Instantiation of existentially quantified variables in inductive specification proofs

Brigitte Pientka* and Christoph Kreitz

Department of Computer Science, Cornell University
Ithaca, NY 14853-7501, U.S.A.
{pientka,kreitz}@cs.cornell.edu

Abstract. We present an automatic approach for instantiating existentially quantified variables in inductive specifications proofs. Our approach uses first-order meta-variables in place of existentially quantified variables and combines logical proof search with rippling techniques. We can avoid the non-termination problems which usually occur in the presence of existentially quantified variables. Moreover, we are able to synthesize conditional substitutions for the meta-variables. We illustrate our approach by discussing the specification of the integer square root.

1 Introduction

Constructive type theory [14] offers the unique advantage of total correctness of synthesized programs. In this setting a specification is of the form

$$\forall input.\ \exists output.\ \ spec(input, output)$$

where *input* is a vector of arguments, *output* is a result and *spec* is a proposition describing the required relation between them. A program meeting this specification can be extracted from its proof via the proofs-as-programs principle [3]. This style is widely advocated [15] and supported in a number of implementations such as NuPRL[10]. The application of such systems however is limited by its low degree of automation. In order to overcome this drawback, we suggest incorporating techniques from inductive theorem proving.

The first difficult step within a proof is the choice of the appropriate induction scheme. Different induction schemes result in algorithms which differ in their complexity. The base cases of the induction correspond to the base cases in the recursive program while the step cases correspond to the recursive cases. Recursion analysis [7, 20] identifies automatically appropriate induction schemes. However there has not much work been done to determine automatically induction schemes which result in efficient algorithms. In this paper we focus on the second crucial step during a specification proof, the instantiation of the existentially quantified variables. The witness for an existentially quantified variable

* The research reported is supported by the Gottlieb Daimler and Karl Benz Foundation with a fellowship to the first author.

corresponds to the recursive calls in the program. Sometimes a case split is necessary before decomposing the existential quantifier. The existentially quantified variables are then instantiated according to the cases.

A standard technique to deal with existentially quantified variables is to use meta-variables in place of the existential witness and allow the application of logical rules that refine the goal. To complete the proof, a unification procedure gives us the instantiation of the meta-variables. In inductive specification proofs, standard unification techniques are not sufficient; we need to rewrite the expression from the induction conclusion towards the application of the corresponding expression in the induction hypothesis. Both expressions are equal after some rewriting steps. The crucial question is how can we find a chain of rewriting steps, such that both expressions can be made equal by rewriting in the presence of meta-variables.

In inductive theorem proving, an annotated rewriting technique, called rippling [9, 6], has been used successfully in order to control the rewriting process. However, only little focus has been devoted to the automatic instantiation of existentially quantified variables. In this paper we suggest combining the logic provided by constructive type theory with inductive theorem proving techniques such as rippling, in order to compute valid instantiations for the existentially quantified variables. We use first-order meta-variables during proof search within the sequent calculus. During rippling the meta-variables are treated in the same way as potential "sink variables". We develop an extended matching procedure that matches the rippled induction conclusion with the induction hypothesis. If this match is successful, it returns an instantiation for the meta-variables and a rippling sequence, which rewrites the instantiated induction conclusion to the induction hypothesis. With this approach we can avoid non-termination problems that usually occur in the presence of existentially quantified variables. Moreover, we check the consistency of the remaining subgoals under the synthesized substitution. During this consistency check, we are able to synthesize constraints that form a case split during the proof. We demonstrate the strength of our approach by discussing the proof of the integer square root specification.

In Section 2, we give a brief introduction to rippling and discuss the rippling approaches for dealing with meta-variables. In Section 3 we describe the general idea of our approach and in 4 we consider the proof of the integer square root. We show step-by-step how we can derive *conditional substitutions* for the existentially quantified variables. In Section 5 we present a more technical description of our method using ML-notation. In Section 6 we describe the formalization in NuPRL. In Section 7 we discuss related work concerning program synthesis and finally in Section 8 we outline future work and draw some conclusions.

2 A brief introduction to rippling

Rippling is an annotated rewriting technique that has been successfully applied in inductive theorem proving. Differences between the induction hypothesis (*given*) and the induction conclusion (*goal*) are marked by meta-level annotations, called *wave annotations*. Expressions that appear both in the goal and in the given are called *skeletons*. Expressions that appear in the goal, but not in the given are called *wave-fronts*. The induction (or recursive) variable that is surrounded by a wave-front is called *wave-hole*. *Sinks* are parts of the goal which correspond to universally quantified variables in the given and are marked by $\lfloor sink \rfloor$. We call the annotated rewrite rules *wave-rules*. To illustrate, consider the following wave-rule which is derived from the recursive definitions of $+$.

$$\boxed{\mathsf{s}(\underline{U})}^{\uparrow} + V \stackrel{R}{\longmapsto} \boxed{\mathsf{s}(\underline{U+V})}^{\uparrow} \tag{1}$$

In this wave-rule $\mathsf{s}(\dots)$ denotes the wave-front that is marked by a box. The underlined part $\underline{U}$ resp. $\underline{U+V}$ marks the wave-hole. Intuitively, the position and orientation of the wave-fronts define the direction in which the wave-front has to move within the term tree. The up-arrow $\uparrow$ indicates that the wave-front has to move from a position within the term tree towards the root of the term tree (*rippling-out*). A down-arrow $\downarrow$ moves the wave-front inwards or sideways towards the sink in the term tree, i.e. the sink is filled with the wave-front (*rippling-in*). If rippling succeeds in moving the annotations either to the root of the term tree or to a sink, then rippling terminates successfully and the induction hypothesis matches the induction conclusion. Rippling terminates unsuccessfully if the rewriting process is blocked, i.e. no wave-rule is applicable anymore and the induction hypothesis (given) does not match the induction conclusion (goal).

In Basin & Walsh [2], a calculus for rippling is presented and well-founded measure, called *wave measure* is defined, under which rippling terminates if no meta-variables occur in the goal. The wave measure associates weights to the wave-fronts to measure the width and the size of the wave-front. The *width* of a wavefront is defined by the number of nested function symbols between the root of the wave-front and the wave-hole. The *size* of a wave-front is the number of function symbols and constants in the wave-front. Rewriting is restricted such that each application of a wave-rule is skeleton preserving and measure decreasing according to the defined wave measure.

For instantiating existentially quantified variables via rippling, mainly two approaches have been suggested in the literature. In Bundy *et al.* [6] special existential wave-rules are suggested. Existential wave-rules can be derived from non-existential wave-rules. For example, the existential wave-rule corresponding to wave-rule (1) takes the following form:

$$\exists\, \boxed{U} : \mathbb{N}.\ \boxed{U} + V \stackrel{R}{\longmapsto} \exists\, \boxed{U'} : \mathbb{N}.\ \boxed{s(\underline{U'+V})}^{\uparrow} \tag{2}$$

Unfortunately, the search problem is exacerbating in the presence of existential quantifiers. Another disadvantage of this approach is that the process of existential rippling does not explicitly record the relationship between the non-existential wave-rule (1) and its existential analogue (2), more precisely the relation between U and U'. This does not matter if we are just interested in provability. In program synthesis, however, the identity of the existential witness plays a vital role of defining the program to be synthesized.

Other approaches [1, 12, 18] use meta-annotations; the existentially quantified variable x is replaced by $\boxed{F(\underline{X})}$ where capital letters indicate meta-variables. Middle-out reasoning [11] is used in order to instantiate the function F and its argument X. This problem requires a computationally expensive higher-order unification. The presence of higher-order variables also leads to non-termination of rippling, as the width and the size of the wave-front cannot be determined.

Our approach overcomes these drawbacks by combining rippling with first–order theorem proving techniques and using an extended matching procedure for finding the witness for the existentially quantified variables.

3 Automatic instantiation of meta-variables

Our research interest is to automate key steps such as the instantiation of the existentially quantified variable in sequent proofs. By the proofs–as–programs paradigm we are then able to extract a program from the proof of a specification. In Figure 1 an overview of our approach is presented.

1. Refinement of the step case formula by applying sequent rules and using meta-variables in place of existential witness
2. Rippling on atomic subgoals
3. Extended matching in order to compute valid substitutions for the meta-variables and a rippling sequence
4. Consistency check to test if the remaining subgoals are true under the found substitution.

Fig. 1. Automatic instantiation of meta-variables – 4 steps

In order to deal with existentially quantified variables, we suggest first decomposing the existentially quantified formula and use first-order meta-variables in place of the existentially quantified variables. Secondly, rippling is used to manipulate the atomic subgoals. It rewrites part of the induction conclusion to some

$$IC \overset{R}{\longmapsto} C_0 \overset{R}{\longmapsto} \ldots \overset{R}{\longmapsto} C_i \overset{R}{\longmapsto} \ldots \overset{R}{\longmapsto} C_n \overset{R}{\longmapsto} IH$$

rippling extended matching

formula C_i. Thirdly, an extended matching procedure returns an instantiation for the meta-variables and a sequence of rippling steps by reasoning backwards from the induction hypothesis towards C_i. For rippling we use the *rippling–distance* strategy [13, 4]. Each wave-front is mapped to a selected (goal)-sink. The distance between a wave-front and its assigned (goal) sink in the term tree is called *distance measure.* Each application of a wave-rule must reduce this distance until the sink is filled with the wave-front and the distance measure is zero. The main advantage of this approach is the uniform efficient treatment of rippling. Hence, we need not treat the various rippling strategies differently, and it is redundant to mark the wave-fronts with up-arrows $\uparrow$ and down-arrows $\downarrow$. For uniformly integrating rippling-out the definition of sink has been generalized to arbitrary term positions. By putting a sink around the whole term rippling-out can be simulated by rippling–distance. This approach can be extended easily to incorporate meta-variables. During rippling, meta-variables are treated like potential sink variables. By adopting rippling for meta-variables and using an extended matching procedure, higher-order variables can be avoided and rippling terminates.

For extended matching, we use a *reverse rippling* process which reasons from the induction hypothesis towards the C_i. The main idea is to use the rippling–distance strategy backwards and synthesize a rippling sequence $C_i \xmapsto{R} \ldots \xmapsto{R} C_n \xmapsto{R} IH$ together with a substitution for the meta-variables.

Finally, a consistency checker, tests if the remaining subgoals can be proven under the synthesized substitution. If necessary, conditions can be synthesized which constrain the substitution. These *conditional substitutions* form a case split in the proof.

Our approach combines rippling techniques with logical proof search. If it succeeds, the rippling proof is translated back into sequent style.

4 Proving the specification of the integer square root

In this section, we illustrate the automatic instantiation of existentially quantified variables by discussing the proof of the following integer square root specification:

$$\forall x:\mathbb{N}.\ \exists y:\mathbb{N}.\ y^2 \leq x \wedge x < (y+1)^2 \tag{3}$$

The top–down sequent proof starts by induction on x. We concentrate on the step case of the induction:

$$x:\mathbb{N},\ \exists y:\mathbb{N}.\ y^2 \leq x \wedge x < (y+1)^2 \quad \vdash \quad \exists y:\mathbb{N}.\ y^2 \leq \mathsf{s}(x) \wedge \mathsf{s}(x) < (y+1)^2$$

This sequent is proved by a proof procedure which searches for a rippling proof and an instantiation for the existentially quantified variable y. It proceeds as described in figure 1. In the first step, logical inference rules are used in order to decompose the induction hypothesis and the conclusion. The existentially quantified variable in the conclusion is replaced by a meta-variable Y. This gives us two subgoals, (4) and (5):

$$x:\mathbb{N},\ y:\mathbb{N}\,\underline{y^2 \leq x},\ x < (y+1)^2 \quad \vdash \quad Y^2 \leq \boxed{\mathsf{s}(\underline{x})} \tag{4}$$

and

$$x:\mathbb{N},\ y:\mathbb{N},\ y^2 \leq x,\ \underline{x < (y+1)^2} \quad \vdash \quad \boxed{\mathsf{s}(\underline{x})} < (Y+1)^2 \tag{5}$$

During the second step the goal is annotated in such a way, that its skeleton matches the corresponding part in the induction hypothesis. Corresponding parts are underlined in this example. Note, that the meta-variable Y matches the variable y in the given. The following wave-rules are derived from the definitions of functions used in the specification:

$$\boxed{\underline{U} + W)} < \boxed{\underline{V} + W)} \stackrel{R}{\longmapsto} U < V \tag{6}$$

$$\boxed{s(\underline{U})} + V \stackrel{R}{\longmapsto} \boxed{s(\underline{U + V})} \tag{7}$$

$$(\boxed{s(\underline{A})})^2 \stackrel{R}{\longmapsto} \boxed{\underline{A}^2 + 2A + 1} \tag{8}$$

Note, while in wave-rule (7) and (8) wave-fronts occur on both sides of the wave-rule, in wave-rule (6) the wave-fronts are dropped on the right hand side of the wave-rule. Wave-rules like (6) are usually used to complete proofs[1]. As no wave-rule is applicable, rippling leaves the subgoal unchanged.

Thirdly, an extended matching procedure is applied, that attempts to find an instantiation for the meta-variable Y and a rippling sequence by reasoning from the induction hypothesis towards the subgoals. In the induction hypothesis y is marked as a sink variable. During the first reverse wave-rule application wave-fronts are created by wave-rules of the same type as wave-rule (6). This wave-front has to move towards the sink variable y by reverse rippling and results in the instantiation of the meta-variable Y. We start with the second subgoal (5) and try to match $\boxed{\mathsf{s}(\underline{x})} < (Y+1)^2$ (induction conclusion IC – goal) and $x < (y+1)^2$ (induction hypothesis IH – given). The induction hypothesis IH represents the final formula in the rippling sequence. In order to determine which formula preceded the induction hypothesis, the wave-rule set is inspected. The given must match the right hand side of a wave-rule. The left hand side of this rule constitutes the predecessor to IH, if further rippling towards the sink variable is possible. By wave-rule (6) it is suggested that the formula before reaching the induction hypothesis $x < (y+1)^2$ is $\boxed{\underline{x} + W} < \boxed{\underline{(y+1)^2} + W)}$. This formula can be rippled by wave-rule (8) and W can be instantiated with $2(y+1)+1$. The inserted wave-front moves closer to the sink variable y. Rippling towards the sink variable y is straightforward and the generated rippling sequence is presented in Figure 2. By wave-rule (7) the wave-front s is moved to a position where it surrounds y; therefore our rippling sequence terminates successfully. As no wave-rule is available to justify the final step, we need to prove that the induction conclusion is implied by the last step[2]. Typically these implications can be proven by decision procedure using standard arithmetic. In this case the proof is trivial. Therefore $\mathsf{s}(y)$ is a valid substitution for Y.

[1] We consider here proofs by strong fertilization, i.e. which aim for total match between the induction conclusion term and induction hypothesis term.

[2] Note that a rippling rule $C_i \stackrel{R}{\longmapsto} C_{i+1}$ corresponds to the opposite of the logical implication $C_{i+1} \Rightarrow C_i$.

$$
\begin{array}{ll}
\boxed{\mathsf{s}(\underline{x})} < (\boxed{\mathsf{s}(\underline{y})} + 1)^2 & \\
 & \text{by decision proc.} \\
\quad \overset{R}{\rightleftarrows} \boxed{\underline{x} + 2(y+1) + 1} < (\lfloor \boxed{\mathsf{s}(\underline{y})} \rfloor + 1)^2 & \\
 & \text{by wr (7)} \\
\quad \overset{R}{\longmapsto} \boxed{\underline{x} + 2(y+1) + 1} < \boxed{\mathsf{s}((\underline{\lfloor y \rfloor + 1}))}^2 & \\
 & \text{by wr (8)} \\
\quad \overset{R}{\longmapsto} \boxed{\underline{x} + 2(y+1) + 1} < \boxed{(\underline{(\lfloor y \rfloor + 1)^2} + 2(y+1) + 1)} & \\
 & \text{by wr (6)} \\
\quad \overset{R}{\longmapsto} x < (\lfloor y \rfloor + 1)^2 &
\end{array}
$$

Fig. 2. Rippling sequence generated by extended matching

This substitution and its corresponding rippling sequence (cp. Figure 2) constitute a successful match if the remaining subgoals are true under the found substitution (step 4 in Figure 1). We use a heuristic to check the subgoals and synthesize case splits if necessary. In order to restrict search space, we require that the subgoals can be proven by standard arithmetic, rippling and equational reasoning. If one of the remaining subgoals is not provable by these techniques, we use this subgoal as a constraint of the substitutions. In order to be consistent, we then prove all the remaining subgoals under the negated constraint. In this example, (4) is the only remaining subgoal. We use the subgoal $(\mathsf{s}(y))^2 \leq \mathsf{s}(x)$ as a constraint, and therefore try to prove both cases (4) and (5) for the case $\neg(\mathsf{s}(y))^2 \leq \mathsf{s}(x)$. We use unfolding of the successor and $\neg(\ldots \leq \ldots)$ function s and normal matching, to derive a substitution. We start again by inspecting subgoal (5). Matching between ; $\mathsf{s}(x) < (Y+1)^2$ and $\mathsf{s}(x) < (y+1)^2$ returns the substitution $[y/Y]$. The subgoal (4) is trivially true under this substitution.

By combining logical proof search with rippling and extended matching, we are able to generate automatically a proof for the step case and to synthesize a set of conditional substitutions:$[(\mathsf{s}(y))^2 \leq \mathsf{s}(x), \mathsf{s}(y)/Y], [\neg((\mathsf{s}(y))^2 \leq \mathsf{s}(x)), y/Y]$.

5 An algorithm for extended matching

In this section, we present a technical description of the steps performed during the automatic instantiation of existentially quantified variables (see Figure 1). We use ML-notation to describe the algorithm. To automate step 1 and 2 standard theorem proving methods can be used. We concentrate on the extended matching procedure (step 3 and 4) which is the core of the automatic instantiation of existentially quantified variables. We present the algorithm top down, starting with the main function extended_match(Figure 3) and proceeding to the sub-parts. Before calling the algorithm extended_matching, the wave-rule set wrs contains annotated wave-rules. The function extended_match returns (conditional) substitutions and proofs, if

```
let extended_matching sgoal-list wrs =
let find_one_match sgoal sgoals =
   let (rip_sequ,subst)= rippling_sequence (conclusion sgoal) (hyp sgoal) wrs in
      check_subgoals sgoal sgoals (rip_sequ,subst)
exists s ∈ sgoal-list (find_one_match s (delete s sgoal-list)) ≠ []
```

Fig. 3. Extended matching algorithm

1. The list of subgoals, sgoal-list, contains at least one element sgoal. In this case rippling_sequence (conclusion sgoal) (hypothesis sgoal) wrs finds a rippling sequence rip_seq and a substitution subst; (conclusion sgoal) gives us the rippled conclusion and (hypothesis sgoal) returns the corresponding hypothesis.
2. the remaining subgoals, sgoal-list$\setminus\{$sgoal$\}$, are consistent with this substitution, which is checked by check_subgoals sgoal-list sgoal subst.

The function rippling_sequence, described in Figure 4, computes a rippling sequence rip_conc $\xmapsto{R} \ldots \xmapsto{R} \ldots C_i \xmapsto{R} \ldots \xmapsto{R}$ ind_hyp and a substitution for the meta-variable. First the variable in the induction hypothesis that corresponds to

```
let rippling_sequence rip_conc ind_hyp wrs =
let mvar = meta-variable-in-term rip_conc in
let ih = mark-sink mvar ind_hyp in
let children = poss_predecessors ih wrs in
letrec reverse_rippling_in path children =
      if children = []
         then let subst = filled_sink (hd path) in
               if not (subst = []) and implies (hd path) rip_conc
                  then (subst,path)
                  else []
               else letrec recursively_rippling list =
                     if list = []
                       then []
                       else let r = reverse_rippling_in (hd list)::path
                                                   (poss_predecessors (hd list) wrs) in
                            if r = []
                              then recursively_rippling (tl list)
                              else r
in reverse_rippling_in [ind_hyp] children
```

Fig. 4. Algorithm for synthesizing rippling sequence $C_n \xmapsto{R} \ldots IH$

the meta-variable in the induction conclusion is marked with a sink. The function poss_predecessors computes possible predecessors C_{i-1} for a given formula C_i in the rippling sequence by inspecting the wave-rule set wrs. It simulates the backwards application of one rippling rule. The function reverse_rippling_in triggers the reverse rippling process. It proceeds recursively by depth first search if there are several predecessors to a formula. It reasons backwards from the induction hypothesis ind_hyp towards the rippled conclusion rip_conc. In each recursion, the next possible predecessors are computed. It is successful if the sink variable is surrounded by a wave-front, i.e. the sink is filled and this formula C_{i+1} implies C_i. Otherwise backtracking is initiated. The function returns a triple [(wr_id wr),subst,C_{i-1}] where (wr_id wr) characterizes the identifier of the applied wave-rule wr and subst describes the substitution under which the wave-rule wr is applicable.

```
let poss_predecessors C_i wrs =
letrec potential_predecessors ih wr in
    let (subst,rhs) = match ih (w_rhs wr) in
    let C_{i-1} = apply subst (wr_lhs wr) in
        if null subst then [] else [(wr_id wr),subst,C_{i-1}]
map (\wr. potential_predecessors C_i wr) wrs
```

Fig. 5. Computing a list of possible predecessors to C_i such that $C_{i-1} \overset{R}{\longmapsto} C_i$

The function reverse_rippling_in triggers the reverse rippling process. The function recursively_rippling proceeds recursively by depth first search if there are several predecessors (children) to a formula. It reasons backwards from the induction hypothesis ind_hyp towards the rippled conclusion rip_conc. In each recursion, the next possible predecessors are computed. The function reverse_rippling_in is successful if the sink variable is surrounded by a wave-front, i.e. the sink is filled and this formula C_{i+1} implies C_i. Otherwise backtracking is initiated. The function rippling_sequence returns a substitution for the meta-variable mvar and a rippling sequence path.path is a list of formulas which denotes the rippling sequence from the induction hypothesis to the current node.

The function check_subgoals (Figure 6) checks if the remaining subgoals are provable under the substitution derived by rippling_sequence. If all the remaining subgoals can be proven under the substitution, then the match is successful. If there are subgoals that are not provable by a simple proof procedure simplify[3], then we use one of the remaining unprovable subgoals as a constraint c and prove

1. each g $\in$ sgoal-list $\backslash$\{sgoal\} is provable by simplify under the constraint c
2. each g $\in$ sgoal-list is provable by simplify and matching under the constraint $\neg$c.

sgoal is the formula that has been proved by rippling and extended matching; sgoals and sgoal denote the list of subgoals. If the subgoals are not provable by a simple proof procedure simplify[4], then check_cond_subgoals $\neg$g (sgoal :: sgoals) tries to prove the subgoals by assuming the truth of one of the subgoals that were not provable, and prove all of the subgoals under the negated condition. In order to avoid explosion of the search space, we require that this proof be developed by simple matching and the procedure simplify.

[3] simplify is a combination of NuPRL's tactics Unfold, SupInf and Auto. It includes a decision procedure for elementary arithmetic.

[4] simplify is a tactic which is a combination of NuPRL tactics Unfold 's', SupInf and Auto. It is a decision procedure that uses standard arithmetic.

```
letrec check_subgoals sgoals sgoal (rip_sequ,subst) =
let subst_sgoals = map (\goal. apply_subst subst goal) sgoals in
let (unproved_gs,proved_gs1) = map (\g. if simplify g then [] else g)
                                      subst_sgoals in
    if unproved_gs = []
      then[(_,subst,(sgoal,rip_sequ) @proved_gs1)]
      else
         if [g] = unprovable_gs
           then let (subst2,proved_gs2) = check_cond_sgoals ¬g (sgoal :: sgoals)
                if subst2 = [] then []
                                else [(g,(subst,(sgoal,rip_sequ)::proved_gs1)),
                                      (¬g,(subst2,proved_gs2))]
           else []
```

Fig. 6. Algorithm for checking the remaining subgoals and synthesizing case splits

With the presented algorithm we are able to instantiate existentially quantified variables and solve the step case of an inductive proof automatically. Moreover, by a simple heuristic, which is integrated in check_subgoals we are able to synthesize conditional substitutions. These conditions form a case split in the proof. For a more detailed version we refer to [17].

6 Integrating in NuPRL

In this section we discuss the integration of our proof method into NuPRL, an interactive, tactic based theorem prover. The described proof procedure is embedded within the tactic Ripple. If the proof procedure finds a substitution for the meta-variables and all the subgoals can be solved, this proof is translated back into sequent style. Due to the nature of reverse rippling, the eigenvariable condition is observed, and does not cause any problems for the back translation. The sequent-style proof in NuPRL for the integer square root specification is presented in Figure 7.

In this proof the user specified the induction scheme. First the universal quantifier on the right hand side was decomposed by the tactic allR. The tactic TNatInd 'x' then splits the conjecture into base and step case. The step case was solved automatically by the tactic Ripple 'x'. This proof displays the subtactics which were applied by Ripple. A case split was performed by tactic Decide based on the conditional substitution set before instantiating the existential quantifier on the right hand side. The tactic exL decomposes the existential quantifier on the left hand side. Applications of andL resp. andR eliminate the conjunction on the left resp. right hand side. The generated rippling sequence is translated back into sequent proof by cut, substitution and lemma applications as described in [13].

Other examples, that can be solved by our method include the specification of quotient remainder, append, half, or last. These examples do not require any case splits. We also can prove the specification for $\log_2$ which results in a similar proof to the integer square root example. Moreover, we used the extended matching

```
⊩ ∀x:ℕ.∃y:ℕ. y²≤x ∧ x<(y+1)²
BY allR
|
1. x: ℕ
⊩ ∃y:ℕ. y²≤x ∧ x<(y+1)²
BY TNatInd `x`
|\
| ⊩ ∃y:ℕ. y²≤0 ∧ 0<(y+1)²
1 BY exR ⌈0⌉
| |
| ⊩ 0*0 ≤ 0 ∧ 0 < (0+1)*(0+1)
1 BY Auto
\
2. ∃y:ℕ. y²≤x ∧ x<(y+1)²
⊩ ∃y:ℕ. y²≤s(x) ∧ s(x)<(y+1)²
BY exL 2 THEN andL 3
|
2. y: ℕ
3. y² ≤ x
4. x < (y+1)²
BY Decide ⌈(y+1)² ≤ s(x)⌉ THENW Auto
|\
| 5. (y+1)² ≤ s(x)
1 BY exR ⌈s(y)⌉
| |
| ⊩ (s(y))²≤s(x) ∧ s(x)<(s(y)+1)²
1 BY andR
| |\
| | ⊩ (s(y))²≤s(x)
1 2 BY Auto
| \
| \
| ⊩ s(x) < (s(y)+1)²
1 BY Cut ⌈x + 2*(y+1) + 1 < (y+1)² + 2*(y+1) + 1⌉
| |\
| | ⊩ x + 2*(y+1) + 1 < (y+1)² + 2*(y+1) + 1
1 2 BY Substitution THEN Lemma wave-rule 2
| \
| 6. x + 2*(y+1) + 1 < (y+1)² + 2*(y+1) + 1
1 BY Cut ⌈x + 2*(y+1) + 1 < (s(y + 1))²⌉
| |\
| | ⊩ x + 2*(y+1) + 1 < (s(y + 1))²
1 2 BY Substitution THEN Lemma wave-rule 4
| \
| 7. x + 2*(y+1) + 1 < s((y + 1))²
1 BY Cut ⌈x + 2*(y+1) + 1 < (s(y)+1)²⌉
| |\
| | ⊩ x + 2*(y+1) + 1 < (s(y)+1)²
1 2 BY Substitution THEN Lemma wave-rule 3
| \
| 8. x + 2*(y+1) + 1 < (s(y)+1)²
1 BY Unfold `s` 0 THEN SupInf THEN Auto
|    5. ¬( (y+1)² ≤ s(x) )
BY exR ⌈y⌉
|
⊩ y² ≤ s(x) ∧ s(x) < (y+1)²
BY andR
|\
| ⊩ y² ≤ s(x)
1 BY Unfold `s` 0 THEN Auto
\
⊩ s(x) < (y+1)²
BY Auto
```

Fig. 7. Proof of the integer square root in NuPRL

procedure to instantiate universally quantified variables in the hypothesis list. With this extension we are also able to prove the specification of the integer square root and $\log_2$ by using non-standard induction schemes allowing us to synthesize while loops and binary search algorithms for these two specifications.

7 Related Work

One of the first approaches to automate the instantiation of existentially quantified variables has been by Biundo [5]. Existentially quantified variables are replaced by Skolem functions which describe the program which is to be synthesized. After induction the formula in the step case is put into clausal form. The synthesis proceeds by *clause–set translations* (e.g. rewriting and case splitting) which induce an AND/OR search space. Because rippling is a highly goal directed rewriting technique with only little search involved, our approach is more controlled.

From the systems which rely on the programs–as–proofs paradigm, only a few systems support automation. The closest system to NuPRL is the Oyster/Clam system[8] which started out as a re-implementation of NuPRL. Two major approaches evolved from this work. In Kraan *et al.*[12], also existentially quantified variables are replaced by Skolem functions for the synthesis of logical programs. Middle–out reasoning [11] is used to construct predicate definitions from specifications in classical logic. In Smail & Green [18], an approach for the synthesis of functional programs within the framework of OYSTER/CLAM is suggested. This approach builds on higher–order embeddings and higher–order rippling. Middle-out reasoning and higher order embeddings have the disadvantage of a big search space, as rippling in the presence of higher-order function variables does not terminate.

As the underlying theory in the approaches from Biundo [5] and Kraan *et al.* [12] does not guarantee that the synthesized program is correct, it has to be verified after the synthesis. We believe that constructive type theory provides a firmer mathematical foundation than is found in these systems. Moreover, our approach is much more controlled because of the rippling and extended matching strategy. By reasoning backwards from the induction hypothesis towards the rippled induction conclusion, we are able to synthesize lemmata during the extended matching process. By combining logical proof search methods with rippling techniques, we are able to synthesize case splits which cannot be derived by other comparable systems.

In approaches from Kraan *et al.* [12] and Smaill & Green [18], the aim is to ripple the whole induction conclusion and aim for a match of the whole induction conclusion with the entire induction hypothesis. The underlying logical calculus is not used to decompose the step case during proof search. This causes major problems when we deal with specifications of more complex formulas. For example, if we prove the integer square root specification by a different induction scheme[5], we yield the following step case (induction proceeds over k):

$$\begin{aligned} &k : \mathbb{N}, \forall x, y : \mathbb{N}.\ x - y < \mathsf{p}(k) \wedge y^2 \leq x \wedge 0 \leq y \rightarrow \exists n : \mathbb{N}.\ y \leq n \wedge n^2 \leq x \wedge x < (n+1)^2 \\ &\qquad \vdash \forall x, y : \mathbb{N}.\ x - y < k \wedge y^2 \leq x \wedge 0 \leq y \rightarrow \exists n : \mathbb{N}.\ y \leq n \wedge n^2 \leq x \wedge x < (n+1)^2 \end{aligned}$$

Rippling would annotate the term $x - y < k$ to give $\lfloor x \rfloor - \lfloor y \rfloor < \boxed{\mathsf{s}(\mathsf{p}(\underline{k}))}$ as it operates on the induction conclusion. However, no rippling proof for the left hand side of the sequent can be found. Our approach first decomposes the right and left hand side and then uses extended matching to find a match between $x - y < k$ in the hypothesis list and $X - Y < \mathsf{p}(k)$ on the conclusion side. By reverse rippling starting from $x - y < k$ we try to generate a rippling sequence and an instantiation for X and Y. The following additional wave-rules are derived from monotonicity laws and the definition of $-$ is provided:

$$V > 0 \wedge U > 0 \rightarrow \boxed{\mathsf{p}(\underline{U})} < \boxed{\mathsf{p}(\underline{V})} \stackrel{R}{\longmapsto} U < V \tag{9}$$

$$\boxed{\mathsf{p}(\underline{U - V})} \stackrel{R}{\longmapsto} U - \boxed{\mathsf{s}(\underline{V})} \tag{10}$$

By wave-rule (9) and wave-rule (10) the extended matching procedure generates the following rippling sequence:

$$x - \boxed{\mathsf{s}(\underline{y})} < \boxed{\mathsf{p}(\underline{k})} \underset{(10)}{\stackrel{R}{\longmapsto}} \boxed{\mathsf{p}(\underline{x - y})} < \boxed{\mathsf{p}(\underline{k})} \underset{(9)}{\stackrel{R}{\longmapsto}} x - y < k$$

[5] This induction scheme will result in a more efficient program, namely a while loop. In each iteration y is incremented until $(y+1)^2 > x$.

This example illustrates that the conventional approach [9] to instantiate universally quantified variables in the induction hypothesis by rippling-in is not expressive enough. Moreover, it supports the strength of our approach. The combination of logical proof search and rippling gives us the flexibility to deal with complex logical formulas.

8 Conclusion and future work

We have presented an approach for the instantiation of existentially quantified variables which provides a significant degree of automation to proofs in constructive type theory. The key idea is to use first-order meta-variables in place of the existential witness during proof search and rippling and instantiate this meta-variable by an extended matching procedure. Because we reason backwards from the induction hypothesis towards the rippled conclusion by reverse rippling, our approach is highly goal directed. Moreover, we are able to synthesize a set of conditional substitutions for the meta-variables. The methods known to us cannot synthesize conditional substitutions.

We see our work in a more general framework of matching: two terms t_1 and t_2 match, if the meta-variables in t_1 can be instantiated in such a way that rippling rewrites t_1 towards t_2. This approach allows us to treat meta-variables uniformly. We plan to extend and refine our method in this direction.

Moreover, we plan to explore the use of specially tailored logical proof search methods such as connection method [16] or resolution [19] instead of direct proof search in the sequent calculus. These proof methods are more goal directed. For future research we aim to combine these techniques with rippling and our extended matching algorithm.

References

1. A. Armando, A. Smaill, and I. Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *Proceedings of the twelfth IEEE International Automated Software Engineering Conference, Incline Village, Nevada*, pages 2–9. IEEE Computer Society, 1997.
2. David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(2):147–180, 1996.
3. Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.
4. Wolfgang Bibel, Daniel Korn, Christoph Kreitz, Ferenc Kurucz, Jens Otten, Stephan Schmitt, and Gerd Stolpmann. A multi-level approach to program synthesis. In *Seventh International Workshop on Logic Program Synthesis and Transformation (LOPSTR'97), Leuven, Belgium, July 1997*, Lecture Notes in Artificial Intelligence, 1998.
5. Susanne Biundo. Automated synthesis of recursive algorithms as a theorem proving tool. In *Proceedings of the 8th European Conference on Artificial Intelligence*, München, Germany, 1988.

6. Alan Bundy, Andrew Stevens, Frank van Harmelen, Alan Smaill, and Andrew Ireland. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, August 1993.
7. Alan Bundy, Frank van Harmelen, Jane Hesketh, Alan Smail, and Andrew Stevens. A rational reconstruction and extension of recursion analysis. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989.
8. Alan Bundy, Frank van Harmelen, C. Horn, and Alan Smaill. The Oyster-Clam system. In *Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern*, pages 647–648. Springer, 1990. LNAI 449.
9. Alan Bundy, Frank van Harmelen, Alan Smaill, and Andrew Ireland. Extensions to the rippling–out tactic for guiding inductive proofs. In *Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern*, pages 132–146. Springer, 1990.
10. R. L. Constable, S. F. Allen, H. M. Bromley, and et al. *Implementing Meta-Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1086.
11. Jane T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* PhD thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1991.
12. I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer, 1993. also available as Max-Planck Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.
13. Ferenc Kurucz. Realisierung verschiedender Induktionsstrategien basierend auf dem Rippling-Kalkül. Master's thesis, Technical University Darmstadt, 1997.
14. Per Martin-Löf. Constructive mathematics and computer programming. In *6-th International Congress for Logic, Methodology and Philosophy of Science, 1979*, pages 153–175. North-Holland, 1982.
15. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löfs Type Theory. An introduction.* Clarendon Press, Oxford, 1990.
16. Jens Otten and Christoph Kreitz. A Uniform Proof Procedure for Classical and Non-classical Logics. In G. G rz & S. H lldobler, editor, *KI-96: Advances in Artificial Intelligence*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 307–319. Springer, 1996. Also technical report AIDA–96–02.
17. Brigitte Pientka. Automatic instantiation of meta-variables in specification proofs. Technical report, Cornell University, 1998.
18. Allan Smaill and Ian Green. Automating the synthesis of functional programs. Research paper 777, Dept. of Artificial Intelligence, University of Edinburgh, 1995.
19. T. Tammet. A resolution theorem prover for intuitionistic logic. In *Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ, July 1996*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 2–16, 1996.
20. Christoph Walther. Computing induction axioms. In *Proceedings of the Conference on Logic Programming and Automated Reasoning, St. Petersburg*, LNAI 624. Springer, 1992.