Automated Synthesis of Induction Axioms for Programs with Second-Order Recursion

Markus Aderhold

Technische Universität Darmstadt, Germany
`aderhold@informatik.tu-darmstadt.de`

Abstract. In order to support the verification of programs, verification tools such as ACL2 or Isabelle try to extract suitable induction axioms from the definitions of terminating, recursively defined procedures. However, these extraction techniques have difficulties with procedures that are defined by second-order recursion: There a first-order procedure f passes itself as an argument to a second-order procedure like *map*, *every*, *foldl*, etc., which leads to *indirect* recursive calls. For instance, second-order recursion is commonly used in algorithms on data structures such as *terms* (variadic trees). We present a method to automatically extract induction axioms from such procedures. Furthermore, we describe how the induction axioms can be optimized (i. e., generalized and simplified). An implementation of our methods demonstrates that the approach facilitates straightforward inductive proofs in a verification tool.

1 Introduction

For the verification of programs one usually needs to show that a program behaves as expected for *all* possible inputs. Therefore formal specifications of expected properties often contain universal quantifications. In order to prove a universal formula $\forall x : \tau.\ \psi[x]$, many theorem provers employ *explicit induction* [4,5,7,10,11,16]. Given a well-founded relation $\succ$ on the domain τ that the quantification ranges over (i. e., a relation without infinite chains $q_0 \succ q_1 \succ q_2 \succ \ldots$), the general schema of *well-founded induction* permits the inference

$$\frac{\forall x : \tau.\ \big(\forall x' : \tau.\ x \succ x' \to \psi[x']\big) \to \psi[x]}{\forall x : \tau.\ \psi[x]}\ . \qquad (1)$$

For a concrete well-founded relation $\succ$, we call (1) an *induction axiom*.[1]

From the infinitely many well-founded relations $\succ$ that exist for each non-trivial data type τ, in general only few relations are suitable to prove $\forall x : \tau.\ \psi[x]$ for a given formula ψ. Thus finding an appropriate well-founded relation $\succ$ for a formula ψ is an essential challenge in program verification.

One particularly successful approach to finding a suitable induction axiom for a formula ψ is *recursion analysis*, which was pioneered by Boyer and Moore [5].

[1] The term "axiom" emphasizes that well-foundedness of $\succ$ need not necessarily be proved within the formal system, but may be assumed when applying (1).

Variants have been developed that are used in current theorem provers, see [4,9,10,13] for instance. The idea is to exploit the strong relationship between recursion and induction by uniformly extracting well-founded relations from terminating, recursively defined procedures occurring in formula ψ.

In this paper we describe a method for recursion analysis of procedures with *second-order recursion.* A procedure f is defined by second-order recursion if f calls a second-order[2] procedure g using f in a function argument for g, e. g., $g(f, \ldots)$ [8,12]. Typical examples of second-order recursion arise in algorithms on variadic trees such as terms; e. g., applying a substitution to a term, counting the variables in a term, computing the size of a term (cf. Figs. 1 and 2). The following examples illustrate how recursion analysis works and why second-order recursion is a challenge for current theorem provers.

Example 1. Fig. 1(a) shows an example program that defines data types *bool*, $\mathbb{N}$, and $list[@A]$ (where $@A$ is a type variable) by enumerating the respective data constructors *true*, *false*, 0, *succ*, ø, and "::". Each argument position of a data constructor is assigned a *selector* function; e. g., selector *pred* denotes the predecessor function. Procedure *sum* computes the sum of all numbers in a list k. An induction axiom for proofs about *sum* can be directly read off from the recursive definition:

$$\frac{\forall k : list[\mathbb{N}].\ k = \emptyset \rightarrow \psi[k] \qquad \forall k : list[\mathbb{N}].\ k \neq \emptyset \wedge \psi[tl(k)] \rightarrow \psi[k]}{\forall k : list[\mathbb{N}].\ \psi[k]} \tag{2}$$

The base case of the recursion becomes a base case of the induction. The recursive call $sum(tl(k))$ gives rise to the induction hypothesis $\psi[tl(k)]$ in the step case.◇

Example 2. In Fig. 1(b), procedure *map* is a second-order procedure that gets a first-order function f as argument. Procedure *varcount* uses second-order recursion to count the number of variables in a term t, modeled by data type *term*. (Expressions of the form $?cons(t)$ check if t denotes a value of the form $cons(\ldots)$.) While it is easy to see that $?var(t)$ is a base case of the recursion, the arguments of the recursive calls of *varcount* are not obvious from the source code. However, this information about the indirect recursion via *map* is necessary to synthesize an induction axiom for *varcount*. ◇

Isabelle builds on the concept of so-called *congruence rules* that tell the system which function calls need to be evaluated [12,8]. For example, a procedure call $map(f, k)$ requires evaluation of $f(z)$ for all $z \in k$. From this knowledge one can infer that *varcount* is recursively called on all terms $z \in args(t)$. A drawback of congruence rules is that the *user* needs to state and prove the corresponding congruence theorems. Moreover, for a fixed set of congruence rules—possibly supplied by libraries—the resulting induction axioms may easily become suboptimal (e. g., due to weak induction hypotheses) [8].

[2] As in [3], we define the order $o(\tau)$ of base types τ like $\mathbb{N}$ or $list[\mathbb{N}]$ as 0; the order of a functional type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is $1 + \max_i o(\tau_i)$ for a base type τ.

```
(a) structure bool <= true, false
    structure ℕ <= 0, succ(pred : ℕ)
    structure list[@A] <= ø, ::(hd : @A, tl : list[@A])
    procedure sum(k : list[ℕ]) : ℕ <=
    if k = ø then 0 else hd(k) + sum(tl(k))

(b) structure variable.symbol <= variable(varID : ℕ)
    structure function.symbol <= func(funcID : ℕ)
    structure term <=
      var(vsym : variable.symbol),
      apply(fsym : function.symbol, args : list[term])
    procedure map(f : @A → @B, k : list[@A]) : list[@B] <=
    if k = ø then ø else f(hd(k)) :: map(f, tl(k))
    procedure varcount(t : term) : ℕ <=
    if ?var(t) then 1 else sum(map(varcount, args(t)))
```

Fig. 1. A functional program with (a) the first-order procedure *sum* and (b) the second-order procedure *map* and second-order recursion in procedure *varcount*

The contributions of this paper

(1) allow the automated extraction of induction axioms from procedures that are defined by second-order recursion (e. g., procedure *varcount*) and
(2) facilitate the optimization (i. e., generalization and simplification) of induction axioms, which permits more straightforward inductive proofs.

The optimization also helps to reveal the essence of the recursion structure of a procedure. This supports the heuristic selection of an induction axiom for a formula ψ (as ψ usually involves more than just one procedure). However, such a heuristic selection is beyond the scope of this paper.

The input for our methods is the source code of the procedures and their termination proofs. In particular, our approach does not require additional user input such as congruence theorems. It has been implemented and integrated into √eriFun, a semi-automated verifier for functional programs [16].

In Sect. 2 we give a brief overview over the programming language and some terminology that we use afterwards. Sect. 3 describes the synthesis of so-called *quantification procedures* that we use to formulate induction hypotheses. The synthesis of induction axioms is presented in Sect. 4. We describe techniques for their optimization in Sect. 5 and compare our methods with related techniques in Sect. 6. We conclude with experimental results in Sect. 7.

2 Programming Language and Terminology

We briefly summarize the relevant features of √eriFun's input language $\mathcal{L}$ [1,15] that roughly corresponds to the second-order fragment of ML or Haskell with strict evaluation; additional details can be found in [1,15].

$\mathcal{L}$ offers definition principles for freely generated polymorphic data types, for first-order and second-order procedures that operate on these data types, and for statements about the data types and procedures. A *base type* is a type variable $@A$ or an expression of the form $\mathit{str}[\tau_1, \ldots, \tau_k]$, where $\tau_1, \ldots, \tau_k$ are base types and str is a k-ary type constructor ($k \geq 0$). A *type* is a base type or an expression of the form $\tau_1 \times \ldots \times \tau_k \to \tau$ for types $\tau_1, \ldots, \tau_k, \tau$. *Type constructors* are defined by expressions of the following form:

$$\texttt{structure}\ \mathit{str}[@A_1, \ldots, @A_k] <=\ \ldots,\ \mathit{cons}(\mathit{sel}_1 : \tau_1, \ldots, \mathit{sel}_n : \tau_n),\ \ldots$$

The τ_j are base types, and str may only occur as $\mathit{str}[@A_1, \ldots, @A_k]$ in the τ_j. Each *cons* is called a *data constructor* and the sel_j are called *selectors*.

Let $\Sigma(P)$ denote the signature of all function symbols defined by an $\mathcal{L}$-program P. As usual, $\mathcal{T}(\Sigma(P), \mathcal{V})$ denotes the set of all *terms* over $\Sigma(P)$ and a set $\mathcal{V}$ of variables. We write $\mathcal{T}(\Sigma(P))$ instead of $\mathcal{T}(\Sigma(P), \emptyset)$ for the set of all *ground terms* over $\Sigma(P)$. $\Sigma(P)^c \subset \Sigma(P)$ contains all data constructors of P. A *literal* is an *if*-free Boolean term or the negation *if b then false else true* of such a term. $\mathcal{CL}(\Sigma(P), \mathcal{V})$ is the set of *clauses* over $\Sigma(P)$, i. e., the set of all finite sets of literals. For a term $t \in \mathcal{T}(\Sigma(P), \mathcal{V})$, we let $\Pi(t) \subset \mathbb{N}^*$ denote the set of all positions of t, i. e., $\Pi(t)$ comprises the positions of all subterms of t. We write $t|_\pi$ for the subterm of t at position $\pi \in \Pi(t)$.

For a ground type[3] τ, $\mathbb{V}(P)_\tau$ denotes the "values" of type τ: If τ is a ground base type, $\mathbb{V}(P)_\tau := \mathcal{T}(\Sigma(P)^c)_\tau$, and for each ground type $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, $\mathbb{V}(P)_\tau$ contains all closed (i. e., no free variables) λ-expressions of type τ; e. g., $\lambda t : \mathit{term}.\ \mathit{varcount}(t) \in \mathbb{V}(P)_{\mathit{term} \to \mathbb{N}}$.

The call-by-value interpreter $\mathsf{eval}_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)$ defines the operational semantics of $\mathcal{L}$ [1] by mapping ground terms $t \in \mathcal{T}(\Sigma(P))_\tau$ to values $\mathsf{eval}_P(t) \in \mathbb{V}(P)_\tau$. It is a partial function, because some procedures in program P may not terminate. A universally quantified formula of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$, where $b \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\mathit{bool}}$, is *true* iff all procedures in P terminate and $\mathsf{eval}_{P'}(b[q_1, \ldots, q_n]) = \mathit{true}$ for each terminating program $P' \supseteq P$ and all $q_1, \ldots, q_n \in \mathbb{V}(P')$.[4]

We implicitly assume procedure bodies to be in η-long form; e. g., $\mathit{map}(f, \mathit{tl}(k))$ abbreviates $\mathit{map}(\lambda z : @A.\ f(z),\ \mathit{tl}(k))$ in Fig. 1, because $f =_\eta \lambda z : @A.\ f(z)$. The following definition formalizes the notion "$f(q)$ requires the evaluation of $g(q')$":

Definition 1. *For a procedure or λ-expression f with body B_f and parameters $x_1, \ldots, x_n$, a procedure or λ-expression g, and $q_1, \ldots, q_n$, $q'_1, \ldots, q'_m \in \mathbb{V}(P)$, we write $f(q_1, \ldots, q_n) \rhd g(q'_1, \ldots, q'_m)$ iff B_f contains a subterm $h(t'_1, \ldots, t'_m)$ under some call context[5] C such that for $\sigma := \{x_1/q_1, \ldots, x_n/q_n\}$, $\sigma(h) =_\eta g$, $\mathsf{eval}_P(\sigma(c)) = \mathit{true}$ for all $c \in C$, and $q'_j = \mathsf{eval}_P(\sigma(t'_j))$ for all $j = 1, \ldots, m$. We write $f(q_1, \ldots, q_n) \rhd_g g(q'_1, \ldots, q'_m)$ iff $f(q_1, \ldots, q_n) \rhd h_1(\ldots) \rhd \ldots \rhd h_k(\ldots) \rhd g(q'_1, \ldots, q'_m)$ such that $h_i \neq_\eta g$ for all $i = 1, \ldots, k$.*

For example, $\mathit{map}(\mathit{varcount},\ t_1 :: t_2 :: t_3 :: \emptyset) \rhd \mathit{varcount}(t_1)$.

[3] A *ground (base) type* is a (base) type without type variables; e. g., $\mathit{list}[\mathbb{N}]$.

[4] Program P' may define additional data types and procedures to instantiate the x_i.

[5] $C \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ consists of the conditions in B_f that lead to the call $h(\ldots)$.

```
procedure $every(f : @A \to bool,\ k : list[@A]) : bool$ <=
$if\ k = \emptyset\ then\ true\ else\ if\ f(hd(k))\ then\ every(f, tl(k))\ else\ false$
procedure $foldl(f : @A \times @B \to @A,\ x : @A,\ k : list[@B]) : @A$ <=
$if\ k = \emptyset\ then\ x\ else\ foldl(f,\ f(x, hd(k)),\ tl(k))$
procedure $groundterm(t : term) : bool$ <=
$if\ ?var(t)\ then\ false\ else\ every(groundterm, args(t))$
procedure $termsize(t : term) : \mathbb{N}$ <=
$if\ ?var(t)\ then\ 1\ else\ foldl(\lambda n : \mathbb{N}, s : term.\, n + termsize(s),\ 1, args(t))$
```

Fig. 2. Second-order recursion in procedures *groundterm* and *termsize*

3 Quantification Procedures

Quantification procedures are system-generated procedures that iterate over certain values z and check if a given predicate p is satisfied for all these values z.

Quantification Procedures for Data Types. Consider the usual structural induction axiom for terms: In the base case, one proves that $\psi[t]$ holds if t is an arbitrary variable. In the step case, t is of the form $f(t_1, \ldots, t_n)$ and one proves $\psi[t]$ under the induction hypothesis that "$\psi[t_i]$ holds for all $i = 1, \ldots, n$". In general a program does not contain procedures to access the i-th element of list $args(t) = t_1 :: \ldots :: t_n :: \emptyset$ or to quantify over all elements of list $args(t)$. Hence we assume that for each data type $str[@A]$ a *quantification procedure*

$$\texttt{procedure}\ forall.str(p : @A \to bool,\ x : str[@A]) : bool \tag{3}$$

is synthesized that returns *true* iff $p(z)$ holds for all items $z : @A$ in x. PVS and √eriFun synthesize such quantification procedures automatically [11,1].

Example 3. For data type $list[@A]$, Fig. 3(a) shows quantification procedure *forall.list* that checks if some predicate p on $@A$ is satisfied for all elements $z : @A$ of a list k. Thus the axiom for structural induction on terms can be expressed (and automatically extracted by PVS and √eriFun) as

$$\frac{\begin{array}{l}\forall t : term.\ ?var(t) \to \psi[t] \\ \forall t : term.\ ?apply(t) \wedge forall.list(\lambda s : term.\, \psi[s],\ args(t)) \to \psi[t]\end{array}}{\forall t : term.\ \psi[t]}$$

where the induction hypothesis $forall.list(\ldots)$ states that $\psi[s]$ may be assumed for all terms s in list $args(t)$. ◇

Quantification Procedures for Second-Order Procedures. As Example 2 shows, the recursion analysis for procedure *varcount* needs to find out which arguments z the second-order procedure *map* calls its first-order parameter $f := varcount$ with. The induction hypothesis in the induction axiom for *varcount* will then quantify over all these arguments z to ensure $\psi[z]$.

```
(a) procedure $forall.list(p : @A \to bool,\ k : list[@A]) : bool <=$
    $if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall.list(p, tl(k))\ else\ false$

(b) procedure $forall.map(p : @A \to bool,\ f : @A \to @B,\ k : list[@A]) : bool <=$
    $if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall.map(p, f, tl(k))\ else\ false$
    procedure $forall.every(p, f : @A \to bool,\ k : list[@A]) : bool <=$
    $if\ k = \emptyset\ then\ true$
          $else\ \ if\ p(hd(k))\ then\ if\ f(hd(k))\ then\ forall.every(p, f, tl(k))\ else\ true$
                                  $else\ \ false$
    procedure $forall.foldl(p : @A \times @B \to bool,\ f : @A \times @B \to @A,$
                            $x : @A,\ k : list[@B]) : bool <=$
    $if\ k = \emptyset\ then\ true$
          $else\ \ if\ p(x, hd(k))\ then\ forall.foldl(p, f, f(x, hd(k)), tl(k))\ else\ false$
```

Fig. 3. Automatically synthesized quantification procedures

For that purpose we introduce a new concept, namely quantification procedures *forall.proc* for second-order procedures *proc*. For the sake of readability, we define *forall.proc* for second-order procedures *proc* with one first-order parameter f and an (optional) second formal parameter x. This definition can be generalized to more parameters in a straightforward way [1].

Definition 2. *For each terminating second-order procedure*

$$\texttt{procedure}\ proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc} <= B_{proc}$$

the quantification procedure *forall.proc* for *proc is defined by*

$$\texttt{procedure}\ forall.proc(p : \tau_1 \times \ldots \times \tau_m \to bool,\\ f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : bool <= \mathsf{ALL}_f(B_{proc})$$

where

$$\begin{aligned}
&\mathsf{ALL}_f(v) := true\\
&\mathsf{ALL}_f(f(t_1, \ldots, t_m)) := p(t_1, \ldots, t_m) \wedge \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_m)\\
&\mathsf{ALL}_f(g(t_1, \ldots, t_n)) := \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_n)\\
&\mathsf{ALL}_f(h(\lambda \boldsymbol{y}.\, t,\, t')) := \mathsf{ALL}_f(t') \wedge forall.h(\lambda \boldsymbol{y}.\, \mathsf{ALL}_f(t),\ \lambda \boldsymbol{y}.\, t,\ t')\\
&\mathsf{ALL}_f(if\ t_1\ then\ t_2\ else\ t_3) := \mathsf{ALL}_f(t_1) \wedge if\ t_1\ then\ \mathsf{ALL}_f(t_2)\ else\ \mathsf{ALL}_f(t_3)
\end{aligned}$$

for any variable v, any first-order function $g \neq if$, $g \neq f$, and any second-order procedure h (including proc). We write $\boldsymbol{y}$ as an abbreviation of $y_1, \ldots, y_k$, and $A \wedge B$ abbreviates "if A then B else false".

Quantification procedure *forall.proc* checks if $p(z_1, \ldots, z_m)$ holds for all tuples $(z_1, \ldots, z_m)$ that occur as arguments of f-calls:

Example 4. Procedure *forall.map* shown in Fig. 3 checks if $p(z)$ is satisfied for all elements z of list k, as procedure *map* applies f to all elements z of k. $\diamondsuit$

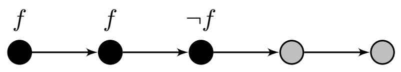

Fig. 4. Procedure *every* examines only the black elements of this list

Example 5. Procedure *every* in Fig. 2 checks if $f(z)$ is satisfied for all elements z of list k. As soon as an element z is encountered with $\neg f(z)$, procedure *every* stops with result *false*. This is illustrated in Fig. 4, where *every* evaluates $f(z)$ only for the black elements of the list. Consequently, procedure *forall.every* in Fig. 3 checks if $p(z)$ is satisfied for the first n elements of k, where $n \in \{1, \ldots, |k|\}$ is the smallest index such that f is *not* satisfied for the n-th element of k. (If there is no element z with $\neg f(z)$, then $n := |k|$, the length of k.) $\diamondsuit$

Example 6. Procedure *forall.foldl* checks if $p(a, b)$ is satisfied for all pairs (a, b) that f is applied to by *foldl*. $\diamondsuit$

The following lemma asserts that the quantification procedures according to Definition 2 compute the expected result. It demands that p and f be *fresh* functions, which means that these functions do not occur in the body of *proc* or in the bodies of auxiliary procedures for *proc*. (Alternatively, one can imagine p and f as uniquely labeled to distinguish these function calls from hard-coded function calls in the procedure bodies.)

Lemma 1. *For all* $\mathsf{x} \in \mathbb{V}(P)$ *and all fresh functions* $\mathsf{p} \in \mathbb{V}(P)$ *and* $\mathsf{f} \in \mathbb{V}(P)$*:*

(1) $\mathsf{eval}_P(\mathit{forall.proc}(\mathsf{p}, \mathsf{f}, \mathsf{x})) \in \{\mathit{true}, \mathit{false}\}$
(2) $\mathsf{eval}_P(\mathit{forall.proc}(\mathsf{p}, \mathsf{f}, \mathsf{x})) = \mathit{true} \iff \mathsf{eval}_P(\mathsf{p}(q_1, \ldots, q_m)) = \mathit{true}$ *for all* $q_1, \ldots, q_m \in \mathbb{V}(P)$ *with* $\mathit{proc}(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$

Proof. The proof is given in [1] (Sect. 3.2.2). $\square$

4 Synthesis of Induction Axioms

In order to synthesize an induction axiom for a procedure

$\texttt{procedure}\ p(x : \tau) : \tau' \texttt{ <= } B_p$

we analyze the recursive calls in the body B_p of procedure p. In case of second-order recursion, the indirect recursive calls are nested in λ-expressions, so in general we need to analyze a subterm t of B_p.

A *result term of* t is a maximal subterm of t that occurs outside of *if*-conditions and λ-expressions and does not contain *if*-expressions. We define $\Pi_p^{\mathsf{base}}(t) \subseteq \Pi(t)$ as the set of the positions of the base cases of p in t, i. e., the positions of those result terms that do not contain calls of p. $\Pi_p^{\mathsf{rec1}}(t) \subseteq \Pi(t)$ denotes the set of the positions of *direct* recursive calls, i. e., calls $p(\ldots)$ outside of λ-expressions. Finally, $\Pi_p^{\mathsf{rec2}}(t) \subseteq \Pi(t)$ denotes the set of positions of *second-order recursive calls*, i. e., calls $p(\ldots)$ inside a λ-expression that is passed to a second-order procedure. For some $\pi \in \Pi_p(t) := \Pi_p^{\mathsf{base}}(t) \cup \Pi_p^{\mathsf{rec1}}(t) \cup \Pi_p^{\mathsf{rec2}}(t)$, we write C_t^π for the call context of the subterm at position π in t (i. e., the set of conditions that lead to π).

In the base and step cases of an inductive proof of $\forall x : \tau.\ \psi[x]$, $\psi[x]$ needs to be shown under certain premises. Given a subterm t of B_p and a position $\pi \in \Pi_p(t)$, the premise $\mathsf{Prem}_p^\pi(\psi, t)$ is constructed as follows:

- If $\pi \in \Pi_p^{\mathsf{base}}(t)$, we get a base case of the induction: $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^\pi$.
- If $\pi \in \Pi_p^{\mathsf{rec1}}(t)$, we have a recursive call $t|_\pi = p(t')$ for some t', which gives rise to an induction hypothesis: $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^\pi \wedge \psi[t']$.
- If $\pi \in \Pi_p^{\mathsf{rec2}}(t)$, then there is a minimal prefix π' of π such that $t|_{\pi'} = h(\lambda \boldsymbol{y}.\, t'',\ t')$ for some second-order procedure h, and t'' contains a recursive call at position $\pi'' \in \Pi_p(t'')$ that is a suffix of π. Thus we use the quantification procedure *forall.h* in the induction hypothesis to assert that $\psi[\ldots]$ holds for the arguments of the respective p-call within $\lambda \boldsymbol{y}.\, t''$:
  $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^{\pi'} \wedge \mathit{forall.h}\big(\lambda \boldsymbol{y}.\, \mathsf{Prem}_p^{\pi''}(\psi, t''),\ \lambda \boldsymbol{y}.\, t'',\ t'\big)$

These premises are used in the induction axiom for procedure p as follows:

Definition 3. *For a terminating procedure* `procedure` $p(x : \tau) : \tau' \mathrel{<=} B_p$*, the* induction axiom *for p is given by*

$$\frac{\big\{\forall x : \tau.\ \mathsf{Prem}_p^\pi(\psi, B_p) \to \psi[x] \ \big|\ \pi \in \Pi_p(B_p)\big\}}{\forall x : \tau.\ \psi[x]}\ .$$

Example 7. The base case of procedure *varcount* (cf. Fig. 1) is given by result term "1" under call context $\{?\mathit{var}(t)\}$. After η-expansion, second-order recursion occurs in $\mathit{map}(\lambda s : \mathit{term}.\ \mathit{varcount}(s),\ \mathit{args}(t))$. Thus the induction axiom is:

$$\frac{\begin{array}{l}\forall t : \mathit{term}.\ ?\mathit{var}(t) \to \psi[t] \\ \forall t : \mathit{term}.\ \neg\, ?\mathit{var}(t) \wedge \mathit{forall.map}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{varcount},\ \mathit{args}(t)) \to \psi[t]\end{array}}{\forall t : \mathit{term}.\ \psi[t]}$$

In the induction hypothesis, procedure *forall.map* asserts $\lambda s : \mathit{term}.\, \psi[s]$ for all calls of $\lambda s : \mathit{term}.\, \mathit{varcount}(s)$ by *map*. ♢

Example 8. In the induction axiom for *groundterm* (cf. Fig. 2), the step case is $\forall t : \mathit{term}.\ \neg\, ?\mathit{var}(t) \wedge \mathit{forall.every}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{groundterm},\ \mathit{args}(t)) \to \psi[t]$. ♢

Definition 3 can easily be generalized to accommodate procedures with more parameters. We illustrate this with two examples:

Example 9. Procedure *termsize* (cf. Fig. 2) is defined by second-order recursion via *foldl*, which receives a third argument that is just passed on to *forall.foldl*: The induction hypothesis

$$\mathit{forall.foldl}\big(\lambda n : \mathbb{N}, s : \mathit{term}.\, \psi[s],\ \lambda n : \mathbb{N}, s : \mathit{term}.\, n + \mathit{termsize}(s),\ 1,\ \mathit{args}(t)\big)$$

asserts $\psi[s]$ for all elements of $\mathit{args}(t)$. ♢

```
structure $predefinedSymbol$ <= $T, CONS, CAR, CDR, LIST, QUOTE, IF, \ldots$
structure $sexpr$ <=
  $nil,\ lispsymbol(name : predefinedSymbol),\ cons(car : sexpr, cdr : sexpr), \ldots$
structure $maybe[@A]$ <= $nothing,\ just(what : @A)$
procedure $mapsx(f : sexpr \to maybe[sexpr],\ x : sexpr) : maybe[sexpr]$ <= $\ldots$
procedure $eval(expr, va, fa : sexpr,\ n : \mathbb{N}) : maybe[sexpr]$ <=
  $\ldots mapsx(\lambda arg : sexpr.\ eval(arg, va, fa, n),\ cdr(expr)) \ldots$
```

Fig. 5. Excerpt from a LISP Interpreter *eval*

Example 10. In [6], Boyer and Moore describe a LISP Interpreter *eval* that evaluates LISP s-expressions (cf. Fig. 5). Since the evaluation of a LISP function call `(F T1 ... Tn)` requires the evaluation of the list `(T1 ... Tn)` of arguments, they introduce an auxiliary procedure

```
procedure $evlist(expr, va, fa : sexpr,\ n : \mathbb{N}) : maybe[sexpr]$
```

that considers *expr* as a list of s-expressions and successively evaluates these s-expressions by calling *eval* on each of them. Thus *eval* and *evlist* are mutually recursive. Due to lacking support of mutual recursion, Boyer and Moore merge both procedures into a single procedure *ev* that is parameterized by a flag to indicate if a single s-expression or a list of s-expressions is to be evaluated.

Second-order recursion provides a much more elegant way to implement the interpreter: Procedure *mapsx* considers parameter x as a list, applies f to $car(x)$, $car(cdr(x))$, $car(cdr(cdr(x)))$, ..., and returns an s-expression that represents the list of the result values. If an application of f yields *nothing*, the iteration stops and *mapsx* returns *nothing*. Procedure *eval* then uses second-order recursion via *mapsx* to evaluate a "list" $cdr(expr)$ of s-expressions.[6]

According to Definition 2 our approach synthesizes a quantification procedure $forall.mapsx(p : sexpr \to bool,\ \ f : sexpr \to maybe[sexpr],\ \ x : sexpr) : bool$ that checks $p(z)$ for all calls $f(z)$ by *mapsx*. In one of the step cases of the induction axiom for *eval* for a proof of $\forall expr, va, fa : sexpr,\ n : \mathbb{N}.\ \psi[expr, va, fa, n]$ the induction hypothesis is

$$\begin{aligned} &forall.mapsx(\lambda arg : sexpr.\ \psi[arg, va, fa, n],\\ &\qquad\qquad \lambda arg : sexpr.\ eval(arg, va, fa, n),\ cdr(expr))\ . \end{aligned} \qquad \diamondsuit$$

Theorem 1. *The induction axiom from Definition 3 for a terminating procedure p is an instance of well-founded induction.*

Proof (sketch). The relation $\succ$ on τ, defined by $x \succ x'$ iff $p(x) \rhd_p p(x')$, is well-founded, because p terminates. This relation can be syntactically represented by

[6] Parameter *va* models the variable assignment, and *fa* associates function symbols with their definition. If the resource limit n for the evaluation of *expr* does not suffice, *eval* returns *nothing* as in [6]. The complete source code is several pages long [1].

a formula that may use the quantification procedures from Sect. 3. This formula can be used to instantiate the schema (1) of well-founded induction to obtain the induction axiom from Definition 3, see [1] (Sect. 5.2.2 and 5.3). □

Hence Definition 3 describes a method to extract induction axioms from the source code of procedures with second-order recursion. These induction axioms precisely mirror the recursive structure of the respective procedure.

5 Optimization of Induction Axioms

Induction axioms from terminating procedures often are overly specific and thus suboptimal [5,8,9,13,14]. This also holds for many induction axioms that are synthesized according to Definition 3. In the following, we describe optimization techniques for the case of second-order recursion.

Similarly to many existing optimization techniques for procedures without second-order recursion, our approach examines the termination proof of the respective procedure to find optimizations: Intuitively, "components" of induction axioms (e. g., subformulas or parameters) that are irrelevant for the termination proof are also irrelevant for the induction axiom, because well-foundedness of the underlying relation obviously does not depend on these components.

5.1 Optimization of Quantification Procedures

Quantification procedures as in Definition 2 play a pivotal role in induction axioms for procedures with second-order recursion. Our approach optimizes quantification procedures along the following three dimensions (in this order):

(1) Reduce the arity of the additional predicate p.
(2) Extend the range of the quantification.
(3) Reduce the number of parameters of the quantification procedure.

Optimizations along dimensions (1) and (3) obviously increase the readability of induction hypotheses by making them syntactically simpler. In addition, they facilitate a final polishing of induction axioms that simplifies their use in proofs. Optimizations along dimension (2) strengthen the induction hypotheses by generalizing them, so $\psi[z]$ may be assumed for further values z.

In the induction axiom for procedure *groundterm*, for example, the induction hypothesis $\mathit{forall.every}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{groundterm},\ \mathit{args}(t))$ only ensures that ψ holds on a prefix of list $\mathit{args}(t)$, because *every* in general only examines a prefix of list k (cf. Example 5). This is suboptimal, because from structural induction we know that it would be safe to assume that ψ holds for *all* elements of $\mathit{args}(t)$.

In a typical termination proof for procedure *groundterm*, one tries to show that the parameter of *groundterm* gets structurally smaller in recursive calls [2,8,11]. Clearly, $\mathit{args}(t)$ is structurally smaller than t, because the leading *apply*-constructor is missing. Procedure *every* applies $f := \mathit{groundterm}$ only to values $s \in \{\mathit{hd}(k),\ \mathit{hd}(\mathit{tl}(k)),\ \mathit{hd}(\mathit{tl}(\mathit{tl}(k))), \ldots\}$ for $k := \mathit{args}(t)$. Since each such value s is structurally not larger than $\mathit{args}(t)$, one concludes that each argument s of a recursive call of *groundterm* is structurally smaller than t, which proves termination of *groundterm*.

Apparently the proof that procedure *every* applies f only to values z that are structurally not larger than k does not use the fact that *every* stops as soon as it encounters an element z with $\neg f(z)$. Formally, condition $f(hd(k))$ from the body of *every* is not used in the proof. Thus *groundterm* would still terminate if *every* continued with the examination of list elements in case $\neg f(hd(k))$. Then the case analysis over $f(hd(k))$ in the body of *forall.every* would become unnecessary, and the induction hypothesis for *groundterm* would assert ψ for all elements of $args(t)$ as desired.

Consequently, we optimize quantification procedures as follows:

Definition 4. *Let* $proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc}$ *be a procedure and* $i \in \{1, \ldots, m\}$. *Let* Prf *be a proof that proc calls f only with values* $q_1, \ldots, q_m$ *such that* q_i *is structurally not larger than* x.[7] *We say that proc is* call-bounded *wrt. the i-th argument of f and define the synthesis of the* optimized quantification procedure $forall_i^{opt}.proc$ for *proc as follows:*

(1) Procedure $forall_i^{opt}.proc(p : \tau_i \to bool,\ f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : bool$ *is derived from forall.proc by replacing all subterms* $p(t_1, \ldots, t_m)$ *in the procedure body with* $p(t_i)$.
(2) For each case analysis over some term c in the body of proc such that c is not *used in* Prf*, the corresponding case analysis over c in the body of* $forall_i^{opt}.proc$ *is replaced with the conjunction of its branches.*
(3) Each unused parameter of $forall_i^{opt}.proc$ *is removed.*

Call-bounded procedures can be identified by the approach in [2], for example. Unused conditions c of case analyses can be read off from proofs Prf.

Example 11. Procedure *foldl* is call-bounded wrt. the 2nd argument of f, so step (1) reduces the arity of p to $p : @B \to bool$. In step (3), parameters x and f are removed from $forall_2^{opt}.foldl$ (in this order). Thus

```
procedure $forall_2^{opt}.foldl(p : @B \to bool,\ k : list[@B]) : bool <=$
$if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall_2^{opt}.foldl(p, tl(k))\ else\ false$
```

checks $p(z)$ for all elements z of k, and $forall_2^{opt}.foldl(p, k) \leftrightarrow forall.list(p, k)$. $\diamondsuit$

Example 12. For *forall.every*, steps (2) and (3) apply: A proof that *every* is call-bounded does not use condition $c := f(hd(k))$ (i. e., the fact that *every* stops the iteration over list k when $\neg f(hd(k))$ holds). Thus the case analysis over $f(hd(k))$ in the body of *forall.every* can be removed, and parameter f is no longer used. Hence $forall^{opt}.every$ in addition checks p for the gray elements in Fig. 4, and $forall^{opt}.every(p, k) \leftrightarrow forall.list(p, k)$. $\diamondsuit$

Example 13. For procedure *forall.map*, only step (3) applies, which removes the unused parameter f, so $forall^{opt}.map(p, k) \leftrightarrow forall.list(p, k)$. $\diamondsuit$

[7] The structural size of values can be determined by a size measure as in [2].

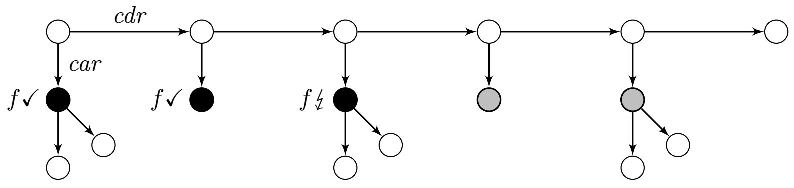

Fig. 6. Procedure *mapsx* applies f to the black entries of this s-expression

Example 14. Fig. 6 shows an exemplary s-expression x. When applying *mapsx* to x, function f is potentially applied to the black and the gray nodes (cf. Example 10). A node z is labeled with "$f\checkmark$" if $?\mathit{just}(f(z))$, whereas "$f\lightning$" means $f(z) = \mathit{nothing}$. As "$f\lightning$" holds for the third black node, procedure *mapsx* stops here and does not apply f to the gray nodes. Since a proof that *mapsx* is call-bounded (i. e., that f is only applied to s-expressions z that are structurally not larger than the whole s-expression x) does not use the fact that the iteration may stop early, the optimized quantification procedure

`procedure` $\mathit{forall}^{opt}.\mathit{mapsx}(p : \mathit{sexpr} \to \mathit{bool},\ x : \mathit{sexpr}) : \mathit{bool}$

checks $p(z)$ for both the black and the gray nodes. ♢

5.2 Optimized Induction Hypotheses For Second-Order Recursion

We optimize induction axioms for procedures with second-order recursion by using the optimized quantification procedures if possible:

Definition 5. *Let p be terminating procedure. If the termination proof for p exploits that some second-order procedure h is call-bounded, the* optimized induction axiom *for p is obtained by replacing forall.h with* $\mathit{forall}^{opt}.h$ *in the induction axiom from Definition 3. If* $\mathit{forall}^{opt}.h$ *is equivalent*[8] *to forall.str for some type constructor str, then forall.str is used instead of* $\mathit{forall}^{opt}.h$*.*

Example 15. The optimized induction axioms for *varcount*, *groundterm*, and *termsize* are equivalent to the structural induction axiom from Example 3:

$$\frac{\begin{array}{l}\forall t : \mathit{term}.\ ?\mathit{var}(t) \to \psi[t] \\ \forall t : \mathit{term}.\ \neg\, ?\mathit{var}(t) \wedge \mathit{forall.list}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{args}(t)) \to \psi[t]\end{array}}{\forall t : \mathit{term}.\ \psi[t]}$$

This induction axiom is significantly stronger than the non-optimized induction axiom for *groundterm*: In the optimized axiom $\psi[s]$ may be assumed for *all* terms s in list $\mathit{args}(t)$ as induction hypothesis. In contrast, in the non-optimized axiom $\psi[s]$ may only be assumed for the first n terms in $\mathit{args}(t)$, where n is the index of the first term s in $\mathit{args}(t)$ with $\neg\, \mathit{groundterm}(s)$. ♢

[8] Syntactical identity up to a renaming of formal parameters is a sufficient and practically useful criterion for equivalence of quantification procedures.

Example 16. For the LISP interpreter of Example 10 and some s-expression $cdr(expr)$ as in Fig. 6, the induction hypothesis asserts $\psi[arg,\ldots]$ only for black entries before the optimization (where f corresponds to the LISP interpreter $eval$). After the optimization, $forall^{opt}.mapsx(\lambda arg : sexpr.\, \psi[arg, n],\ cdr(expr))$ asserts $\psi[arg,\ldots]$ also for the gray nodes (i. e., for all elements of the "list"). $\diamondsuit$

As the examples demonstrate, the optimization leads to intuitive induction axioms. The induction hypotheses correspond to the recursive calls of the respective procedure without being restricted by unnecessary preconditions.

Theorem 2. *The optimized induction axiom from Definition 5 for a terminating procedure p is an instance of well-founded induction.*

Proof (sketch). The optimization drops case analyses (in quantification procedures $forall.h$) on conditions that are irrelevant for the termination proof of p. Thus there is a modified copy p' of p where these case analyses are dropped in h (cf. Sect. 5.2.3 in [1]). Procedure p' terminates and the non-optimized induction axiom for p' is equivalent to the optimized induction axiom for p. $\square$

6 Related Work

In Isabelle [8,10,12] induction theorems are synthesized (and proved within Isabelle's higher-order logic) for terminating procedures and data types. Since higher-order logic is *not* a programming language and thus lacks an operational semantics, Isabelle cannot determine which function calls are required to evaluate a given term. Therefore, induction axioms for procedures with second-order recursion cannot be synthesized from just the source code. To solve this problem, the user can specify *congruence rules* by proving *congruence theorems* such as $k = k' \wedge \big(\forall z : @A.\ z \in k \to f(z) = f'(z)\big) \to map(f, k) = map(f', k')$, which tells Isabelle that for $map(f, k)$ the values $f(z)$ for at most all $z \in k$ are relevant. The resulting induction theorem for procedure $varcount$ is equivalent to our induction axiom from Example 15. Syntactically, the quantification over the elements s in $args(t)$ is expressed by $\forall s : term.\ s \in args(t) \to \psi[s]$, where the notion "$\in$" of list membership stems from the user's congruence rule. Thus the induction theorems directly depend on the congruence rules, and the only way to optimize induction theorems is to (manually) modify the congruence rules. However, this becomes impossible when two function calls require different sets of congruence rules (e. g., see the example with procedure *testany* in [8]), so "in general, there is no 'best' or 'complete' set of congruence rules" [8]. Apart from that, the induction theorem for data type *term* is different from the usual structural induction and targets the simultaneous proof of two formulas $\forall t : term.\ \phi[t]$ and $\forall k : list[term].\ \psi[k]$ based on the mutual recursion of types $term$ and $list[term]$.

In contrast, PVS [11] synthesizes quantification procedures for parameterized data types such as $list[@A]$ and uses these procedures for structural induction axioms (e. g., for data type *term*). While PVS uses *constructor induction*, our induction axioms use *destructor induction.* PVS does *not* synthesize induction axioms for (terminating) procedures and hence does not offer techniques to optimize induction axioms.

In ACL2 [5,9] induction axioms are synthesized for data types and for terminating procedures. Induction axioms are optimized using various techniques (e. g., [9]). However, procedures cannot be defined by second-order recursion.

For Coq, Barthe et al. [4] describe a tool that synthesizes induction axioms for terminating procedures, but second-order recursion is not supported.

Bundy et al. [7] developed a technique to construct induction axioms for the *synthesis* of procedures. In their approach, the goal is to find *novel* induction axioms that do not correspond to the recursive structure of existing procedures. Second-order recursion is not considered in this approach.

7 Conclusion

Our approach to automatically extract induction axioms from terminating procedures consists of two main steps: Firstly, it synthesizes induction axioms that precisely mirror the recursive structure of the procedures. For procedures with second-order recursion, the indirect recursive calls are captured using so-called *quantification procedures* that are synthesized automatically for the respective second-order procedures. Secondly, induction axioms are optimized automatically (i. e., generalized and simplified) by inspecting the termination proofs of the respective procedures. For that purpose our approach in particular optimizes the quantification procedures to strengthen the induction hypotheses.

The vision behind our approach is that a degree of automation can be achieved for the verification of *second-order programs* that is comparable to highly automated verification tools for *first-order programs*, e. g., ACL2. Practical experiments in √`eriFun`[9] (involving 21 procedures with second-order recursion, 14 main theorems and 28 auxiliary lemmas) showed that our methods in fact synthesize induction axioms that are neither too specific (as "precise" induction axioms tend to be) nor too general (as axioms for *complete induction* would be). This facilitates intuitive proofs, i. e., proofs that are quite similar to what one would do using paper and pencil. Hence our approach contributes to achieving such a high degree of automation.

For example, the optimization of induction axioms considerably simplifies the proof that $varcount(t) = 0$ implies $groundterm(t)$. With the optimization, a simple auxiliary lemma is required: If $p(z_i)$ and $p(z_i) \to q(z_i)$ hold for all elements z_i of a list k, then $q(z_i)$ holds for all elements z_i of k. Without the optimization, the user needs to discover and prove a much more complicated auxiliary lemma: Let n be the index of the first element z_n of a list k with $\neg\, q(z_n)$, or $n := |k|$ if there is no such element in k; if $p(z_i)$ and $p(z_i) \to q(z_i)$ hold for the first n elements z_i of k, then $q(z_i)$ holds for *all* elements z_i of k.

We expect that our approach can be transferred to other programming languages with call-by-value semantics; for ML, this might require to also consider

[9] see `http://www.mais.informatik.tu-darmstadt.de/Markus_Aderhold.html`

axioms for constructor-style induction. Our commitment to an evaluation strategy makes it possible to uniformly determine which function calls need to be evaluated for a given term. In contrast, Isabelle does not commit to an evaluation strategy; the price for this increased flexibility is that the user needs to formulate and prove additional theorems that at least approximate an evaluation strategy for particular functions.

Procedures in *continuation passing style* provide numerous additional examples of second-order recursion, because there *each* procedure has a function parameter (representing the continuation). However, in certain cases this may involve indirect recursive calls in continuations of direct recursive calls, which we leave as an area for further research.

Acknowledgment. I am grateful to Nathan Wasser for the implementation of the approach and to the anonymous referees for constructive feedback.

References

1. M. Aderhold. *Verification of Second-Order Functional Programs.* Doctoral dissertation, TU Darmstadt, 2009.
2. M. Aderhold. Automated termination analysis for programs with second-order recursion. In *Proceedings of TACAS-16*, volume 6015 of *LNCS*. Springer, 2010.
3. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Kluwer Academic Publishers, 2002.
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 *Proceedings of FLOPS-2006*, volume 3945 of *LNCS*, pages 114–129, 2006.
5. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, Inc., 1979.
6. R. S. Boyer and J S. Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 31(3):441–458, 1984.
7. A. Bundy, L. Dixon, J. Gow, and J. Fleuriot. Constructing induction rules for deductive synthesis proofs. In *Proceedings of Constructive Logic for Autom. Softw. Engineering 2005*, volume 153 of *ENTCS*, pages 3–21. Elsevier, 2006.
8. A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic.* Doctoral dissertation, TU München, Germany, 2009.
9. P. Manolios and A. Turon. All-termination(T). In *Proceedings of TACAS-2009*, volume 5505 of *LNCS*, pages 398–412. Springer, 2009.
10. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2009.
11. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, November 2001.
12. K. Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, TU München, Germany, 1999.
13. C. Walther. Computing induction axioms. In Andrei Voronkov, editor, *Proceedings of LPAR-3*, volume 624 of *LNAI*, pages 381–392. Springer-Verlag, 1992.
14. C. Walther. Mathematical induction. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Oxford University Press, 1994.
15. C. Walther, M. Aderhold, and A. Schlosser. The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, TU Darmstadt, 2006.
16. C. Walther and S. Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.