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 and specialized in the functional programming language Clean. It is in the first place intended to be used by programmers on small parts of Clean-programs, combining theorem proving and programming into one process. It should of course also be usable by logicians interested in proving properties of larger programs.
This paper presents an example proof about a small Clean-program. It will be shown that building this proof in Sparkle is very easy and will require little effort from Clean-programmers. The most important features of Sparkle will be illustrated.
Two features in particular are helpful for programmers. Firstly, Sparkle is integrated in Clean and has a semantics based on lazy graph-rewriting. This allows the reasoning to take place on the program itself, rather than on a translated version which uses different concepts. Secondly, Sparkle supports automated reasoning. Trivial goals will automatically be discarded and suggestions will be given on more difficult goals.

1 Introduction

Formulating properties of (parts of) functional programs can be very useful. These properties can help to understand the behavior of the program and the intentions of the programmer, making maintenance easier. One can even go a step further and *prove* the correctness of these properties in a formal framework. This can guarantee the partial correctness of the program, and, in case the property could not be proven, help in finding errors in the program.

Unfortunately, building formal proofs is not an easy task. Not only a deep understanding of the program in question is needed, but also a profound knowledge of logic and proving techniques. Moreover, formal reasoning forces one to deal with a lot of uninteresting details, making it a tedious and lengthy exercise.

A good theorem prover can make formal reasoning considerably easier. A lot of well-established theorem provers are available, such as Pvs[8], Coq[4] and Isabelle[9]. However, these are general purpose theorem provers that lack support for expressing functional programs and reasoning about them. Functional programs typically use concepts as laziness, strictness annotations, sharing, overloading and partial functions. If a theorem prover does not make these concepts available, reasoning in it becomes unnecessarily difficult.

To fill this gap, SPARKLE was developed. Work on SPARKLE started after a successful experiment with a prototype[7], which could only be used for proving a limited set of properties, but was very easy to use. SPARKLE is a semi-automatic theorem prover which specializes in a subset of the functional programming language CLEAN[5]. This subset is very basic and is also available in similar functional programming languages like HASKELL[10], making it in principle possible to use SPARKLE to reason about basic HASKELL-programs as well.

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 provides a suggestion mechanism which is able to hint users in the right direction. SPARKLE is written in CLEAN; with approximately 55.000 lines of CLEAN source code (counting libararies even ± 130.000; note that *all* lines were counted, also those containing comments only) 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. It is a stand-alone application and can be downloaded at `http://www.cs.kun.nl/Sparkle`.

The ultimate goal of the project is to combine programming and theorem proving into one process, enabling programmers to prove properties *on-the-fly*. On-the-fly proving can only be accomplished if reasoning is easy and does not require much effort or 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. Of course, SPARKLE also supports reasoning about larger programs, but this may require more expertise and effort.

In this paper a proof about a small CLEAN-program will be presented. It will be shown that building the proof in SPARKLE is very easy, making it possible for a programmer to do the proof on-the-fly. The process of building the proof will be described in detail, highlighting the specialized features that SPARKLE offers and their effect on the reasoning process.

The rest of this paper is structured as follows. First the example program and the property about this program are presented. Then it is described how this program and property can be expressed in SPARKLE. A description of a proof of this property in SPARKLE, is given next; this is the biggest part of the paper. At the end, after the conclusions and related work, an appendix is given in which the tactics needed to build the proof are described.

2 The example program

The example program involves the well-known functions `drop` and `take`. These functions are also defined in the standard environment of CLEAN, but slightly differently. The versions below handle negative arguments more consistently. The next distribution of CLEAN will use these corrected versions.

```
take :: Int ![a] -> [a]
take n [x:xs]
```

```
   | n > 0          = [x: take (n-1) xs]
   | otherwise      = []
take n []
   = []

drop :: Int ![a] -> [a]
drop n [x:xs]
   | n > 0          = drop (n-1) xs
   | otherwise      = [x:xs]
drop n []
   = []
```

Note that the first integer argument of both `drop` and `take` is not strict, because the second alternative produces a result that does not depend on it. This means that `take undef []` will reduce to `[]`. The list argument in both cases is *head-strict*, which means that it should be known whether the list starts with a `Cons` or a `Nil` before `take` or `drop` may be reduced. If the list itself is undefined, the result of `drop` and `take` will also be undefined. The elements of the list do not have to evaluated however.

Since the function `-` is used in both `take` and `drop`, it is also part of the example. In CLEAN, the type `Int` is predefined and can not be accessed algebraically. Therefore, the function `-` on integers is defined by machine code:

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

The goal of the example is to prove that `drop` and `take` negate each other, which is expressed by the following property:

$$\forall_{n \in \texttt{Int}} \forall_\alpha \forall_{xs \in [\alpha]} [\texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs]$$

Although this formulation seems probable, it is incorrect. This can detected by examining the behavior when `n=`$\perp$ and `xs=[7]`. In that case, the left-hand-side of the equality will reduce to $\perp$ and the right-hand-side to `[7]`. These kind of problems with undefined expressions occur frequently and can be very hard to detect beforehand. However, they will always be revealed in the reasoning process.

An attempt to correct this problem, trying to leave the property to be proven intact, is to manually make the first argument of both `drop` and `take` strict. This does not help, however, because the case `n=`$\perp$ and `xs=[]` now falsifies the property. A solution that does work is to add an implication in which the left-hand-side excludes the problematic case. In this paper a slightly simplified version of this approach is used, excluding **all** cases with an undefined `n`:

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

Because the function ++ is used, it must also be part of the example:

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

3 Expressing the program

Expressing the example program in SPARKLE is trivial, because it is accepted as it is. In general, however, CLEAN-programs have to be simplified for SPARKLE. Still, this *simplification* is much easier than the *translation* required when expressing the program in other theorem provers. In fact, this translation is one of the main obstacles for using other provers and one of the main arguments to use SPARKLE. This will be explained in more detail in the following subsections.

3.1 Demands on the specification language; other provers

In order to reason about a program in a theorem prover, it must first be translated to its *specification language*. 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.

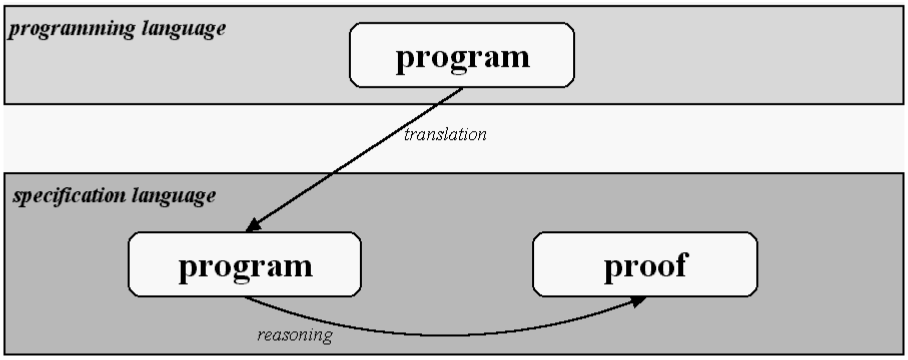

Fig. 1. Reasoning takes place in the specification language

For effective reasoning, one must understand the program and its behavior well. Programmers usually understand the programs they write very well. However, this may no longer be the case if the program is translated to a different language. 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.

The closer the specification language is to the programming language, the better. There is, however, still a big gap between an executable language which is useful in practice and a formal language which is useful in theory. Differences between the specification language and the programming language are inevitable, as are differences between the original program and its translated version. 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 definition. 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.

3.2 The specification language of Sparkle: Core-Clean

The specification language of Sparkle is 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 are represented by the constant expression $\perp$. Also, efficient basic values and operations on basic values are available in Core-Clean.

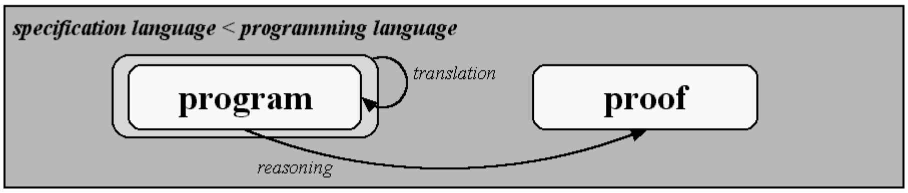

Fig. 2. Reasoning takes place in a subset of the programming language

The differences between CLEAN and CORE-CLEAN are not big. The criteria from the previous subsection can be used for a comparison:

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 basic values. In CORE-CLEAN, an idealized notion of numbers is used. It is assumed that all whole numbers (no bounds) can be represented as `Ints` and all real numbers (infinite precision, no bounds) can be represented as `Reals`. It is also assumed that operations on basic numbers are perfect (same behavior as mathematical counterparts). 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. For other programs the behavior is the same.
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 ZF-expressions (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 ZF-expressions 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.

All in all, CORE-CLEAN resembles CLEAN a lot and is well suited to reason about CLEAN-programs. The translation of ZF-expressions is, however, still problematic. This could be solved by using a different translation-scheme or by directly interpreting ZF-expressions; further investigation is required here.

3.3 Translating CLEAN to CORE-CLEAN

SPARKLE automatically translates each CLEAN-program to CORE-CLEAN. This is actually accomplished by invoking the real CLEAN-compiler, which is possible because the new compiler is written in CLEAN and uses a variant of CORE-CLEAN as intermediate language for compilation. Translating CORE-CLEAN to CLEAN is by no means an easy task. Writing a translation from a functional language to CORE-CLEAN by hand would require a huge effort. 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.

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.

4 Expressing the property

SPARKLE provides a standard logic extended with equalities on expressions. The usual logical connectives ($\neg, \rightarrow, \wedge, \vee, \leftrightarrow$) and quantors ($\forall, \exists$) are available. Quantification, either existential or universal, is possible over expressions of an arbitrary type, as well as over basic propositions. Predicates and quantification over predicates are not allowed.

The example property can be expressed in this logic without difficulties. Several features in SPARKLE are available to make expressing 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.
- 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.
- Quantifying over type variables (such as the α in the example program) is not needed. A universal quantification over all free type variables will be added implicitly by the theorem prover. Note that it is not possible to explicitly specify such quantifications.
- Universal quantifications are completely optional. A universal quantor will be added for all free variables found.

Using these features to full effect, it is possible to state the property in SPARKLE as follows:

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

This property will automatically be transformed by SPARKLE. Because n is the first variable to occur in the property, the quantor over n will be added first. Quantification over type-variables is still omitted. This results in the following property, which will be the starting point of the proof:

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

5 Building the proof

In this section a proof of the example property in SPARKLE will be described. First, however, the reasoning style of SPARKLE has to be explained. An important feature of SPARKLE, the hint mechanism, is also introduced first.

5.1 Reasoning style in Sparkle

Reasoning in Sparkle is similar to reasoning in other theorem provers. The reasoning process 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 first element of this list is called the *current goal*; the others are called *subgoals*. Reasoning always takes place on the current goal. It is possible to exchange the current goal with one of the subgoals.

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.

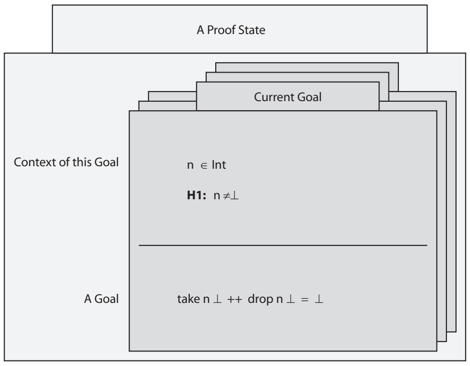

Fig. 3. A proof state

Sparkle implements a total of 42 tactics. The behavior of most of these tactics can be adjusted by means of parameters. Many of the tactics can also be found, or expressed, in other theorem provers. All, however, are specifically geared towards proving properties of functional programs and are tailored for usage by programmers. 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 $\perp$); (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). A more detailed description of these tactics can be found in the appendix.

5.2 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 examines the current goal and determines which tactics can be applied. Not all tactics (and variations of tactics) are checked, but the most important ones are. A probability score will be given to each applicable tactic. This score ranges from 1 to 100. The higher the score, the more likely the application of the tactic will help the proof. A score of 100 is reserved for tactics that prove the current goal in one step.

The determination of the probability scores depends on the current goal, but is otherwise hard-coded in the theorem prover. However, when a theorem has been proved, the user can instruct the hint mechanism to check for applications of it. In that case, the probability score is determined by the user.

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 good probability 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 very fast. This reduces the typing (or clicking) effort for making a proof considerably.
2. It is possible to set a threshold for automatic application. Each time a tactic is found that has a probability score higher than the threshold, it will be applied automatically. By setting a low threshold one can let the theorem prover attempt the proof on its own. A medium threshold can be used for semi-automated proving: if the theorem prover is certain about an applicable tactic it may apply it.

5.3 Proof of the example program

In this subsection a proof of the example property will be presented. This is a proof that can be built in SPARKLE. 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 multiple 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). 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.

$$\boxed{\dfrac{-}{\forall_{n\in\texttt{Int}}\forall_{xs\in[\texttt{a}]}[n \neq \bot \rightarrow \texttt{take}\ n\ xs\ \texttt{++}\ \texttt{drop}\ n\ xs = xs]}} \tag{1}$$

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 $\boxed{\texttt{Induction xs}}$. Three new goals(1.1,1.2,1.3) are created: one for the case that xs is $\perp$; one for the case that xs is [] and one for the case that xs is a non-empty list. Note that $\perp$ is treated as a constructor for all algebraic types; therefore induction creates three new goals instead of two.

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

It is best to introduce variables and hypothesis in the goal context as soon as possible, so that reasoning on the 'real goal' can commence. The only exception is a variable on which induction should be performed. In the current goal the variable n and the hypothesis $n \neq \perp$ can, however, safely be introduced. This is accomplished by the tactic $\boxed{\texttt{Introduce n H1}}$.

$$\boxed{\frac{\begin{array}{c} n \in \texttt{Int} \\ \textbf{H1: } n \neq \perp \end{array}}{\texttt{take } n \perp \texttt{ ++ drop } n \perp = \perp}} \quad (1.1')$$

Due to the strictness of `take` and `++`, redexes are present in the current goal. These can be reduced using the tactic $\boxed{\texttt{Reduce NF All}}$, which will 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.

$$\boxed{\frac{\begin{array}{c} n \in \texttt{Int} \\ \textbf{H1: } n \neq \perp \end{array}}{\perp = \perp}} \quad (1.1'')$$

This is clearly a trivial goal, because equality is a reflexive relation. Such reflexive equalities are proved immediately with the tactic $\boxed{\texttt{Reflexive}}$.

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

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

$$\boxed{\frac{\begin{array}{c} n \in \texttt{Int} \\ \textbf{H1: } n \neq \perp \end{array}}{\texttt{take } n \; [] \texttt{ ++ drop } n \; [] = []}} \quad (1.2')$$

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

$$\boxed{\begin{array}{c} n \in \texttt{Int} \\ \textbf{H1: } n \neq \bot \\ \hline [] = [] \end{array}} \quad (1.2'')$$

This is another example of a reflexive equality; therefore $\boxed{\texttt{Reflexive}}$.

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

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): $\boxed{\texttt{Introduce x xs IH m H1}}$.

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

In proofs by induction, the goal should be transformed so that the induction hypothesis can be applied. In this case, there is only a difference in the list argument of `take` and `drop`: in the induction hypothesis it is xs, but in the goal it is $[x{:}xs]$. This difference can be overcome by means of a simple reduction: $\boxed{\texttt{Reduce NF All}}$. 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.

$$\boxed{\begin{array}{c} x \in \texttt{a}, xs \in [\texttt{a}], m \in \texttt{Int} \\ \textbf{IH: } \forall_{n \in \texttt{Int}} [n \neq \bot \rightarrow \texttt{take } n \; xs \texttt{ ++ } \texttt{drop } n \; xs = xs] \\ \textbf{H1: } m \neq \bot \\ \hline \left(\begin{array}{l} \text{case } (0 < m) \text{ of} \\ \quad \texttt{True} \quad \rightarrow [x{:}\texttt{take } (m\text{-}1) \texttt{ xs}] \\ \quad \texttt{default} \rightarrow [] \end{array} \right) \texttt{++} \left(\begin{array}{l} \text{case } (0 < m) \text{ of} \\ \quad \texttt{True} \quad \rightarrow \texttt{drop } (m\text{-}1) \texttt{ xs} \\ \quad \texttt{default} \rightarrow [x{:}xs] \end{array} \right) = [x{:}xs] \end{array}}$$

(This proof state is also shown in Fig. 4.)

The natural next step is a case distinction on $0 < m$, because that will allow the reduction of both case-expressions in the current goal. A special tactic is used for this purpose: $\boxed{\texttt{SplitCase 1}}$. This tactic will examine the first case-expression in the current goal. Three cases are distinguished: (1) $\bot$ (for when $0 < m$ 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.

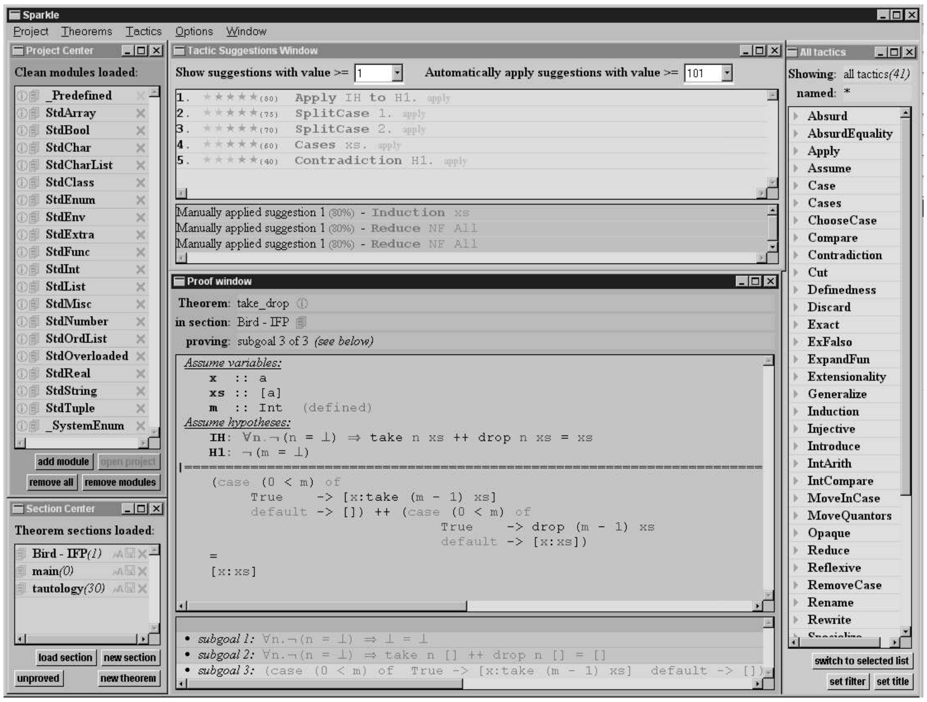

Fig. 4. The theorem prover in action

$$\boxed{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \bot \\ \hline \bot\ \mathtt{++}\ \bot = [x{:}xs] \end{array}} \qquad (1.3.1)$$

This is the goal created by `SplitCase 1` for the case that $0 < m = \bot$. This goal can be proved in one step, because hypotheses H1 and H2 are contradictory. This is due to a special property of the function $<$: the result of $x < y$ can only be $\bot$ if either $x = \bot$ or $y = \bot$. Hypothesis H2 states that $0 < m = \bot$, thus either $0 = \bot$ or $m = \bot$. Of course, $0 = \bot$ 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 hyper-strictness of functions as $-$ and $<$. If an expression is found which is both be defined and undefined, the goal is proved immediately. Note that Sparkle will always automatically detect if such a contradiction can be found in the current goal and will report that to the user.

$$\begin{array}{|c|}\hline x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{True} \\ \hline [x{:}\mathtt{take}\ (m\text{-}1)\ xs]\ \mathtt{++}\ \mathtt{drop}\ (m\text{-}1)\ xs = [x{:}xs] \\ \hline \end{array} \quad (1.3.2)$$

This is the goal created by `SplitCase 1` for the case that $0 < m =$ `True`. The first case-expression has been replaced by a 'cons', making it possible to reduce the `++`. Therefore: $\boxed{\texttt{Reduce NF All}}$.

$$\begin{array}{|c|}\hline x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{True} \\ \hline [x{:}\mathtt{take}\ (m\text{-}1)\ xs\ \mathtt{++}\ \mathtt{drop}\ (m\text{-}1)\ xs] = [x{:}xs] \\ \hline \end{array} \quad (1.3.2')$$

In this goal it is finally possible to use the induction hypothesis. When $m - 1$ is substituted for n and the condition $m - 1 \neq \bot$ is satisfied, the induction hypothesis can be used to substitute `take` $(m\text{-}1)$ xs `++` `drop` $(m\text{-}1)$ xs by xs in the current goal. This is accomplished in Sparkle by an application of the backwards tactic `Rewrite`, to be precise by $\boxed{\texttt{Rewrite IH}}$. Using this tactic, the mentioned substitution is applied immediately, which leaves the trivial goal 1.3.2.1 to be proven. An additional goal (1.3.2.2) is created in which the condition $m - 1 \neq \bot$ must be proved.

$$\begin{array}{|c|}\hline x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{True} \\ \hline [x{:}xs] = [x{:}xs] \\ \hline \end{array} \quad (1.3.2.1)$$

This trivial goal is proved immediately by $\boxed{\texttt{Reflexive}}$.

$$\begin{array}{|c|}\hline x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{True} \\ \hline (m - 1) \neq \bot \\ \hline \end{array} \quad (1.3.2.2)$$

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 $\boxed{\texttt{Contradiction}}$, which creates a hypothesis that is the negation of the current goal and replaces the current goal by the proposition $False$.

$$\boxed{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n \in \mathtt{Int}}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{True} \\ \textbf{H3: } (m - 1) = \bot \\ \hline False \end{array}} \quad (1.3.2.2')$$

This goal is similar to goal 1.3.1. Here, the same reasoning can be applied using hypothesis H3, because $-$ has the same property as $<$. Therefore, hypothesis H3 implies that m must be equal to $\bot$, but hypothesis H1 states the opposite. Again, the application of $\boxed{\mathtt{Definedness}}$ will use this contradiction and prove the goal immediately.

$$\boxed{\begin{array}{c} x \in \mathtt{a}, xs \in [\mathtt{a}], m \in \mathtt{Int} \\ \textbf{IH: } \forall_{n}[n \neq \bot \rightarrow \mathtt{take}\ n\ xs\ \mathtt{++}\ \mathtt{drop}\ n\ xs = xs] \\ \textbf{H1: } m \neq \bot \\ \textbf{H2: } (0 < m) = \mathtt{False} \\ \hline [\,]\ \mathtt{++}\ [x{:}xs] = [x{:}xs] \end{array}} \quad (1.3.3)$$

This final goal is the third goal created by `SplitCase 1`, for the default case. It is, however, clearly a trivial goal, which can be proved by a reduction followed by an application of `Reflexive`. These two actions can be combined by $\boxed{\mathtt{Reduce\ NF\ All;\ Reflexive}}$, which finishes the proof!

5.4 Remarks concerning the reasoning process

The proof presented was not difficult to build. The decision which tactic to use next was always motivated by an examination of the current goal; no overview of the proof as a whole was required. This actually turns out to be the case for many small proofs about functional programs. With good support, most notably a well-designed user interface and a good hint mechanism, these proofs can be built with minimal effort.

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 the right hint. This is a lot easier than selecting the right tactic, because there are less options to choose from. Right now, there are 41 different tactics which can have arguments that change the behavior 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 succesfully. Further improving the hint mechanism will be one of the speerheads in the further development of SPARKLE.

A proof of (almost) the same property is also presented in [3]. The proof presented there only takes positive integer arguments into account. It was given by an induction on the integer argument, while the proof in this paper is given by induction on the list argument. This is only a minor difference and the proofs are actually 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 been translated to SPARKLE without difficulties. No problems are expected for translating the others.

6 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 actions to users, reducing the required expertise even further. All in all, a programmer can build this proof in a short time and without many difficulties.

The two features of SPARKLE that attribute 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. Most importantly, the user guidance must be improved by adding documentation to the system and by researching possibilities to improve on the hint suggestion mechanism.

Also, work needs to be done on the formal framework of the theorem prover. This framework incorporates the reduction semantics of CLEAN and defines a semantics for properties about CLEAN-expressions. It defines equality as a bisimulation and covers both finite and potentially infinite structures. The effect of the tactics is already formally described in this framework, but the soundness of these effects with respect to the semantics must still be proved. Of particular importance is the soundness of `Induction` for all lazy structures.

7 Related work

Widely used generic theorem provers are PVS[8], COQ[4] and ISABELLE[9]. 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.

Somewhat closer related work is described in [6], in which a description is given of a proof tool which is dedicated to HASKELL[10]. 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.

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. The Coq Development Team. *The Coq Proof Assistant Reference Manual (version 7.0)*, Inria, 1998. `http://pauillac.inria.fr/coq/doc/main.html`
5. 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`
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. 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.
8. 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`
9. L. C. Paulson. *The Isabelle Reference Manual*, Cambridge, 2001. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`
10. S. Peyton Jones(editor), J. Hughes(editor) et al. *Report on the programming language Haskell 98*, Yale, 1999. `http://www.haskell.org/definition/`
11. 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. The description of a tactic consists of the following parts:

Type: A tactic can be categorized in three ways:

Safe vs. Unsafe - A tactic is safe if the reverse action is also a valid tactic. This can only be the case if the created goals are logically equivalent to the original goal. An unsafe tactic creates goals which are logically stronger than the original goal.

Forwards vs. 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). A forwards tactic only changes the hypotheses, while a backwards tactic always transforms the current goal.

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 vs. Logic - Programming tactics act on expressions and are based on the semantics of CLEAN; logic tactics implement logical reasoning steps.

Syntax: The syntax is used to describe the parameters. Some parameters are fixed in order to simplify the description.

Info: Describes the tactic itself.

Effect: Describes the effect of the tactic on a goal.

Example: Gives an example application of the tactic.

A.1 Contradiction

Type: Safe; backwards; logic.
Syntax: `Contradiction.`
Info: Builds a proof by contradiction.
Effect: Replaces the current goal by the absurd proposition $False$ and add its negation as a hypothesis in the context. If a double negation is produced, it will be removed automatically.

Example: $xs \vdash xs \texttt{ ++ } [] \neq xs$

$\Longrightarrow$ `Contradiction` $\Longrightarrow$

(1) $xs, \langle xs \texttt{ ++ } [] = []\rangle \vdash False$

A.2 Definedness

Type: Instantaneous; logic.
Syntax: `Definedness.`
Info: Determines two sets of expressions: (1) the set of defined expressions, which are expressions that are unequal to $\perp$; (2) the set of undefined expressions, which are expressions that are equal to $\perp$. These sets are determined by examining equalities in hypotheses and using strictness information. In addition, the *hyper-strictness* of certain functions is used. A function `f` is hyper-strict if the application of `f` is only equal to $\perp$ if one of its arguments is equal to $\perp$.
Effect: If an expression is found that is both defined and undefined, the goal is proved instantaneously. Otherwise nothing happens.

Example: $xs, ys, zs, \langle xs = \perp\rangle, \langle xs \texttt{ ++ } ys = [1{:}zs]\rangle \vdash False$

$\Longrightarrow$ `Definedness` $\Longrightarrow$

$\square$

A.3 Induction

Type: Unsafe; backwards; programming.
Syntax: `Induction <variable>.`
Info: Performs structural induction on a variable. A goal is created for each root-normal-form the variable may have, including $\perp$. For recursive variables induction hypotheses are created. The root-normal-forms a variable may have are determined by its type. Induction is most commonly used for variables of an algebraic type. The root-normal-forms in this case are applications of its constructors.
Effect: The variable is replaced by its root-normal-form in the current goal and universal quantors are added for new variables. Additionally, induction hypotheses are added as implications in the current goal for recursive variables. This process is repeated for each root-normal-form examined.
Example: $\vdash \forall_{xs}[xs \texttt{ ++ } [] = xs]$

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

A.4 Introduce

Type: Safe; backwards; logic.
Syntax: `Introduce <name1> <name2> ... <namen>.`
Info: Moves as many universally quantified variables and hypotheses to the 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 $x_1 \ldots x_a$ are introduced in the context, using the names from the list. The hypotheses $P_1 \ldots P_b$ are also introduced in the context, again using the names from the list.
Example: $\vdash \forall_x[x = 7 \rightarrow \forall_y[y = 7 \rightarrow x = y]]$

$\Longrightarrow$ `Introduce p H1 q H2` $\Longrightarrow$
(1) $p, q, \langle \texttt{H1: } p = 7\rangle, \langle \texttt{H2: } q = 7\rangle \vdash p = q$

A.5 Reduce

Type: Safe; backwards; programming.
Syntax: `Reduce NF All.`
Info: Reduces all expressions in the current goal to normal form. An eager reduction mechanism is used, but there is a limit to the number of reductions allowed. Replacing the `NF` by `RNF` results in lazy reduction to root-normal-form.
Effect: All redexes are replaced by their reducts.
Example: $x, xs, ys, zs \vdash [x{:}xs] \texttt{ ++ } ys = \texttt{reverse } zs$

$\Longrightarrow$ `Reduce NF All` $\Longrightarrow$
(1) $x, xs, ys, zs \vdash [x{:}\ xs \texttt{ ++ } ys] = \texttt{reverse } zs$

A.6 Reflexive

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

A.7 Rewrite

Type: Safe; backwards/forwards; logic.
Syntax: `Rewrite <hypothesis>.`
Info: Rewrites the current goal using an equality in a hypothesis.
Effect: The hypothesis must be of the form $\forall_{x_1} \ldots \forall_{x_n}[E_1 = E_2]$. 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.
Example: xs, $\langle \texttt{H1:}\ xs = [] \rangle \vdash xs \texttt{ ++ } xs = xs$
$\Longrightarrow$ `Rewrite H1` $\Longrightarrow$
(1) xs, $\langle \texttt{H1:}\ xs = [] \rangle \vdash [] \texttt{ ++ } [] = []$

A.8 SplitCase

Type: Unsafe; backwards; programming.
Syntax: `SplitCase <num>.`
Info: Performs a case distinction based on a case-expression in the current goal. Each pattern, is transformed to a case. Two cases are always created: one for errors when evaluating the case-expression and one for the case that no pattern matches (replaced by the default pattern if one is available).
Effect: A goal is created for each case. In each goal, the case-expression is replaced by the result of the pattern chosen (or $\perp$ for the erroneous case). Hypotheses are introduced in the context to reflect which case was chosen. The case for the default alternative is optimized: instead of negating all other alternatives, an investigation of the remaining possibilities is used. If another equivalent case-expression is present in the current goal, it is filled in as well.
Example: $xs \vdash (\texttt{case}\ xs\ \texttt{of}\ [y{:}ys] \rightarrow y;\ \texttt{default} \rightarrow 12) = 0$
$\Longrightarrow$ `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, $\langle xs = [], \vdash 12 = 0$