Theorem proving for functional programmers

SPARKLE: a functional theorem prover

Maarten de Mol, Marko van Eekelen and Rinus Plasmeijer

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

Abstract. SPARKLE is a new theorem prover written in and specialized for the functional programming language CLEAN. It is mainly intended to be used by programmers for proving properties of parts of programs, combining programming and reasoning into one process. It can also be used by logicians interested in proving properties of larger programs.
Two features of SPARKLE are in particular helpful for programmers. Firstly, SPARKLE is integrated in CLEAN and has a semantics based on lazy graph-rewriting. This allows reasoning to take place on the program itself, rather than on a translation that uses different concepts. Secondly, SPARKLE supports automated reasoning. Trivial goals will automatically be discarded and suggestions will be given on more difficult goals.
This paper presents a small example proof built in SPARKLE. It will be shown that building this proof is easy and requires little effort.

1 Introduction

It has often been stated that functional programming languages are well suited for formal reasoning. In practice, however, there is little support for reasoning about functional programs. Existing theorem provers, such as PVS[10], COQ[5] and ISABELLE[12], do not support the full semantics of functional languages and can only be used if the program is translated first, making them difficult to use.

Still, formal reasoning can be a useful tool for any programming language. To make reasoning about programs written in the functional programming language CLEAN[6] feasible, SPARKLE was developed. Work on SPARKLE started after a successful experiment with a restricted prototype[9]. SPARKLE is a semi-automatic theorem prover that can be used to reason about any CLEAN-program.

SPARKLE supports all functional concepts and has a semantics based on lazy graph-rewriting. It puts emphasis on tactics which are specifically useful for reasoning about functional programs and automatically provides suggestions to guide users in the reasoning process. SPARKLE is written in CLEAN; with approximately 130.000 lines of source code (also counting libraries and comments) it is one of the larger programs written in CLEAN. It has an extensive user interface which was implemented using the Object I/O library[1]. SPARKLE is prepared for CLEAN 2.0 and will be integrated in the new IDE. Currently, SPARKLE is a stand-alone application and can be downloaded at `http://www.cs.kun.nl/Sparkle`.

The ultimate goal of the project is to include formal reasoning in the programming process, enabling programmers to easily state and prove properties of parts of programs. This *on-the-fly* proving can only be accomplished if reasoning requires little effort and time. This is already achieved by SPARKLE for smaller programs, mainly due to the possibility to reason on source code level and the support for automatic proving.

In this paper a global description of SPARKLE and its possibilities will be presented. For this purpose a desired property of a small CLEAN-program will be formulated. It will be shown that building a formal proof for this property is very easy in SPARKLE. The specialized features of SPARKLE that in particular assist programmers in building this proof will be highlighted.

The rest of this paper is structured as follows. First, the specification language of SPARKLE will be introduced and the example program will be expressed in it. A comparison with the specification languages of other theorem provers will be made. Then, the logical language of SPARKLE is introduced and the property to prove will be defined in it. Then, a detailed description is given of a proof for this property in SPARKLE. Finally, after the conclusions and related work, an appendix is given in which the tactics needed to build the proof are explained.

2 The specification language of SPARKLE

Although SPARKLE can be used to prove properties about arbitrary CLEAN-programs, the reasoning process itself takes place on a simplified representation of the CLEAN-program. In this section the CLEAN-program to reason about will be presented and its simplification for SPARKLE will be described. Then, the specification of functional programs in other theorem provers will be examined and compared to SPARKLE.

2.1 The CLEAN-program

The proof that will be constructed in this paper relates the functions `take` and `drop` by means of the function `++`. These functions are defined in the standard environment of CLEAN. For this proof, however, the definitions of `take` and `drop` have been improved to handle negative arguments more consistently. The next distribution of CLEAN will use the improved definitions.

```
take :: Int ![a] -> [a]
take n [x:xs]
    | n < 1     = []
    | otherwise = [x:take (n-1) xs]
take n []
    = []

++ :: ![a] [a] -> [a]
++ [x:xs] ys
    = [x:xs++ys]
++ [] ys
    = ys
```

```
drop :: Int ![a] -> [a]
drop n [x:xs]
    | n < 1     = [x:xs]
    | otherwise = drop (n-1) xs
drop n []
    = []

- :: infixl 6 !Int !Int -> Int
- x y = code inline {subI}

< :: infix 4 !Int !Int -> Bool
< x y = code inline {ltI}
```

These definitions are very straightforward, but it is important to take note of the use of *strictness*. Because there is no exclamation mark in front of the first argument type of `take` (or `drop`), the expression `take` $\perp$ `[]` reduces to `[]` and not to $\perp$. Adding an exclamation mark would change this behavior. The exclamation mark in front of the second argument of `take` (or `drop`), however, is superfluous, because a pattern match is carried out on this argument.

The functions above also make use of several *predefined* concepts: the integer type `Int`, the integer denotations `0` and `1`, the list type `[a]`, the nil constructor `[]` and the cons constructor `[_:_]`. Furthermore, the auxiliary functions `-` and `<`, also from the standard environment of CLEAN, are defined in machine code.

2.2 Simplification for SPARKLE

Reasoning in SPARKLE takes place on CORE-CLEAN, which is a subset of CLEAN. CORE-CLEAN is a simple functional programming language, basically containing only application, sharing and case distinction. Its semantics is based on lazy graph-rewriting and it supports strictness annotations. Reductions leading to an error and non-terminating reductions are represented by the constant $\perp$.

SPARKLE automatically translates each CLEAN-program to CORE-CLEAN. For this purpose functions from the source code of the new CLEAN-compiler are used. These functions transform CLEAN to a variant of CORE-CLEAN which is used internally in the compiler. Translating CLEAN to CORE-CLEAN is by no means an easy task and would require a huge effort by hand. Using the real compiler saves a lot of work and has an additional advantage as well: it is trivially guaranteed that the translation preserves the semantics of the program.

The program to reason about is a very basic CLEAN-program and can be expressed in CORE-CLEAN almost immediately. The only concept used that is not supported by CORE-CLEAN is pattern matching. The patterns in the functions therefore have to be transformed to case distinctions. The effect of this translation is minimal and is further reduced by SPARKLE, which is able to hide top-level case distinctions and present them as patterns.

The translation of predefined concepts is not a problem, because these are also made available by CORE-CLEAN. The semantics of the representation of numbers, however, is different. SPARKLE disregards overflow and rounding errors, because these would complicate the reasoning process too much. Instead, an idealized representation of numbers is assumed, resulting in an `Int` type without bounds and a `Real` type without bounds and with infinite precision.

Translating *delta rules*, which are functions written in machine code, to CORE-CLEAN is problematic, however. SPARKLE is not able to translate an arbitrary delta rule to CORE-CLEAN. Instead, a fixed set of delta rules occurring in the standard environment of CLEAN is recognized. The translation of recognized delta rules is hard-coded in the theorem prover, usually by referring to mathematical definitions working on idealized numbers. This is for example the case for the subtract function from the example program.

2.3 Specification in other theorem provers

In order to reason about a program in a theorem prover, it must first be translated to its *specification language*, which is Core-Clean for Sparkle. This language is a very important aspect of a theorem prover, because the reasoning process takes place on the translated program in the specification language.

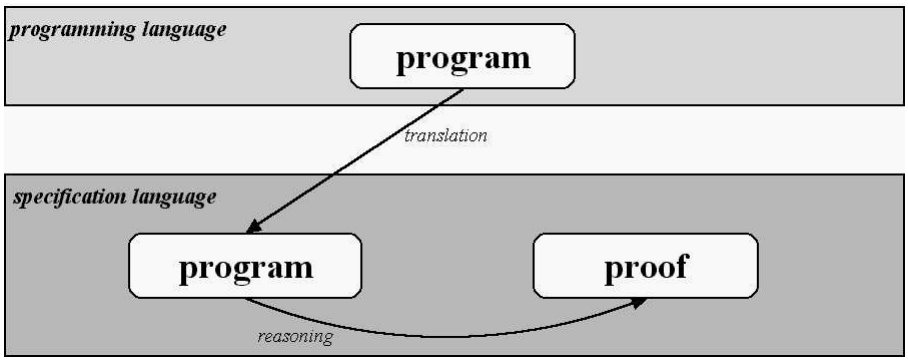

Fig. 1. Reasoning takes place in the specification language

For effective reasoning, a good understanding of the translated program is required. Programmers usually understand the programs they write very well, but this may not be the case for the translated version. If the differences are too big, knowledge of the original program is completely lost and proving will be a lot more difficult. Moreover, a new specification language must be mastered. These obstacles will likely lead to programmers giving up on formal reasoning.

Unfortunately, there is still a big gap between an executable (programming) language which is useful in practice and a formal (specification) language which is useful in theory. Differences between the specification language and the programming language are inevitable. The following differences can be distinguished, in decreasing order of importance:

1. *Differences in semantics.* These are quite serious, because understanding the translated program may become very difficult. Firstly, the concepts in the specification language may not be known to the programmer. Secondly, the relation between the original program and its translation may be lost, making it difficult to re-use the expertise of the original program.
2. *Differences in notational expressivity.* Sometimes complicated concepts have to be translated to simpler ones, such as translating notational sugar to ordinary function applications. These differences can again make it difficult to relate the translated program to the original program.
3. *Differences in syntax.* These are not so serious and can often be solved easily. However, it can still be very annoying to programmers.

The specification languages of existing theorem provers are very powerful but score badly on the points mentioned above. Most importantly, there are usually

many differences in semantics. For instance, Coq supports both reasoning about finite (inductive) and infinite (co-inductive) objects, but these objects can not be combined into one datatype. Strictness annotations are not supported by any existing theorem prover. Writing a translation from Clean to for instance the specification language of Pvs would require a huge effort and may in fact be as difficult as developing a new theorem prover.

All in all, using an existing theorem prover to reason about Clean-programs is very problematic for programmers.

2.4 Suitability of Core-Clean for reasoning about Clean

Reasoning in Sparkle takes place on Core-Clean, which is not a *new* language but only a subset of Clean:

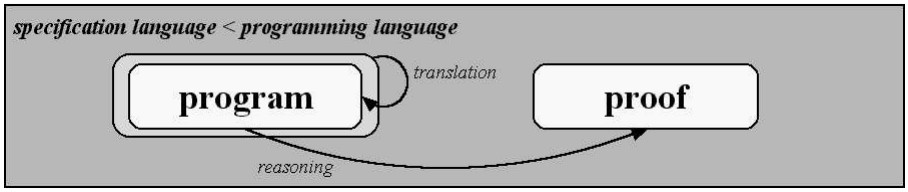

Fig. 2. Reasoning takes place in a subset of the programming language

In contrast to the specification language of other theorem provers, Core-Clean is very similar to Clean. There will not be many differences between a Clean-program and its simplification in Core-Clean:

1. *Semantics.* Core-Clean borrows its semantics from Clean[2], using a lazy term-graph rewriting system to reduce expressions. All programs written in Core-Clean are valid Clean as well and will therefore easily be understood by experienced Clean-programmers. The only difference in semantics lies in the handling of numbers. This is only a problem for programs in which overflow or rounding occurs. If one wants to reason about these programs, a different representation of numbers must be chosen.
2. *Concepts.* In Core-Clean all basic constructs of Clean are available. Notational sugar is translated to these basic concepts, including pattern matching (translated to case distinctions), overloading (translated to dictionaries), dot-dot-expressions (translated to functions) and comprehensions (translated to functions). The translated versions are usually recognized and understood easily by programmers, because they are not that different. There is, however, one exception: the translation of comprehensions to functions is not transparent at all. The functions created here are hard to understand and almost impossible to relate to the original program.
3. *Syntax.* Core-Clean uses the same syntax as Clean.

Due to these similarities, Core-Clean is a good specification language for reasoning about Clean-programs. The translation of comprehensions is, how-

ever, still problematic. This could be solved by using a different translation-scheme or by interpreting comprehensions; further investigation is required here.

3 The specification of the property

Properties can be specified in SPARKLE using a simple first-order propositional logic which is extended with equalities on expressions. The logical connectives $\neg, \rightarrow, \wedge, \vee, \leftrightarrow$ and the quantors $\forall, \exists$ are available. Quantification, either existential or universal, is possible over propositions and expressions of an arbitrary type. Predicates and quantification over predicates are not allowed.

A standard semantics for propositional logics is used. The semantics of the equality on expressions is defined using the operational reduction semantics of CLEAN. Two expressions are equal if for all reductions of one expression there exists a reduction of the other expression that produces the same constructors and basic values (and possibly more). This semantics covers both the equality between finite and infinite structures.

SPARKLE offers the following features to make the specification of properties as easy as possible:

- The same syntax may be used as in CLEAN, meaning that infix applications are allowed and no superfluous brackets have to be supplied.
- Top-level universal quantors may be omitted. For each free variable in the proposition, a top-level universal quantor will automatically be created by SPARKLE.
- It is optional to specify the types of the variables in a $\forall$ or $\exists$. If the type is left out, it will be inferred by the theorem prover. The property will always be type-checked.
- Quantification over type-variables is implicit and must not be specified. Properties will always be interpreted as polymorphic as possible.

The property that is going to be proved in this paper relates the functions `take` and `drop`. Using the described features it can be specified as follows:

$$\texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs$$

There is, however, a problem with this property. If $n = \bot$ and $xs = [7]$, the left-hand-side of the equation will reduce to $\bot$ while the right-hand-side is $[7]$. These kind of problems with undefined expressions occur frequently and can be very hard to detect beforehand. They will always be revealed in the reasoning process, though. An easy solution is to simply demand that n is always defined:

$$n \neq \bot \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs$$

This property contains two free variables, n and xs, for which universal quantors will be created automatically by SPARKLE. The type of n will be inferred as `Int` and the type of xs will be inferred as `[a]`. A universal quantor for the type variable `a` will be omitted. This results in the following property, which will be the starting point of the proof:

$$\forall_{n \in \texttt{Int}} \forall_{xs \in [\texttt{a}]} [n \neq \bot \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs]$$

4 Building the proof

In this section the process of building a proof for the given property in SPARKLE will be described. Before the proof itself is given, the reasoning style of SPARKLE (how proofs are constructed) and its hint mechanism (a mechanism to assist users in building proofs) will be explained.

4.1 Reasoning style in SPARKLE

Reasoning in SPARKLE is similar to reasoning in other theorem provers and consists of the repeated application of tactics on goals until all goals are discarded.

A *goal* is a property that still has to be proven. Each goal is associated with a *goal context*. In a goal context variables are declared and local hypotheses are stored. The *proof state* consists of a list of goals. The active goal being manipulated is called the *current goal*; the others are called *subgoals*. Changing the active goal is always allowed.

A *tactic* is a function from a single goal to a list of goals. Applying a tactic on the current goal will lead to a new proof state, which consists of the created goals and the old subgoals. All tactics must be *sound* with respect to semantics, meaning that the validity of the created goals must logically imply the validity of the original goal.

SPARKLE implements a total of 42 tactics. Although all of these tactics can also be found, or expressed, in other theorem provers, their behavior is specifically geared towards proving properties of lazy functional programs. The `Induction` tactic, for example, can only be applied to admissible propositions (see [11]) and is valid for both finite and infinite structures.

A proof of the example property can be constructed using a subset consisting of eight tactics, which are: (1)**Contradiction**(proof by contradiction); (2)**Definedness**(use absurd hypotheses concerning $\bot$); (3)**Induction**(structural induction); (4)**Introduce**(elimination of $\forall$ and $\rightarrow$); (5)**Reduce**(reduction to root-normal-form); (6)**Reflexive**(prove reflexive equality); (7)**Rewrite**(rewrite according to a hypothesis); (8)**SplitCase**(case distinction). See the appendix for a more detailed description of these tactics.

4.2 The hint mechanism

Successfully building a proof in SPARKLE depends on the selection of the right tactics. For this, knowledge of the available tactics and their effect is needed, as well as expertise in proving. To make the selection of tactics easier, a hint mechanism is available in SPARKLE.

The hint mechanism is activated each time the current goal changes. It automatically produces a list of applicable tactics. Based on built-in heuristics only the most important applicable tactics are suggested. Each tactic is assigned a score between 1 and 100 that indicates the likelihood of that tactic being helpful in the proof. A score of 100 is reserved for tactics that prove the current goal in one step. The assignment of scores to tactics is hard-coded in SPARKLE.

The hint mechanism is a valuable tool, especially for those with little expertise in proving. However, it is by no means a failsafe feature. Sometimes the right tactic is not suggested or several wrong tactics get high scores. Programmers can use the mechanism to their advantage but should not completely rely on it. Future work will concentrate on improving the hint mechanism.

On top of making users aware of useful applicable tactics, there are two additional advantages offered by the hint mechanism:

1. Suggested tactics are assigned a hot-key and can be applied instantly. This reduces the typing (or clicking) effort for building proofs considerably.
2. A threshold for automatic application can be set. If the best applicable tactic has a score higher than this threshold, it will be applied automatically. This process continues until no tactic with a high enough score can be found. A low threshold can be used for automatic proving; a medium threshold for semi-automatic proving and a high threshold for manual proving.

4.3 Proof of the example program

In this subsection a proof of the example property built with SPARKLE will be presented. The description will focus on the goals that have to be proved. At each goal, a tactic to be applied is chosen. An argument for this choice will be given. The description then continues with the first goal that is created; if several goals are created, they will be proved later. The order in which the goals are proved is the same as in SPARKLE. (to be more precise: all unproved goals are stored in a proof tree, which is traversed from left to right and top-down). A numbering system is used to keep track of the goals.

The initial goal is simply the property to be proven. It has an empty context.

$$\boxed{\frac{\text{-}}{\forall_{n \in \texttt{Int}} \forall_{xs \in [\texttt{a}]} [n \neq \bot \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs]}} \quad (1)$$

Because of the definitions of `take` and `drop`, which are tail-recursive in the list argument, structural induction on xs is likely to be useful here. This is accomplished by applying the tactic `Induction xs`. Three new goals(1.1,1.2,1.3) are created: one for the case that xs is $\bot$; one for the case that xs is [] and one for the case that xs is a non-empty list. Note that $\bot$ is treated as a constructor for all algebraic types; therefore induction creates three new goals instead of two.

$$\boxed{\frac{\text{-}}{\forall_{n \in \texttt{Int}} [n \neq \bot \rightarrow \texttt{take}\ n\ \bot\ \texttt{++}\ \texttt{drop}\ n\ \bot = \bot]}} \quad (1.1)$$

The *goal context* is used to store introduced variables and hypotheses. It is actually just a prettier representation of a chain of $\forall$'s and$\rightarrow$'s, which allows the reasoning to focus on the interesting part of the goal. Another induction is not needed in the current goal. The variable n and the hypothesis $n \neq \bot$ can therefore safely be moved to the goal context using the tactic `Introduce n H1`.

$$\frac{n \in \texttt{Int} \quad \textbf{H1:}\ n \neq \bot}{\texttt{take}\ n\ \bot\ \texttt{++}\ \texttt{drop}\ n\ \bot = \bot} \quad (1.1')$$

Due to the strictness of `take` and `++` and the presence of $\bot$ arguments, redexes are present in the current goal. The tactic `Reduce NF All` can be used to reduce all redexes in the current goal to normal form (eager reduction). With other parameters, the tactic `Reduce` can also be used for stepwise reduction, lazy reduction, reduction of one particular redex and reduction in the goal context.

$$\frac{n \in \texttt{Int} \quad \textbf{H1:}\ n \neq \bot}{\bot = \bot} \quad (1.1'')$$

This is clearly a trivial goal, because equality is a reflexive relation. Such reflexive equalities are proved immediately with the tactic `Reflexive`.

$$\frac{\text{-}}{\forall_{n \in \texttt{Int}}[n \neq \bot \rightarrow \texttt{take}\ n\ []\ \texttt{++}\ \texttt{drop}\ n\ [] = []]} \quad (1.2)$$

This is the second case of the induction, created for the case that $xs = []$. Again, introduction in the context should be done first: `Introduce n H1`.

$$\frac{n \in \texttt{Int} \quad \textbf{H1:}\ n \neq \bot}{\texttt{take}\ n\ []\ \texttt{++}\ \texttt{drop}\ n\ [] = []} \quad (1.2')$$

There are again redexes present in the current goal, due to the pattern matching performed by `take` and `drop`. Therefore: `Reduce NF All`.

$$\frac{n \in \texttt{Int} \quad \textbf{H1:}\ n \neq \bot}{[] = []} \quad (1.2'')$$

This is another example of a reflexive equality; therefore `Reflexive`.

$$\frac{\text{-}}{\begin{array}{l}\forall_{x \in \texttt{a}} \forall_{xs \in [\texttt{a}]}[\\ \quad \forall_{n \in \texttt{Int}}[n \neq \bot \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs] \\ \quad \rightarrow \forall_{n \in \texttt{Int}}[n \neq \bot \rightarrow \texttt{take}\ n\ [x{:}xs]\ \texttt{++}\ \texttt{drop}\ n\ [x{:}xs] = [x{:}xs]\ ]]\end{array}} \quad (1.3)$$

This is the third goal created by the induction; the induction step. The current goal looks quite complicated, but introduction can make things a lot clearer. For reasons of clarity, the first hypothesis will be called IH (induction hypothesis) and the variable n will be introduced as m (to avoid name conflicts with the n already present in the induction hypothesis): `Introduce x xs IH m H1`.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \to \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \end{array}}{\mathtt{take}\ m\ [x{:}xs]\ \mathtt{++}\ \mathtt{drop}\ m\ [x{:}xs] = [x{:}xs]} \quad (1.3')$$

Again, the current goal contains redexes that can be removed by applying the tactic `Reduce NF All`. Note that a lazy reduction (to root-normal-form) will not suffice here, because ++ is lazy in its second argument and therefore `drop` m $[x{:}xs]$ as a whole will not be reduced at all.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \to \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \end{array}}{\left(\begin{array}{ll} \text{case } (m < 1) \text{ of} & \\ \quad \mathtt{True} & \to [] \\ \quad \mathtt{default} & \to [x{:}\mathtt{take}\ (m\text{-}1)\ xs] \end{array}\right) \mathtt{++} \left(\begin{array}{ll} \text{case } (m < 1) \text{ of} & \\ \quad \mathtt{True} & \to [x{:}xs] \\ \quad \mathtt{default} & \to \mathtt{drop}\ (m\text{-}1)\ xs \end{array}\right) = [x{:}xs]}$$

(This proof state is also shown in Fig. 3.)

The natural next step is a case distinction on $m < 1$, because that will allow the reduction of both case-expressions in the current goal. A special tactic is used for this purpose: `SplitCase 1`. This tactic will examine the first case-expression in the current goal. Three cases are distinguished: (1) $\bot$ (for when $m < 1$ can not be properly evaluated); (2) `True` (for the first alternative); (3) `False` (for the default alternative). For each case a new goal(1.3.1,1.3.2,1.3.3) is created, in which the appropriate alternatives of the case-expressions are chosen. Also, in each goal hypotheses are introduced to reflect the case chosen.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \to \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \bot \end{array}}{\bot\ \mathtt{++}\ \bot = [x{:}xs]} \quad (1.3.1)$$

This is the goal created by `SplitCase` for the case that $m < 1 = \bot$. This goal can be proved in one step, because hypotheses H1 and H2 are contradictory. This is due to the totality of $<$, which ensures that $x < y$ can only be $\bot$ if either $x = \bot$ or $y = \bot$. Hypothesis H2 states that $m < 1 = \bot$, thus either $m = \bot$ or $1 = \bot$. Of course, $1 = \bot$ is not true, thus from hypothesis H2 it may be concluded that $m = \bot$. This contradicts with hypothesis H1. In SPARKLE, a specialized tactic is available to handle these cases: `Definedness`. This tactic searches for expressions (most notably, variables) that are *defined* (known to be unequal to $\bot$) and expressions that are *undefined* (known to be equal to $\bot$). The analysis makes use of the hypotheses, the ordinary strictness information of functions and the totality of functions such as $-$ and $<$. If an expression is found which is both defined and undefined, the goal is proved by contradiction.

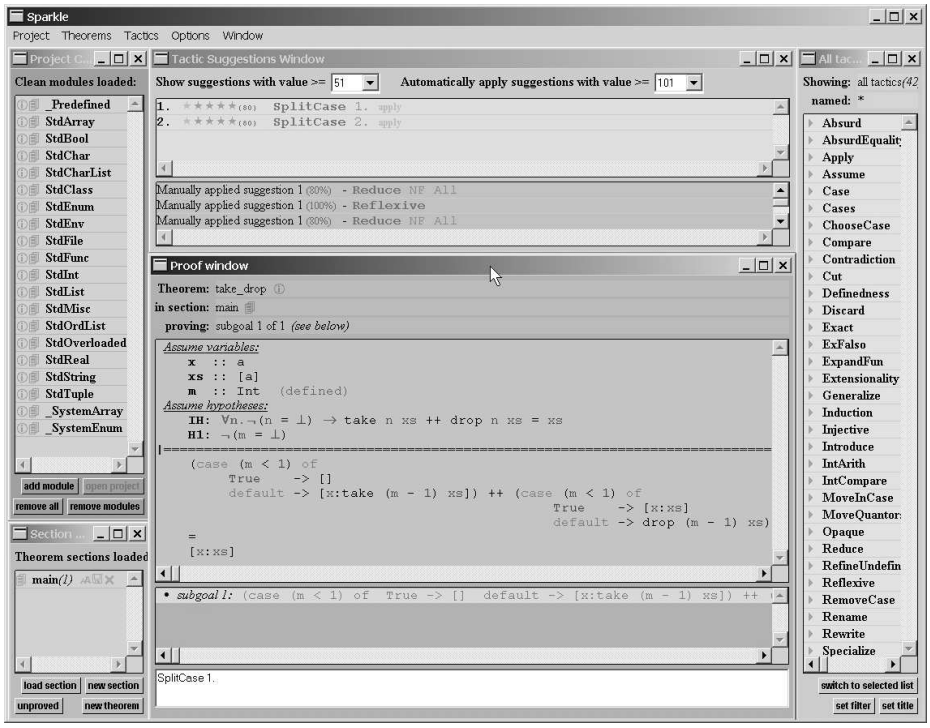

Fig. 3. The theorem prover in action

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_n [n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{True} \end{array}}{[]\ \mathtt{++}\ [x{:}xs] = [x{:}xs]} \quad (1.3.3)$$

This is the goal created by `SplitCase` for the case that $m < 1 = \mathtt{True}$. The appropriate case alternatives have been chosen and the resulting goal is clearly a trivial one. It can be proved by a reduction followed by an application of `Reflexive`. These two can be combined by `Reduce NF All; Reflexive`.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}} [n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{False} \end{array}}{[x{:}\mathtt{take}\ (m\text{-}1)\ xs]\ \mathtt{++}\ \mathtt{drop}\ (m\text{-}1)\ xs = [x{:}xs]} \quad (1.3.3)$$

This is the goal created by `SplitCase` for the case that $m < 1 = \mathtt{False}$. Filling in the proper case alternatives has resulted in a goal which contains a redex (`++` can be reduced); therefore: `Reduce NF All`.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{False} \end{array}}{[x{:}\mathtt{take}\ (m\text{-}1)\ xs\ \mathtt{++}\ \mathtt{drop}\ (m\text{-}1)\ xs] = [x{:}xs]} \quad (1.3.3')$$

In this goal it is finally possible to use the induction hypothesis, using $(m-1)$ as value for n. This results in the substitution of `take` $(m-1)$ xs `++ drop` $(m-1)$ xs by xs in the current goal. This is accomplished in SPARKLE by the tactic `Rewrite IH`. This tactic will create two new goals, one for the goal after substitution (1.3.3.1) and one for the condition $m-1 \neq \bot$ (1.3.3.2).

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{False} \end{array}}{[x{:}xs] = [x{:}xs]} \quad (1.3.3.1)$$

This trivial goal is proved immediately by `Reflexive`.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{False} \end{array}}{(m-1) \neq \bot} \quad (1.3.3.2)$$

This goal is proved by contradiction: the negation of the current goal will lead to an absurd situation. This action is performed by the tactic `Contradiction`, which creates a hypothesis that is the negation of the current goal and replaces the current goal by the proposition $False$.

$$\frac{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (m < 1) = \mathtt{False} \\ \textbf{H3: } (m-1) = \bot \end{array}}{False} \quad (1.3.3.2')$$

This goal is similar to goal 1.3.1. Here, a contradiction can be found by examining hypotheses H1 and H3 and using the totality of $-$. An application of `Definedness` will therefore prove the current goal and finish the proof!

4.4 Remarks concerning the reasoning process

The presented proof was not difficult to build. An examination of the current goal always resulted in a tactic to apply; no overview of the proof as a whole

was ever required. This actually turns out to be the case for many small proofs about functional programs.

The hint mechanism is especially useful for building such 'goal-directed' proofs. In fact, *all steps in the presented proof were given as hints by* SPARKLE. Building the proof is therefore reduced to selecting hints, which is a lot easier than selecting tactics, simply because there are far less options to choose from. Right now, there are 42 different tactics which can have arguments as well, whereas there are typically less than 15 hints given for a small-sized goal.

Automatic proving is possible in SPARKLE by letting it automatically apply the hint with the highest score. *The example property can be proved automatically with the hint mechanism.* Of course, larger and more difficult proofs can not be built automatically, although often suggestions given by SPARKLE can be used successfully. Further improving the hint mechanism will be one of the spearheads in the further development of SPARKLE.

A proof of (almost) the same property is also presented in Bird's Introduction to Functional Programming[3]. The proof presented there only takes positive integer arguments into account, but is otherwise quite similar. Note that building such a formal proof with the aid of a theorem prover is much easier than doing it on paper. In [3], a lot of proofs of properties about functional programs are given. A lot of these proofs (about 80%) have already been translated to SPARKLE without difficulties. No problems are expected for translating the others.

5 Conclusions and further work

Building the proof required little effort and little expertise. The proving action could always be found by examining the current goal and following a few ground rules. The theorem prover is able to follow these same ground rules and suggest the correct tactics to users, reducing the required expertise even more. All in all, a programmer can build this proof in a short time and without many difficulties.

The two features of SPARKLE that contribute the most to this are:

- The possibility to reason about the source program. Starting with proving is trivial: state what you want to prove and run the theorem prover.
- The hint mechanism. Selecting suggested hints is very easy. An application of a hint can easily be undone, making playing with hints possible. This is not only a fast way of learning how to use the system, but also a fast way of actually constructing the proof.

There are, however, lots of things that still need to be done. Although SPARKLE can already be used to build proofs, it is by no means finished. For instance, documentation must still be added to the system. Furthermore, the hint mechanism must be compared to the automatic reasoning abilities of other theorem provers and possibilities to improve the mechanism must be researched.

Also, work needs to be done on the formal framework of the theorem prover. The effect of the tactics must be described formally in this framework and their soundness with respect to the semantics of CLEAN must be proved. Of particular importance is the soundness of `Induction` for all lazy structures.

6 Related work

In many textbooks (for instance [3]) properties about functional programs are proved by hand. Also, several articles (for instance [4]) make use of reasoning about functional programs. It seems worthwhile to attempt to formalize these proofs in SPARKLE. In programming practice, however, reasoning about functional programs is scarcely used.

Widely used generic theorem provers are PVS[10], COQ[5] and ISABELLE[12]. They are not tailored towards a specific programming language. Reasoning in these provers requires using a syntax and semantics that are different from the ones used in the programming language. For instance, strictness annotations as in CLEAN are not supported by any existing theorem prover. This makes it rather hard for a programmer to use. On the other hand, these well established theorem provers offer features that are not available in SPARKLE. Most notably, the tactic language and the logic are much richer than in SPARKLE.

Somewhat closer related work is described in [8], in which a description is given of a proof tool which is dedicated to HASKELL[13]. 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.

Further related work concerns a proof tool specialized for HASKELL, called ERA, which stands for Equational Reasoning Assistant. This proof tool is still in development, although a working prototype is available. ERA, however, is intended to be used for equational reasoning, and not for theorem proving in general. Additional proving methods, including induction or any logical tactics, are not supported. ERA is a stand-alone application.

Another theorem prover which is dedicated to a functional programming language is EVT[7], the Erlang Verification Tool. It differs from SPARKLE because ERLANG is a strict, untyped language which is mainly used for developing distributed applications. EVT has been applied in practice to larger examples.

We do not know of any other theorem prover than SPARKLE that is integrated, tailored towards a lazy functional language and semi-automatic.

References

1. P. Achten and M. Wierich. *A Tutorial to the CLEAN Object I/O Library (version 1.2)*, Nijmegen, February 2000. CSI Technical Report, CSI-R0003.
2. E. Barendsen and S. Smetsers. *Strictness Typing*, Nijmegen, 1998. In proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, 1998, pages 101-116.
3. R. Bird. *Introduction to Functional Programming using Haskell, second edition*, Prentice Hall Europe, 1998, ISBN 0-13-484346-0.
4. A. Butterfield and G. Strong. *Proving Correctness of Programs with I/O - a paradigm comparison*, Dublin, 2001. In proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, 2001, pages 319-334.

5. The Coq Development Team. *The Coq Proof Assistant Reference Manual (version 7.0)*, Inria, 1998. `http://pauillac.inria.fr/coq/doc/main.html`
6. M. van Eekelen and R. Plasmeijer. *Concurrent Clean Language Report (version 1.3)*, Nijmegen, June 1998. CSI Technical Report, CSI-R9816. `http://www.cs.kun.nl/~clean/Manuals/manuals.html`
7. T. Noll, L. Fredlund and D. Gurov. *The EVT Erlang Verification Tool*, Stockholm, 2001. In proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Lecture Notes in Computer Science, Vol. 2031, Springer, 2001, ISBN 3-540-41865-2, pages 582-585.
8. 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.
9. M. de Mol and M. van Eekelen. *A proof tool dedicated to Clean - the first prototype*, 1999. In proceedings of Applications of Graph Transformations with Industrial Relevance 1999, Lecture Notes in Computer Science, Vol. 1779, Springer, 2000, ISBN 3-540-67658-9, pages 271-278.
10. S. Owre, N.Shankar, J.M. Rushby and D.W.J. Stringer-Calvert. *PVS Language Reference (version 2.3)*, 1999. `http://pvs.csl.sri.com/manuals.html`
11. L. C. Paulson. *Logic and Computation*, Cambridge University Press, 1987. ISBN 0-52-134632-0.
12. L. C. Paulson. *The Isabelle Reference Manual*, Cambridge, 2001. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`
13. S. Peyton Jones(editor), J. Hughes(editor) et al. *Report on the programming language Haskell 98*, Yale, 1999. `http://www.haskell.org/definition/`
14. N. Winstanley. *Era User Manual, version 2.0*, Glasgow, 1998. `http://www.dcs.gla.ac.uk/~nww/Era/Era.html`

A A short description of 8 tactics (appendix)

This appendix provides a short description of the tactics used in the example proof. In this description the following categorization of tactics is used:

Safe/Unsafe - A tactic is safe if its inverse is also a valid tactic. This can only be the case if the created goals are logically equivalent to the original goal, whereas an unsafe tactic creates goals which are logically stronger.

Forwards/Backwards - Forwards reasoning brings hypotheses closer to the current goal (top-down), while backwards reasoning brings the current goal closer to the hypotheses (bottom-up).

Instantaneous - An instantaneous tactic proves a goal in one single step. Such a tactic will not be categorized as safe/unsafe or forwards/backwards.

Programming/Logic - Logic tactics are based on the semantics of the logical connectives. Programming tactics are based on the semantics of CLEAN.

```
Contradiction.
```

Type: Safe; backwards; logic.
Info: Builds a proof by contradiction.
Effect: Replaces the current goal by the absurd proposition *False* and adds its negation as a hypothesis in the context. If a double negation is produced, it will

be removed automatically.

Example: $xs \vdash xs$ `++` $[] \neq xs$
$\Longrightarrow$ `Contradiction` $\Longrightarrow$
(1) xs, $\langle xs$ `++` $[] = xs\rangle \vdash False$

`Definedness.`

Type: Instantaneous; logic.

Info: Determines two sets of expressions: (1) the set of defined expressions, which are expressions that are unequal to $\bot$; (2) the set of undefined expressions, which are expressions that are equal to $\bot$. These sets are determined by examining equalities in hypotheses and using strictness information. In addition, the totality of certain predefined functions is used.

Effect: If an expression is found that is both defined and undefined, the goal is proved instantaneously. Otherwise nothing happens.

Example: $xs, ys, zs, \langle xs = \bot\rangle, \langle xs$ `++` $ys = [1{:}zs]\rangle \vdash False$
$\Longrightarrow$ `Definedness` $\Longrightarrow$
$\Box$

`Induction <variable>.`

Type: Unsafe; backwards; programming.

Info: Performs structural induction on a variable. A goal is created for each root-normal-form the variable may have, including $\bot$. The type of the variable must be `Int`, `Bool` or algebraic. The root-normal-forms of an algebraic type are determined by its constructors.

Effect: In each created goal the variable is replaced by its root-normal-form. Universal quantors are created for new variables. Additionally, induction hypotheses are added (as implications) for all recursive variables.

Example: $\vdash \forall_{xs}[xs$ `++` $[] = xs]$
$\Longrightarrow$ `Induction xs` $\Longrightarrow$
(1) $\vdash \bot$ `++` $[] = \bot$
(2) $\vdash []$ `++` $[] = []$
(3) $\vdash \forall_x \forall_{xs}[xs$ `++` $[] = xs \rightarrow [x{:}xs]$ `++` $[] = [x{:}xs]]$

`Introduce <name(1)> <name(2)> ... <name(n)>.`

Type: Safe; backwards; logic.

Info: Moves as many universally quantified variables and hypotheses to the goal context as there are names given.

Effect: The current goal must be of the form $\forall_{x_1} \ldots \forall_{x_a}[P_1 \rightarrow \ldots P_b \rightarrow Q]$, where $a + b = n$. The quantors and implications may be mixed. The variables $x_1 \ldots x_a$ and the hypotheses $P_1 \ldots P_b$ are deleted from the current goal and are added to the goal context using the names given.

Example: $\vdash \forall_x[x = 7 \rightarrow \forall_y[y = 7 \rightarrow x = y]]$
$\Longrightarrow$ `Introduce p H1 q H2` $\Longrightarrow$
(1) $p, q, \langle$`H1:` $p = 7\rangle, \langle$`H2:` $q = 7\rangle \vdash p = q$

`Reduce NF All.`

Type: Safe; backwards; programming.

Info: Reduces all expressions in the current goal to normal form. This is accom-

plished by a reduction to root-normal-form, followed by the recursive reduction to normal form of all top-level lazy arguments. The functional reduction strategy is used. An artificial limit on the maximum number of reduction steps is imposed in order to safely handle non-terminating reductions. Replacing the `NF` by `RNF` results in reduction to root-normal-form only.
Effect: All redexes are replaced by their reducts.
Example: $x, xs, ys, zs \vdash [x{:}xs]$ `++` $ys =$ `reverse` zs
$\Longrightarrow$ `Reduce NF All` $\Longrightarrow$
(1) $x, xs, ys, zs \vdash [x{:}$ xs `++` $ys] =$ `reverse` zs

`Reflexive.`

Type: Instantaneous; logic.
Info: Proves any reflexive equality instantaneously.
Effect: Proves any goal of the form $E = E$. Additional quantors and implications are allowed in front of the equality.
Example: $\vdash \forall_{xs} \exists_{ys} [xs = ys \rightarrow xs$ `++` $ys = xs$ `++` $ys]$
$\Longrightarrow$ `Reflexive` $\Longrightarrow$
$\Box$

`Rewrite <hypothesis>.`

Type: Safe; backwards/forwards; logic.
Info: Rewrites the current goal using an equality in a hypothesis.
Effect: The hypothesis must be of the form $\forall_{x_1} \ldots \forall_{x_n} [P_1 \rightarrow \ldots P_m \rightarrow E_1 = E_2]$. For all substitutions $\overrightarrow{x_i} = \overrightarrow{e_i}$ such that $E_1[\overrightarrow{x_i} := \overrightarrow{e_i}]$ occurs in the current goal, $E_1[\overrightarrow{x_i} := \overrightarrow{e_i}]$ is replaced by $E_2[\overrightarrow{x_i} := \overrightarrow{e_i}]$. Variables in the context are treated as constants. Additionally, goals are created for all conditions $P_1 \ldots P_m$.
Example: $xs, \langle$`H1:` $xs = [] \rangle \vdash xs$ `++` $xs = xs$
$\Longrightarrow$ `Rewrite H1` $\Longrightarrow$
(1) $xs, \langle$`H1:` $xs = [] \rangle \vdash []$ `++` $[] = []$

`SplitCase <number>.`

Type: Unsafe; backwards; programming.
Info: Performs a case distinction based on the case-expression in the current goal that is indicated by the argument number. For each alternative a goal will be created. Two goals are always created: (1) for the case that evaluation of the condition produces an error; (2) for the case that no alternative matches (replaced by the default alternative if one is available).
Effect: In each created goal, the case-expression is replaced by the result of the alternative chosen (or $\perp$ for the erroneous case). Hypotheses are introduced in the context to reflect which alternative was chosen. The goal for the default alternative is optimized: a negation of all other alternatives is transformed to an ordinary alternative if possible.
Example: $xs \vdash$ (`case` xs `of` $[y{:}ys] \rightarrow y$; `default` $\rightarrow 12) = 0$
$\Longrightarrow$ `SplitCase 1` $\Longrightarrow$
(1) $xs, \langle xs = \perp \rangle \vdash \perp = 0$
(2) $xs, y, ys, \langle xs = [y{:}ys] \rangle \vdash y = 0$
(3) $xs, \langle xs = [] \rangle \vdash 12 = 0$