to prove the congruence condition. Congruence rules are used to build up the right context for the recursive calls.

Example 1 We consider the function *mirror* given in §3. The extraction process is illustrated by a tree. Each node consists of a context and a term, written $\Gamma \vdash t$. At the root we have the complete right-hand side of the equation and the empty context. Then the term is split into components via congruence rules:



Since there is no special congruence rule for the *Node* constructor, the *app-cong* rule is applied, and it simply splits the application in two parts. The *Node* constructor on the left-hand side is uninteresting, since there is no recursive call here. On the right hand side, we have *map mirror* (*rev xs*), which now matches the *map-cong* rule. Following the structure of that rule, we get two branches, one for *rev xs*, which is again uninteresting, and one for *mirror t*, which now appears in a context extended by a new variable *t* and an assumption $t \in$ *set* (*rev xs*). At this point, we have found a recursive call, since *mirror* is now fully applied. We continue this search on the subterm *t*, where it immediately terminates, since there are no more occurrences of *mirror*.

If the rule *map-cong* were not present, we would still get a tree, but it would instead look like this:



Now the extracted recursive call would be $[\bigwedge x.\ \rightsquigarrow x]$, which is not helpful, as we have seen.

This example demonstrates that the extraction mechanism depends on the configuration via congruence rules, which encode instructions for dealing with higher-order constructs. This has the advantage that it makes the package very flexible. The disadvantage however is that users sometimes need to know how the extraction works, in order to feed it the right congruence rules.

The result of the extraction process is a set of calls $[\Gamma_1 \rightsquigarrow r_1], \ldots, [\Gamma_k \rightsquigarrow r_k]$. This set is used in the definition process as we have seen in §3. In particular it determines the definition of the graph and the domain and, as a consequence, the form of the induction rule and the termination proof obligations.

The way the calls are constructed guarantees that the congruence condition given in §3 is provable automatically. The straightforward proof simply follows the tree structure above.

7.3 Congruence rules and evaluation order

Higher-order logic differs from functional programming languages in that it has no built-in notion of evaluation order. A program is just a set of equations, and it is not specified how they must be evaluated. However, when reasoning about termination of recursive functions, an implicit notion of evaluation order sneaks in. The evaluation order is specified by the congruence rules we are using. For example, consider the following simple recursion on natural numbers:

$$f\,n = (n = 0 \vee f\ (n - 1))$$

Whether this is a total function or not depends on how we interpret $\vee$. We could use the semantics known from ML, where *orelse* and *andalso* are strict in the left argument but non-strict in the right one. Then the function terminates because $n \neq 0$ implies $n - 1 < n$. However, if the disjunction is strict in both arguments we get nontermination. (Recall that on natural numbers, $n - 1$ may be equal to n if n is zero.)

HOL itself does not make this distinction, since there is no explicit notion of undefinedness. Instead, the congruence rules that we use to extract the recursive calls will determine which function we get out. Without any congruence rules, the extraction will regard disjunction as strict in both arguments, and our function has the empty domain. However, we can give a congruence rule for disjunction that gives the behaviour known from ML:

$$\frac{P = P' \qquad \neg P' \Longrightarrow Q = Q'}{(P \vee Q) = (P' \vee Q')} \ (\textit{disj-cong})$$

Now the definition of f above gives us the total function we expect. The termination proof will use the extra condition that we obtained from the congruence rule.

However, as evaluation is not a hard-wired concept, we could just turn everything around by declaring a different congruence rule:

$$\frac{\neg Q' \Longrightarrow P = P' \qquad Q = Q'}{(P \vee Q) = (P' \vee Q')} \ (\textit{disj-cong2})$$

This would allow us to make the reverse definition:

$$f\,n = (f\ (n - 1) \vee n = 0)$$

This already shows that the congruence rules that we need might depend on the function we are defining. Note that the meaning of disjunction does not change.

One could argue that *disj-cong2* is unnatural, and that *disj-cong* should be enabled by default. Adding this rule will always make the termination proof simpler, since the recursive calls are restricted by an extra condition. However, as another consequence we get a weaker induction rule. Consider for example a function that checks if some element in a list satisfies some (fixed) predicate *test*:

$$\textit{testany}\ [] = \textit{False}$$
$$\textit{testany}\ (x \# xs) = (\textit{test}\ x \vee \textit{testany}\ xs)$$

Obviously, *testany* terminates just by structural recursion over the list, so *disj-cong* is not needed here. If we still add it, the function is not changed, but in the induction rule, the inductive hypothesis is now guarded by a condition:

$$\frac{P\ [] \qquad \bigwedge x\ xs.\ (\neg\ \textit{test}\ x \Longrightarrow P\ xs) \Longrightarrow P\ (x \# xs)}{P\ a}$$

When we do induction with this rule, we will always have to show $\neg\ \textit{test}\ x$ before we can apply the induction hypothesis. So in this case we get a better induction principle by avoiding the unnecessary congruence rule. Of course, for this example we can just use standard list induction, but in other situations the custom induction rule might be important.

These examples show that, in general, there is no "best" or "complete" set of congruence rules. The default setup in Isabelle is rather conservative, relying on the user to manually add rules when needed. Nonetheless, the basic set of predefined congruence rules often proves sufficient.

8 Extensions

The core recursion infrastructure described above is already quite powerful. In this section we describe some useful extensions: default values, tail recursion, pattern matching, mutual recursion and currying.

8.1 Default values

Recall that we model partial functions as underspecified total functions. Outside their domain, we cannot determine their value. Sometimes this is not desirable, as there may be a natural completion of the function that better suits the needs of the application.

Here, we note the difference between the *algorithm* that is specified by the recursive equations and the *function* that we define in the logic. While the algorithm cannot return anything when it does not terminate, the function actually has a value, and we can specify that value (which we call the *default value*) at definition time.

The only change that is needed is to replace the description operator *THE* which we used to define the function (cf. §3.2) by a variant that takes a default value:

THE-default :: $\alpha \Rightarrow (\alpha \Rightarrow bool) \Rightarrow \alpha$
THE-default d P = (if $\exists!x$. *P x* then *THE x*. *P x* else *d*)

Then we can define functions with a user-specified default value *d*, which may even depend on *x*. We can then derive the following theorem:

$x \notin dom_f \Longrightarrow f\,x = d\,x$

To motivate the use of default values, consider a function that checks some kind of certificate:

checker :: *cert* $\Rightarrow$ *bool*

The implementation of *checker* may be a very complicated algorithm, for which we can only prove partial correctness. Hence we have a theorem

$c \in dom_{checker} \Longrightarrow checker\ c = True \Longrightarrow P\ c$

for some interesting property *P*.

Now, if we can define *checker* such that it returns *False* when given a value that is not in the domain, then we can remove the domain condition from the theorem above, which can make subsequent reasoning simpler:

$checker\ c = True \Longrightarrow P\ c$

Note that default values are just a logical concept and have no operational meaning. If the function *checker* is run on something outside its domain, it will still loop instead of returning *False*. However, we have a theorem that it is equal to *False*, logically. Here the difference between the logical view and the algorithmic view of a function becomes very apparent.

8.2 Tail recursion

The partial simplification rules generated by the function package are guarded by domain conditions. If the function does not always terminate, it is usually not possible to remove them (recall the example $U\,x = U\,x + 1$, which is obviously inconsistent). However, there is an important special case for which unguarded recursion equations are derivable even for partial functions. This is the case when the function is tail-recursive, a fact that was first noticed and exploited by Manolios and Moore [35]. In HOL, tail-recursive functions could previously be defined by instantiating a *while* combinator, but that was a tedious manual process.

While it would be possible to automate the definition of tail-recursive functions using a *while* combinator, it turns out that we can achieve the same effect with the definition framework presented here, by deriving the unconstrained simplification rules afterwards. For this we use the default value feature described above, and give the function an arbitrary default value d that is independent from the input.

Now, the unconstrained recursion equation $f\,x = F\,f\,x$ can be proved as follows: For $x \in dom_f$, we just need to apply the partial simplification rule. Consider the case $x \notin dom_f$. By tail recursion we know that $F\,f\,x = f\ (g\ x)$, for some expression g. Now $g\ x$ cannot be in dom_f, since otherwise x would also be in dom_f. Since both x and $g\ x$ are not in dom_f, we have $f\,x = d = f\ (g\ x) = F\,f\,x$ which is our recursive equation.

This reasoning has been automated, and thus we can provide unconstrained recursion equations for the user, if the function is tail-recursive.

One important motivation for removing the domain conditions even for partial functions is that Isabelle/HOL's code generator can only handle unconditional equations. When we are able to derive them as theorems, then we can use all the existing code generation facilities [5,27] to convert our Isabelle/HOL specifications to ML or Haskell programs. Note that this translation only preserves partial correctness, as the resulting code may be nonterminating.

8.3 Pattern matching

So far, our example functions were given by just one equation with a variable as argument. Often, it is more convenient to write a function in multiple equations using pattern matching.

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

We could adopt the same restrictions in a theorem prover, and then compile away the pattern matching into nested case expressions. After the definition, the pattern matching equations must be proved from the equation with the case expression, and the induction rule must be modified to reflect the different cases. Slind goes this way in his thesis [45], where he describes an algorithm similar to those used in compilers.

However, in an extensible logical framework, the restriction to datatype patterns seems artificial. Isabelle/HOL contains various extensions that are not plain inductive datatypes, but behave similarly and suggest some form of pattern matching, e.g., records [38], and nominal datatypes [47]. Ideally, the function definition package deals with pattern matching in a general way.

One way to achieve this is by allowing patterns to be arbitrary expressions, and additionally have conditions attached to them. This radical approach was first suggested by John Harrison and implemented in HOL Light [28]. It again exploits the non-computational view of HOL: We are able to define the function and derive the recursive equations, even if we do not know how to compile the function as a functional program.

8.3.1 Arbitrary expressions as patterns

Up to now, the specification of the function was given by a single fixed-point equation of the form $f\,x = F\,f\,x$. With pattern matching, the specification consists of several conditional equations:

$$C_1\ vs_1 \Longrightarrow f\ (p_1\ vs_1) = r_1\, f\, vs_1$$
$$\vdots$$
$$C_n\ vs_n \Longrightarrow f\ (p_n\ vs_n) = r_n\, f\, vs_n$$

In each equation, p_i is the pattern, r_i is the right-hand side, and C_i is a condition. The function f may occur on the right-hand sides as a recursive call, but not in the patterns or conditions.

If we allow arbitrary expressions as patterns, we must generate proof obligations to ensure that the pattern matching uniquely defines a function. The proof obligations are as follows:

Completeness: Every value is matched by at least one pattern.
Compatibility: If a value is matched by two patterns, then the corresponding right hand-sides are equal in both cases.

Requiring pattern completeness may seem surprising at first, since we usually allow our functions to be partial. But the completeness property is necessary in order to justify the case distinction which is built into the induction rule. Incomplete patterns can usually be easily eliminated by mapping the missing cases to some arbitrary value.

The compatibility condition ensures that the equations do not contradict each other. This is a slightly weaker requirement than disjointness: We do allow patterns to overlap, but only if the overlap causes no harm.

Formally, completeness is described by the following elimination rule, which just expresses that any x must have at least one of the given forms:

$$\frac{\bigwedge vs_1.\ x = p_1\ vs_1 \Longrightarrow C_1\ vs_1 \Longrightarrow P \qquad \ldots \qquad \bigwedge vs_n.\ x = p_n\ vs_n \Longrightarrow C_n\ vs_n \Longrightarrow P}{P}$$

For each pair of equations i and j, the compatibility condition models the informal description above. If the patterns are non-overlapping, this condition is trivially satisfied:

$$\bigwedge vs_i\ vs_j'.\ p_i\ vs_i = p_j\ vs_j' \Longrightarrow C_i\ vs_i \Longrightarrow C_j\ vs_j' \Longrightarrow r_i\, f\, vs_i = r_j\, f\, vs_j'$$

8.3.2 Matching combinator

We can encode arbitrary pattern matching in a single definition using a special matching combinator:

$$MATCH\ ::\ (\gamma \Rightarrow bool \times \alpha \times \beta) \Rightarrow \beta \Rightarrow \alpha \Rightarrow \beta$$

$$MATCH\ M\ d\ x =$$
$$(\text{if } \exists!r.\ \exists\, v.\ M\ v = (True, x, r) \text{ then } THE\ r.\ \exists\, v.\ M\ v = (True, x, r) \text{ else } d)$$

The argument M is a *matching clause* which should be of the form $\lambda v.\ (C\ v, p\ v, r\ v)$, where C is a condition, p is a pattern and r is the corresponding right-hand side. For example the equation $x < y \Longrightarrow f\ (x, Suc\ y) = 2 * x + y$ is expressed by the matching clause $\lambda(x, y).\ (x < y, (x, Suc\ y), 2 * x + y)$. Note how the pattern variables are combined into a tupled abstraction.

The *MATCH* combinator takes a value and if the value matches the pattern, the corresponding right-hand side is returned. Otherwise, a default value is returned. By nesting

MATCH expressions, we can describe sequential pattern matching where patterns are tried in a fixed order.

A matching clause $\lambda v.\ (C\ v,\ p\ v,\ r\ v)$ does not tell us how to effectively *compute* the result of the match. However, it captures the logical essence of pattern matching. Since we are using the definite description operator *THE* again, the result must be uniquely defined for the match to succeed. This is always the case when the pattern p is injective, but injectivity is sometimes too strong a requirement: For example, when modeling α-equated lambda-terms as a nominal datatype, the lambda constructor is not injective: For example, $Lam\ [a].(Var\ a) = Lam\ [b].(Var\ b)$). Nevertheless it can make sense to do pattern matching on nominal datatypes, and the function can be applied here in principle. However, the resulting proof obligations can be tricky to solve as they sometimes require induction. Making (general recursive) function definitions over nominal datatypes work smoothly in practice is an area of future work.

8.4 Mutual recursion and currying

Our package implements mutual recursion by first reducing it to simple recursion on a suitable sum type. This is a standard trick, which was already described by Boyer and Moore [16], and adapted to higher-order logic by Slind [45], so we just give a small example: The functions *even* and *odd* with the equations

$$\begin{array}{lcl} even\ 0 & = & True \\ even\ (Suc\ n) & = & odd\ n \\ odd\ 0 & = & False \\ odd\ (Suc\ n) & = & even\ n \end{array}$$

are reduced to a single function $even\text{-}odd :: nat + nat \Rightarrow bool$ with the equations

$$\begin{array}{lcl} even\text{-}odd\ (Inl\ 0) & = & True \\ even\text{-}odd\ (Inl\ (Suc\ n)) & = & even\text{-}odd\ (Inr\ n) \\ even\text{-}odd\ (Inr\ 0) & = & False \\ even\text{-}odd\ (Inr\ (Suc\ n)) & = & even\text{-}odd\ (Inl\ n) \end{array}$$

Then the individual functions are defined as

$$\begin{array}{lcl} even\ n & = & even\text{-}odd\ (Inl\ n) \\ odd\ n & = & even\text{-}odd\ (Inl\ n) \end{array}$$

and we easily derive the original recursive equations from this.

We must also produce an appropriately modified induction principle. The (total) induction rule for *even* and *odd* involves two induction predicates P and Q:

$$\frac{P\ 0 \qquad \bigwedge n.\ Q\ n \Longrightarrow P\ (Suc\ n) \qquad Q\ 0 \qquad \bigwedge n.\ P\ n \Longrightarrow Q\ (Suc\ n)}{P\ a \wedge Q\ a}$$

We can handle currying in a similar way: If a function has multiple arguments, we first define the corresponding uncurried function which takes a tuple. From that function we then define the curried function and derive the equations.

These transformations can be used as wrappers around the core of the package. They reduce currying and mutual recursion to simple functions, such that the rest of the definition infrastructure just needs to handle a single function with one argument.

9 Limitations

In this section, we briefly discuss some general limitations of our package.

9.1 Higher-order nesting

We have seen how our package gracefully handles higher-order and nested recursion. However, by combining the two, we can take the difficulty to a new level and make the automation fail. Here is a simple example — yet another silly way of defining the constant zero function:

zero n = fun-pow n zero 0

It is easy to see that *zero 0 = id 0 = 0* and hence *zero n = zero* (*zero* (... (*zero 0*) ...) = *0*. The problem is that we cannot give a useful congruence rule for *fun-pow*, the function exponentiation. Intuitively, such a rule should express that in *fun-pow n f x*, the function f is called on the values *fun-pow i f x* for all $i < n$. But this contains the very same pattern again, which makes the extraction of recursive calls loop.

This example shows that the extraction of recursive calls using congruence rules is just an approximation that works well in practice but may also fail.

We can circumvent this problem by expanding the higher-order recursion into a mutual recursion by adding the recursion equations for *fun-pow* to the definition of *zero*:

zero n = zeropow n 0

zeropow 0 x = x
zeropow (*Suc n*) *x = zero* (*zeropow n x*)

Then we no longer need a congruence rule and can proceed in the normal way to prove termination of the mutual and nested recursion.

9.2 Undefinedness does not propagate

If we expect that the domain dom_f models the set of values where f terminates, it can be a little surprising to see that for the function

g x = U x

the associated domain dom_g is the universal set, although g calls U, whose domain is empty. The reason is that dom_g arises only from the analysis of the recursion in the definition of g, and a non-recursive function always terminates in our sense.

It is possible to change the analysis to a more global one, where the domain of g would also depend on the domain of U. However, recall that we introduced the domain primarily to simplify partial correctness proofs and not as a faithful model of termination with respect to some evaluation mechanism. Note that to obtain the latter, we must also settle on a fixed evaluation order and a fixed set of congruence rules.

It seems that the practical benefit of getting stronger induction and simplification rules outweighs the somewhat unintuitive property that undefinedness does not propagate.

9.3 Other forms of recursion

There are other forms of recursive definitions that are not based on well-founded recursion. One example is corecursion, where the result type of a function is a coinductive datatype, which can be infinite.

For example, the function

$$from\ n = n \# from\ (Suc\ n)$$

defines an infinite sequence of numbers. This definition can only work for coinductive lists, not for normal inductive ones, which are finite by construction.

Well-founded recursion cannot define functions like *from*, and other tools would be required to introduce them.

9.4 Code generation for partial functions

A practically relevant limitation of the domain predicate approach is the fact that only unconditional equations can be used by Isabelle's code generator [5, 27], which produces functional programs from logical specifications. However, the recursive equations for partial functions carry the domain conditions, which makes them currently unsuitable for code generation. A notable exception are the tail-recursive functions, for which unconditional equations can be generated (§8.2). Although many functions have a natural tail-recursive definition, this situation is unsatisfactory. We hope to address this issue in future work.

10 Related Work

General recursion in proof assistants. Generating termination conditions and induction schemes from recursive function definitions was already done by Boyer and Moore in NQTHM [15], the predecessor of ACL2 [29]. Today, ACL2 still works in essentially the same way: Functions must be proved total at definition time by giving the appropriate measure, which can sometimes be inferred by the system. As opposed to a definitional extension, recursion is built into the system itself and must be trusted. As ACL2 uses first-order logic, there is no higher-order recursion. ACL2 also supports the definition of (possibly partial) tail-recursive functions [35]. Then no termination proof is needed.

Both Isabelle and HOL4 [24] include (different versions of) the definitional recursion package TFL, a work by Slind [44, 45]. TFL supports the definition of total recursive functions by using the specialized fixed-point combinator *wfrec* and a well-founded relation given by the user. Proving termination amounts to showing that the relation is well-founded and recursive calls are decreasing. Optionally, termination arguments can be deferred by replacing the relation by its specification using a choice operator.

HOL Light [28] provides a similar mechanism, also based on a fixed-point combinator. Furthermore, by a clever combination with tail recursion, termination proof obligations only arise from non-tail calls, even if the function as a whole is not tail recursive. The drawback of this approach is that no induction principles can be generated. There is no general support for higher-order recursion.

In Coq [8], a package by Barthe, Forest, Pichardie, and Rusu [4] allows definitions in a manner similar to TFL. However, nested and higher-order recursion are not supported.

Partiality and domain predicates. The idea of generating an explicit description of a function's domain is by no means new, and appeared already in various previous approaches.

Finn, Fourman, and Longley [21] describe how partial functions can be axiomatized consistently in a total higher-order logic, by having their equations guarded by domain predicates. The domain predicates are again specified by recursive equations, but (as is justified semantically), a domain predicate for the domain predicate is not required.

Dubois and Donzeau-Gouge [19] replace the recursive domain with an inductive one, which makes the approach (in principle) amenable to implementation as a definitional extension to Coq. However, they did not provide an implementation.

Giesl [23] studies the use of functional induction for partial functions, and shows that it is applicable and useful. The partial induction rule is similar to the one we are using, and is proved sound with respect to the semantics of programs. As this employs a modified notion of truth for formulae, it is not directly applicable to higher-order logic, whose semantics is fixed. However, Giesl managed to extend existing induction provers with his calculus with little effort.

Bove and Capretta [10, 12] investigate how general recursion can be treated in constructive type theory, which by default only admits structural recursion. Their approach is also based on an inductively defined domain predicate which is similar to our dom_f. However, then the actual function is defined by *structural* recursion on that inductive predicate, which is possible in the dependently typed setting. This approach also underlies the implementation of Coq's function package [4]. The disadvantage is that there is no general type of partial recursive functions, but instead each function has its own private type of the form $\sigma \to dom_f\ \sigma \to \tau$. Either impredicativity [13] or a coinductive construction [14] can be used to overcome this. Note that in our simply-typed framework this is not an issue, since partial functions are just underspecified total functions (of type $\sigma \to \tau$).

Another related approach using a recursive domain predicate is given by Cowles, Greve and Young [18] for ACL2. Although the construction is different, similar ideas are present here, and a conditional equation and induction rule is proved in the end. Recently, Greve presented some refinements of the approach, together with an implementation [25].

A different approach for dealing with non-termination is to work in a logic that features a "native" notion of partiality. One such logic is domain theory, where any computable function can easily be defined, since general fixed points exist. On the other hand, reasoning in domain theory comes with a certain overhead, since induction is restricted to admissible predicates. This leads to additional admissibility and definedness proof obligations that can make reasoning harder.

Nested recursion. Nested recursion is currently not well-supported in theorem provers. Slind's TFL package provides some support using a provisional induction rule [46], but the resulting proof obligations are clumsy.

The approach sketched by Dubois and Donzeau-Gouge [19] supports a user-specified *post condition* that can encode the property required for the termination proof of a nested recursive definition. Later, Krstić and Matthews [32] suggested a very similar notion they called *inductive invariant*. Inductive invariants are given by the user at definition time and are used to approximate the results of nested recursive calls to make the termination proof work. However, no convenient reasoning principles are given for them, and one must resort to general well-founded induction, which is more awkward to use than functional induction.

In our approach, such properties are simply expressed as ordinary lemmas about possibly partial functions, constrained by a domain predicate and provable by the partial induc-

tion rule. This follows Giesl's approach [22,23], but it does not require a new notion of truth, since all constructions happen in standard HOL.

Bove and Capretta [11] can only support nested recursion by defining the domain and the function simultaneously, which requires extending the underlying theory to support simultaneous inductive-recursive definitions as described by Dybjer [20]. In contrast, our classical setting avoids this issue and works in standard higher-order logic, since the domain is not required for the function definition but only introduced for the purpose of convenient reasoning. However, since we make use of the definite description operator, it is not clear whether a similar construction could work in constructive type theory, which does not have such an operator.

In ACL2, nested recursive functions must be transformed by adding extra tests to the body of the function, before they can be defined. These checks can later be removed when the function is executed due to a special mechanism [26].

11 Conclusion

We have presented a methodology for recursive function definitions, based on inductive definitions of the function's graph and domain. Our treatment of nested recursion facilitates defining and reasoning about such functions, since the domain predicate allows to express partial correctness statements naturally, and the partial induction principle is available early.

The implementation of our approach has become the principal tool for defining recursive functions in Isabelle/HOL since version 2007. Together with the termination provers [17] that can find termination proofs automatically for most of the function definitions that occur in practice, our package considerably facilitates defining and reasoning about recursive functions.

Acknowledgements I would like to thank Tobias Nipkow and Makarius Wenzel for many fruitful discussions. Jasmin Christian Blanchette, Amine Chaieb, Norbert Schirmer, and Christian Urban commented on drafts of this paper. The anonymous referees gave valuable feedback.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Academic Press, 1986.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer Verlag, 2004.
4. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming (FLOPS 2006)*, volume 3945 of *Lecture Notes in Computer Science*, pages 114 – 129. Springer Verlag, 2006.
5. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer Verlag, 2000.
6. S. Berghofer and M. Reiter. Formalizing the logic-automaton connection. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 147–163. Springer Verlag, 2009.

7. 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)*, volume 1690 of *Lecture Notes in Computer Science*, pages 19–36. Springer Verlag, 1999.
8. Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer Verlag, 2004.
9. A. Bove. Programming in Martin-Löf type theory: Unification - A non-trivial example, November 1999. Licentiate Thesis, Department of Computer Science, Chalmers University of Technology.
10. A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 39–58. Springer Verlag, 2002.
11. A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer Verlag, 2001.
12. A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.
13. A. Bove and V. Capretta. Recursive functions with higher-order domains. In P. Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2007)*, volume 3461 of *Lecture Notes in Computer Science*, pages 116–130. Springer Verlag, 2005.
14. A. Bove and V. Capretta. Computation by prophecy. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA 2007)*, volume 4583 of *Lecture Notes in Computer Science*, pages 70–83. Springer Verlag, 2007.
15. R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
16. R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.
17. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag, 2007.
18. J. Cowles, D. Greve, and W. Young. The while-language challenge: First progress. In *ACL2 Workshop Proceedings*, 2007.
19. 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.
20. P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbolic Logic*, 65(2):525–549, 2000.
21. S. Finn, M. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.
22. J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, Aug. 1997.
23. J. Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 26(1):1–49, 2001.
24. M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for Higher Order Logic*. Cambridge University Press, 1993.
25. D. Greve. Assuming termination. In *ACL2 Workshop Proceedings*, 2009.
26. D. A. Greve, M. Kaufmann, P. Manolios, J. S. Moore, S. Ray, J.-L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1):15–46, 2008.
27. F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 08 2007.
28. J. Harrison. The HOL Light theorem prover. `http://www.cl.cam.ac.uk/users/~jrh13/hol-light`.
29. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
30. A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 589–603. Springer Verlag, 2006.
31. A. Krauss. Certified size-change termination. In F. Pfenning, editor, *Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Computer Science*, pages 460–476. Springer Verlag, 2007.
32. 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)*, volume 2758 of *Lecture Notes in Computer Science*, pages 253–269. Springer Verlag, 2003.
33. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages (PoPL 2001)*, pages 81–92, 2001.

34. Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.
35. P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.
36. C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.
37. O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.
38. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. C. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs '98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 349–366. Springer Verlag, 1998.
39. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
40. T. Nishihara and Y. Minamide. Depth first search. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Depth-First-Search.shtml`, June 2004. Formal proof development.
41. S. Owens and K. Slind. Adapting functional programs to higher-order logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.
42. L. C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–170, 1985.
43. L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer Verlag, 1994.
44. 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)*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer Verlag, 1996.
45. K. Slind. *Reasoning About Terminating Functional Programs*. PhD thesis, Institut für Informatik, Technische Universität München, 1999.
46. K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLS 2000)*, volume 1869 of *Lecture Notes in Computer Science*, pages 498–518. Springer Verlag, 2000.
47. C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.
48. C. Walther. On proving the termination of algorithms by machine. *J. Artificial Intelligence*, 71(1):101–157, 1994.
49. M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2002.