A Prototype Dedicated Theorem Prover for Clean

M.J. de Mol, M.C.J.D. van Eekelen

Computing Science Institute/

CSI-R9913 October

A Prototype Dedicated Theorem Prover for Clean

Maarten de Mol and Marko van Eekelen

Department of Computer Science
University of Nijmegen, the Netherlands
{maartenm, marko}@cs.kun.nl

Abstract. This paper examines an approach to computer assisted formal reasoning in relation to functional programming. Instead of using a generic proof tool which may differ on some points from the functional language used, a new proof tool is to be developed which is solely intended for proving properties of programs written in one specific language. This proof tool is intended to be inserted in the Integrated Development Environment of the programming language, which ensures a seamless integration. A prototype approximating such a proof tool for the pure, lazy functional programming language Clean has been implemented and will be described in this paper. It will be shown how this prototype can be used and examples of theorems that can be proven with it will be given. An examination will be made of the work that needs to be done to extend the prototype to an integrated programming tool.

1 Introduction

A method to guarantee the absence of errors in computer programs is *formal reasoning*. In this approach the source code of the program to reason about is needed as input, as well as a description of its desired behavior in the form of a theorem. Formal reasoning then is the construction of a formal proof for this theorem in some kind of formal logic system, using both standard reasoning steps from logic and reduction steps corresponding to the semantics of the program. Formal reasoning could be a useful tool for any programmer to ensure the correctness of written applications.

The problem with formal reasoning is that it is hard to do for human beings. To overcome this problem many sophisticated proof tools have been developed. Although it has often been stated that formal reasoning for programs written in pure functional programming languages (like Clean[1] or Haskell[2]) should be easier than for programs written in traditional programming languages (like C or Java), experiments conducted for Clean using the proof tools COQ[3] and PVS[4] have shown that it can still not easily be used in these cases.

Some of the experienced problems seemed to be caused by the genericity of the used proof tool. Programs had to be completely translated to a generic specification language, profound knowledge of the logical framework was needed to give the proof tool the right commands and many actions which had to be executed by the user could easily have been handled by the system. In general, the distance between the proof tool and the programming language added problems to the formal reasoning and prevented the giving of specialized support.

To fully exploit the possibilities for formal reasoning in relation to Clean, the problems in using a proof tool should be eliminated. This can be accomplished by aiding users in operating an existing generic theorem prover (see for instance [5] for this approach in Haskell), but in Clean another approach, which seems more flexible, will be employed: the development of a new proof tool which is completely dedicated to Clean alone. It should be possible in this tool to directly reason about Clean-programs, without having to translate to a separate specification language. It should be possible to easily (preferably even automatically) prove useful theorems about these programs, like for instance:

- $\forall_{x,y,z}[x\texttt{++}(y\texttt{++}z) = (x\texttt{++}y)\texttt{++}z]$
- $\forall_{f,x,y}[\texttt{Map}\ f\ (x\texttt{++}y) = (\texttt{Map}\ f\ x)\texttt{++}(\texttt{Map}\ f\ y)$
- $\forall_{x,y}[\texttt{Length}(x\texttt{++}y) = (\texttt{Length}\ x)\texttt{++}(\texttt{Length}\ y)]$

This idea will be carried out even a step further. The proof tool will not only be dedicated to Clean, it will actually be incorporated in its Integrated Development Environment(IDE). In this way there is no distance at all between the proof tool and the programming language. The aim is to provide the proof tool as a development tool for anyone who is programming in Clean.

Building a new proof tool is however a lot of work and a risky venture. In order to research whether it is possible at all to implement a dedicated proof tool for Clean (using Clean as development language), a prototype was built. This prototype has been restricted to make its implementation easy. For the final version these restrictions will be removed.

The most important restriction is that to eager term-rewriting. Equational rewriting is one of the most basic parts of a theorem prover for a functional programming language. It is used not only to model function reduction, but also to model the application of proven equalities. It is therefore necessary to have a lazy graph-rewrite system as the basis of a theorem prover for Clean. Other restrictions are the narrowing of the input to a subset of Clean and the acceptance of finite reductions only.

In this paper the prototype will be described. First the idea of providing a proof tool as a tool for the programmer will be presented. Then a description of the existing prototype will be given, followed by an examination of what needs to be done in order to add it to the IDE and to use it as a complete reasoning assistant for Clean. Finally a conclusion will be drawn on whether it is sensible to continue further developing this new proof tool.

2 A proof tool as a tool for the programmer

In the *Integrated Development Environment(IDE)* of Clean tools which can be used for the development of applications in Clean are combined. Already part of this IDE are for instance an editor, a profiler and of course the compiler. The components in the IDE are linked to make the combined usage of different tools easier; the compiler can for instance be invoked from the editor.

The idea is to insert a proof tool in the IDE of Clean. To allow for the easy use of the prover when developing Clean-programs, it must be linked to other components of the IDE. Two of these links will be established as follows:

1. *By allowing the prover to be started from the editor.*
   The source code of the program to reason about can be automatically transferred to the proof tool by the editor. Assisted by the tool the user can then construct a proof for a desired property of the program.
2. *By adding theorems to Clean-programs which are checked at compile-time.*
   By extending the syntax of Clean, theorems about a program can be inserted in the program itself. These theorems can then be automatically checked at compile-time without user assistance.

A programmer developing a program in the editor of the IDE can then start the theorem prover by pressing a single button. He will be presented with a window in which it is possible to specify a theorem about the program in development and to issue commands to prove this theorem. It is not necessary to specify the program to reason about, because this is taken care of automatically by the IDE. Beginning with formal reasoning is therefore very easy for programmers, but in order to make the proof tool really useful it should be possible to obtain results from the tool fast and easily as well. For this purpose the automated construction of proofs is very important, as well as a sophisticated user interface which guides users through the reasoning process.

A proof tool can be a really usable tool for any programmer, if it is tightly integrated in the IDE, provides a powerful automatic proof construction algorithm and has a sophisticated and easy-to-use user interface.

3 The first prototype

The first prototype[8] is a small-scale proof tool, which must be run as a stand-alone application. A screen dump of the prototype can be found in Fig. 1. It supports a subset of valid Clean-programs, for which theorems formulated in a stripped first-order predicate logic can be proven. These theorems basically state equalities between expressions which are related to the program one is reasoning about. The reasoning is done by applying predefined proving actions that rewrite the theorem to prove. Both reasoning rules known from logic and the semantics of the defined program are expressed in these proving actions. Once the theorem has been rewritten to `TRUE`, the proof is complete.

In this section the prototype, and how it can be used, will be described. First the logical framework implemented in the prototype is described. Next the way theorems and programs must be provided as input is explained. Then it is shown how proofs can be constructed using the prototype, both interactively and automatically. Finally some examples are given of proofs constructed with the prototype.

Prop	$= \rho$	**Expr**	$= x$
	$\mid True \mid False$		$\mid Fun[E_1 \ldots E_n]$
	$\mid E_1 = E_2$		$\mid DataCons[E_1 \ldots E_n]$
	$\mid \neg P$		$\mid E[E_1 \ldots E_n]$
	$\mid P_1 \wedge P_2$		
	$\mid P_1 \rightarrow P_2$	**Type**	$= \alpha$
	$\mid \forall_{x::T}.P$		$\mid T_1 \rightarrow T_2$
	$\mid \forall_{\alpha}.P$		$\mid TypeCons[T_1 \ldots T_n]$

Table 1. Syntax of logical language

3.1 The logical framework

The logical framework in the prototype provides a *logical language* for formalizing theorems and programs and a *derivation system* for constructing proofs.

In the logical language three kinds of terms are defined: *propositions*, *expressions* and *types*. For all kinds of terms variables may be used. These proposition-, expression- and type-variables are used for unification in the rewriting, representing arguments of functions, representing polymorphic types and quantification. The precise syntax of the logical language is described in Table 1.

Propositions are statements in a first-order predicate logic which is extended with an '=' on expressions. The basic cases are $True$, $False$ and $E_1 = E_2$, and complex propositions can be built using the operators $\neg$ (negation), $\wedge$ (conjunction), $\rightarrow$ (implication) and $\forall$ (quantification). Expressions consist entirely of applications of symbols and variables. Empty argument lists will be omitted and sometimes infix notation will be used. Sharing and lambda-abstraction have not been implemented in the prototype. Types are also simple and can only be constructed using type-constructors and the $\rightarrow$-operator.

The formalization of the definitions in the program to reason about is shown in Table 2. All symbols that are defined are assumed to be unique regardless of their definition, meaning that no two different functions or data-constructors can ever be equal. In the implementation this is of course accomplished by storing unique names. The translation of the actual program to its formal specification is straightforward and performed automatically by the prototype.

A proposition that one wants to prove is called a *goal*. Goals are always proven in a *goal context*, which is a set containing introduced hypotheses, expression-variables and type-variables. For each expression-variable also its type is stored.

Function	$= [T_1 \ldots T_n] \leadsto T \oplus [Pattern_1 \ldots Pattern_m]$
Pattern	$= [E_1 \ldots E_n] \leadsto E \mid [E_1 \ldots E_n] \leadsto P$
TypeCons	$= [\alpha_1 \ldots \alpha_n] \oplus [DataCons_1 \ldots DataCons_m]$
DataCons	$= [T_1 \ldots T_n] \leadsto T$

Table 2. Formalization of program

Name	Derivation rules	Conditions
Start	$\Gamma \vdash \mathit{True}$	
Split	$\dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P \quad \Gamma \vdash Q} \quad \dfrac{\Gamma, P \wedge Q \vdash R}{\Gamma, P, Q \vdash R}$	
Introduction	$\dfrac{\Gamma \vdash P \rightarrow Q}{\Gamma, P \vdash Q} \; \dfrac{\Gamma \vdash \forall_{x::T}.P}{\Gamma, x :: T \vdash P} \; \dfrac{\Gamma \vdash \forall \alpha.P}{\Gamma, \alpha \vdash P}$	
Cut	$\dfrac{\Gamma, P \vdash Q}{\Gamma, P \vdash P \rightarrow Q}$	
Generalize	$\dfrac{\Gamma \vdash P}{\Gamma \vdash \forall_{x::T}.P[E \hookrightarrow x]}$	$E :: T$ x is fresh
Rewrite(E)	$\dfrac{\Gamma \vdash P}{\Gamma \vdash P[E_1[x_i \mapsto E'_i]_i \hookrightarrow E_2[x_i \mapsto E'_i]_i]}$	$(\forall_{x_i::T_i})_i.E_1 = E_2 \in \Gamma \cup \Delta$
Rewrite(P)	$\dfrac{\Gamma \vdash P}{\Gamma \vdash P[P_1[\rho_i \mapsto P'_i]_i \hookrightarrow P_2[\rho_i \mapsto P'_i]_i]}$	$P_1 = P_2 \in \Delta$ $\mathit{PropVars}(P_1, P_2) = \{\rho_i\}_i$
Induction	$\dfrac{\Gamma \vdash \forall_{x::TC[T_1 \ldots T_p]}.P}{\Gamma \cup_{j \in R_i} \{IH_{ij}\} \cup_j \{\mathit{Type}_{ij}\} \vdash_i IS_i}$	$TC = [\alpha_k]_k \oplus [DC_1 \ldots DC_n]$ $DC_i = [T_{i1} \ldots T_{in_i}] \rightsquigarrow TC[\alpha_k]_k$ $R_i = \{j \mid T_{ij}$ is recursive$\}$ $IH_{ij} = P[x \mapsto x_{ij}]$ $\mathit{Type}_{ij} = x_{ij} :: T_{ij}[\alpha_k \mapsto T_k]_k$ $IS_i = P[x \mapsto DC_i[x_{i1} \ldots x_{in_i}]]$ all x_{ij} are fresh

Table 3. The derivation system

Next to the goal context belonging to a specific goal, there is also a *global context* (denoted by Δ) that can be used in all goals. This is a set containing global hypotheses, which are either an equality between propositions or an equality between expressions. These equalities can be regarded as rewrite-rules.

Now a derivation system for $\Gamma \vdash P$ is defined, where $\Gamma \vdash P$ should be interpreted as 'a proof of the goal P which uses the goal context Γ and the global context Δ exists'. The derivation rules that are available in this system are described in Table 3, where the rules should be interpreted as: if the statements on the bottom are true, then also the statement on the top is true.

Very important are the Rewrite()-rules, which will also be used for function-reduction. Note that a reduction strategy is not yet described. The notation $P[x \hookrightarrow y]$ is used to denote any P' that can be obtained by replacing occurrences of x in P by y. If *all* occurrences must be rewritten, the '$\hookrightarrow$' is replaced by '$\mapsto$'.

Using this derivation system statements of the form $\Gamma \vdash P$ can be obtained. When Γ is empty the desired output of the proof tool is reached: a formal confirmation of the correctness of the theorem P. The list of derivation rules that were applied to obtain $\vdash P$ can be regarded as a proof of the theorem.

3.2 Specification of the program

Two kinds of definitions from Clean-programs can be specified in the prototype: type-definitions and function-definitions. Macro-definitions and class-definitions are not supported. For each definition its formalization is automatically constructed.

Each type-definition must be an algebraic type-definition. The standard types of Clean are not supported, unless explicitly specified as algebraic type in the standard library of the prototype. This has been done for `Lists` and `Booleans`, and also a type for positive natural numbers called `Peano` is defined. The syntax for algebraic type-definitions is the same as in Clean, with the exception that a dot must be written to conclude a definition.

Syntactical Specification	`:: Bool   = True | False.`
	`:: List a =  Nil | Cons a (List a).`
	`:: Peano  = Zero | Succ Peano.`	

Formal Specification	$Bool = [] \oplus [True, False]$	$True = [] \rightsquigarrow Bool$ $False = [] \rightsquigarrow Bool$
	$List = [\alpha] \oplus [Nil, Cons]$	$Nil = [] \rightsquigarrow List[\alpha]$ $Cons = [\alpha, List[\alpha]] \rightsquigarrow List[\alpha]$
	$Peano = [] \oplus [Zero, Succ]$	$Zero = [] \rightsquigarrow Peano$ $Succ = [Peano] \rightsquigarrow Peano$

Example of type-definitions

Functions are defined by pattern-matching. For each function a type and one or more patterns must be given. Patterns and expressions as described in the logical framework are permitted, with the extra condition that on the left-hand-side of a pattern only applications of data-constructors may occur. Not supported in the prototype are list-comprehensions, dot-dot-expressions and local definitions.

Syntactical Specification	`Map :: (a -> b) (List a) -> (List b)`
	`Map f []     = []`
	`Map f [x:xs] = [f x:Map f xs]`
	
	`Sum :: (List Peano) -> Peano`
	`Sum []     = 0`
	`Sum [x:xs] = (x + (Sum xs))`

Formal Specification	$Map = [\alpha_1 \rightarrow \alpha_2, List[\alpha_1]] \rightsquigarrow List[\alpha_2] \oplus [Map_1, Map_2]$ $Map_1 = [x_1, Nil] \rightsquigarrow Nil$ $Map_2 = [x_1, Cons[x_2, x_3]] \rightsquigarrow Cons[x_1[x_2], Map[x_1, x_3]]$
	$Sum = [List[Peano]] \rightsquigarrow Peano \oplus [Sum_1, Sum_2]$ $Sum_1 = [Nil] \rightsquigarrow Zero$ $Sum_2 = [Cons[x_1, x_2]] \rightsquigarrow x_1 + Sum[x_2]$

Example of function-definitions

Again the syntax resembles Clean. The notations `[]`, `[x:y]` and `0` are internal aliases for `Nil`, `Cons x y` and `Zero`. If explicitly specified, functions may also be used as infix operators which has been done for `+` in the example. It may sometimes be necessary to write down more brackets in the prototype than is required in Clean.

3.3 Specification of the theorem

The theorem that one wants to prove must simply be expressed as a proposition. The logical operators $\neg$, $\wedge$, $\rightarrow$ and $\forall$ are respectively denoted by `NOT`, `AND`, `->` and `[ :: ]`.

Syntactical Specification	`[a::SET] [x::a] [y::a]`
	`    (x == y) = True -> (y == x) = True`

Formal Specification	$\forall\alpha.\forall_{x_1,x_2::\alpha}.(x_1 == x_2) = True \rightarrow (x_2 == x_1) = True$

Example of theorem-definitions

The symbol `SET` is used to introduce a quantification over types. In the prototype there is a clear distinction between types (which all have the artificial type `SET`), expressions (which all have a normal type) and propositions (which all have the artificial type `Prop`).

The theorem above could be easier specified with the help of predicates. If the predicate `Symmetric` would have been defined somehow, the theorem could be expressed as `Symmetric ==`. For this reason functions which have a proposition as result may be used:

Specification	`Symmetric :: (a -> (a -> Bool)) -> Prop`
	`Symmetric f :=`
	`    [x::a] [y::a] f x y = True -> f y x = True`

Formalization	$Symmetric = [\alpha \rightarrow (\alpha \rightarrow Bool)] \rightsquigarrow Prop \oplus [Sym_1]$ $Sym_1 = [x_1] \rightsquigarrow \forall_{x_2,x_3::\alpha}.x_1[x_2,x_3] = True \rightarrow x_1[x_3,x_2] = True$

Example of a predicate-definition

In fact, these functions have an expression with type *Prop* as result. To make a proposition out of such an expression the prototype automatically extends it with '$= True$'. Additionally, this suffix will not be shown in the output.

3.4 Constructing a proof

Proofs are constructed by repeatedly transforming *proof states* by the use of *tactics*. In the proof state the goals with belonging contexts are stored, as well as the history of the proof constructed so far. Initially there is only one goal, the initial theorem to prove, with an empty context and an empty history. The available global context Δ contains the following rules:

1. *The currying rules.* For all $n, m \geq 0$:
   - $(x_{n+m+1}[x_1 \ldots x_n])[x_{n+1} \ldots x_{n+m}] \;=\; x_{n+m+1}[x_1 \ldots x_{n+m}]$
2. *The discriminating rules.* For all $DataCons_1 \neq DataCons_2$ and $n, m \geq 0$:
   - $(DataCons_1[x_1 \ldots x_n] = DataCons_2[x'_1 \ldots x'_m]) \;=\; False$
3. *The pattern rules.*
   (a) For each $Pattern = [E_1 \ldots E_n] \rightsquigarrow E$ belonging to Fun:
      - $Fun[E_1 \ldots E_n] \;=\; E$

   (b) For each $Pattern = [E_1 \ldots E_n] \rightsquigarrow P$ belonging to $Pred$:
      - $(Pred[E_1 \ldots E_n] = True) \;=\; P$
4. *The semantics rules.*
   These rules describe how the logical operators behave with $True$- or $False$-arguments and with similarity between arguments, for instance:
   - $True \wedge \rho \;=\; \rho$
   - $\rho \wedge \rho \;=\; \rho$
5. *The injection rules.* For all $DataCons$ and $n \geq 0$:
   - $(DataCons[x_1 \ldots x_n] = DataCons[x'_1 \ldots x'_n]) \;=\; True \wedge_i x_i = x'_i$
6. *The lift rules.*
   These rules describe the relation between the standard functions `&&` and `==` in Clean and the logical operators in the prototype, for instance:
   - $((x_1 \;\&\&\; x_2) = True) \;=\; (x_1 = True \wedge x_2 = True)$
7. *The goal rules.* For each proven theorem $(\forall \alpha_i)_i.(\forall_{x_j::T_j})_j.E_1 = E_2$:
   - $E_1 \;=\; E_2$

   To prevent some infinite reduction paths, no rules are created when there exist suitable E'_j such that $E_1[x_j \hookrightarrow E'_j]$ occurs in E_2.
8. *The lemma rules.* The user can define own equalities in the global context.
9. *The function-arguments rules.* For each Fun and $n \geq 0$:
   - $(Fun[x_1 \ldots x_n] = Fun[x'_1 \ldots x'_n]) \;=\; True \wedge_i x_i = x'_i$

   This rule is only valid from left to right.

Instead of applying derivation rules, proofs are constructed by applying tactics. These are predefined combinations of derivation rules, providing an interface to the logical framework. All theorems that can be proven using the derivation rules can also be proven using the tactics, although the found proofs may differ.

It is not possible to specify arguments to tactics, but some tactics (called *multi-tactics*) do generate a list of possible proof states of which one must be chosen. All proofs have to be constructed using the predefined set of tactics made available by the prototype. The following tactics can be used:

1. *Curry.* Rewrites in the current goal using the currying rules from the global context. As many redices as possible are rewritten. This tactic has an efficient implementation which does not make explicit use of the global context.
2. *Move On.* Applies the start rule.
3. *Split.* Applies the split rule.
4. *Unequal-Constructors.* Efficiently rewrites the current goal using the discriminating rules. Only the goal as a whole is considered as redex.
5. *Induction.* Applies the induction rule.
6. *Introduction.* Applies the introduction rule.

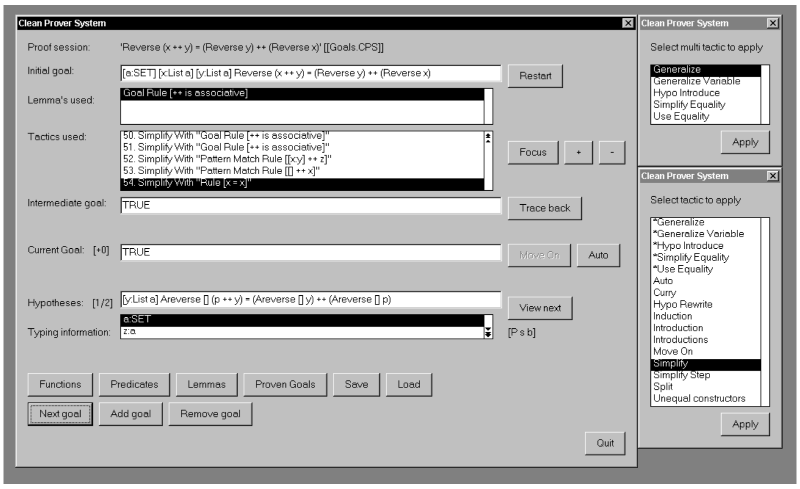
7. *Simplify-Step.* Rewrites in the current goal using the pattern, semantics, injection, lift, goal and lemma rules of the global context. Only one redex is rewritten: the leftmost-outermost redex of the first rule that can be applied.
8. *Hypo-Step.* This tactic creates temporary rules in the global context and then rewrites in the current goal using these rules. Again only the leftmost-outermost redex of the first applicable rule is rewritten. For each hypothesis of the form $(\forall \alpha_i)_i.(\forall x_i :: T_i).P \to Q$ the rule $P \to Q \;\; = \;\; True$ is created.
9. *Simplify-Equality (multi-tactic).* Rewrites in the current goal using the function-arguments rules of the global context. This tactic has a separate implementation, which only rewrites the leftmost-outermost redex (checking all rules). It is implemented as a multi-tactic which generates one option.
10. *Hypo-Introduce (multi-tactic).* Applies the cut rule once or twice in the current goal, trying all hypotheses and all combinations of hypotheses.
11. *Generalize (multi-tactic).* Applies the generalize rule once or twice in the current goal, trying all suitable subexpressions and all combinations of suitable subexpressions. A subexpression is suitable if it is not a variable, not an application of a data-constructor and either appears more than once or is the only subexpression in which a certain variable occurs.
12. *Generalize-Variable (multi-tactic).* Applies the generalize rule using a free variable as subexpression. The tactic can only be applied when the goal does not start with a quantification and generates options for each variable.
13. *Use-Equality (multi-tactic).* Rewrites once or twice in the current goal using the hypotheses in the goal context. For each hypothesis stating an equality its counterpart (switching $E_1 = E_2$ to $E_2 = E_1$) is temporarily added to the goal context. Again a rewrite can only take place on the leftmost-outermost redex. Rewrites according to all hypotheses and according to all combinations of hypotheses are tried.

A proof is always constructed backwardly as a combination of these basic tactics. To make building proofs easier, additionally composed tactics have been defined. Applications of composed tactics are translated to applications of basic tactics and can be used to reduce the number of user-applications needed to finish a proof. The most important composed tactics are *Introductions* (repeated Introduction), *Simplify* (repeated Curry, Split, Unequal-Constructors and Simplify-Step) and *Auto* (automatic proof construction).

Because Simplify and Auto do repeated Simplify-Step, the rewrite-system as implemented in the prototype is eager. To fully match the semantics of Clean a lazy graph rewrite-system is needed. This has not been implemented, making proofs constructed in the prototype not valid for programs that make use of infinite data-structures. The implemented rewrite-system is however equivalent to Clean when only finite structures are used. Note that to reason about infinite structures also another induction scheme is required.

3.5 Automatic proof construction

The automatic construction of proofs is modeled in the prototype by the composed tactic *Auto*. Ideally, one would like an application of Auto to generate and



```
Proof session:          'Reverse (x ++ y) = (Reverse y) ++ (Reverse x)' [[Goals.CPS]]
Initial goal:           [a:SET] [x:List a] [y:List a] Reverse (x ++ y) = (Reverse y) ++ (Reverse x)
Lemma's used:           Goal Rule [++ is associative]
Tactics used:           50. Simplify With "Goal Rule [++ is associative]"
                        51. Simplify With "Goal Rule [++ is associative]"
                        52. Simplify With "Pattern Match Rule [[x:y] ++ z]"
                        53. Simplify With "Pattern Match Rule [[] ++ x]"
                        54. Simplify With "Rule [x = x]"
Intermediate goal:      TRUE
Current Goal:   [+0]    TRUE
Hypotheses:     [1/2]   [y:List a] Areverse [] (p ++ y) = (Areverse [] y) ++ (Areverse [] p)
Typing information:     a:SET
                        z:a                                                      [P s b]
```

```
Select multi tactic to apply
Generalize
Generalize Variable
Hypo Introduce
Simplify Equality
Use Equality
```

```
Select tactic to apply
*Generalize
*Generalize Variable
*Hypo Introduce
*Simplify Equality
*Use Equality
Auto
Curry
Hypo Rewrite
Induction
Introduction
Introductions
Move On
Simplify
Simplify Step
Split
Unequal constructors
```

Fig. 1. Screen dump of prototype

test all possible combinations of basic tactics, ensuring that as many theorems can be proven automatically in the prototype as interactively. Unfortunately there are too many of these combinations, making it necessary to restrict the search space in order to obtain an algorithm which runs in reasonable time. This does mean that there will be proofs which can be constructed interactively but can not be found automatically.

Generating and testing combinations of basic tactics is a typical backtrack-problem. The number of considered combinations can be reduced by deciding not to backtrack at all on certain points, that is, deciding not to try other tactics when the successful application of a certain tactic does not lead to a finished proof. For this purpose all basic tactics are assigned to either the *BackYes-class* or the *BackNo-class*. On each point in the algorithm where a tactic from the BackNo-class is tried, backtracking is disabled completely.

The precise algorithm for automatic proof construction can be found in Fig. 2. Tactics from the BackNo-class are always tried first. If one of them can be applied, instead of building in the possibility to backtrack, an unconditional jump back to the start of the algorithm is made. The tactics from the BackNo-class are tried in a fixed order. This order is important, since tactics in the front of the order can completely block out tactics in the rear. This is for instance the case when both Introduction and Induction are assigned to the BackNo-class.

When none of the tactics of the BackNo-class can be applied, the tactics from the BackYes-class are tried. On this class also an order is needed, but due to the backtracking this order only affects speed and not the number of combinations considered. Following the successful application of a basic tactic from the BackYes-class, a recursive call to the Auto-tactic is made. Backtracking is then performed if this recursive call fails to complete the proof. However, recursively trying the Auto-tactic turns out to be too time-consuming. Therefore

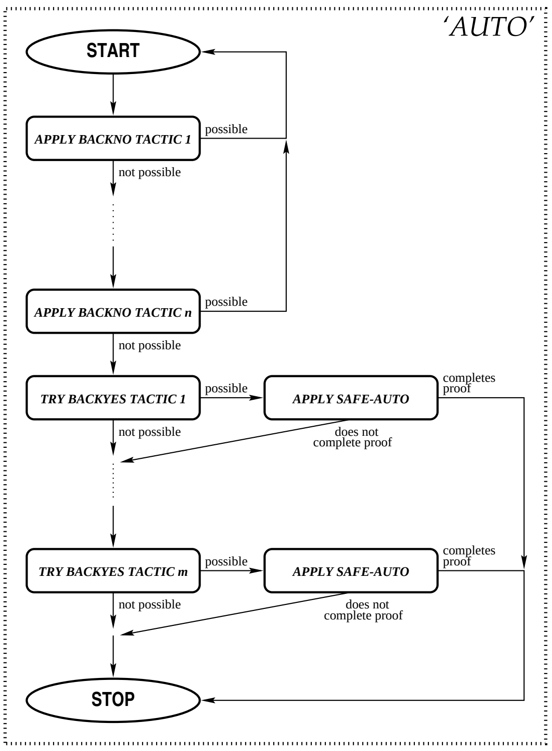


Fig. 2. Algorithm for automatic proof construction

a stripped version of the Auto-tactic, called the *Safe-Auto*-tactic is used. In this safe version the tactics Hypo-Introduce and Use-Equality have been removed, restricting the number of applications of these tactics in a proof to only one.

It is thus a very important decision whether to put a basic tactic in the BackYes- or the BackNo-class. Putting it in the BackNo-class ensures no considerable increase in search time will be experienced, but it may block out applications of other tactics which are needed to finish a proof as well. Putting it in the BackYes-class ensures that no other tactics will be blocked out, but may also lead to a considerable increase in search time.

In the prototype it was decided to only put tactics in the BackYes-class when that is absolutely necessary. This is the case for tactics which generate more than one alternative and for tactics which logically strengthen the goal. These are implemented as multi-tactics and assigned to the BackYes-class and all ordinary tactics are assigned to the BackNo-class. The prototype is thus optimized for efficiency rather than for expressivity.

Another important decision to make is the order of the basic tactics in the BackNo-class. The following order is used in the prototype: (1) Curry, (2) Simplify-Step, (3) Unequal-Constructors, (4) Split, (5) Hypo-Step, (6) Induction, (7) Introduction, (8) Move-On. Thus in the automated proof-construction simplification is performed instantly after each application of another tactic, and each quantification of an expression-variable in a type constructed with a type-constructor is tackled with induction.

3.6 Examples

In the following some examples will be given to demonstrate both the interactive and automated usage of the prototype. The first example is an easy theorem for which an interactive proof will be constructed by applying basic tactics only. The following definitions from the standard library of the prototype are required:

```
&& :: INFIX Bool Bool -> Bool
&& True  x = x
&& False x = False

Listeq :: (List a) (List a) -> Bool
Listeq []     []     = True
Listeq [x:xs] []     = False
Listeq []     [y:ys] = False
Listeq [x:xs] [y:ys] = (x == y) && (Listeq xs ys)

Reflexive :: (a -> (a -> Bool)) -> Prop
Reflexive f := [x::a] (f x x) = True
```

Goal :

$\vdash \mathit{Reflexive}[\mathit{Listeq}]$

Aliases in the proof :

$\Gamma_1 := \alpha, x_2 :: \alpha, x_3 :: \mathit{List}[\alpha]$

$\Gamma_2 := \Gamma_1, \mathit{Listeq}[x_3, x_3] = \mathit{True}$

$\mathit{IS} := \mathit{Listeq}[x_3, x_3] = \mathit{True} \rightarrow \mathit{Listeq}[\mathit{Cons}[x_2, x_3], \mathit{Cons}[x_2, x_3]] = \mathit{True}$

Proof :

$\{\textit{Simplify-Step}\} \quad \vdash \forall\alpha.\forall_{x_1::\mathit{List}[\alpha]}.(\mathit{Listeq})[x_1, x_1] = \mathit{True}$

$\{\textit{Curry}\} \quad \vdash \forall\alpha.\forall_{x_1::\mathit{List}[\alpha]}.\mathit{Listeq}[x_1, x_1] = \mathit{True}$

$\{\textit{Introduction}\} \quad \alpha \vdash \forall_{x_1::\mathit{List}[\alpha]}.\mathit{Listeq}[x_1, x_1] = \mathit{True}$

$\{\textit{Induction}\} \quad \alpha \vdash \mathit{Listeq}[\mathit{Nil}, \mathit{Nil}] = \mathit{True}, \mathit{IS}$

$\{\textit{Simplify-Step}\} \quad \alpha \vdash \mathit{True} = \mathit{True}, \mathit{IS}$

$\{\textit{Simplify-Step}\} \quad \alpha \vdash \mathit{True}, \mathit{IS}$

$\{\textit{Move-On}\} \quad \Gamma_1 \vdash \mathit{Listeq}[x_3, x_3] = \mathit{True} \rightarrow$
$\quad \mathit{Listeq}[\mathit{Cons}[x_2, x_3], \mathit{Cons}[x_2, x_3]] = \mathit{True}$

$\{\textit{Introduction}\} \quad \Gamma_2 \vdash \mathit{Listeq}[\mathit{Cons}[x_2, x_3], \mathit{Cons}[x_2, x_3]] = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash ((x_2 == x_2)\ \&\&\ \mathit{Listeq}[x_3, x_3]) = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash (x_2 == x_2) = \mathit{True} \land \mathit{Listeq}[x_3, x_3] = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash x_2 = x_2 \land \mathit{Listeq}[x_3, x_3] = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash \mathit{True} \land \mathit{Listeq}[x_3, x_3] = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash \mathit{Listeq}[x_3, x_3] = \mathit{True}$

$\{\textit{Use-Equality}\} \quad \Gamma_2 \vdash \mathit{True} = \mathit{True}$

$\{\textit{Simplify-Step}\} \quad \Gamma_2 \vdash \mathit{True}$ $\square$

The second example is a bit trickier. Again an interactive proof will be constructed, but now use will be made of the Simplify-tactic to make the proof shorter. The definitions of `Map` and `Sum` which were given earlier and the following definition of `+` from the standard library are required:

```
+ :: INFIX Peano Peano -> Peano
+ 0        y = y
+ (Succ x) y = Succ (x + y)
```

The goal is now to prove that it makes no difference to (1) take the sum of a list of numbers and then add the length of the list to it, or (2) map the `Succ` on the list and then take the sum.

Goal :

$\vdash \forall_{x_1::List[Peano]}.Sum[x_1] + Length[x_1] = Sum[Map[Succ, x_1]]$

Aliases in the proof :

$\Gamma_1 := x_2 :: Peano, x_3 :: List[Peano]$

$\Gamma_2 := \Gamma_1, Sum[x_3] + Length[x_3] = Sum[Map[Succ, x_3]]$

$IS := Sum[x_3] + Length[x_3] = Sum[Map[Succ, x_3]] \rightarrow$
$\quad Sum[Cons[x_2, x_3] + Length[Cons[x_2, x_3]] =$
$\quad Sum[Map[Succ, Cons[x_2, x_3]]$

Proof :

$$\begin{array}{rl}
\{Induction\} & \vdash Sum[Nil] + Length[Nil] = \\
& \quad Sum[Map[Succ, Nil]], IS \\
\{Simplify\} & \vdash True, IS \\
\{Move\text{-}On\} & \Gamma_1 \vdash Sum[x_3] + Length[x_3] = Sum[Map[Succ, x_3]] \rightarrow \\
& \quad Sum[Cons[x_2, x_3] + Length[Cons[x_2, x_3]] = \\
& \quad Sum[Map[Succ, Cons[x_2, x_3]] \\
\{Introduction\} & \Gamma_2 \vdash Sum[Cons[x_2, x_3]] + Length[Cons[x_2, x_3]] = \\
& \quad Sum[Map[Succ, Cons[x_2, x_3]] \\
\{Simplify\} & \Gamma_2 \vdash (x_2 + Sum[x_3]) + Length[x_3] = \\
& \quad x_2 + Sum[Map[Succ, x_3]] \\
\{Use\ Equality\} & \Gamma_2 \vdash (x_2 + Sum[x_3]) + Length[x_3] = \\
& \quad x_2 + (Sum[x_3] + Length[x_3]) \\
\{Simplify\} & \Gamma_2 \vdash True
\end{array}$$ □

In the last two simplification steps it is assumed that the associativity of $+$ and the theorem $\forall_{x_1,x_2::Peano}.x_1 + Succ[x_2] = Succ[x_1 + x_2]$ have already been proven. This is the case since these auxiliary theorems are defined in the standard library of the prototype as well.

For these two given examples proofs can also be generated automatically. Other theorems which can also be proven automatically are for instance:

1. $Symmetric[Listeq]$
2. $Transitive[Listeq]$
3. $Associative[++]$
4. $\forall\alpha.\forall_{x_1,x_2::List[\alpha]}.Length[x_1{+}{+}x_2] = Length[x_1] + Length[x_2]$
5. $\forall\alpha.\forall_{x_1,x_2::List[\alpha]}.Reverse[x_1{+}{+}x_2] = Reverse[x_2]{+}{+}Reverse[x_1]$

6. $\forall \alpha . \forall_{x::List[\alpha]} . Length[Reverse[x]] = Length[x]$
7. $\forall \alpha . \forall_{x::List[\alpha]} . Reverse[Reverse[x]] = x$

For the last two it is needed to use the proof of the fifth theorem as a lemma. There are also some theorems which can be proven interactively but not automatically, like the following one:

$$\forall_{x_1,x_2::Peano} . Sum[x_2] + (x_1 * Length[x_2]) = Sum[Map[(+x_1), x_2]]$$

A lot of interesting theorems can be proven using the prototype, and a large portion of these even automatically.

4 Upgrading the prototype: further work

A lot of work needs to be done to connect the prototype fully to Clean:

1. *Matching Clean-semantics by using a lazy graph rewrite-system.*
   At present function-definitions are rewritten using an eager term rewrite-system. In order to reason about all Clean-programs, it is necessary to implement lazy evaluation. Such a rewrite-algorithm, which makes use of the strictness information, is already available in the new IDE (Clean 2.0, written in Clean itself). Rewriting in other situations must however still be done eagerly.
2. *Extending the accepted programs to full Clean.*
   The prototype supports a subset of Clean-programs, but for integration in the IDE it is needed to accept all. For this purpose the logical language must be extended, as well as the derivation system. By reusing the parser of the IDE it can be ensured that all Clean-programs are syntactically accepted. Things that are not presently supported include:
   (a) *Class-definitions and priorities of infix operators.* These can be transformed to function calls by using algorithms available in the compiler.
   (b) *Standard types of Clean: Integers, Reals, Records, Arrays, Tuples.* New derivation rules have to be added to reason about these types.
   (c) *Uniqueness and strictness information.* Uniqueness information can be discarded, since it does not affect the semantics of the program. Strictness information is needed for the rewriting.
   (d) *Local definitions.* These can be lifted to global definitions using the algorithm available in the compiler.
   (e) *Cyclic definitions, sharing and lambda-abstraction.* Supporting these requires extending the logical language with graphs and abstraction.
   (f) *Dot-dot expressions and list-comprehensions.* These can be translated to calls of functions in the standard library.
3. *Extending the accepted logic.*
   The logical operators $\vee$ and (optionally) $\exists$ have to be added. Additional derivation rules have to be constructed for these operators.

4. *Extending the set of tactics.*
   The changes in derivation rules require changing the set of tactics as well. Tactics have to be constructed to apply the new derivation rules. Also the expressivity of the available set of tactics has to be assessed (for instance by comparing to other proof tools) and upgraded accordingly.
5. *Preventing infinite reductions by using weight-based rewrite-rules.*
   Using certain lemma's or proven goals can lead to infinite reduction paths, for instance in the case of $\forall_{x_1,x_2::Peano}.x_1 + x_2 = x_2 + x_1$. By defining a suitable weight measure on graphs, e.g. a kind of lexicographical ordering, it is possible to use these rules without risking non-termination.
6. *Designing a suitable user-interface.*
   No work at all has been done to design a good GUI. This is however very important to allow for usage by programmers.
7. *Interfacing with parser, editor, compiler.*
   The prototype has to be linked to other components of the IDE and accept output given by these components, but also algorithms already available in the IDE can be reused. For this purpose it is necessary to internally use an equivalent representation of programs and expressions.

These changes require a lot of work. Especially the change of internal representation affects all parts of the prototype and in fact requires a new implementation. A lot of algorithms already implemented in other components of the IDE can however be reused, making the enterprise as a whole more feasible.

5 Conclusions and related work

The prototype is a fully operational proof tool with which it is possible to prove many theorems about programs written in Clean automatically. The results from the initial usage of the prototype are very encouraging. It is easy to use and the proven theorems can really be useful to programmers.

But in order to obtain a proof tool which can be provided as part of the Clean-package and can be used by any programmer, a lot of work still needs to be done. Most importantly all of Clean should be supported instead of a subset, and the logical framework and offered tactics have to be extended to match the semantics of Clean. Shared and cyclic graphs and lazy evaluation have to be implemented to support reasoning about infinite data-structures.

The end-result seems very promising, offering formal reasoning as an easy-to-use and fully integrated programming development tool. The work that needs to be done is for a large part very technical, and it was already shown that with seemingly little effort a powerful proof tool can be developed. Therefore the development of an integrated proof tool for Clean will continue and we hope to report on more results soon.

Related work is described in [6], in which a description is given of a proof tool which is dedicated to Haskell. It supports a subset of Haskell and needs no guidance of users in the proving process. The user can however not manipulate a proof state by the use of tactics to help the prover in constructing a proof, and induction is only applied when the corresponding quantifier has been explicitly marked in advance. No plans were made to insert the tool in an IDE.

Further related work concerns a theorem prover for Haskell, called the Haskell Equational Reasoning Assistant[7], which is still under development. This proof tool is also dedicated to Haskell and supports Haskell 1.4. Proofs can only be constructed using equational reasoning and case analysis. No other proof methods, like induction or generalization, are supported. ERA is a stand-alone application.

Availability

The source code of the prototype and a precompiled version for Windows 95 can be obtained from:

`http://www.cs.kun.nl/~maartenm`

Suggestions and remarks can be directed to `maartenm@cs.kun.nl`.

References

1. R. Plasmeijer and M. van Eekelen. *Concurrent Clean Language Report (version 1.3)*, Nijmegen, 1998. `http://www.cs.kun.nl/~clean/Clean.Cleanbook.html`
2. J. Peterson(editor), K. Hammond(editor) et al. *Report on the programming language Haskell (version 1.4)*, Yale, 1997. `http://www.haskell.org/definition/`
3. B. Barras et al. *The Coq Proof Assistant Reference Manual (version 6.2)*, Inria, 1998. `http://pauillac.inria.fr/coq/doc/config-eng.html`
4. S. Owre, N.Shankar, J.M. Rushby and D.W.J. Stringer-Calvert. *PVS Language Reference (version 2.2)*, 1998. `http://www.csl.sri.com/pvsweb/manuals.html`
5. G. Collins. *A proof tool for reasoning about functional programs*, Glasgow, 1996. Submitted to 1996 International Conference on Theorem Proving in Higher Order Logics. `http://www.dcs.gla.ac.uk/~grmc/`
6. S. Mintchev. *Mechanized reasoning about functional programs*, Manchester, 1994. In K. Hammond, D.N. Turner and P. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 151-167. Springer-Verlag.
7. N. Winstanley. *Era User Manual, version 2.0*, Glasgow, 1998. `http://www.dcs.gla.ac.uk/ nww/Era/Era.html`
8. M. de Mol. *Clean Prover System: a proof tool for interactively and automatically proving properties of functional programs*, Master's Thesis no. 442, Nijmegen, 1998. `http://www.cs.kun.nl/~maartenm`