Zeno: An Automated Prover for Properties of Recursive Data Structures

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach

Imperial College London

Abstract. Zeno is a new tool for the automatic generation of proofs of simple properties of functions over recursively defined data structures. It takes a Haskell program and an assertion as its goal and tries to contruct a proof for that goal. If successful, it converts the proof into Isabelle code. Zeno searches for a proof tree by iteratively reducing the goal into a conjunction of sub-goals, terminating when all leaves are proven true.

This process requires the exploration of many alternatives. We have adapted known, and developed new, heuristics for the reduction of the search space. Our new heuristics aim to promote the application of function definitions, and avoid the repetition of similar proof steps.

We compare with the rippling based tool IsaPlanner and the industrial strength tool ACL2s on the basis of a test suite from the IsaPlanner website. We found that Zeno compared favourably with these tools both in terms of theorem proving power and speed.

1 Introduction

Proving algebraic properties of recursive functions usually requires inductive reasoning. SMT solvers[6], while successfully applied in imperative program verification[1,2], can only construct such proofs when supplied with induction schemata. Recent work[12] automatically sets up the base case and the induction step, and then passes the proof obligation to an SMT solver, and has been successful in proving several such properties. Nevertheless, such an approach runs into difficulties with proofs which require several inductive sub-proofs.

Such cases require proof systems which explicitly handle induction, such as ACL2s[3,7] or IsaPlanner[8]. ACL2 is an industrial strength proof system based on the Boyer-Moore technique, recently extended to ACL2s, the "Sedan Edition". IsaPlanner is a proof-planning framework for the Isabelle[13] proof system.

To address the huge search space ensuing from the fact that at each proof step several induction steps and case-splits are applicable, ACL2 uses *recursion-analysis*[3] while IsaPlanner enumerates every free variable or potential split. IsaPlanner features the rippling technique for applying function definitions, "preferring" steps which make it possible to apply the induction hypothesis. IsaPlanner can also discover auxiliary lemmas needed for a larger proof by appealing to *proof critics* when a proof search is unable to progress [9].

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 407–421, 2012.

We propose a novel approach, which differs from those above in the following aspects: First, in contrast to rippling, we "prefer" steps which make it possible to apply function definitions, and thus we "bring the proof forwards". Second, through intelligently chosen generalization or CUT steps, we introduce intermediary auxiliary lemmas which the tool tries to prove. Third, we avoid revisiting proof steps which have recently been tried out, and thus we reduce the search to finite space. Furthermore, we adopted some known techniques, e.g. a search for counterexamples before trying to prove new sub-goals.

To support our approach, we introduce a concept called a *critical term*, which is either a variable which appears in the original term (guiding the tool to apply induction on this variable), or a new term which was not a part of the original term (guiding the tool to apply a case-split on this new term), or a "non-minimal term" (guiding the tool to discover an auxiliary lemma). We also introduce *critical paths*, which reflect the cases already visited in a proof branch and avoid applying steps whose paths expand those of earlier steps.

Based on these ideas, we built Zeno, a fully automated verification tool which requires no extra lemmas to be supplied by the user, and often discovers the necessary auxiliary lemmas. Zeno supports **HC**, a minimal functional language with a small language of properties which allows for algebraic properties with entailment. From the constructed proof tree, Zeno creates a proof in Isabelle.

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

This paper is organised as follows. **Section 2** defines the input language **HC**. **Section 3** describes the steps Zeno uses to construct its proofs. **Section 4** describes the heuristics which trim the search space. **Section 5** compares Zeno, IsaPlanner and ACL2s and discusses our Isabelle proof output. In **Section 6** we conclude and discuss future work.

Download files and instructions are at `haskell.org/haskellwiki/Zeno`, and try out Zeno online at `tryzeno.org`.

2 Zeno's Internal Functional Language HC

In this section we describe **HC**, Zeno's internal language. **HC** is annotated with labels, which are used by the heuristics for trimming the search space. These labels will not be of interest before **Section 4**, and are written in this colour.

Fig. 1 describes **HC**, a slightly simplified version of GHC Core, the internal language of the Glasgow Haskell Compiler. **HC** is created from GHC Core through an almost direct translation through the GHC API - Zeno uses GHC for parsing and type-checking. For simplicity, in this paper we do not present polymorphic typing, even though Zeno is able to handle it.

Fig. 2 contains an example Haskell program, while **Fig. 3** contains its representation in **HC** (GHC has inlined the definition of `(&&)` in `ord`). We use infix operator syntax in **HC** in the same way as Haskell, as well as the built-in `Boolean` data type and list type and syntax - `[]` for the empty list and `(:)` for cons.

Fig. 1. Zeno's internal language **HC**

	`x,y` $\in Var$	`f,g` $\in Fun$	`K` $\in Con$	`T` $\in TypeVar$	`i` $\in Id$
$E \in Expr$	::=	Var`<`$Path^*$`>` | Fun | Con	Variable/Function/Constructor
	|	`(`$Expr$ $Expr$`)`	Application
	|	`\`Var `->` $Expr$	Lambda abstraction
	|	`case<`Id`>` $Expr$ `of { `Alt^* `}`	Pattern
Alt	::=	Con Var^* `->` $Expr$	A pattern match
	|	`_ ->` $Expr$	`_` is the default pattern
$Bind$	::=	`let` Fun `=` $Expr$	Non-recursive definition
	|	`letrec` Fun `=` $Expr$ (`and` Fun `=` $Expr$)*	(Mutually) recursive definitions
$TypeDef$	::=	`data` $TypeVar$ `=` Con $Type^*$ (`|` Con $Type^*$)*	Data-type definition
$\tau \in Type$	::=	$TypeVar$	Simple type
	|	$Type$ `->` $Type$	Function type
$Prog$	::=	$TypeDef^*$ $Bind^*$	An **HC** program
$p \in Path$	::=	$[] \mid Id : Path$	A critical path
$P \in Prop$	::=	`all x`* `.` $Cls \mid Cls$	Properties
$\Phi \in Cls$	::=	$Prop^*$ `==>` $Eq \mid Eq$	Clauses
$\varphi \in Eq$	::=	$Expr$ `=` $Expr$	Equations

Fig. 2. Example program in Haskell

```
data Nat  = Zero | Succ Nat

(<=) :: Nat -> Nat -> Bool
Zero <= _ = True; Succ x <= Zero = False
Succ x <= Succ y = x <= y

ord :: [Nat] -> Bool
ord [] = True; ord [x] = True
ord (x:y:ys) = x <= y && ord (y:ys)

ins :: Nat -> [Nat] -> [Nat]
ins n [] = [n]
ins n (x:xs) | n <= x = n:x:xs | otherwise = x:(ins n xs)

sort :: [Nat] -> [Nat]
sort [] = []; sort (x:xs) = ins x (sort xs)
```

Fig. 3. The interpretation of **Fig. 2** in **HC**, all uses of a variable have implicitly empty paths

```
data Nat = Zero | Succ Nat

letrec (<=) = \x -> \y -> case<lq1> x of
  { Zero -> True; Succ x' -> case<lq2> y of
    { Zero -> False; Succ y' -> x' <= y' } }

letrec ord = \ns -> case<o1> ns of
  { [] -> True; x:xs -> case<o2> xs of
    { [] -> True; y:ys -> case<o3> (x <= y) of
      { True -> ord (y:ys); False -> False } } }

letrec ins = \n -> \ns -> case<i1> ns of
  { [] -> n:[]; x:xs -> case<i2> (n <= x) of
    {  True -> n:x:xs; False -> x:(ins n xs) } }

letrec sort = \ns -> case<s1> ns of
  { [] -> True; x:xs -> ins x (sort xs) }
```

Fig. 1 also defines the language in which we express properties P. These have the obvious meaning, where free variables are implicitly universally quantified. Thus, `ord (sort as) = True` asserts that `sort` returns an ordered list.

Fig. 4 defines reduction, $\overline{P} \vdash E \rightsquigarrow E'$, which means that E reduces to E' given the facts $\overline{P}$. The first rule uses call-by-value reduction ($\overset{bv}{\rightsquigarrow} \subseteq Expr \times Expr$). For example, as shown in **Fig. 5**, `ord (ins b (d:ds))` reduces to `ord (b:d:ds)`, using an intermediate step which applies `b <= d = False`. Even though our input syntax is Haskell, the evaluation is eager - thus our proofs talk about finite structures only. Some expressions, *e.g.* pattern matching, are not conducive to proofs. To distinguish those expressions that *are* conducive, we introduce in **Fig. 4** *terms*, $Term \subseteq Expr$, which are expressions with a name leftmost, and *normal terms*, $NormalTerm \subseteq Term$, which cannot be further reduced to other terms.

Notation. We use $\rightsquigarrow_+$ for the transitive, and $\rightsquigarrow_*$ for the reflexive transitive closure of $\rightsquigarrow$. For symbols s ranging over S, we use $\overline{s}$ to range over $\wp(S)$, e.g. $\overline{P} \in \wp(Prop)$. Functions are lifted to sets in the obvious way, e.g. for $\overline{s} \in \wp(S)$ and $f \in S \to X$, we have $\overline{f(s)} \in \wp(X)$. We use a syntactic notion of expression equality, where we ignore critical pairs, since, as we will see, these are annotations and do not change the semantics of an expression; for example, `f x<`p_2, p_3`> = f x<`p_1`>`. Set membership operators between expressions ($E \in E'$) denote the reflexive sub-expression relationship (ignoring critical paths), e.g. `f x<`p_1`> ∈ g (f x<`p_2`>) y`.

Fig. 4. Reduction modulo rewriting, $Terms$ and $NormalTerms$

$$\frac{E \overset{bv}{\rightsquigarrow} E'}{\overline{P} \vdash E \rightsquigarrow E'} \qquad \frac{(E' = E'') \in \overline{P} \ \vee \ (E'' = E') \in \overline{P}}{\overline{P} \vdash E \rightsquigarrow E[E' \coloneqq E'']_c}$$

$$Term = Var \cup Fun \cup Con \cup \{\ (E\ E') \mid E \in Term,\ E' \in Expr\ \}$$
$$NormalTerm = \{\ E \in Term \mid \nexists E' \in Term\ .\ E \overset{bv}{\rightsquigarrow}_+ E'\ \}$$

Fig. 5. An example of reduction modulo rewriting

`(b <= d) = False` $\vdash$ `ord (ins b (d:ds))`	Starting expression
$\rightsquigarrow_*$ `ord (case<i2> b <= d of { False -> b:d:ds; ... })`	Unfold `ins` definition
$\rightsquigarrow$ `ord (case<i2> False of { False -> b:d:ds; ... })`	Apply fact as rewrite
$\rightsquigarrow_*$ `ord (b:d:ds)`	Reduce pattern match

3 Proof Steps

In this section we discuss the individual proof steps used in Zeno's proofs. We define these steps through the rules in **Fig. 6**.

Zeno constructs proof-trees by applying these rules "backwards". As usual, in a given situation, several different rules may be applicable, and a rule may be applicable in several different ways. Zeno searches for a proof in a depth-first manner. We reduced the search space considerably by prioritizing some rules over others, and by restricting the applicability of some of the rules by requiring further conditions. These further conditions are expressed through premises in the rules written in this colour. In this section we ignore the extra conditions, and will discuss them in **Section 4**.

Fig. 7 describes parts of Zeno's proof that `ord (sort as) = True`, i.e. that our insertion sort function produces ordered lists. For simplicity, we write E to mean E `= True` and `not` E to mean E `= False`, e.g. `b <= d` means `(b <= d) = True`. We use Greek letters between α and μ to denote particular steps in the proof.

Steps (EQL) and (CON) are the only two not to follow from a sub-proof and so can close a proof branch. (EQL) means that both sides of the property consequent are syntactically equal so the goal is true, e.g. in step $[\delta]$ and $[\theta]$ of our example we close these branches as we have `True = True` as our goal. (EXP) applies our previously defined reduction rule to a property, e.g. in step $[\lambda]$ we apply the rewrite shown in **Fig. 5**. (FAC) means that with expression application on both sides of our goal equation it suffices to prove equality between both functions and both arguments respectively - known as "factoring". (USE) converts an antecedent in $Prop$ to one in Eq, i.e. `all` $\overline{\mathtt{x}}$ `.` $\overline{P'}$ `==>` φ' is converted to $\varphi'[\overline{\mathtt{x}} \coloneqq E_{\overline{\mathtt{x}}}]$, by choosing a value for each quantified variable ($\overline{\mathtt{x}}$) and proving all its antecedents ($\overline{P'}$); φ' can now be used in a later step like (EXP) or (CON).

Fig. 6. Zeno's proof steps

$$(\text{EQL})\ \frac{E =_c E'}{\vdash \overline{P} \texttt{ ==> } (E \texttt{ = } E')} \qquad (\text{CON})\ \frac{\texttt{K} \neq \texttt{K}' \qquad (\texttt{K}\ \overline{E} \texttt{ = K}'\ \overline{E}') \in \overline{P}}{\vdash \overline{P} \texttt{ ==> } \varphi}$$

$$(\text{EXP})\ \frac{\vdash (\overline{P} \texttt{ ==> } \varphi)\,[E \coloneqq E']_c \qquad \overline{P} \vdash E \rightsquigarrow_* E' \qquad E' \in \mathit{NormalTerm}}{\vdash \overline{P} \texttt{ ==> } \varphi} \qquad (\text{FAC})\ \frac{\vdash \overline{P} \texttt{ ==> } (E_f \texttt{ = } E_f') \qquad \vdash \overline{P} \texttt{ ==> } (E_a \texttt{ = } E_a')}{\vdash \overline{P} \texttt{ ==> } (E_f\ E_a \texttt{ = } E_f'\ E_a')}$$

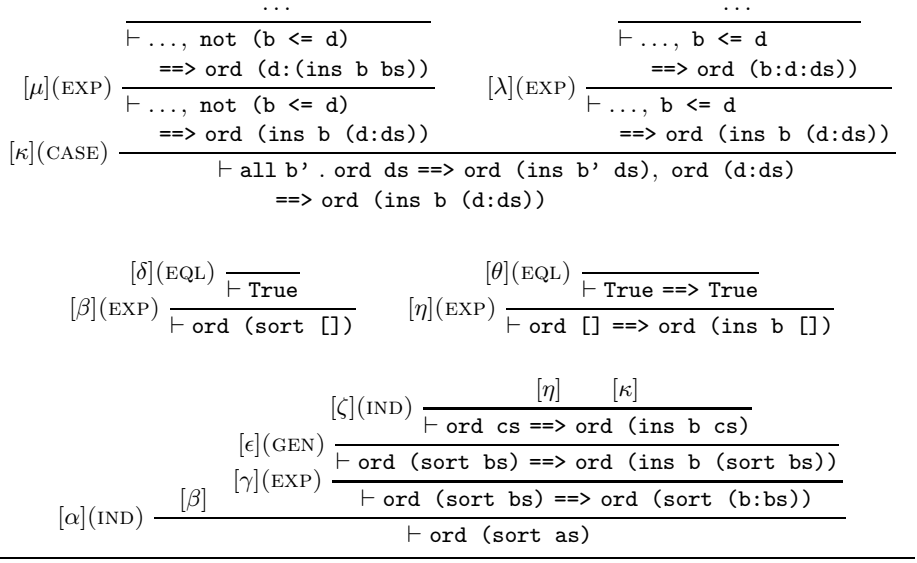
$$(\text{USE})\ \frac{\vdash \left\{\varphi'[\overline{\texttt{x} \coloneqq E_{\texttt{x}}}]\right\} \cup \overline{P} \texttt{ ==> } \varphi \qquad \left(\texttt{all } \overline{\texttt{x}}\ .\ \overline{P}' \texttt{ ==> } \varphi'\right) \in \overline{P} \qquad \mathit{foreach}\ \left(\texttt{all } \overline{\texttt{y}}\ .\ \overline{P}'' \texttt{ ==> } \varphi''\right) \in \overline{P}'\ .\ \left\{\begin{array}{r}\vdash (\overline{P} \cup \overline{P}''') \texttt{ ==> } \varphi''' \\ \text{where } \overline{P}''' = \overline{P}''[\overline{\texttt{x} \coloneqq E_{\texttt{x}}}]\ \overline{[\texttt{y} \coloneqq E_{\texttt{y}}]} \\ \varphi''' = \varphi''[\overline{\texttt{x} \coloneqq E_{\texttt{x}}}]\ \overline{[\texttt{y} \coloneqq E_{\texttt{y}}]}\end{array}\right\}}{\vdash \overline{P} \texttt{ ==> } \varphi}$$

$$(\text{GEN})\ \frac{\vdash \Phi[E \coloneqq x]_c \quad \text{fresh } \texttt{x} : \tau \qquad E : \tau \qquad E \in \mathit{gens}(\Phi)}{\vdash \Phi} \qquad (\text{CUT})\ \frac{\vdash \overline{P} \texttt{ ==> } E \texttt{ = } E' \qquad \vdash (\overline{P} \cup \{E \texttt{ = } E'\}) \texttt{ ==> } \varphi \qquad \langle E, \_\rangle \in \mathit{cases}(\overline{P} \texttt{ ==> } \varphi) \qquad \texttt{K} \in \mathit{cons}(\texttt{T}) \qquad (E', \_) = \mathit{inst}(\texttt{K})}{\vdash \overline{P} \texttt{ ==> } \varphi}$$

$$(\text{CASE})\ \frac{\mathit{foreach}\ \texttt{K} \in \mathit{cons}(\texttt{T})\ .\ \left\{\begin{array}{c}\vdash (\overline{P} \cup \{E \texttt{ = } E_{\texttt{K}}'\}) \texttt{ ==> } \varphi \\ \text{where} \\ (E_{\texttt{K}}, \_) = \mathit{inst}(\texttt{K}) \\ E_{\texttt{K}}' = \mathit{addHistory}(E_{\texttt{K}}, \{p\})\end{array}\right\} \qquad E : \texttt{T} \qquad \langle E, p\rangle \in \mathit{cases}(\overline{P} \texttt{ ==> } \varphi)}{\vdash \overline{P} \texttt{ ==> } \varphi}$$

$$(\text{IND})\ \frac{\mathit{foreach}\ \texttt{K} \in \mathit{cons}(\texttt{T})\ .\ \left\{\begin{array}{c}\vdash \overline{P}_h \cup \overline{P}[\texttt{x} \coloneqq E_{\texttt{K}}']_c \texttt{ ==> } \varphi[\texttt{x} \coloneqq E_{\texttt{K}}']_c \\ \text{where} \\ (E_{\texttt{K}}, \overline{\texttt{r}}) = \mathit{inst}(\texttt{K}) \quad E_{\texttt{K}}' = \mathit{addHistory}(E_{\texttt{K}}, p) \\ \overline{\texttt{y}} = (\mathit{FV}(\overline{P} \texttt{ ==> } \varphi) \backslash \{\texttt{x}\}) \quad \overline{\texttt{y}'} \text{ all fresh} \\ \overline{P}_h = \{\ \texttt{all } \overline{\texttt{y}'}.(\overline{P}' \texttt{ ==> } \varphi)[\overline{\texttt{y} \coloneqq \texttt{y}'}][\texttt{x} \coloneqq \texttt{r<}p\texttt{>}]_c \mid \texttt{r} \in \overline{\texttt{r}}\ \}\end{array}\right\} \qquad E : \texttt{T} \qquad \langle \texttt{x}, p\rangle \in \mathit{inds}(\overline{P} \texttt{ ==> } \varphi)}{\vdash \overline{P} \texttt{ ==> } \varphi}$$

Fig. 7. Parts of Zeno's proof for `ord (sort as)`. Proof steps are annotated by the name of the rule applied, and by a different Greek letter.



(GEN) and (CUT) both discover necessary sub-lemmas of our goal. Generalisation replaces an expression with a fresh variable of the same type - it corresponds to ∀-elimination. E.g., step [ϵ] "discovers" the sub-lemma `ord cs ==> ord (ins b cs)` by generalising `sort bs`. (CUT) is cumulative transitivity; it adds a new antecedent by proving it from the existing ones - this proof is our discovered sub-lemma.

The partial function *inst* in **Fig. 8** takes an expression of function type and applies fresh argument variables until the expression is simply typed - returning the new simply typed expression and the set of every recursively typed fresh variable applied, i.e. those variables whose type is the simple type of the returned expression. For example $inst(\texttt{(:)}) = (\texttt{b:bs}, \{\texttt{bs}\})$; this follows from $inst(\texttt{b:}) = (\texttt{b:bs}, \{\texttt{bs}\})$; which, in its turn, follows from $inst(\texttt{b:bs}) = (\texttt{b:bs}, \emptyset)$.

(CASE)-splitting proves a goal by choosing a simply typed expression (E : `T`) and proving a branch for each value this expression could take, viz. each constructor of its type ($cons(\texttt{T})$). The value this expression has been assigned down each branch is added as an antecedent. In step [κ] we case-split upon `b <= d` creating two branches - [μ] `(b <= d) = False`, and [λ] `(b <= d) = True`.

(IND) applies structural induction on a variable `x`, proving branch for every constructor of its type where an inductive hypothesis is added for every recursive variable in that constructor - $\overline{P}_h$ is the set of all these new hypotheses. Every free variable that is not `x` becomes ∀-quantified in our new hypotheses. In step [α] we apply induction on `as`, creating two branches - [β] `as = []` and [γ] `as = (b:bs)` which gains the hypothesis `ord (sort bs)`. In step [ζ] we apply induction on

Fig. 8. Instantiation function

$$rtype : Type \to TypeVar \qquad rtype(\texttt{T}) = \texttt{T} \qquad rtype(\tau_1 \texttt{ -> } \tau_2) = rtype(\tau_2)$$

$$inst : Expr \rightharpoonup Expr \times \wp(Var)$$

$$inst(E) = \begin{cases} (E, \emptyset) & \text{if } E : \texttt{T} \\ (E', \{\texttt{x}\} \cup \overline{\texttt{x}}) & \text{if } E : (rtype(\tau) \texttt{ -> } \tau) \\ & \text{where } \texttt{x} : rtype(\tau) \text{ is fresh, } (E', \overline{\texttt{x}}) = inst(E\ \texttt{x}) \\ (E', \overline{\texttt{x}}) & \text{if } E : (\tau_a \texttt{ -> } \tau_r),\ \tau_a \neq rtype(\tau_r) \\ & \text{where } \texttt{x} : \tau_a \text{ is fresh, } (E', \overline{\texttt{x}}) = inst(E\ \texttt{x}) \end{cases}$$

`cs`, creating branches $[\eta]$ and $[\kappa]$, where the latter gains a hypothesis in which `b` has been replaced by a fresh `b'` which is $\forall$-quantified.

Soundness. We believe, but have not yet proven, that the proof-steps presented in this section are sound, *i.e.,* that any property provable using these steps is provable in first order logic enhanced with structural induction, provided that induction or case splits are only applied on terms guaranteed to terminate. Unsoundness through non-terminating functions does not arise in our case, because, as we will see in **Section 4**, the calculation of critical terms for expressions containing such functions would not terminate. Thus, when faced with non-terminating functions, Zeno might loop for ever but will not produce erroneous proofs. We want to adopt termination checkers in further work. Moreover, created proofs are checked by Isabelle; this gives a strong guarantee of soundness.

4 Heuristics

In **Section 3** we discussed the proof rules without discussing the highlighted, further conditions. In this section we describe the most important heuristics which trim the search space, and in particular the further conditions.

4.1 Prioritize (EQL) and (CON), and Counterexamples

The steps (EQL) and (CON) are applied whenever possible, as they immediately close their proof branch.

When generating a new proof goal, before attempting to prove it, Zeno searches for counterexamples, and abandons the proof search if it finds any. Our approach is similar to SmallCheck[14], in that both use execution to generate values, but differs in that SmallCheck uses depth of recursion to restrict to a finite set, whereas we use our critical pair technique, described later on. In contrast, ACL2s generates a constant number of random values, much more like QuickCheck[5].

4.2 Applying (CUT) Only When (CASE) is also Possible

In principle, (CUT) is applicable at *any* point during proof search, and for *any* intermediate goal $E = E'$, which follows from the current antecedents, and which implies the current goal. This makes (CUT) highly non-deterministic.

Fig. 9. A (CUT) step

$$[\nu](\text{CUT})\ \dfrac{[\sigma]\ \dfrac{\ldots}{\vdash \ldots,\ \texttt{not(b<=d) ==> d<=b}} \qquad [\xi]\ \dfrac{\ldots}{\vdash \ldots,\ \texttt{not(b<=d), d<=b ==> ord [d, b]}}}{\vdash \ldots,\ \texttt{not(b<=d) ==> ord [d, b]}}$$

Our heuristic resticts the applicability of (CUT), and the search for an appropriate intermediate goal, by requiring that this step should be chosen only when a (CASE) would have been applicable too (i.e. when $E.. \in \mathit{cases}(...)$), and only when the intermediate goal $E = E'$ can be inferred from the current antecedents. Thus, (CUT) discovers necessary sub-lemmas.

In **Fig. 9** we use a (CUT) in the proof of `not (b <= d) ==> ord [d,b]`; the latter is a subproof for $[\mu]$ from **Fig. 7**. At $[\nu]$, a case analysis on `b<=d` is possible, however, since `b<=d` follows from the antecedents, our heuristic prefers a (CUT) instead. In the process, it proves the sublemma that `not (b <= d) ==> d <= b`.

4.3 Critical Terms

Critical pairs[1], defined in **Fig. 11**, are pairs of terms and paths, and are used to select between induction, case analysis, generalization and cut steps. We will first discuss the first component of these pairs, i.e. the *critical terms*, and then, in 4.4 we will refine the picture and introduce the *critical paths*. As our running example we use **Fig. 10**, which revisits the example from **Fig. 7**, this time annotated with critical paths.

For a motivation for critical terms, consider the bottom of **Fig. 10**, annotated with $[\alpha]$, where – ignoring the empty critical path `<>` – the aim is to prove that `ord (sort as)`. At this point, it would be possible to apply induction on `as`, or case analysis on `as`, or generalization on `sort as`. Nevertheless, Zeno only considers the (IND) step. It does this based on the *critical term* of the expression.

We focus in **Fig. 11** on the term E' such that $\mathit{pair}(\overline{P}, E) = \langle E', ... \rangle$. Then, E' is crucial for the evaluation of E, i.e. the evaluation of E can continue only if we have some more information about the value of E' than currently available in E' itself or in $\overline{P}$. For example, because `sort as` $\rightsquigarrow_*$ `case as of { ... }`, and `ord (sort as)` $\rightsquigarrow_*$ `case as of { ... }`, we have $\mathit{pair}(\emptyset, \texttt{sort as}) = \langle \texttt{as}, ... \rangle$ and $\mathit{pair}(\emptyset, \texttt{ord (sort as)}) = \langle \texttt{as}, ... \rangle$. Also, `ord (ins b (d:ds))` $\rightsquigarrow_*$ `case b <= d of { ... }`, and thus $\mathit{pair}(..., \texttt{ord (ins b (d:ds))}) = \langle \texttt{b <= d}, ... \rangle$. With *pairs* (**Fig. 11**) critical pairs are lifted to equations, clauses and properties, allowing for more than one critical pair per goal.

Now we focus again on the proof steps in **Fig. 6**, and consider the extra conditions in (IND), (CASE), (GEN), which restrict the applicability of these steps. For example, induction is applicable only on $\mathit{inds}(...)$. In **Fig. 12** we define functions *inds*, *cases* and *gens*. We will discuss paths in the next section, but we can see already that if a critical term is a variable, then Zeno uses it for

[1] Not to be confused with the *critical pairs* of term rewriting.

Fig. 10. First part of the example from **Fig. 7** revisited.
Here Φ_α, Φ_γ, Φ_ϵ, and Φ_ζ, stand for the proof goals at $[\alpha]$, $[\gamma]$, $[\epsilon]$, and $[\zeta]$. The goals are annotated with paths.
We use names for paths: $p_1 = \texttt{o1:s1:[]}$, $p_2 = \texttt{o1:i1:s1:[]}$, $p_3 = \texttt{o1:i1:[]}$, and $p_4 = \texttt{s1:[]}$.
Then we have $p_1 \sqsubseteq p_2$, and $p_1 \not\sqsubseteq p_3$, and $p_4 \sqsubseteq p_3$.
Also, $inds(\Phi_\alpha) = pairs(\Phi_\alpha) = \{\ \langle \texttt{as}, p_1 \rangle\ \}$, and $cases(\Phi_\alpha) = gens(\Phi_\alpha) = \emptyset$.
Also, $gens(\Phi_\epsilon) = \{\ \texttt{isort bs<}p_1\texttt{>}\ \}$, and $inds(\Phi_\epsilon) = cases(\Phi_\epsilon) = \emptyset$.
Also, $pairs(\Phi_\zeta) = \{\ \langle \texttt{bs<}p_1\texttt{>}, p_2 \rangle, \langle \texttt{bs<}p_1\texttt{>}, p_1 \rangle\ \}$, and $pairs(\Phi_\zeta) \cap MinPairs = \emptyset$.
Thus, $pairs(\Phi_\zeta) \cap MinPairs = \{\ \langle \texttt{cs<}p_1\texttt{>}, p_3 \rangle,\ \langle \texttt{cs<}p_1\texttt{>}, p_4 \rangle\ \}$.
Thus, $inds(\Phi_\zeta) = \{\ \langle \texttt{cs<}p_1\texttt{>}, p_3 \rangle\ \}$, and $cases(\Phi_\zeta) = gens(\Phi_\zeta) = \emptyset$.

$$[\kappa](\text{CASE})\ \frac{[\mu] \qquad [\lambda]}{\vdash \texttt{(all b' . ord ds<}p_1, p_3\texttt{> ==> ord (ins b' ds<}p_1, p_3\texttt{>)),}\ \texttt{ord (d<}p_1, p_3\texttt{>:ds<}p_1, p_3\texttt{>) ==> ord (ins b<}p_1\texttt{> (d<}p_1, p_3\texttt{>:ds<}p_1, p_3\texttt{>))}}$$

$$[\zeta](\text{IND})\ \frac{[\eta] \qquad [\kappa]}{\vdash \texttt{ord cs<}p_1\texttt{> ==>}}$$

$$[\epsilon](\text{GEN})\ \frac{\texttt{ord (ins b<}p_1\texttt{> cs<}p_1\texttt{>)} \quad (\Phi_\zeta)}{\vdash \texttt{ord (sort bs<}p_1\texttt{>) ==>}}$$

$$[\gamma](\text{EXP})\ \frac{\texttt{ord (ins b<}p_1\texttt{> (sort bs<}p_1\texttt{>))} \quad (\Phi_\epsilon)}{\vdash \texttt{ord (sort bs<}p_1\texttt{>) ==>}}$$

$$[\alpha](\text{IND})\ \frac{[\beta] \qquad \texttt{ord (sort (b<}p_1\texttt{>:bs<}p_1\texttt{>))} \quad (\Phi_\gamma)}{\vdash \texttt{ord (sort as<>)} \quad (\Phi_\alpha)}$$

Fig. 11. Defining critical pairs

$$Pair = NormalTerm \times Path$$

$$pair : \wp(Prop) \times Expr \rightharpoonup Pair$$

$$pair(\overline{P}, E) = \begin{cases} \langle E, [] \rangle & \text{if } E \in NormalTerm,\ E : \texttt{T},\ \nexists \texttt{K}. \exists \overline{E}. E = \texttt{K}\ \overline{E} \\ \langle E', [\texttt{i}] \rangle & \text{if } \overline{P} \vdash E \rightsquigarrow_* \texttt{case<i>}\ E'\ \texttt{of \{ ... \}},\ E' \notin E, \\ & \quad E' \in NormalTerm \\ \langle E'', \texttt{i} : p \rangle & \text{if } \overline{P} \vdash E \rightsquigarrow_* \texttt{case<i>}\ E'\ \texttt{of \{ ... \}},\ E' \in E \\ & \text{where } \langle E'', p \rangle = pair(\overline{P}, E') \end{cases}$$

$$\begin{array}{lll} exprs : Prop & \to \wp(Expr) & \\ exprs(E_1\ \texttt{=}\ E_2) & = \{\ E_1,\ E_2\ \} & \\ exprs(\overline{P}\ \texttt{==>}\ \varphi) & = \bigcup \overline{exprs(P)}\ \cup\ exprs(\varphi) & \\ exprs(\texttt{all}\ \overline{\texttt{x}}\ .\ \Phi) & = \{\ E \in exprs(\Phi) \mid \nexists \texttt{x} \in \overline{\texttt{x}}\ .\ \texttt{x} \in E\ \} & \end{array}$$

$$\begin{array}{lll} pairs : Cls & \to \wp(Pair) & \\ pairs(\overline{P}\ \texttt{==>}\ \varphi) & = \{\ pair(\overline{P}, E) \mid E \in exprs(\overline{P}\ \texttt{==>}\ \varphi)\ \} & \end{array}$$

induction, if it is not a sub-term of the current goal, then Zeno uses it for case analysis. This is why, in step $[\alpha]$ of **Fig. 10** the only applicable step is induction on `as`. Similarly, in step $[\kappa]$ of **Fig. 10** a case-split on `b <= d` is applicable.

4.4 Critical Paths

Although critical terms are essential in pruning the search space, they do not prevent repeated application of what is essentially the same step. Consider, e.g., step $[\gamma]$ in **Fig. 10**; here $pair(..., \texttt{ord (sort bs)}) = \langle \texttt{bs}, ... \rangle$. Naïve application of critical terms as we have considered them so far would be applying induction again, and in fact, would be applying induction for ever!

For this, we built into Zeno a way of remembering which cases in a function definition it has tried so far, and then avoiding covering the same cases when selecting the next step. We use the notion of *path*, which consists of a sequence of labels; the labels indicate cases in the definition of functions. For this, the full syntax definition in **Fig. 1** prescribes distinct labels for each case in a function definition, c.f. the labels `o1`, `o2`, `c1` etc., in **Fig. 3**. Furthermore, **Fig. 1** prescribes each variable in an expression to be decorated with a set of paths. We call these paths the *history* of an expression, as they record for each of these variables the reason why this this variable has been introduced. The variables in the function definitions and in the original property have an empty history (e.g. we start with `ord (sort as<>)` in $[\alpha]$ in **Fig. 10**). Then, as the proof progresses, new variables gain history.

We now read the full definition of critical pairs in **Fig. 11**. We call the second component of the critical pair the *intention* of the pair. It is a path, which describes the function cases that would be covered if that term were used in the next step of the proof. For example, $pair(\emptyset, \texttt{ord (sort as<>)}) = \langle \texttt{as}, p_1 \rangle$ where $p_1 = \texttt{o1:s1:[]}$, which means that selecting the variable `as` would progress the proof along the cases `o1` and `s1`. Also, $pair(..., \texttt{ord (sort bs<}p_1\texttt{>)}) = \langle \texttt{bs<}p_1\texttt{>}, p_2 \rangle$, where $p_2 = \texttt{o1:i1:s1:[]}$. Finally, $pair(..., \Phi_\gamma) = \{ \langle \texttt{bs<}p_1\texttt{>}, p_1 \rangle, \langle \texttt{bs<}p_1\texttt{>}, p_2 \rangle \}$.

When we use a critical pair in a (CASE) or (IND) step we store its intention in the corresponding variables in the new goal (see **Fig. 6**, *addHistory* is given in **Fig. 12**). For example, in step $[\gamma]$ in **Fig. 10**, we store the path p_1 in `bs`.

In **Fig. 12** we define the partial order $\sqsubseteq$ on paths, where $p \sqsubseteq p'$ if p' contains all cases from p extended at places with further cases. For example, `o1:s1:[]` $\sqsubseteq$ `o1:i1:s1:[]`. We also define $Min\text{imal}Pairs$, so as to remove any pairs where the history of the term "covers" (in the sense of $\sqsubseteq$) the intention of the pair. Thus, $MinPairs$ are critical pairs which do *not* represent a previously applied similar path. For example, the pair $\langle \texttt{bs<}p_1\texttt{>}, p_2 \rangle$ is *not* minimal.

For induction and case splits, we only take the minimal pairs, c.f. **Fig. 12**, and thus we avoid chosing paths which have been essentially covered in earlier proof steps. Therefore, at $[\gamma]$ in **Fig. 10** induction and case analysis are not applicable, but generalization is. On the other hand, the purpose of generalisation, when discovering intermediary lemmas, w.r.t. to critical paths is to shorten the critical path of the critical pair of an expression. Thus, in **Fig. 10** in $[\gamma]$ we apply generalisation and turn Φ_ϵ into Φ_ζ which has the critical pair $\langle \texttt{cs<}p_1\texttt{>}, p_3 \rangle$, in $MinPairs$ since $p_1 \not\sqsubseteq p_3$ - here generalisation has shortened p_2 to p_3, yielding a less complex step ($p_3 \sqsubseteq p_2$) and no longer covered by the previous p_1 step.

Fig. 12. Critical-pair-based heuristics: *inds*, *cases* and *gens*, and auxiliary definitions

$$p_1.p_2...p_n \sqsubseteq q_1.p_1.q_2.p_2...p_n.q_{n+1} \quad \text{where } p_i, q_i \in Path$$

$$\begin{array}{ll}
history : Expr & \to \wp(Path) \\
history(E) & = \bigcup\{\ \overline{p} \mid \texttt{x<}\overline{p}\texttt{>} \in FV(E)\ \} \\
\\
addHistory : Expr \times \wp(Path) & \to Expr \\
addHistory(E, \overline{p}) & = E\{\ [\texttt{x<}\overline{p}'\texttt{>} := \texttt{x<}\overline{p} \cup \overline{p}'\texttt{>}] \mid \texttt{x<}\overline{p}'\texttt{>} \in FV(E)\ \} \\
\\
MinPairs & \subseteq \wp(Pair) \\
MinPairs & = \{\ \langle E, p\rangle \in Pair \mid \nexists p' \in history(E)\ .\ p' \sqsubseteq p\ \} \\
\\
maxPaths : \wp(Pair) & \to \wp(Pair) \\
maxPaths(\overline{\pi}) & = \{\ \langle E, p\rangle \in \overline{\pi} \mid \nexists \langle E, p'\rangle \in \overline{\pi}\ .\ p \sqsubseteq p'\ \} \\
\\
inds : Cls & \to \wp(Pair) \\
inds(\Phi) & = maxPaths(\{\ \langle \texttt{x}, p\rangle \in (pairs(\Phi) \cap MinPairs)\ \}) \\
\\
cases : Cls & \to \wp(Pair) \\
cases(\Phi) & = maxPaths(\{\ \langle E, p\rangle \in (pairs(\Phi) \cap MinPairs) \\
 & \qquad\qquad \mid \nexists E_\Phi \in exprs(\Phi)\ .\ E \in E_\Phi\ \}) \\
\\
gens : Cls & \to \wp(Expr) \\
gens(\Phi) & = \{\ E \mid E_c \in E \in E_\Phi \in exprs(\Phi),\quad E : \texttt{T} \\
 & \qquad \langle E_c, p\rangle \in (pairs(\Phi) \setminus MinPairs), \\
 & \qquad \nexists E' : \texttt{T}'\ .\ E_c \in E' \in E \wedge E' \neq E\ \}
\end{array}$$

Fig. 13. Partial definition of substitution preserving critical paths

$$E =_c E' \text{ iff } [\![E]\!] = [\![E']\!] \text{ where } [\![]\!] \text{ removes all stored critical paths}$$

$$E[E' := E'']_c = addHistory(E'', history(E) \cup history(E')) \quad \text{if } E =_c E'$$

$$\vdots$$

$$(\texttt{\textbackslash x -> } E)[E' := E'']_c = \begin{cases} \texttt{\textbackslash x -> } E[E' := E'']_c & \text{if } \texttt{x} \notin E' \wedge \texttt{x} \notin E'' \\ \texttt{\textbackslash x -> } E & \text{otherwise} \end{cases}$$

$$\vdots$$

Notice, that although induction is not applicable on Φ_γ, it is applicable on Φ_ζ: Our critical pairs technique blocked induction until we had generalised to our intermediary lemma, and then re-enabled it.

Substitution and comparison in the presence of paths. In order to preserve the history of an expression after substitution - such as in (GEN)eralisation or (IND)uction - we have defined "capture avoiding substitution preserving critical paths" ($E[E' := E'']_c$) in **Fig. 13**. The first line is all we have changed from regular capture avoiding substitution; we have left out most of the definition since it is as you would expect and we give the rule for abstraction as an example. We define this, and some of our earlier rules in terms of "equality modulo critical paths" ($=_c$), which is syntactic equality ignoring critical paths stored in variables, as these are an annotation and do not affect execution.

5 Comparisons and the Output of Isabelle Proofs

We now compare Zeno, IsaPlanner[4,8,10], ACL2s[3,7], and Dafny's extension with induction [12], in terms of their respective performance on the 87 lemmas from a test-suite from the IsaPlanner website[2], which also appears in [10].

Of the 87 lemmas, 2 are false. Zeno can prove 82 lemmas, IsaPlanner can prove 47, while ACL2s can prove 74. All lemmas unprovable by Zeno are unprovable for the other tools too. ACL2s can prove 28 lemmas unprovable by Isaplanner, and Isaplanner can prove 1 lemma unprovable by ACL2s; the latter over a binary tree – something with a more natural representation in IsaPlanner. See below for percentages of proven, and lists of, unproven lemmas:

Tool	**Percent. Proven**	**Id's of unproven lemmas**
Dafny+indct.	53.5%	45 - 85
IsaPlanner	55%	48 - 85
ACL2s	86%	47, 50, 54, 56, 67, 72, 73, 74, 81, 83, 84, 85
Zeno	96%	72, 74, 85

Zeno's proofs take between 0.001s and 2.084s on an Intel Core i5-650 processor. ACL2s and Isaplanner produce proofs in similar times. The Haskell code to test Zeno and the LISP code to test ACL2s can be found at `tryzeno.org/comparison`. As functions in ACL2s are untyped, we supplied the type information ourselves through proven theorems. Without this information, ACL2s is unable to prove 6, 7, 8, 9, 15, 18, and 21.

To avoid ending up "training" Zeno towards the specific IsaPlanner test-suite, we developed a suite of 71 further lemmas. We tried to find lemmas to differentiate the tools, and show their respective strengths. Indeed, we found lemmas provable by ACL2s but not by Zeno, and lemmas provable by IsaPlanner but not by ACL2s, but none which IsaPlanner could prove and Zeno couldn't. Below we discuss five lemmas from our suite:

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

Nr.	Lemma	Nr.	Lemma
P1	`sort (sort xs) = sort xs`	P2	`x * (S 0) = x`
P3	`x * (y + z) = (x * y) + (x * z)`	P4	`x ^ (y + z) = (x ^ y) * (x ^ z)`
P5	`even(x), even(y) ==> even(x+y)`		

Zeno can prove `P1`, while IsaPlanner and ACL2s can not. Zeno takes around 2s to find the proof, has a proof tree 14 steps deep and discovers the sub-lemmas `ins x (ins y xs) = ins y (ins x xs)` and `sort (ins x xs) = ins x (sort xs)`. Both Zeno and IsaPlanner can prove `P2`, `P3`, `P4`, but ACL2s can not. Because Zeno does not support strong induction, it cannot prove `P5`, while ACL2s can.

Our (CUT) makes Zeno robust to multiple names of the same function. For example, when we defined `ord` so that it uses `leq` for comparison, while `ins` uses `<=`, where `leq` and `<=` are semantically equivalent, then ACL2s goes into an infinite loop, whereas Zeno applies several (CUT) steps, discovers the sub-lemmas `leq x y ==> x <= y`, and `x <= y ==> leq x y`, and `not (x <= y) ==> leq y x`, and `leq x y, leq y z ==> x <= z`, and finally finds the proof.

Isabelle Output Zeno translates its internal proof tree into an Isar[16] proof - every sub-goal of its proof becomes a new line and each step has a natural counterpart in Isabelle. The **HC** functions and data-types are easily converted into Isabelle's ML. Zeno's internal proof is purely backwards reasoning; we kept high-level structure of the proof output this way but we mixed in forwards reasoning for small internal sections of non-branching proof steps. At certain points we restart the output in a new sub-lemma, to keep the proof tree from becoming too deep and to display important sub-lemmas to the user. All lemmas from this section have had their proof output checked by Isabelle.

6 Conclusions and Future Work

We have described Zeno's proof steps and its heuristics; in particular, how critical pairs guide the selection of proof rules and avoid revisiting earlier proof steps. Zeno requires no further lemmas to be suggested to it, and indeed often discovers interesting auxiliary lemmas. We found that Zeno compared favourably with other tools both in terms of theorem proving power and speed.

On the other hand, Zeno cannot use auxiliary lemmas, while IsaPlanner and ACL2s can, and therefore are better suited for larger, human guided proofs. In particular, ACL2s has been used in the verification of properties of real-world systems. Integration of Zeno into a proof system like Isabelle, perhaps as a tactic for IsaPlanner, would be a useful next step – we also plan to adapt Zeno so that it can use background lemmas. Furthermore, ACL2s can prove properties over full first-order logic whereas Zeno lacks negation and existentials - another potential extension. Currently, Zeno does not check for function termination before attempting a proof, leaving us with a potential infinite loop in the critical pair discovery step; we plan to integrate techniques for termination checking.

The three properties from our test suite which Zeno is unable to prove require auxiliary lemmas which are not generalisations of sub-goals. Therefore, we

want to develop intelligent methods of finding such necessary lemmas, through techniques like IsaCoSy's[11] random perturbation of property terms.

Acknowledgements. We thank D. Ancona, SLURP research group, K. Broda, M. Johansson, R. Leino, and most particularly T. Allwood for useful feedback.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., Leino, K., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Boyer, R.S., Moore, J.S.: A theorem prover for a computational logic. In: CADE (1990)
4. Bundy, A., Stevens, A., Harmelen, F.V., Ireland, A., Smaill, A.: Rippling: A Heuristic for Guiding Inductive Proofs. Art. Intell. (62) (1993)
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: ICFP, pp. 268–279 (2000)
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: "The ACL2 Sedan". In: ICSE, pp. 59–60 (2007)
8. Dixon, L., Fleuriot, J.: IsaPlanner: A Prototype Proof Planner in Isabelle. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 279–283. Springer, Heidelberg (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: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 291–306. Springer, Heidelberg (2010)
11. Johansson, M., Dixon, L., Bundy, A.: Conjecture Synthesis for Inductive Theories. Journal of Automated Reasoning 47, 251–289 (2011)
12. Leino, K.R.M.: Automating Induction with an SMT Solver. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 315–331. Springer, Heidelberg (2012)
13. Paulson, L.C.: The foundation of a generic theorem prover. Journal of Automated Reasoning 5 (1989)
14. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and lazy Smallcheck: automatic exhaustive testing for small values. In: First ACM SIGPLAN Symposium on Haskell, pp. 37–48 (2008)
15. Walther, C., Schweitzer, S.: About VeriFun. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 322–327. Springer, Heidelberg (2003)
16. Wenzel, M.: Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–183. Springer, Heidelberg (1999)
17. Xu, D., Peyton-Jones, S., Claesen, K.: Static Contract Checking for Haskell. In: POPL (2009)