Inductive Invariants for Nested Recursion[⋆]

Sava Krstić and John Matthews

OGI School of Science & Engineering at Oregon Health & Sciences University

Abstract. We show that certain input-output relations, termed *inductive invariants* are of central importance for termination proofs of algorithms defined by nested recursion. Inductive invariants can be used to enhance recursive function definition packages in higher-order logic mechanizations. We demonstrate the usefulness of inductive invariants on a large example of the BDD algorithm APPLY. Finally, we introduce a related concept of *inductive fixpoints* with the property that for every functional in higher-order logic there exists a largest partial function that is such a fixpoint.

1 Introduction

To prove termination of a recursively defined program, we usually argue that its execution on any given input x will make recursive calls only to arguments smaller than x, where "smaller" is specified by some wellfounded relation. When the pattern of recursive calls is simple enough, we can derive from the program text a finite set of inequalities sufficient to guarantee termination.

This simple scheme has a difficulty with nested recursion—the occurrence of recursive calls within recursive calls, as in the following simple program (of type $\mathsf{nat} \Rightarrow \mathsf{nat}$) taken from [17].

$$g\,x \equiv \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } g\,(g\,(x-1)) \tag{1}$$

Even though this program clearly terminates for all inputs, the reason for it is not just a couple of easy inequalities. The simplest rigorous explanation would probably be proving by strong induction that "for all x, the algorithm terminates with input x and returns the value 0". This pattern of reasoning requires knowing something in advance about the function being defined, and proving that fact simultaneously with termination. Proving termination of functions defined by nested recursion can thus be a challenge for a human prover, not to mention automated tools.

A recursive declaration for a function $f\colon A \Rightarrow B$ is naturally represented by a functional $F\colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$, so that defining f amounts—more or less—to showing that F has a unique fixpoint. The inductive invariants we introduce in this paper are specific invariants of the functional. They are predicates on

⋆ The research reported in this paper was supported by the National Science Foundation Grants EIA-0072761 and CDA-9703218, Hewlett-Packard Corporation, and Intel Corporation.

$A \Rightarrow B$ that are defined by input-output relations (predicates on $A \times B$). We have found that "the extra information about the function being defined that needs to be known in order to prove termination" is invariably in the form of an invariant relation. That is, there is an inductive invariant playing an essential part in every (nested) termination proof that we have seen. It is as important a part of the termination proof as is the wellfounded relation that captures the dependency pattern of recursive calls. The inductive invariant is sometimes easy to guess, but ultimately it is the user's responsibility to come up with, just as is the wellfounded relation.

The main contribution of the paper is perhaps the discovery of this role of inductive invariants for structuring termination proofs for nested recursive definitions. While there have been attempts in the literature to formulate general principles behind such termination proofs [4, 6, 19], the methodology of inductive invariants appears not to have been observed as yet.

The paper is organized as follows. In Section 2, we give an overview of the state-of-the-art recursion package *recdef* by Konrad Slind [15, 18]. This puts us in the framework of higher-order logic and in the context of Slind's work on nested recursion. In Section 3, we introduce inductive invariants and prove Theorem 1, which opens the way to incorporating them into *recdef*. We also demonstrate inductive invariants in a few standard small examples.

Section 4 presents a small theory of recursive definitions in HOL. We introduce the notion of *inductive fixpoints* of functionals that is arguably the closest natural concept in HOL to "terminating functional program". We also discuss *contraction conditions* as an abstraction of *recdef*'s proof obligations when presented with a non-nested recursive definition. We then extend the notion of contraction condition with an inductive invariant parameter. In Theorem 2 we show that a functional satisfying a contraction condition necessarily has an inductive fixpoint. Theorem 2 is independent of *recdef* and can be used as an alternative in situations when *recdef* is difficult to apply. This development leads naturally to general notion of a fixpoint operator in higher-order logic that given any functional produces its largest (in an appropriate sense) inductive fixpoint. Existence of such a fixpoint is stated as Theorem 3.

In Section 5 we apply the theory of Section 4 to a large and difficult example of the imperative program Apply that is a key component of any efficient BDD (Boolean decision diagram) package. This example also reveals that nested recursion is not a rarity: it occurs in disguise—because of the hidden state variable—in every recursively defined imperative program in which there is a sequence of commands containing two recursive calls. Another example of nested recursion that occurs when modeling imperative programs is given in [3].

Section 6 draws some conclusions and comparisons with related work.

2 A Summary of *Recdef*

In this section we give a quick description of the theory underlying *recdef*, a powerful recursive function definition package designed by Slind [15, 17–19].

Given as input a recursive declaration of the form

$$f\,x = M, \tag{2}$$

where M is some term that contains no free variables other than f and x, *recdef* attempts to produce a valid HOL definition of a function f that satisfies equation (2) for every x. It is convenient to abstract the variables in M and use the associated functional $F = \lambda f\,x.\ M \colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ as the input. Thus equation (2) translates into the *fixpoint equation* (or *recursion equation*)

$$\forall x.\ f\,x = F\,f\,x. \tag{3}$$

2.1 Partiality and WFREC

We will use the notation $f \upharpoonright D$ for the restriction of a function $f \colon A \Rightarrow B$ on a subset D of A; this partial function is represented in *HOL* as a total function, and is defined by

$$(f \upharpoonright D)\,x \;\equiv\; \textbf{if } x \in D \textbf{ then } f\,x \textbf{ else } \mathsf{Arb},$$

where $\mathsf{Arb} \equiv \varepsilon z.\mathsf{True}$ is an arbitrary element of the domain of f. We will write $f =_D g$ as an abbreviation for $f \upharpoonright D = g \upharpoonright D$, which is equivalent to $\forall y \in D.\ f\,y = g\,y$. We will also write $\rho^{-1}x$ as an abbreviation for the set $\{y \mid \rho\,y\,x\}$, where ρ is a binary relation. This notation will be used throughout the paper.

The principle of definition by wellfounded recursion (e.g., [22], Theorem 10.19) can be expressed as follows: Given a functional $F \colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ and a wellfounded relation ρ on A, there exists a unique function $f \colon A \Rightarrow B$ satisfying

$$\forall x.\ f\ x = F\ (f \upharpoonright \rho^{-1}x)\ x. \tag{4}$$

Nipkow has given an elegant formalization of this principle in HOL [17]. The theory defines a "controlled fixpoint operator" WFREC, and contains the theorem

$$\mathsf{WF}\,\rho \longrightarrow \forall x.\ \phi\ x = F\ (\phi \upharpoonright \rho^{-1}x)\ x, \tag{5}$$

where $\phi = \mathsf{WFREC}\,\rho\,F$ and WF is the wellfoundedness predicate.

We do not need to know the exact definition of the function WFREC. The upshot is that, for any given wellfounded relation ρ, the function $\phi = \mathsf{WFREC}\,\rho\,F$ satisfies (4). Given a recursive declaration (2), we can start by introducing the function ϕ as a newly defined constant (for a suitable ρ), obtain equation (4) for free, and then work to transform that equation into the desired form of (3).

2.2 Termination Conditions

The *recdef* algorithm begins with theorem (5) in the form

$$(f = \mathsf{WFREC}\,\rho\,F) \wedge (\mathsf{WF}\,\rho) \longrightarrow \forall x.\ f\ x = F\ (f \upharpoonright \rho^{-1}x)\ x, \tag{6}$$

leaving ρ unspecified at the beginning, and analyzes the structure of the term M (recall that $F = \lambda f\, x.\, M$) in order to find occurrences $f\, t_1, \ldots, f\, t_m$ of recursive calls as subterms of M. For simplicity, let us assume that every occurrence of f in M is applied to some argument. For each of the m calling sites, *recdef* also gathers the conjunction of conditionals Γ_i controlling it, and extracts the *termination condition* $TC_i \equiv \Gamma_i \longrightarrow \rho\, t_i\, x$. For example, the declaration (1) produces two termination conditions (where we write $<$ in place of ρ):

$$x \neq 0 \longrightarrow x - 1 < x \quad \text{and} \quad x \neq 0 \longrightarrow g\,(x-1) < x. \tag{7}$$

By definition of function restriction, we have

$$\rho\, t_i\, x \longrightarrow (f \upharpoonright \rho^{-1} x)\, t_i = f\, t_i.$$

Thus, it is possible by a bottom-up traversal of the term M to transform theorem (6) into one where all occurrences of $f \upharpoonright \rho^{-1} x$ are replaced with f, at the price of adding termination conditions as assumptions:

$$(f = \mathsf{WFREC}\, \rho\, F) \wedge (\mathsf{WF}\, \rho) \wedge TC_1 \wedge \cdots \wedge TC_m \longrightarrow f\, x = F\, f\, x. \tag{8}$$

The desired recursion theorem for $\phi = \mathsf{WFREC}\, \rho\, F$ is now in sight: it only remains to eliminate the antecedent of implication (8), which amounts to proving the termination conditions TC_i for a suitably instantiated wellfounded relation ρ. Assuming ρ has been instantiated, the system can try to prove each termination condition $\forall x.\, TC_i$, or pass this obligation to the user. However, if the original recursive declaration is nested, then there will be occurrences of f in the termination conditions, and it is not clear at all how to systematically discharge them. So after disposing of immediately provable termination conditions, we are left with a theorem of the form

$$TC \longrightarrow \phi\, x = F\, \phi\, x, \tag{9}$$

where TC is a formula that may contain occurrences of ϕ.

2.3 Slind's Method for Nested Recursion

Normally, to prove a property of a recursively defined function f, one would use the recursion equation (3) as the main auxiliary result. Unfortunately, we are now in a position that we need to prove TC in order to derive (3). But note that there is no genuine circularity here: the function has already been defined (as $\mathsf{WFREC}\, \rho\, F$), and it may well be possible to prove that it satisfies the property TC before one proves that it satisfies (3). Slind discovered that the constrained recursion equation (9) that we have at our disposal can sometimes be used to prove the termination condition TC by induction. The induction principle he uses is not standard, however. It is given by a *provisional induction theorem* that can be automatically generated at the same time the constrained equation (9) is derived [18, 19].

Space does not permit us to go into a description of provisional induction theorems. The reader is referred to the cited Slind's work for details and case studies. They include termination proofs for nested recursive definitions as complex as the first order unification algorithm.

The exact scope of Slind's method is difficult to characterize. Just for illustration, the method deals easily with the simple function given by equation (1), but not so with the following, just slightly changed equation:

$$g\,x \equiv \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } g\,(x-1) + g\,(g\,(x-1)) \tag{10}$$

3 Inductive Invariants

In both examples, given by recursive equations (1) and (10), the nested termination condition is $x \neq 0 \longrightarrow g\,(x-1) < x$. What makes it provable by Slind's method in one case, but not in the other, is the fact that this condition expresses an *inductive invariant* for the first equation, but not for the second.

Definition 1. *A predicate $S\colon A \Rightarrow B \Rightarrow \mathsf{bool}$ is an* inductive invariant *of the functional $F\colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ with respect to the wellfounded relation ρ iff the following condition is satisfied:*

$$\forall f\,x.\ (\forall y.\ \rho\,y\,x \longrightarrow S\,y\,(f\,y)) \longrightarrow S\,x\,(F\,f\,x) \tag{11}$$

Predicates of the form $S\colon A \Rightarrow B \Rightarrow \mathsf{bool}$ express input-output relations for functions $f\colon A \Rightarrow B$. We will say that f *satisfies* S if $S\ x\ (f\ x) = \mathsf{True}$ holds for all $x \in A$. Thus, if f satisfies an inductive invariant S, it follows from (11) that $F\,f$ satisfies S as well. In this sense, inductive invariants are invariants of their associated functionals.

Checking inductive invariance is an inductive method of proving properties of $\mathsf{WFREC}\,\rho\,F$, as the following key result shows.

Theorem 1. *If ρ is wellfounded and S is an inductive invariant of F with respect to ρ, then $\mathsf{WFREC}\,\rho\,F$ satisfies S.*

Proof. Denote $\phi = \mathsf{WFREC}\,\rho\,F$. By instantiating f in (11) with $\phi \restriction \rho^{-1}x$, we get

$$\forall x.\ (\forall y.\ \rho\,y\,x \longrightarrow S\,y\,\big((\phi \restriction \rho^{-1}x)\,y\big) \longrightarrow S\,x\,(F\,(\phi \restriction \rho^{-1}x)\,x).$$

This is equivalent to

$$\forall x.\ (\forall y.\ \rho\,y\,x \longrightarrow S\,y\,(\phi\,y)) \longrightarrow S\,x\,(\phi\,x) \tag{12}$$

because $(\phi \restriction \rho^{-1}x)\,y = \phi\,y$ is clearly true when $\rho\,y\,x$, and $F\,(\phi \restriction \rho^{-1}x)\,x = \phi\,x$ holds from equation (5). The formula (12) is exactly what is needed for a wellfounded-inductive proof of the formula $\forall x.\ S\,x\,(\phi\,x)$ saying that ϕ satisfies S. $\square$

Theorem 1 suggests the possibility of treating nested recursive definitions automatically with an extension of *recdef*: Use *recdef* to generate termination conditions, then check if they can be expressed as input-output relations, and if so then prove their inductive invariance.

We conjecture that termination proofs of most functions defined by nested recursion can be naturally based on some inductive invariant property. Termination conditions themselves are not necessarily inductive invariants, or not the most convenient ones. Rather, there exists some inductive invariant that *implies* the termination conditions. Coming up with such an invariant in every concrete nested recursive definition seems as fundamental for understanding its termination as coming up with an appropriate wellfounded relation. So just like the user-supplied wellfounded relations are essential for *recdef*, a user-supplied input-output relation (inductive invariant) could be essential for an extension of *recdef* that will treat nested cases successfully.

3.1 Examples

We give several small examples to demonstrate ubiquity of inductive invariants. In Section 5, we will give a more substantial example. The interested reader can find yet another example at the beginning of the termination proof of the unification algorithm given in [19].[1]

Nested Zero. The functional

$$G\,g\,x \equiv \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } g\,(g\,(x-1)) \tag{13}$$

corresponds to the example in equation (1). The termination condition can be written as $g\,x < x+1$, and it is an inductive invariant. (More precisely, the inductive invariant is defined by: $S\,x\,y$ iff $y < x+1$.) Indeed, one needs to check

$$\forall g\,x.\ (\forall y < x.\ g\,y < y+1) \longrightarrow G\,g\,x < x+1.$$

This is clearly true for $x = 0$, while for $x > 0$ it reads as

$$\forall g.\ (\forall y < x.\ g\,y < y+1) \longrightarrow g\,(g\,(x-1)) < x+1$$

and the conclusion of this formula follows by using the assumption twice.

The slightly different functional

$$G'\,g\,x \equiv \textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } g\,(x-1) + g\,(g\,(x-1))$$

corresponding to the equation (10) has the same termination condition $g\,x < x+1$, but now this condition is not an inductive invariant. We leave checking this fact to interested readers.

Note also that $g\,x = 0$ is an inductive invariant for both G and G'.

[1] The proof is due to Manna and Waldinger; its first mechanized version is due to Paulson. See [19] for history and references.

McCarthy's Ninety-One Function. This classic example is a recursive definition of a function of type $\mathsf{nat} \Rightarrow \mathsf{nat}$, given by the functional

$$F\,f\,x \equiv \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f(f(x+11))$$

The termination conditions for F are

$$x \leq 100 \longrightarrow x + 11 \prec x \quad \text{and} \quad x \leq 100 \longrightarrow f(x+11) \prec x,$$

for a suitable wellfounded relation $\prec$. A relation that works is the ordering defined by $1 \succ 2 \succ 3 \succ \cdots \succ 99 \succ 100$ and $100 \succ 100 + i$ for all $i \geq 1$. This ordering discharges the first termination condition, while the second termination condition can be rewritten as

$$11 \leq x \leq 111 \longrightarrow x < f(x) + 11.$$

This expresses an obvious input-output relation and one can check that the relation is in fact an inductive invariant. We will check that an even stronger (but simpler) relation, namely

$$S\,x\,y \equiv x < y + 11 \tag{14}$$

is an inductive invariant for F. Thus, we need to prove

$$z < (F\,f\,z) + 11 \tag{15}$$

assuming

$$x < f(x) + 11 \tag{16}$$

holds for all $x \prec z$. For $z > 100$, the relation (15) reduces to $z < (z - 10) + 11$, which is true. In the remaining case $z \leq 100$, the relation (15) rewrites as

$$z < f\,(f\,(z + 11)) + 11. \tag{17}$$

Now we can assume that (16) holds for all $x > z$, since $x \prec z$ and $x > z$ are equivalent when $z \leq 100$. In particular, $z+11 < f\,(z+11)+11$ must hold, giving us $z < f\,(z + 11)$. Instantiating x with $f\,(z + 11)$ in (16) is now legitimate and gives us $f\,(z + 11) < f\,(f\,(z + 11)) + 11$, so (17) follows by transitivity of $<$.

HOL *Version of the While Combinator.* This example illustrates the utility of inductive invariants even for non-nested recursion. In *Isabelle/HOL* [15], the function $\mathsf{while}\,b\,c \colon A \Rightarrow A$ is defined for any predicate $b \colon A \Rightarrow \mathsf{bool}$ and function $c \colon A \Rightarrow A$, and it is a fixpoint of the functional

$$W\,f\,x \equiv \textbf{if } b\,x \textbf{ then } f\,(c\,x) \textbf{ else } x \tag{18}$$

One can check that, given any two predicates $P \colon A \Rightarrow \mathsf{bool}$ and $Q \colon A \Rightarrow \mathsf{bool}$, the following relation S is an inductive invariant for W:

$$\begin{array}{rl} S\,x\,y & \equiv \ P\,x \\ & \wedge \ (\forall z.\, P\,z \wedge b\,z \longrightarrow P\,(c\,z) \wedge c\,z \prec z) \\ & \wedge \ (\forall z.\, P\,z \wedge \neg(b\,z) \longrightarrow Q\,z) \\ & \longrightarrow Q\,y \end{array}$$

The theorem `while_rule` supplied with *Isabelle/HOL* says precisely that $\mathsf{while}\, b\, c$ satisfies every relation S of this form, provided the parameter $\prec$ is a wellfounded relation. Thus, `while_rule` is essentially an instance of Theorem 1.

4 Fixpoint Operators in HOL

When we use *recdef* to prove termination of an algorithm, we represent the algorithm as a functional $F\colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$, find a wellfounded relation ρ, and then prove that the function $f = \mathsf{WFREC}\, \rho\, F$ satisfies the recursion equation

$$\forall x.\; f\, x = F\, f\, x.$$

Simple examples like $F\, f\, x \equiv f\,(x-1)$, where every constant function is a fixpoint, and $\mathsf{WFREC}\;(<)\, F$ is just one of them, show that the truth of the recursion equation does not quite correspond to termination.However, in situations when *recdef* succeeds (at least in the non-nested cases), the fixpoint is unique, and even more is true. In this section we define a simple HOL concept of *inductive fixpoints* that is related to *recdef* and that corresponds more closely to termination.

4.1 Inductive Fixpoints

The graph S_h of a function h is an input-output predicate that only h satisfies: $S_h\, x\, y$ iff $y = h\, x$. Condition (11) with S_h in place of S is equivalent to

$$\forall f\, x.\; f =_{\rho^{-1}x} h \longrightarrow F\, f\, x = h\, x. \tag{19}$$

Definition 2. *We say that h is an* inductive fixpoint *of F if S_h is an inductive invariant of F for some ρ (equivalently: if equation (19) holds).*

Lemma 1. *If h is an inductive fixpoint of F, then*

(a) h is the unique fixpoint of F;
(b) $h = \mathsf{WFREC}\, \rho\, F$ for some wellfounded relation ρ.

Proof. *(a)* If h' is another fixpoint of F, then we can prove $\forall x.\; h'\, x = h\; x$ by wellfounded induction using (19) with f instantiated with h'.
(b) Suppose ρ is such that (19) holds and let $\phi = \mathsf{WFREC}\, \rho\, F$. Instantiating f with $\phi \restriction \rho^{-1}x$ in (19), we get

$$\forall x.\; (\phi \restriction \rho^{-1}x) =_{\rho^{-1}x} h \longrightarrow F\,(\phi \restriction \rho^{-1}x)\, x = h\, x.$$

The antecedent here is clearly equvalent to $\phi =_{\rho^{-1}x} h$, while the consequent is, in view of the WFREC recursion equation (5), equivalent to $\phi\, x = h\, x$. Now $\phi = h$ follows by wellfounded induction. $\square$

Consider the example

$$F\,f\,x \equiv \textbf{if } x = 0 \textbf{ then } f(1) - f(0) \textbf{ else } f(x-1).$$

The constant zero function is equal to $\mathsf{WFREC}\,(<)\,F$ and it is also the only fixpoint of the functional F. Yet, the constant zero function is not an inductive fixpoint of F. The example demonstrates that $\mathsf{WFREC}\,\rho\,F$ is not necessarily an inductive fixpoint even when it is a unique fixpoint.

4.2 Contraction Conditions

The (unrestricted) *contraction condition* for the functional F with respect to the wellfounded relation ρ is

$$\forall f\,g\,x.\ f =_{\rho^{-1}x} g \longrightarrow F\ f\ x = F\ g\ x. \tag{20}$$

Harrison proved in [7] that the contraction condition implies the unique existence of a fixpoint of F. In Theorem 2 below, we show that this condition actually implies that $\mathsf{WFREC}\,\rho\,F$ is an inductive fixpoint of F. For now we check that in the case of non-nested recursive declarations the contraction condition follows from the conjunction of termination conditions generated by *recdef*. Indeed, assuming a wellfounded relation ρ is fixed, we can think of *recdef* as transforming the theorem

$$f\ x = F\ (f \restriction \rho^{-1}x)\ x \tag{21}$$

into the theorem

$$TC_1 \wedge \cdots \wedge TC_m \longrightarrow f\,x = F\,f\,x. \tag{22}$$

It is true that f in these equations is the constant $\mathsf{WFREC}\,\rho\,F$, but that fact is never used in the sequence of steps (transforming the right-hand side progressively, while keeping the left-hand side fixed) that lead from (21) to (22). Consequently, the same sequence of steps would transform the trivial theorem

$$F\ (f \restriction \rho^{-1}x)\ x = F\ (f \restriction \rho^{-1}x)\ x,$$

in which f is free, into the theorem

$$TC_1 \wedge \cdots \wedge TC_m \longrightarrow F\ (f \restriction \rho^{-1}x)\ x = F\,f\,x.$$

Thus, if the (universally quantified) termination conditions are true, then

$$\forall f\,x.\ F\,(f \restriction \rho^{-1}x)\,x = F\,f\,x$$

is true as well. It is easy to see that this last equation is equivalent to saying that F satisfies the contraction condition with respect to ρ

The contraction condition does not hold for even the simplest nested cases, like the one in (13). However, a weaker form of it that uses an inductive invariant to restrict one of the arguments may still be provable, and still be strong enough to guarantee the existence of an inductive fixpoint. We define the *restricted* *contraction condition* for the functional F with respect to a wellfounded relation ρ and an inductive invariant S to be the formula

$$\forall f\, g\, x.\ f =_{\rho^{-1}x} g \wedge (\forall y.\ S\, y\, (g\, y)) \longrightarrow F\ f\ x = F\ g\ x. \tag{23}$$

Note that the contraction condition (20) is a special case of (23) corresponding to the trivial (constantly true) inductive invariant S.

Theorem 2. *Suppose the restricted contraction condition (23) is satisfied and S is an inductive invariant of F associated with ρ. Then $\mathsf{WFREC}\,\rho\,F$ is an inductive fixpoint of F.*

Proof. Instantiate (23) with $g = \mathsf{WFREC}\,\rho\,F$. The second conjunct is true by Theorem 1, so we obtain

$$\forall f\, x.\ f =_{\rho^{-1}x} g \longrightarrow F\, f\, x = F\, g\, x. \tag{24}$$

Instantiating f here with $g \restriction \rho^{-1}x$ we obtain $F\,(g \restriction \rho^{-1}x)\, x = F\, g\, x$, and then $g\, x = F\, g\, x$ by combining with the constrained recursion equation (4). Now (24) can be rewritten as

$$\forall f\, x.\ f =_{\rho^{-1}x} g \longrightarrow F\, f\, x = g\, x,$$

finishing the proof. □

4.3 Partial Inductive Fixpoints

The techniques described in these sections readily generalize to allow termination proofs for recursive algorithms that terminate only on a specific subset of their input type. We just summarize the main definitions and results. The proofs are straightforward modifications of those already seen, as is their formalization in *Isabelle/HOL*.

Given a functional $F\colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ and a subset D of A, we say that h is an *inductive fixpoint of F on D*, if there exists a wellfounded relation ρ such that

$$\forall f\, x.\ x \in D \wedge f =_{D \cap (\rho^{-1}x)} h \longrightarrow F\, f\, x = h\, x. \tag{25}$$

The generalized Lemma 1 asserts that such an inductive fixpoint satisfies the guarded recursive equation

$$\forall x.\ x \in D \longrightarrow h\, x = F\, h\, x, \tag{26}$$

that $h =_D \mathsf{WFREC}\,\rho\,F$ for some wellfounded relation ρ, and that h is unique in the sense that $h =_D h'$ holds for any other function h' satisfying (25).

The definition of an *inductive invariant for F on D* is given by the same equation (11), except that the variables x, y need to be restricted to D. Theorem 1 generalizes to say that every such inductive invariant is satisfied by $\mathsf{WFREC}\,\rho\,F$ for every input in D.

The *restricted contraction condition for F with respect to ρ, S, and D* is

$$\forall f\, g.\ \forall x \in D.\ f =_{D \cap (\rho^{-1} x)} g \wedge (\forall y \in D.\ S\, y\, (g\, y)) \longrightarrow F\ f\ x = F\ g\ x. \qquad (27)$$

The generalized Theorem 2 states that when this contraction condition is satisfied and S is an inductive invariant for F on D, then $\mathsf{WFREC}\, \rho\, F$ is an inductive fixpoint of F on D.

4.4 HOL Fixpoints

If h and h' are inductive fixpoints of F on D and D' respectively, let us say that h' *extends* h if $D \subseteq D'$ and $h =_D h'$.

Theorem 3. *For any functional $F\colon (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ there exists an inductive fixpoint h (defined on some domain $D \subseteq A$) that extends all other inductive fixpoints of F.*

We omit the proof for space reasons. It is more involved than the others in this paper, and we have not yet mechanized it.

Theorem 3 shows that there is a natural fixpoint operator that associates a partial function to every recursive declaration in HOL. The properties and usefulness of this fixpoint operator are left for future research.

5 Case Study: The BDD Apply function

In this section we survey the proof of termination of the imperative BDD program Apply. This proof has been verified in *Isabelle/HOL* and reported on in [9]. Our *HOL* proof did not use the *recdef* mechanism, but is instead based on the techniques described in the previous section.

Binary decision diagrams (BDDs) are a widely used representation of Boolean functions. Intuitively, a BDD is a finite rooted directed acyclic graph in which every node except the special nodes $TrueNode$ and $FalseNode$ is labeled by a *variable* and has two children: *low* and *high*. Special nodes represent the constant Boolean functions, and the function f_u represented by any other node u is defined recursively by $f_u = \textbf{if}\ x\ \textbf{then}\ f_l\ \textbf{else}\ f_h$, where x is the variable associated with u, and l, h are its left and right children respectively. Bryant [5] originally proved that every function is represented by a unique reduced ordered BDD, where *reduced* means that no two nodes represent the same function, and *ordered* means that variable names are totally ordered and that every node's variable name precedes the variable names of its children. Efficient BDD packages implement reduced ordered BDDs. An abstract, but detailed presentation of such a package of programs is given in [1]. Our work [9] contains a *HOL* model of a significant part of the package. Referring to these papers for more detail, we will now describe just the minimum required to define the Apply program.

The global state used by any BDD package contains a pool of BDD nodes. We assume there is an abstract type Node representing node addresses, and a type Var of variables. A primitive procedure *active*: Node $\Rightarrow$ bool indicates the presence of a node in the current state, and the accessor functions *var*, *low*, *high* take a node as an argument and return the associated variable and children. What these functions return if the argument is not an active node is left unspecified. For simplicity we will assume that Var is the type of natural numbers whose natural ordering corresponds to the ordering of variables needed to implement the concept of ordered BDDs.

The BDD routine Mk takes a variable x and two nodes l, h as inputs and returns a node u such that $var(u) = x$, $low(u) = l$, and $high(u) = h$. If a node with these three attributes already exists in the state, the state is left unchanged; otherwise Mk adds u to the state. We gloss over the details how Mk tests whether it needs to add a node to the existing state and the possibility that Mk can raise an out-of-memory exception.

The crucial routine Apply takes a binary operation *op*: bool $\times$ bool $\Rightarrow$ bool and two nodes u and v, and returns a node w which represents the Boolean function f_w specified by

$$\forall x.\ f_w\, x = op(f_u\, x, f_v\, x). \tag{28}$$

A recursive declaration of Apply is given in pseudocode in Figure 1.

```
Appply[$T$]($op, u, v$) =
if $u, v \in \{TrueNode, FalseNode\}$ then $op(u, v)$
else if $var(u) = var(v)$ then
    $w \longleftarrow$ Mk($var(u)$, Apply($op, low(u), low(v)$), Apply($op, high(u), high(v)$))
else if $var(u) < var(v)$ then
    $w \longleftarrow$ Mk($var(u)$, Apply($op, low(u), v$), Apply($op, high(u), v$))
else
    $w \longleftarrow$ Mk($var(v)$, Apply($op, u, low(v)$), Apply($op, u, high(v)$))
return $w$
```

Fig. 1. The program Apply, as in [1], omitting the memoization part, inessential for the purpose of proving termination. The global variable T is the table of BDD nodes. The *var* accessor function is assumed to return 0 only for $TrueNode$ and $FalseNode$.

Clearly, the variable *op* is of little significance for proving the termination of Apply. Assuming *op* is constant, we can think of Apply as being a function of type Node $\times$ Node $\times$ State $\Rightarrow$ State $\times$ Node.[2] Recursion makes Apply one of the most complicated programs in the package. Pondering the algorithm in Figure 1, one realizes that even a hand proof of termination requires effort. The ultimate reason for termination is clear: in an ordered BDD (and these are the only ones we would like to consider), the level (that is, the *var* value) decreases when

[2] In [9], we show how to use monadic interpretation to hide "state threading" and translate imperative programs to visibly equivalent *HOL* counterparts.

passing to child nodes, so in all recursive calls of APPLY the level decreases either for both node arguments, or decreases for the "higher", while the other stays the same. Thus, in order to prove that the arguments decrease in recursive calls, it is necessary to work with a restricted set of states, described by a predicate goodSt that needs to be preserved by APPLY. A workable invariant goodSt asserts that the associated BDD to each active node is ordered and reduced. Clearly, we cannot expect termination for all input-"good state" pairs. We need to add at least the restriction that the two input nodes be active in the input state. These restrictions define a subset D of Node × Node × State on which we can reasonably expect termination by means of the wellfounded relation defined by the measure that associates to an input-state triple (u, v, T) the maximum of the values $var(u)$, $var(v)$ in T.

Next we need to deal with nesting. Nesting is not immediately seen in Figure 1 because the state is not explicitly mentioned in the program text, being thus an extra hidden argument. Consider line 6; in expanded form, this piece of code could read like this:

```
$6_1$    $l_u \longleftarrow low(u)$
$6_2$    $h_u \longleftarrow high(u)$
$6_3$    $x \longleftarrow var(u)$
$6_4$    $l \longleftarrow \text{APPLY}(op, l_u, v)$
$6_5$    $h \longleftarrow \text{APPLY}(op, h_u, v)$
$6_6$    $w \longleftarrow \text{MK}(x, l, h)$
```

If we made the state explicit, these lines would look as follows, with primes denoting the appropriate modifications of functions representing programs:

```
$6_1$    $(l_u, T_1) \longleftarrow low'(u, T)$
$6_2$    $(h_u, T_2) \longleftarrow high'(u, T_1)$
$6_3$    $(x, T_3) \longleftarrow var'(u, T_2)$
$6_4$    $(l, T_4) \longleftarrow \text{APPLY}'(op, l_u, v, T_3)$
$6_5$    $(h, T_5) \longleftarrow \text{APPLY}'(op, h_u, v, T_4)$
$6_6$    $(w, T_6) \longleftarrow \text{MK}'(x, l, h, T_5)$
```

Lines 6_4 and 6_5 expose the nesting in the definition of h. Desugaring the pattern-matching of tuples, we see h is defined as

$$h = \mathsf{fst}(\text{APPLY}'(op, h_u, v, \mathsf{snd}(\text{APPLY}'(op, l_u, v, T_3)))).$$

To prove termination, we need to find input-output properties of APPLY that are sufficient to show that the measure decreases in each recursive call, and then prove that these properties are inductive invariants of the functional defining APPLY. It turns out that the input-output predicate S defined by

$$\begin{aligned} S\,(u, v, T)\,(w, T') \equiv\ & \text{active nodes in } T \text{ are active in } T' \\ & \wedge\ w \text{ is active in } T' \\ & \wedge\ var'(w, T') \leq var'(u, T), var'(v, T) \end{aligned} \tag{29}$$

is such an inductive invariant. We have carried out the proof of inductive invariance and the proof of the corresponding restricted contraction condition in *Isabelle/HOL* [9]. Other formalizations of BDD algorithms are reported in [8, 20, 21].

6 Conclusion and Related Work

We have described a simple method for proving termination of functions defined by nested recursion. For a given functional F representing the recursive declaration, the user is required to supply a wellfounded relation ρ and an inductive invariant S. The user then has two possibilities to complete the definition.

(A) Generate termination conditions associated with F and ρ (e.g., using the *recdef* definition package) and prove that these conditions are true for all functions satisfying S.
(B) Prove that the restricted contraction condition (23) holds for F, ρ, S.

The challenge of justifying nested recursive definitions has attracted a great deal of attention and [4, 6, 12, 16, 19] are but a few examples of interesting case studies and general methods. They are presented in various formal system, but are all related to our work for the simple reason that inductive invariants (even if not recognized as such) are at the core of most of the known termination proofs of nested recursion.

Our work builds on Slind's [19]. Slind has discovered a method of proving termination conditions based on a specific induction theorem and a constrained recursion theorem. This method can be seen as a variation of the alternative (A) above; instead of proving that a certain property is an inductive invariant, it attempts to prove directly that the property is satisfied by $\mathsf{WFREC}\,\rho\,F$. Our method seems to be at least of comparable power, but is conceptually much simpler.

Alternative (B) is a general mathematical method. It can be used for termination proofs done by hand, but since the underlying theory has been formalized, (B) can be used in formal proofs as well, as we demonstrated in Section 5. Note, however, that since *recdef* normally automatically performs the task that amounts to checking the contraction condition, alternative (B) should be used only if one has difficulties with *recdef*. Such difficulties may arise because *recdef*'s working depends heavily on the requisite simplifiers of the theorem prover, and in particular on an adequate supply of *congruence rules*. There are also cases where not all occurrences of f in the recursive declaration (2) are applied; for example, f may rather occur as an argument to some higher-order function.[3] In such cases, if it is not clear what congruence theorems are needed by *recdef*, it may be advantageous to go directly to the proof of the contraction condition.

[3] A simple example: formalization of the common definition of Catalan numbers $C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$ would require a congruence rule for the higher-order function $sum\, n\, f \equiv (f\,0) + (f\,1) + \cdots + (f\,n)$.

While the search for an inductive invariant can be the most demanding part of a termination proof, often in this search one does not need to go further than the termination conditions themselves. The method described by Giesl [6] follows a similar route: heuristic generation of "induction lemmas" that correspond to nested termination conditions, then proving their "partial correctness", which amounts to inductive invariance. A high level of automation, which is the main virtue of Giesl's method, is also its limitation; in complex cases like the one discussed in Section 5 a human-supplied inductive invariant may be necessary. Compared with Giesl's method, ours is more general, and our proofs are considerably simpler than those in [6].

Giesl and Slind [6, 19] make the point that contrary to common wisdom it is generally possible to prove termination of functions defined by nested recursion without simultaneously proving their correctness/specification. Our work corroborates this point, but note that the notions of correctness and specification are vague. The fact is that *some form* of specification is invariably being used, namely the inductive invariant! As for the full specification, it may even be totally unusable for the termination proof. An example is the Apply algorithm specified by equation (28). The full specification uses an interpretation function associating Boolean functions to BDD nodes, but what is needed for the termination proof of Apply is inductive invariance of the much simpler predicate given in (29).

Contraction conditions are a standard way of proving fixpoint theorems (*à la* Banach) in various contexts involving a metric. Contraction conditions along a wellfounded relation were introduced by Harrison [7] and called *admissibility conditions*. A more general version is introduced by Matthews in [11] to support recursive function definitions over types with coinductive structure.

Modeling partiality and program termination in HOL can be done in several ways, surveyed by Müller and Slind in [14]. The most accurate representation can be achieved in HOLCF [13], the HOL version of domain theory. Following *recdef*, we have adopted the simplest approach where partial functions are modeled as total functions taking an arbitrary value on arguments outside the specified domain of definition. A limitation of this approach is the difficulty in giving unique interpretations (as partial functions) of recursively specified algorithms. We offer a solution with the concept of inductive fixpoints introduced in Section 4.4. The topic seems to deserve further study. For a related concept of *optimal fixpoints*, see [10]. For recent related work in type theory, see [2, 4].

Acknowledgment. We would like to thank the anonymous referees for their detailed and useful comments.

References

1. H. R. Andersen. An Introduction to Binary Decision Diagrams (Lecture Notes). `www.itu.dk/people/hra/bdd97.ps.gz`, October 1997.
2. A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées francophones des langages applicatifs, JFLA'02*. INRIA, 2002.

3. Y. Bertot, V. Capretta, and K. D. Barman. Type-theoretic functional semantics. In V. A Carreno, C. A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLS 2002)*, volume 2410 of *LNCS*, pages 83–98. Springer, 2002.
4. 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 *LNCS*, pages 121–135. Springer, 2001.
5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, 1997.
7. J. Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 200–213. Springer, Berlin,, 1995.
8. F. W. von Henke et al. Case Studies in Meta-Level Theorem Proving. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLS 1998)*, volume 1749 of *LNCS*, pages 461–478. Springer, 1998.
9. S. Krstić and J. Matthews. Verifying BDD algorithms through monadic interpretation. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI 2002)*, volume 2294 of *LNCS*, pages 182–195. Springer, 2002.
10. Z. Manna and A. Shamir. The optimal approach to recursive programs. *Communications of the ACM*, 20(11):824–831, 1977.
11. J. Matthews. Recursive function definition over coinductive types. In Y. Bertot et al., editor, *Theorem Proving in Higher Order Logics (TPHOLS 1999)*, volume 1690 of *LNCS*, pages 73–90. Springer, 1999.
12. J. S. Moore. A mechanical proof of the termination of Takeuchi's function. *Information Processing Letters*, 9:176–181, 1979.
13. O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
14. O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal, 40(10)*, 1997.
15. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
16. L. C. Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2(1):63–74, 1986.
17. K. Slind. Function definition in higher order logic. In J. von Wright et al., editor, *Theorem Proving in Higher Order Logics (TPHOLS 1996)*, volume 1125 of *LNCS*, pages 381–397. Springer, 1996.
18. K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik, Technische Universität München, 1999.
19. K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLS 2000)*, volume 1869 of *LNCS*, pages 498–518. Springer, 2000.
20. R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. Technical Report TR-00-29, The University of Texas at Austin, Department of Computer Sciences, 2000.
21. K. N. Verma et al. Reflecting BDDs in Coq. In J. He and M. Sato, editors, *Proc. 6th Asian Computing Science Conference (ASIAN)*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000.
22. G. Winskel. *The Formal Semantics of Programming Languages: an Introduction*. MIT Press, 1993.

Appendix: Proof of Theorem 3

Lemma 2. *If h and h' are inductive fixpoints of F on D and D' respectively, and if $D \subseteq D'$, then $h =_D h'$.* $\square$

Let us say that D is a *domain* for F if F has an inductive fixpoint on D. In view of Lemma 2, it suffices to prove that there exists the largest domain for F.

For every domain D, there exists a wellfounded relation ρ (not necessarily unique) such that $\mathsf{WFREC}\,\rho\,F$ is an inductive fixpoint of F on D. We will say that (D, ρ) is a *germ* of F in such cases. The following fact about germs is straightforward to check.

Lemma 3. *Suppose (D, ρ) is a germ of F and D' is a downward closed subset of D with respect to ρ. Then (D', ρ') is a germ of F, where ρ' is the restriction of ρ on D'.* $\square$

Consider the ordering on the set of germs given by: $(D, \rho) \preceq (D', \rho')$ iff $D \subseteq D'$ and the restriction of ρ' on D is equal to ρ. For every chain $\mathcal{C}$ in this ordering, consider the pair $(\cup D, \cup\rho)$, where the unions are taken over all elements of the chain $\mathcal{C}$. It is easy to see that this pair is a germ and an upper bound for $\mathcal{C}$. By Zorn's Lemma, there exists a maximal germ, say (D, ρ). We claim that D is the largest domain for F.

Assuming the contrary of the claim, suppose (D', ρ') is germ of F such that D' is not contained in D. Now, there exists $z \in D' \setminus D$ such that all smaller elements (with respect to ρ') than z in D' belong to D as well. In view of Lemma 3, it is no loss of generality to assume that $D' = E \cup \{z\}$, where $E \subseteq D$ and z is the greatest element in D'.

Let $\phi = \mathsf{WFREC}\,\rho\,F$ and $\phi' = \mathsf{WFREC}\,\rho'F$. By Lemma 3, E is a domain for F, as downward closed in D'. Now E is a subdomain of both D and D', so by Lemma 2 we obtain that the restrictions of ϕ and ϕ' on E coincide.

Define the function ψ by

$$\psi\,x \equiv \textbf{if } x \in D \textbf{ then } \phi\,x \textbf{ else } \phi'\,x.$$

Since ϕ and ϕ' have equal restrictions on $E = D \cap D'$, we have that $\psi =_D \phi$ and $\psi =_{D'} \phi'$. Thus, ψ is an inductive fixpoint of F on both D and D'.

We want to prove that ψ is an inductive fixpoint of F on $D \cup D' = D \cup \{z\}$. Using the (obviously wellfounded) relation $\sigma = \rho \cup \rho'$, it suffices to prove that

$$f =_{(D\cup\{z\})\cap(\sigma^{-1}x)} \psi \longrightarrow F\,f\,x = \psi\,x \tag{30}$$

holds for every f and every $x \in D \cup \{z\}$. Consider first the case when $x \in D$. Then $(D \cup \{z\}) \cap (\sigma^{-1}x) = D \cap \rho^{-1}x$ and (30) follows since ϕ is an inductive fixpoint of F on D. In the remaining case when $x = z$, we have $(D \cup \{z\}) \cap (\sigma^{-1}x) = E$ and (30) follows since ϕ is an inductive fixpoint of F on D'.

[**end of appendix**]