Zeno: A tool for the automatic verification of algebraic properties of functional programs

William Sonnex, Sophia Drossopoulou, Susan Eisenbach

Imperial College London

Abstract. Most functional programs rely heavily on recursively defined structures and pattern matching thereupon. Proving properties of such programs often requires a proof by induction, which many theorem provers have difficulty addressing.
In this paper we present Zeno, a new tool for the automatic verification of simple properties of functional programs. We define a minimal functional language along with a subset of first order logic in which to express properties to be proven. Zeno constructs a proof tree by iteratively reducing the goal into an equivalent conjunction of several simpler sub-goals, terminating when all leaves are trivially true. Building this tree requires the exploration of many alternatives and we give sophisticated techniques for the reduction of this search space. We also present an alternative to existing methods of generating inductive schemata, which builds them gradually based on function definitions.
We provide a comparison with the rippling based tool IsaPlanner and the industrial strength tool ACL2s. Using a test suite from the IsaPlanner website, we found that Zeno could prove strictly more properties than either, and in as good times.

1 Introduction

Proving algebraic properties of functions requires proof steps such as induction or case-splitting. Tools already exist which can prove such properties using these methods[3, 2, 7, 8]. ACL2 is an industry strength proof system with a powerful automated prover, which uses untyped first-order Common LISP as its input language. More recently this was extended to ACL2s, the "Sedan Edition", which simplifies its usage and adds more powerful automated techniques. IsaPlanner is a generic proof-planning framework for the Isabelle[12] proof system. Its main proof tactic is a Rippling[5] based inductive theorem prover and is what we will be referring to when we speak of IsaPlanner's theorem proving capacity. It features an ML style input language and, unlike ACL2, allows for higher-order functions and user-defined recursive data-types.

These verification tools need to address the huge search space which ensues from the fact that in each proof step many different induction schemata and case-splits are applicable. Approaches to trim the space are recursion-analysis[2] (used by ACL2(s)), and ripple-analysis[4] (used by CLAM and IsaPlanner). These approaches generate the induction schemes before creating the proofs; IsaPlanner

backtracks on a failed proof and amends its scheme. The rippling technique "prefers" steps which make it possible to apply the induction hypothesis.

Fig. 1 Our functional language **HC** and properties language **PHC**

Prog	::=	$(TypeDef \mid FunType \mid FunDef)^*$
FunDef	::=	`f` $\mathtt{x}^*$ `=` *Expr*
τ	::=	`x` | `f` | `K` | `(`τ_1 τ_2`)`
Expr	::=	τ | `case` τ `of { ` *Alt* `(; ` *Alt*`)`* `}`
Alt	::=	`K` $\mathtt{x}^*$ `->` *Expr*
t	::=	`T` | `(`$\mathtt{t}_1$ `->` $\mathtt{t}_2$`)`
TypeDef	::=	`data T = K` $\mathtt{t}^*$ `(| K` $\mathtt{t}^*$`)`*
FunType	::=	`f :: t`
φ	::=	$\tau_1 = \tau_2$
Prop	::=	φ | φ `:-` φ `(,` φ`)`*

We propose an alternative technique, which differs from those above in the following two aspects: First, we build up the induction scheme only gradually through consecutive, separate proof steps. Second, we "prefer" steps which make it possible to apply function definitions, and thus we "bring the proof forwards". To support our technique, we introduce a concept called a *critical term*, which is either a variable which appears in the original term (guiding the tool to add this variable to the induction scheme), or a new term which was not part denitions of the original term (guiding the tool to apply a case-split on this new term).

Based on these ideas, we built Zeno, a fully automated verification tool which requires no extra lemmas to be input by the user to complete its proofs, and often discovers the necessary auxiliary lemmas. Zeno breaks the proof of a property down into the proof of zero or more sub-properties by applying ten dierent kinds of steps, and iterates until every branch of the ensuing tree has no further sub-properties left. Zeno supports **HC**, a minimal functional language, and **PHC**, a small language of properties which allows for algebraic properties with entailment.

We evaluated Zeno against IsaPlanner and ACL2s using a test suite from the IsaPlanner website. We found that Zeno could prove strictly more properties than either, and with similar computation times.

This paper is organised as follows. In Section 2 we define **HC** and **PHC**. In Section 3 we explain Zeno's proof output with reference to an example. In Section 4 we explain each of the steps that Zeno can take to construct a proof. In Section 5 we describe our heuristic for the selection of steps, and define critical terms.

In Section 6 we present a comparison between Zeno, IsaPlanner and ACL2s. In Section 7 we present our conclusions and discuss future work.

One can try Zeno online at `http://www.doc.ic.ac.uk/~ws506/tryzeno`, and the sources are available from `http://code.google.com/p/zeno`.

Fig. 2 Example types and functions in **HC**

```
1  data Bool = True | False
2  data Nat = 0 | S Nat
3
4  (+) :: Nat -> Nat -> Nat
5  n + m = case n of { 0 -> y;
6                      S n' -> S (n' + m) }
7
8  (<=) :: Nat -> Nat -> Bool
9  n <= m = case n of { 0 -> True ;
10                      S n' -> case m of {
11                         0 -> False ;
12                         S m' -> n' <= m' } }
13
14 max :: Nat -> Nat -> Nat
15 max n m = case n <= m of { True -> m; False -> n }
```

2 The languages HC and PHC

In this section we define **HC**, our core functional language, and **PHC**, the language of properties over **HC** terms.

2.1 The core functional language HC

HC, defined at the top of **Fig. 1**, is a minimal non-polymorphic[1] subset of Haskell and can be run by any Haskell98[1] compliant compiler. We use `x` for variable names, `f` for function names, `K` for constructor names, and `T` for type names. In **Fig. 2** we give a working example.

A function definition, $FunDef$, introduces a new function `f` with 0 or more parameters, $\mathtt{x}^*$. Lines 9 and 10 in **Fig. 2** define a function `add` with parameters `n` and `m`. A term, τ, is a variable (`x`), or a function (`f`), or a constructor (`K`), or the *application* of a *function term* (τ_1) to an *argument term* (τ_2). Term application is implicitly left-associative, e.g., `f x y` $\equiv$ `((f x) y)`. An expression, $Expr$, is a term (τ), or the *case-analysis* of another term (`case` τ `of { ... }`) giving one of many *Alt*ernative expressions depending on the value of the term being analysed.

[1] Zeno does in fact support polymorphism but we have removed it here for simplicity.

A type, $\mathbf{t}$, is either a type name `T` or a function type $\mathbf{t}_1$ `->` $\mathbf{t}_2$. Function types are implicitly right-associative, e.g., `Nat -> Nat -> Nat` ≡ `(Nat -> (Nat -> Nat))`. A *TypeDef* introduces a new type name `T`, and one or more constructors `K`. The constructors take 0 or more arguments of a given type. Although not represented in our grammar, functions consisting of an operator surrounded by parentheses can be used infix without the parentheses, e.g., `x + y` ≡ `(+) x y`.

Execution of **HC** expressions is defined by the judgement $(E_1 \rightsquigarrow E_2)$, given in **Fig. 3**. In **Fig. 4** we show the evaluation of `(S 0) + 0` to `S 0`.

2.2 The properties language PHC

The language **PHC**, whose syntax is given at the end of **Fig. 1**, supports definite clauses of equality between terms under universal quantification. We follow a Prolog-style notation whereby φ `:-` $\overline{\varphi}$ stands for $\overline{\varphi} \Rightarrow \varphi$, and a comma `,` stands for "and" ($\wedge$). We refer to the property we are attempting to prove as the *goal*. We refer to the φ to the left of the `:-` as the *consequent*, and the $\overline{\varphi}$ to the right of the `:-` as the *antecedents* or *conditions*.

Note that the term `True` is not logical truth; it is just one constructor of the two constructor data type `Bool`. It is possible to express full propositional logic in our syntax, since we can express its operators as functions in **HC** using the data type `Bool`. However, it is not possible to express FOL, as we have no way of expressing existential quantification, nested universal quantification, or negated equality.

All variables within a property that are not constructors or defined functions are implicitly universally quantified. For example,

```
x <= z = True :- x <= y = True, y <= z = True
```

expresses the transitivity of `(<=)`, i.e. that

$$\forall x.\forall y.\forall z.\ \left[\ (x \texttt{ <= } y = \texttt{True}) \wedge (y \texttt{ <= } z = \texttt{True})\ \right] \Rightarrow (x \texttt{ <= } z = \texttt{True})\ .$$

Fig. 3 Expression evaluation in **HC**

$$\frac{(\texttt{f}\ \overline{\texttt{x}} = E) = definition(\texttt{f})}{\texttt{f}\ \overline{\tau_a} \rightsquigarrow E[\overline{\tau_a/x}]} \qquad \frac{(\texttt{K}\ \overline{\texttt{x}} \texttt{ -> } E) \in \overline{A}}{\texttt{case K}\ \overline{\tau_a}\ \texttt{of \{}\ \overline{A}\ \texttt{\}} \rightsquigarrow E[\overline{\tau_a/x}]}$$

$$\frac{\tau_1 \rightsquigarrow \tau_2}{\texttt{case}\ \tau_1\ \texttt{of \{}\ \overline{A}\ \texttt{\}} \rightsquigarrow \texttt{case}\ \tau_2\ \texttt{of \{}\ \overline{A}\ \texttt{\}}}$$

3 Zeno's proof output

In this section we explain Zeno's proof output using the example proof in **Fig. 5**. Zeno's output is a textual representation of the proof tree constructed by Zeno, where each line is a node in the tree. The topmost node of the tree and the first line of our output is the property we are trying to prove.

Fig. 4 Evaluation of `(S 0) + 0`

```
(S 0) + 0
⇝ case (S 0) of { ...; S n' -> S (n' + 0) }
⇝ S (0 + 0)
        0 + 0
        ⇝ case 0 of { 0 -> 0; ... }
        ⇝ 0
⇝ S 0
```

Zeno uses indentation and line breaks to represent the tree structure. Thus, a node with one child is separated from its child by a new line, and appears at the same indentation level as the child. For example, the node at line 3 has a unique child which appears on line 4. A node with multiple children (or, in other words, a node with multiple branches) is represented by indentation for each branch, and by separating the branches by an empty line. For example, the node at line 1 has two children, one at line 3 and the other at line 7. Each line of the proof output contains the *proof step* taken to reach this line from its parent and is given in square brackets `[...]` at the start of the line (these steps are explained in the next Section), and the property that is to be proven (`True` indicates that the proof branch is complete).

Fig. 5 Zeno's proof that `0` is a right-identity for `(+)`

```
1  [goal] x + 0 = x
2
3   > [ind x => 0] 0 + 0 = 0
4   > [def] 0 = 0
5   > [eql] True
6
7   > [ind x => S x'] S x' + 0 = S x'
8       with x' + 0 = x'
9   > [def] S (x' + 0) = S x'
10  > [hyp x' + 0 = x'] S x' = S x'
11  > [eql] True
12
13 Proven: x + 0 = x
```

The parent relationship expresses entailment; the property at a node is a consequence of the conjunction of the properties of all its children. For example, `0 + 0 = 0` from line 3 is a consequence of `0 = 0` from line 4. Similarly, the property `x + 0 = x` from line 1 is a consequence of the conjunction of `0 + 0 = 0` from line 3 and of `x' + 0 = x` ⇒ `S x' + 0 = S x'` from line 7. (Note that the `with` on

line 8 shows we have added a property to our list of inductive hypotheses down this branch and is always part of a previous inductive step.)

At the end of the output Zeno gives all the auxiliary lemmas that it has proven. For example, as we see on line 19 of **Fig. 7**, in order to prove that `0` is a right-identity for `max`, Zeno discovers and proves the auxiliary lemma that `0` is the only number less than or equal to `0`.

4 Proof steps

We now describe the possible steps that Zeno can apply to a goal. Each step reduces the current property into the conjunction of one or more simpler properties, or directly proves the property to be true.

4.1 `[eql]` - Reflexivity of equality

This step reduces the goal τ `=` τ `:-` $\overline{\varphi}$ to `True`. That is, if the two sides of the goal's consequent are syntactically equal, then the goal is trivially true. This step is trivially sound, because **HC** is a pure functional language, where syntactic equality implies equality.[2] Examples of this step appear on lines 5 and 11 of the proof in **Fig. 5**.

4.2 `[def]` - Applying function definitions

This step applies function definitions, and thus reduces the goal to a simpler one. This step is sound, because function definitions can be seen as background lemmas for any proof and can be applied as such.

Examples of this step can be found on lines 4 and 9 of **Fig. 5**. On line 9, for example, Zeno applied the definition of `(+)`, and reduced the term `S x' + 0` to the term `S (x' + 0)`.

When trying to apply a function definition we can also use the antecedents of our goal to rewrite any case-analysed expressions. This is particularly useful after a *case-split* step – more later.

4.3 `[ind x =>` τ`]` - Proof by structural induction

This step describes the inductive step where the variable `x` has the same value as term τ. This line in the proof output will be followed by zero or more lines of the form "`with` φ", one for each induction hypothesis φ added down this branch. Multiple nested `[ind]` steps represent the step by step construction of an induction scheme for a proof.

To apply structural induction on a variable `x` of a type `T`, Zeno constructs a separate proof branch for each constructor of `T`[3]. For each such proof branch,

[2] In an imperative language such a step would not be sound and we would need to make its application conditional on constancy annotations or framing.

[3] Obviously, induction is not applicable for function types.

and for each recursively typed argument of the branch's constructor, Zeno adds an inductive hypothesis down that branch. The inductive hypotheses is identical to the original goal, except that the inductive variable `x` is replaced by each recursively typed argument variable in turn.

Lines 3 and 7 of our example proof in **Fig. 5** represent the two branches needed for an inductive proof over `x`. As `x` is of type `Nat`, Zeno constructs one branch for each of the two constructors of `Nat`, i.e. one branch for when `x` is `0` and the other for when `x` is `S x'` for some new `x'` of type `Nat`. Because `x'` has the same type as `x`, the inductive hypothesis down this branch is `x' + 0 = x'`, as shown by line 8.

Note that in a new inductive hypotheses every variable will be universally quantified except for the inductive one, since every variable is implicitly universally quantified in the goal from which it was generated. Take for example the property `x + y = y + x`, which is really ∀`x`.∀`y`.`x + y = y + x`. If Zeno were to perform induction on `x` then down the `S x'` branch it would get the hypothesis ∀`y`.`x' + y = y + x'`. It can then match any variable to the `y` when using this hypothesis, rather than just the original `y` from the goal.

In order that this preserves a well-founded ordering for our induction scheme when a variable is inducted upon, if it exists ∀-quantified in an existing induction hypothesis this quantifier is removed and the variable replaced with its new value down this branch. For example if we had the induction hypothesis ∀`y`.`x' + y = y + x'` and we perform an induction step on `y`, down the branch `[ind y => 0]` this hypothesis would become `x' + 0 = 0 + x'`.

4.4 `[hyp` φ`]` - Application of an inductive hypothesis

This step reduces the goal by applying the induction hypothesis φ to rewrite the goal. On line 9 of **Fig. 5** Zeno reduced the goal `S (x' + 0)` to `S x'` by using the hypothesis `x' + 0 = x'`.

When performing induction on a goal with antecedents, Zeno creates a hypothesis which also has antecedents. This means we can only apply the hypothesis if the antecedents have been satisfied by the antecedents of the current goal. An example of this could be seen in Zeno's proof of that `(<=)` is a total ordering, i.e. `x <= y = True :- y <= x = False`[4].

4.5 `[fac]` - Goal factoring

Goal factoring is applicable on goals where the same function appears outermost on both sides of the equation of the consequent. For example, `(x + 0) + y = x + y`, can be factored to `x + 0 = x` and `y = y`.

The formal presentation of this step demonstrates its soundness:

$$\bigwedge_i (\tau_i = \tau_i' \text{ :- } \overline{\varphi}) \quad \Rightarrow \quad (f\ \overline{\tau} = f\ \overline{\tau'} \text{ :- } \overline{\varphi})$$

[4] This is not detailed here but can be seen by proving `prop_nat_leq_total` on TryZeno `http://www.doc.ic.ac.uk/~ws506/tryzeno`

4.6 `[gen τ => x]` - Generalising term τ to variable `x`

This step reduces a goal to a more general one by replacing all instances of a sub-term τ in the goal with a new variable `x`. For example, the application of `[gen x + y => z]` on `(x + y) + 0 = x + y` gives us `z + 0 = z`.

Generalisation is a well established technique in inductive theorem proving[9] and is usually the step which leads Zeno to discover important auxiliary lemmas. For example, the proof that the application of list reversal twice returns the original list, highlighted in **Fig. 6**, depends on the lemma `rev (ys ++ (x : [])) = x : rev ys`, which is found by our generalisation step and subsequently proven.

Fig. 6 Generalization discovers auxiliary lemmas - highlighted example

```
rev (rev xs) = xs
 ...
 > [hyp ...] rev (rev xs' ++ (x : [])) = x : rev (rev xs')
 > [gen rev xs' => ys] rev (ys ++ (x : [])) = x : rev ys
 ...
Proven: rev (ys ++ (x : [])) = x : rev ys
        rev (rev xs) = xs
```

Fig. 7 Proof that `0` is a right-identity for `max`

```
1  [goal] max x 0 = x
2
3   > [cse x <= 0 => False] max x 0 = x :- x <= 0 = False
4   > [def] x = x :- x <= 0 = False
5   > [eql] True
6
7   > [cse x <= 0 => True] max x 0 = x :- x <= 0 = True
8   > [def] 0 = x :- x <= 0 = True
9
10   >> [ind x => 0] 0 = 0 :- 0 <= 0 = True
11   >> [def] 0 = 0 :- True = True
12   >> [eql] True
13
14   >> [ind x => S x'] 0 = S x' :- S x' <= 0 = True
15       + 0 = x' :- x' <= 0 = True
16   >> [def] 0 = S x' :- False = True
17   >> [con] True
18
19 Proven: 0 = x :- x <= 0 = True
20         max x 0 = x
```

4.7 [cse τ => τ'] - Case-split on τ

This step corresponds to case-splitting on a term τ, and in particular to the branch where τ is taken to have the form τ'. In case-splitting, as with induction, Zeno creates one branch for each different constructor of the type of τ. Zeno then adds the equality $\tau = \tau'$ to the antecedents of the goal.

Lines 3 and 7 of the proof in **Fig. 7** represent the two branches of a case-split on `x <= 0`. Line 3 introduces the branch in which `x <= 0` is `False`, and line 7 introduces the branch in which it is `True`. On line 4 Zeno then uses the new antecedent to apply the definition of `max` to `max x 0`, reducing it to `x`.

4.8 [con] - Contradiction

This step reduces a goal to `True` by finding a contradiction in its antecedents, and is sound because $\bot \Rightarrow \varphi$ is true for any φ, even a false φ.

A contradiction is found when we have an equality between two different constructor terms in one of our antecedents. In the last line of **Fig. 7** we have proven our goal to be true since `False = True` is in the antecedents.

4.9 [hcn φ] - Adding an inductive hypothesis to the goal conditions

This step moves an inductive hypothesis into the list of goal conditions and will only be applicable if it allows us to perform a subsequent generalisation step with a common sub-term of the goal. This also helps us ground any universally quantified hypothesis variables within this common sub-term. For example, in **Fig. 8** it finds the necessary auxiliary lemma `sorted (insert y zs) = True :- sorted ys = True`.

Fig. 8 Highlighted proof that insertion sort (`sort`) produces a `sorted` list

```
[goal] sorted (sort xs) = True

 > [ind xs => y : ys] sorted (sort (y : ys)) = True
     with sorted (sort ys) = True
 > [def] sorted (insert y (sort ys)) = True
 > [hcn sorted (sort ys) = True]
     sorted (insert y (sort ys)) = True
       :- sorted (sort ys) = True
 > [gen sort ys => zs]
     sorted (insert y zs) = True :- sorted zs = True
 ...

Proven: sorted (insert y zs) = True :- sorted zs = True
        sorted (sort xs) = True
```

4.10 [icn φ] - Inferring a new goal condition

This step adds a new goal condition φ by inferring it from the existing conditions. This step is used instead of a case-split when one case is a theorem (i.e. is a consequence) of the goal conditions. Before Zeno starts a case-split on τ it checks whether it can prove that τ = τ' for any τ' that is a constructor term of the type of τ.

For example, in **Fig. 9** the first step could be a case-split on `x <= y` (because of the definition of `max`). Zeno however can prove `x <= y = True :- y <= x = False`, meaning that `x <= y = True` is a theorem of the conditions of the goal; therefore Zeno discounts the branch for `x <= y = False`. This discounted branch could also have been proven by a contradiction in the goal conditions, but we found that this method of checking before a condition is added afforded a massive performance increase. This step will always have two branches, the first one showing the proof of the new condition and the second continuing the original proof with this new condition added.

Fig. 9 Inferring a new goal condition - highlighted example

```
[goal] max x y = y :- y <= x = False

  [icn] x <= y = True :- y <= x = False
  ...

  [icn x <= y = True]
    max x y = y :- y <= x = False, x <= y = True
  [def] y = y :- y <= x = False, x <= y = True
  [eql] True

Proven: x <= y = True :- y <= x = False
        max x y = y :- y <= x = False
```

5 Trimming the search space

For each goal, several different proof steps are applicable, and each proof step may be applicable in different parts of a term. As an example, for the goal `(x + y) + (u + v) = (x + u) + (y + v)`, there are seventeen different possibilities, i.e. goal factoring, induction for each of the four variables, generalization for each of the six subterms, as well as case analysis for each of the of the six subterms.

In general Zeno will try any applicable step, backtracking if a step does not lead to a proof. We have however developed several heuristics for reducing the number of applicable steps.

5.1 Deterministic steps - `[eql]`, `[con]`, `[def]`

These three steps have the highest priority; Zeno applies them whenever it can do so. Whenever Zeno generates a new goal it applies any function definitions it can (`[def]`), then checks whether the two sides of the consequent are syntactically equal (`[eql]`), or whether the antecedents contain any contradictions (`[con]`). If neither is the case, the other proof steps will then be tried.

This eager application of function definitions could cause an infinite loop when presented with a function that does not terminate for a certain input, thus restricting Zeno to operating only over total functions.

5.2 Controlling `[ind]`, `[cse]` and `[gen]` with critical terms

The *critical term* within our goal guides the decision of *which step* to apply, and *how* to apply it. It makes the choice so as to ensure function definitions will be applicable later on.

Fig. 10 Critical term of a term

$$\frac{}{\texttt{x} \gg \texttt{x}} \qquad \frac{\tau_1 \rightsquigarrow^* \texttt{case}\ \tau_2\ \texttt{of}\ \{\ \overline{A}\ \} \qquad \tau_2 \notin \tau_1}{\tau_1 \gg \tau_2}$$

$$\frac{\tau_1 \rightsquigarrow^* \texttt{case}\ \tau_2\ \texttt{of}\ \{\ \overline{A}\ \} \qquad \tau_2 \in \tau_1 \qquad \tau_2 \gg \tau_3}{\tau_1 \gg \tau_3}$$

A critical term τ_c of a term τ is given by the relationship $\tau \gg \tau_c$ defined in **Fig. 10**, where $\in$ stands for the sub-term relationship, i.e. `g x` $\in$ `f (g x)`. A critical term is one we would have to replace with a constructor term in order to apply a function definition.[5] Therefore, if the critical term is a variable, Zeno tries performing induction on it, and if it is another term, Zeno tries performing case-splitting on it. The usage of critical terms to direct these steps means that the proof is moving towards being able to apply a function definition later on.

In **Fig. 11** we give two examples of the derivations of critical terms. The first example is `(x + y) + z` which has the same critical term as `x + y`, since this is a sub-term. The critical term of `x + y` is `x`, since this is a case-analysed variable. The second example, `max x y`, has the critical term `x <= y`, since evaluation of `max x y` will need to perform case-analysis on `x <= y`, and `x <= y` is not a sub-term of `max x y`.

Every term has no more than one critical term. Terms without critical terms are those which do not have at the outermost level a function whose definition could be applied e.g., a constructor term like `S x`, or those which can be evaluated

[5] Note that as for the `[def]` steps, the calculation of critical terms may not terminate when we consider non-terminating functions.

Fig. 11 Finding the critical terms of `(x + y) + z` and `max x y`

$$\dfrac{\dfrac{\dfrac{}{\texttt{x} \gg \texttt{x}}}{\texttt{x} \in \texttt{x + y} \quad \texttt{x + y} \rightsquigarrow \texttt{case x of \{ } \overline{A} \texttt{ \}}}}{\dfrac{\texttt{x + y} \gg \texttt{x}}{\texttt{x + y} \in \texttt{(x + y) + z} \quad \texttt{(x + y) + z} \rightsquigarrow \texttt{case (x + y) of \{ } \overline{A} \texttt{ \}}}}{\texttt{(x + y) + z} \gg \texttt{x}}$$

$$\dfrac{\texttt{x <= y} \notin \texttt{max x y} \quad \texttt{max x y} \rightsquigarrow \texttt{case (x <= y) of \{ } \overline{A} \texttt{ \}}}{\texttt{max x y} \gg \texttt{x <= y}}$$

fully to a new term. The former we can ignore whereas the latter should not occur, since we have already performed any available `[def]` step.

The critical terms of $\tau_1 = \tau_2$ are the union of the critical terms of τ_1 and τ_2. The critical terms of the goal φ_1 `:-` $\varphi_2, ..., \varphi_n$ are the union of the critical terms of φ_1 ... φ_n. For example the critical terms of `max x y = x :- x <= y = False` consists of those from `max x y`, from `x`, from `x <= y` and from `False`; this gives us two critical terms in total, i.e. `x` and `x <= y`.

5.3 Usable critical terms

Every critical term is accompanied by a *critical path*, which represents the symbolic execution that would have led to that term. When a critical term is chosen, its critical path is stored in the corresponding node of the proof tree. A critical term is only usable if its critical path is not similar to to any critical path stored anywhere in the current branch.

Controlling Induction Zeno tries induction only on a variable `x` appearing among the usable critical terms of the current goal. For example, in the first line of the proof of `x + 0 = x` in **Fig. 5**, variable `x` is the critical term of both sides. Therefore Zeno can perform induction on `x`; this step allows the application of the definition of `(+)` as the next step. On the other hand in line 3 of **Fig. 6** the only critical term is `xs'`, but this is not usable as it is the product of an earlier inductive step `[ind xs => x : xs']` which had a similar critical path, so Zeno must choose another step.

Controlling Case-Splitting Zeno tries a case-split only on a usable critical term that is not a variable. For example, in the the first line of **Fig. 7**, the critical term of `max x y` is `x <= y`, and therefore Zeno tries a case-split on `x <= y`. After the case split, the function definition of `max` is applicable down either branch.

Controlling Generalization Zeno tries generalisation only on a term containing a variable which is a (not necessarily usable) critical term of the goal. In our generalisation example in **Fig. 6** the goal `rev (rev xs' ++ (x : [])) = x : rev (rev xs')` has the critical term `xs'`. The term `rev xs'` contains this critical term so we can apply generalisation to it.

5.4 Counterexamples

Zeno searches for counterexamples to its original goal, and also to every newly generated goal which is a sufficient but not necessary condition of the previous goal. Such sufficient but not necessary new goals are created by the proof steps `[gen]`, `[fac]` and `[hyp]`. This approach stops Zeno from going down "false paths" in a proof search. The inspiration to this approach came from ACL2s.

In its search for counterexamples, Zeno creates a set of test values by applying the critical terms technique, and restricts itself to a finite set by applying the usability technique. Our approach is similar to SmallCheck[13], in that both use symbolic execution to generate values, but differs in that SmallCheck uses depth of recursion to restrict to a finite set. In contrast, ACL2s generates a constant number of completely random values, much more like the tool QuickCheck[6].

5.5 The effect of trimming

If we revisit our property from the start of this section, `(x + y) + (u + v) = (x + u) + (y + v)`, we had seventeen applicable proof steps. Our critical term technique reduces four potential induction steps and six potential case-splits to just one applicable induction step, `x`. It also reduces our potential generalisations to four, `x + y`, `x + u` and both sides of the equality, but these all have discoverable counterexamples so are discounted. The potential factoring to `x + y = x + u` and `u + v = y + v` is also dismissed with a counterexample. Zeno is therefore left with only a single applicable step, induction on `x`.

6 Comparison with IsaPlanner and ACL2s

We now compare Zeno, IsaPlanner[5, 8, 10] and ACL2s[2, 7] in terms of their respective performance on a set of 87 properties listed on a page from the IsaPlanner website[6]. This set also appears in one of the IsaPlanner authors' papers[10].

Of the 87 properties Zeno could prove 82, and 2 are false, leaving 3 properties unproven. IsaPlanner could prove 47 properties, while ACL2s was able to prove 74 properties. There were no properties which ACL2s or IsaPlanner could prove and Zeno could not. There were 28 which ACL2s could prove over IsaPlanner and one property IsaPlanner could prove over ACL2s, though this was over a binary tree which has a much more natural representation in IsaPlanner. Refer to **Fig. 12** for list of properties for which each tool could *not* find a proof.

The longest Zeno proof took 2.084s, while most proofs took less than 0.001s, running on an Intel Core i5-650 processor. The other two tools produced proofs in similar times. The Haskell code used to test Zeno and the LISP code given to ACL2s can be found at `http://www.doc.ic.ac.uk/~ws506/tryzeno/comparison`. As functions in ACL2s are untyped we supplied the type information ourselves

[6] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/case-analysis-rippling.txt`

through proven theorems. We noted that without this type information ACL2s was unable to prove properties 6, 7, 8, 9, 15, 18, and 21.

A complex property Zeno was able to prove over the other tools which is not from this suite is the idempotence of insertion sort, i.e. `sort (sort xs) = sort xs`. This takes Zeno around 13s to complete, has a proof tree 14 steps deep and discovers two sub-properties: `insort x (insort y xs) = insort y (insort x xs)` and `sort (insort x xs) = insort x (sort xs)`.

Properties which IsaPlanner and Zeno could prove over ACL2s, also not from this suite, were `x * (S 0) = x`, `x * (y + z) = (x * y) + (x * z)` and `x ^ (y + z) = (x ^ y) * (x ^ z)`. We also found a property which IsaPlanner could prove over both ACL2s and Zeno, the symmetry of multiplication, i.e. `x * y = y * x`.

One property from the test suite which no tool could solve was `rev (drop i xs) = take (len xs - i) (rev xs)`. The proof is blocked by the required sub-property `take (len xs) (rev xs) = take (len xs) (rev xs ++ (x : []))`. This seems to need the lemma `len xs = len (rev xs)` to generalise it to `take (len ys) ys = take (len ys) (ys ++ (x : [])`, which can then be solved.

Fig. 12 Properties *not* proven by each tool

Tool	**Properties not proven**	**Total**
IsaPlanner	48 - 85	38
ACL2s	47, 50, 54, 56, 72, 73, 74, 81, 83, 84, 85	11
Zeno	72, 74, 85	3

7 Conclusions and Future Work

We described our tool Zeno along with the steps it uses in constructing proofs and the heuristics we have developed for trimming the search space for these steps. Rather than pre-compute induction schemata we allow our tool to gradually build one up through multiple steps, guided by critical terms. It discovers proofs in a fully automated way, requiring no auxiliary lemmas to be suggested by the user, and indeed often discovers many interesting lemmas within a larger proof.

Zeno has been developed to explore only fully automated proof techniques for Haskell style functional programs. Although it compared favourably to IsaPlanner and ACL2s in this area these tools have a much wider domain.

IsaPlanner uses rippling to guide the application of background lemmas. Zeno does not support any background lemmas aside from function definitions. In future work we plan to integrate a technique such as rippling to allow for these. IsaPlanner is also more generally a proof-planner for the Isabelle system, allowing it to be used within larger, human guided proofs, and ensuring any generated proofs are sound. Integration into a proof system like Isabelle, perhaps as a tactic for IsaPlanner, would be a useful next step.

ACL2s is an industrial strength theorem proving environment which has been used in the verification of properties of real-world systems. Unlike IsaPlanner it

can prove properties over full first-order logic, which we might consider as a future extension.

The three properties which Zeno was unable to prove from our test suite are those requiring lemmas which are not a generalisation of a sub-goal. In future we would like to be able to prove properties such as these through intelligent methods of finding these necessary lemmas. One such technique might be the random perturbation of property terms as in IsaCoSy[11].

Acknowledgements We thank Davide Ancona and the SLURP group for feedback, Tristan Alwood for lots of interesting discussions, Moa Johansson for her help with IsaPlanner, and Krysia Broda for her knowledgeable advice.

References

1. Haskell 98 - A non-strict, purely functional language. Available from `http://www.haskell.org/definition` (February 1999)
2. Boyer, R.S., Moore, J.S.: A theorem prover for a computational logic (1990)
3. Bundy, A.: The use of explicit plans to guide inductive proofs. In: Proceedings of the 9th International Conference on Automated Deduction. pp. 111–120. Springer-Verlag, London, UK (1988)
4. Bundy, A., Harmelen, F.V., Hesketh, J., Smaill, A., Stevens, A.: A rational reconstruction and extension of recursion analysis. In: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. pp. 359–365. Morgan Kaufmann (1992)
5. Bundy, A., Stevens, A., Harmelen, F.V., Ireland, A., Smaill, A.: Rippling: A Heuristic for Guiding Inductive Proofs (1993)
6. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. pp. 268–279. ICFP '00 (2000)
7. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: "The ACL2 Sedan". International Conference on Software Engineering Companion pp. 59–60 (2007)
8. Dixon, L., Fleuriot, J.: IsaPlanner - a prototype proof planner in Isabelle. In: Proceedings of CADE03, LNCS. pp. 279–283. Springer (2003)
9. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. Journal of Automated Reasoning 16, 16–1 (1995)
10. Johansson, M., Dixon, L., Bundy, A.: Case-Analysis for Rippling and Inductive Proof. In: ITP. pp. 291–306 (2010)
11. Johansson, M., Dixon, L., Bundy, A.: Conjecture Synthesis for Inductive Theories (2010), `http://dream.inf.ed.ac.uk/projects/isaplanner/papers/synth-ind-theories-draft-2010.pdf`
12. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5 (1989)
13. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In: Proceedings of the first ACM SIGPLAN symposium on Haskell. pp. 37–48. Haskell '08 (2008)