Proving Properties of Lazy Functional Programs with SPARKLE

Maarten de Mol, Marko van Eekelen, and Rinus Plasmeijer

Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
{maartenm,marko,rinus}@cs.ru.nl

Abstract. This tutorial paper aims to provide the necessary expertise for working with the proof assistant SPARKLE, which is dedicated to the lazy functional programming language CLEAN. The purpose of a proof assistant is to use formal reasoning to verify the correctness of a computer program. Formal reasoning is very powerful, but is unfortunately also difficult to carry out.

Due to their mathematical nature, functional programming languages are well suited for formal reasoning. Moreover, SPARKLE offers specialized support for reasoning about CLEAN, and is integrated into its official development environment. These factors make SPARKLE a proof assistant that is relatively easy to use.

This paper provides both theoretical background for formal reasoning, and detailed information about using SPARKLE in practice. Special attention will be given to specific aspects that arise due to lazy evaluation and due to the existence of strictness annotations. Several assignments are included in the text, which provide hands-on experience with SPARKLE.

1 Introduction

In 2001, the distribution of the lazy functional programming language CLEAN [5,28,29] was extended with the dedicated proof assistant SPARKLE [11]. The purpose of a proof assistant is to verify the correctness of a computer program without executing it. This is accomplished by means of the mathematical process of formal reasoning, which makes use of the source code of the program and the semantics of the programming language.

SPARKLE is intended as an additional tool for the CLEAN-programmer and aims to make formal reasoning accessible. It is conveniently integrated into the official Development Environment of CLEAN, allows reasoning on the level of the programming language itself and offers dedicated support for dealing with CLEAN-programs. Unfortunately, formal reasoning is a complex mathematical process that requires specialized expertise. Therefore, it is often still difficult to carry out, even in dedicated proof assistants such as SPARKLE.

In practice, SPARKLE has already been applied for various purposes. It has been used for proving properties of I/O-programs by Butterfield [7] and Dowse [14]. In [30,19], Tejfel, Horváth and Kozsik have proposed an extension for it for

Z. Horváth et al. (Eds.): CEFP 2007, LNCS 5161, pp. 41–86, 2008.

dealing with temporal properties. Support for class-generic properties has been added to it by van Kesteren [21]. Furthermore, it has also been used in education at the Radboud University of Nijmegen.

The purpose of this paper is to provide the information that is necessary for functional programmers to start making use of Sparkle. A combination of both theoretical and practical expertise will be provided. No special knowledge is required to understand the contents of this paper: a basic understanding of lazy functional languages and elementary logic suffices. Upon completion of this paper, the reader will be able to use Sparkle to prove basic properties of small Clean-programs with minimal effort. Furthermore, a solid foundation will be laid for proving properties that are more complex.

This paper is structured as follows. First, the concept of formal reasoning will be explained independently of Sparkle in Section 2. Then, the important design principles of Sparkle will be summarized in Section 3, and their effect on the way that formal reasoning is implemented will be explained. Then, in Sections 4 and 5 a tutorial of the use of Sparkle in practice is presented. The first part (Section 4) presents a step-by-step introduction of all the basic features of Sparkle; the second part (Section 5) describes several advanced features that are specific for Sparkle. We discuss related work in Section 6 and draw conclusions in Section 7. Finally, the complete tactic library of Sparkle is summarized separately in Appendix A.

The tutorial is written in an explanatory style and contains assignments with which the provided theory can be put into practice. The assignments require the standard Clean 2.2 distribution to be installed, and the Sparkle version from http://www.cs.ru.nl/~marko/research/sparkle/SparkleCEFP2007.zip must be merged in it. The worked out answers to the assignments are available online at http://www.cs.ru.nl/~marko/research/sparkle/cefp2007/.

2 Formal Reasoning

In the following sections, a general introduction to formal reasoning will be presented independently of Sparkle. In Section 2.1, formal reasoning will first be described as an abstract process that transforms input to desired output. In Section 2.2, the underlying formal framework will be identified; this framework is a prerequisite for carrying out formal reasoning. The most important component of the framework is the proof language, which will be explored in more detail in Section 2.3. Finally, the soundness of formal reasoning will be discussed in Section 2.4.

2.1 The Abstract Process of Formal Reasoning

Formal reasoning is a mathematical process that fully takes place on the formal level. The goal of formal reasoning is to verify the correctness of some kind of formal object by means of reasoning about it. The process as a whole can roughly be characterized as follows:

1. Formalize an object o;
2. Formalize a property p that says something about o;
3. Build a formal proof that shows that p holds for o.

If formal reasoning succeeds and a formal proof is built, then it is shown with absolute certainty that the formalized object o behaves as specified by means of property p. This holds for all environments in which o may occur, because the formal proof is obliged to take all possible circumstances into account. As such, a positive result of formal reasoning is more powerful than for instance a positive result of testing, which is restricted by the test-set that was used.

If formal reasoning does not succeed in building a proof, however, then not much information has been gained. It may either be the case that o is incorrect, or it may be the case that the desired behavior of o was incorrectly specified by p, or it may simply be the case that the proof builder did not build the proof in the right way. A negative result of formal reasoning is hard to interpret correctly and is therefore less useful than a negative result of for instance testing.

2.2 Formal Framework

Formal reasoning makes use of the formal representations of the object to reason about (input), the property to prove (input) and the proof to be built (output). Moreover, to ascertain the soundness of reasoning (see Section 2.4), a formal semantics that assigns a meaning to properties must be available as well. The combination of these prerequisites of reasoning will be called a formal framework:

Definition 2.2.1. *(Formal framework)*
A formal framework is a tuple $(O, P, \models_o, \vdash_o)$ such that:

- O is the set that contains all possible objects to reason about;
  ⇢ ($o \in O$ denotes that o is a valid object to reason about)
- P is the set that contains all possible properties that may be specified;
  ⇢ ($p \in P$ denotes that p is a valid property to prove)
- $\models_o$ is the relation that defines the semantics of properties;
  ⇢ ($\models_o p$ denotes that $p \in P$ holds in the context of $o \in O$)
- $\vdash_o$ is the derivation system that defines proofs of properties.
  ⇢ ($\vdash_o p$ denotes that a proof of $p \in P$ exists in the context of $o \in O$)

(The formal framework of SPARKLE is described completely in [13]. In the remainder of this paper, it will be treated implicitly only.)

Note that the elements of a framework are connected: it must be possible to refer to components of objects within properties; the semantics of a property can only be determined in the context of a given object; and the derivation of a proof depends on a given object as well.

Using the notations introduced by the formal framework, formal reasoning can now be characterized as follows:

Definition 2.2.2. *(Formal reasoning)*
Formal reasoning is the process that given a formal framework $(O, P, \models_o, \vdash_o)$, a specific object $o \in O$ and a specific property $p \in P$, attempts to determine

whether $\vdash_o p$ holds or not. From the soundness of the formal framework it then follows that $\models_o p$ holds as well.

In other words, the goal of formal reasoning is to determine $\models_o$ by means of $\vdash_o$. This approach only makes sense for frameworks in which $\vdash_o$ is less complicated than $\models_o$, which is often the case, because derivation systems are usually simpler than semantic relations.

2.3 Proof Language

The most important component of the formal framework is the proof language, which is usually represented by means of a derivation system. The derivation rules of this system are reasoning steps that form the building blocks of proofs. Building proofs is basically the repeated application of these reasoning steps, and can be characterized as follows:

- **Goal**: prove a property p.
- **Apply**: reasoning step R. This transforms p to $p_1, \ldots, p_n$. If $n = 0$, then the proof is complete (R proves p). Otherwise, $p_1, \ldots, p_n$ become the new goals which *all* have to be proved recursively by the same reasoning process.
- **Goal**: prove all properties $p_1, \ldots, p_n$.

In other words, reasoning steps are functions that transform propositions into (possibly more) propositions, and the proof language is the set of functions that one is allowed to apply during reasoning. Furthermore, reasoning itself is 'goal-busting': at each point in time a number of propositions (goals) have to be proved, and these propositions can be simplified (busted) by means of the repeated application of predefined reasoning steps.

The result of reasoning is a derivation tree in which the nodes are propositions (and the root node is the initial proposition to prove) and each set of edges leading from a single node corresponds with a reasoning step. Edges in this tree do not necessarily have to lead to another node, because reasoning steps may produce the empty list of propositions. The leaves of the tree are the propositions that still have to be proved.

The derivation tree is of course the formal representation of a proof. It can easily be serialized, provided that the reasoning steps are named. A serialized proof can be transferred to anyone with knowledge of the formal framework that it uses. Furthermore, the receiver can even automatically check the validity of the proof by re-running it. Note that validating proofs is easy, because it only requires the formal framework, but building proofs is difficult, because it requires the continuous selection of the 'right' reasoning step.

2.4 Soundness of Formal Reasoning

Building formal proofs is an exercise in the repeated simplification of propositions according to predefined reasoning steps. This, however, is a purely syntactic exercise that does not take the actual meaning of propositions into account in any way. In order for the results of reasoning to be meaningful, the underlying formal framework must be sound as well:

Definition 2.4.1. *(Soundness of formal frameworks (1))*
A formal framework $(O, P, \models_o, \vdash_o)$ is sound if for all $o \in O$ and $p \in P$ it holds that $\vdash_o p$ implies $\models_o p$.

Because $\vdash_o$ is composed of individual derivation rules, the soundness of a formal framework as a whole can be determined by verifying these rules as follows:

Definition 2.4.2. *(Soundness of a derivation rule)*
A derivation rule $R \in\vdash_o$ is sound if for all $p \in P$ it holds that $\models_o (p_1 \wedge \ldots \wedge p_n)$ implies $\models_o p$, assuming that $R(p) = p_1, \ldots, p_n$.

Definition 2.4.3. *(Soundness of formal frameworks (2))*
A formal framework $(O, P, \models_o, \vdash_o)$ is sound if all its derivation rules $R \in\vdash_o$ are sound.

Formal reasoning only makes sense if the underlying formal framework is sound. Soundness should therefore preferably be proved explicitly. If the complexity of the derivation system makes this too difficult, then some degree of confidence can still be gained from practice ('no untrue propositions have ever been proved, so it must be correct'), but this weakens the results of formal reasoning considerably. The soundness of the formal framework of SPARKLE has been proved in [13].

Finally, note that for the usefulness of formal reasoning it is important that the reverse property of completeness (for all properties p, $\models_o p$ implies $\vdash_o p$) holds too. Full completeness is extremely difficult to achieve for complex frameworks. Using proof theory, however, it can usually be approximated quite closely.

3 Design Principles of SPARKLE

The main purpose of SPARKLE is to allow functional programmers to reason about the CLEAN-programs that they are developing, which improves the quality of the program as a whole. The reasoning support that SPARKLE offers is in the first place tailored towards this main purpose, although in general SPARKLE is also useable for anyone who would like to reason about functional programs. In particular, a frontend for HASKELL'98 is currently being added to CLEAN, which in the future would allow reasoning about mixed CLEAN /HASKELL-programs.

In the following sections, the effect that the main purpose of SPARKLE has on its design will be explored closely. In Section 3.1, first the intended users of SPARKLE will be analyzed in detail. Then, in Section 3.2 a list of resulting consequences for the design will be presented. Finally, the important consequence of dedicated reasoning will be explored in detail in Sections 3.3 and 3.4.

3.1 Intended Users: Functional Programmers

The intended users of Sparkle are functional programmers, or more specifically anyone who has downloaded the Clean-distribution and is developing programs with it. Of course, there is much diversity in this group, and there is no such thing as 'the functional programmer'. Still, for the sake of design, we will make the following tentative assumptions about the intended users of Sparkle:

- They do not necessarily have much experience with formal reasoning, and may not even know about it at all;
- They often have some theoretical background, and usually have at least a basic understanding of elementary logic;
- They usually have good knowledge of functional programming in general and of Clean (and its semantics) in specific;
- They are not necessarily aware of the benefits of formal reasoning for the purpose of improving the quality of software;
- They are mainly interested in the programs that they develop.

Other proof assistants may be geared towards different users; for instance, the major independent proof assistants (such as for instance Pvs [24] and Coq [31]) are mainly intended for logicians who already know about formal reasoning and are interested in it as well.

3.2 Design Choices

Sparkle implements a theoretically sound formal framework, and therefore fully supports general formal reasoning on the fundamental level. In its design, however, Sparkle focuses mainly on functional programmers as its intended users. The most important choices in the design of Sparkle are:

- The *object language* should be Clean, because this allows programmers to reason on the level of the programming language, which is their area of expertise. Although this has not been realized fully, a good approximation by means of Core-Clean has been adopted by Sparkle (see Section 3.4).
- For the *property language*, it suffices to use a standard first-order logic which has been extended with an equality on arbitrary program expressions. In such a logic most common properties can be expressed easily. Moreover, functional programmers are likely to be capable of handling standard first-order logic. The property language will be introduced in the tutorial in Section 4.3.
- The *semantics* of the property language should conform to the semantics of Clean. This ensures that properties that are proved with Sparkle hold for the real-world Clean-program as well. This is achieved by giving '$e_1 = e_2$' the meaning 'it is possible to interchange e_1 with e_2 in any program without changing its observational behavior'. The full semantics will be introduced on an informal level in the tutorial in Section 4.4.
- Formal reasoning should be *integrated* with programming, such that switching between the two activities becomes easy. This makes formal reasoning

more attractive, because it is linked to an activity that is carried out continuously. The integration of SPARKLE is realized by allowing it to be started directly from the IDE (Integrated Development Environment) of CLEAN, in which case the current project is loaded automatically in SPARKLE.
- The reasoning steps of SPARKLE should be *specialized* for dealing with lazy functional programs in general, and for dealing with CLEAN in specific. In particular, lazy evaluation and explicit strictness have a profound influence on semantics, and therefore on reasoning as well. The specialized features of SPARKLE will be described in Section 5.
- The *first impression* of SPARKLE should be positive, and should entice programmers to continue with formal reasoning. This is realized by SPARKLE's attractive user interface (see tutorial), and by allowing small proofs to be carried out automatically with the hint mechanism (see Section 4.5).
- SPARKLE should have up-to-date and extensive documentation. This paper is the first attempt to achieve this goal.

The design choices with the most profound influence on SPARKLE are the level of the object language and the specialization of the reasoning steps. The consequences of the level of the object language will be examined further in Sections 3.3 and 3.4; the specialized features of SPARKLE will be described in detail later in Section 5.

3.3 Dedicated vs General-Purpose Formal Reasoning

If one wants to add support for formal reasoning to a specific programming language, two different approaches can be taken:

1. Build one's own *dedicated* proof assistant that directly supports reasoning on the level of the programming language itself; or
2. Build a shell around an existing *general-purpose* proof assistant, combined with a translation mechanism to and from its object language.

Currently, several good general-purpose proof assistants are available in practice, such as for instance PVS [24], COQ [31] and ISABELLE [26]. These proof assistants all have a large user base and make use of well-developed formal frameworks that are extremely expressive and powerful. In the shell approach, such a well-established formal framework is re-used automatically, which is a major advantage.

Unfortunately, general-purpose proof assistants have a major disadvantage as well: none have an object language that fully supports the semantics of CLEAN, which is based on lazy graph-evaluation with explicit strictness. Therefore, the evaluation mechanism of the proof assistant cannot be re-used, and an interpreter for CLEAN has to be built completely within the object language of the general-purpose proof assistant. This has the following important drawback:

> *actual reasoning no longer takes place on the level of the* CLEAN*-program, but instead on a meta-representation of it in the object language of the general-purpose proof assistant*

From the programmer's point of view, however, it is crucial that reasoning at least *appears* to be taking place on the level of the Clean-program. In the case that a general-purpose proof assistant is used, it is therefore the task of the shell to hide the underlying meta-level completely from the end user. Consequently, applying a reasoning step in a shell actually requires three activities: (1) translate the program and the reasoning step to the meta-level; (2) execute the reasoning step on the meta-level; (3) translate the feedback back to the programming level.

To summarize, the shell approach has the advantage that a well-established formal framework is re-used, but the disadvantage that an interpreter and a two-way translation and communication mechanism have to be realized. We feel that the general-purpose approach poses more practical problems than it offers advantages; therefore, we have chosen to make use of the dedicated approach.

In hindsight, Sparkle has been the result of only about 18 'man-months' of work, which shows that writing one's own dedicated proof assistant is certainly doable. We estimate that writing a shell would have taken considerably more effort. On the other hand, the formal framework of Sparkle does lack some expressiveness, but this has turned out to be only a slight disadvantage for reasoning about functional programs.

3.4 Sparkle's Approximation of Dedicated Reasoning

Sparkle is a dedicated proof assistant and aims to support formal reasoning on the level of the programming language itself. For this purpose it allows reasoning on the level of Clean, but with the following restrictions:

- All uniqueness annotations are removed automatically from the program;
- I/O-operations have no semantic model and are meaningless;
- Overflow and rounding is disregarded;
- Programs are syntactically simplified to an essential subset before reasoning.

Due to the first restriction, it is not possible in Sparkle to specify properties that make use of uniqueness. Programs with uniqueness, however, can still be loaded: the uniqueness check is first performed as usual, and then the uniqueness annotations are simply removed. Due to the second restriction, it is not possible to use Sparkle for proving properties of I/O. Due to the third restriction, many laws about numbers (such as for instance $\forall_n . n < n+1$) hold in Sparkle, but are falsified by programs in which overflow/rounding occurs. Adding user-friendly support for uniqueness, I/O, overflow and rounding is still future work.

The fourth restriction differs from the first three. Firstly, it does not restrict the scope of reasoning, because it allows all programs to be simplified without loss of meaning. Secondly, it always has an influence on reasoning, because *every* program is simplified implicitly. Thirdly, it is almost impossible to avoid, because defining reasoning support (both on the theoretical and on the practical level) for all of the many syntactic constructs of Clean is practically undoable.

The simplification of programs is performed automatically by Sparkle for all programs that are loaded. The target of the simplification is Core-Clean, which is the intermediate language of the Clean-compiler. From the user's point of view, it seems that Sparkle operates on the level of Clean, but reasoning actually takes place on the level of Core-Clean. Still, the level of Core-Clean approximates dedicated reasoning very well, because:

- Core-Clean has the *same expressive power* as Clean.

  Without loss of meaning, any Clean-program can be transformed to an equivalent Core-program, on which reasoning with Sparkle is possible. Furthermore, the transformation itself has already been implemented in the actual Clean-compiler. Because both Sparkle and the compiler are written in Clean, the existing transformation can be re-used. This not only saves a lot of time, but also ensures soundness of the transformation.
- Core-Clean is a *subset* of Clean.

  Programs in Core-Clean can easily be understood by Clean-programmers, because they make use of the syntax and semantics of Clean. Understanding the program to reason about is vital for the success of formal reasoning.
- Programs in Core-Clean are *very similar* to their Clean-originals.

  The changes between the Core-program and the Clean-original are mainly syntactical in nature, and can in many cases even be hidden by Sparkle. Moreover, the structure of the program is unchanged. As a result, much of the programmer's expertise of the source program is still valid on the Core-Clean level. Again, this increases the understanding of the program to reason about.

Of the four restrictions, the lack of support for dealing with I/O is the most significant, as I/O is an important component of many programs and one would like to reason about it. On the other hand, the usefulness of properties that make use of uniqueness still has to be established, and rounding and overflow are not an issue for the majority of programs. Furthermore, Core-Clean is a suitable intermediate reasoning level.

The differences between Core-Clean and Clean, as well as the feature of Sparkle to present Core-programs as if they were Clean-programs, will be explained further in the Tutorial in Section 4.1.

4 Tutorial Part I: Getting Started with Sparkle

In the following sections, a step-by-step introduction of the basic functionality of Sparkle will be presented. The introduction covers the user interface, the specification of programs and properties, the semantics, and the three different supported styles of reasoning. At various places assignments are included, with the purpose of giving the reader the opportunity to gain hands-on experience with the Sparkle proof assistant.

The tutorial will be continued in Section 5, in which the specialized features of Sparkle will be described. A summary of all available reasoning steps is given in Appendix A.

In order to carry out the assignments in the tutorial, the standard Clean 2.2 distribution (available at http://clean.cs.ru.nl) must be installed, and the files from http://www.cs.ru.nl/~marko/research/sparkle/SparkleCEFP2007.zip must be merged in it. This additional package contains both a full version of Sparkle, and the used example programs `undefined` and `primes` (which will be placed in the `Examples\CEFP` folder of the Clean distribution). Note that Sparkle is available for Windows only. The answers to the assignments are available at http://www.cs.ru.nl/~marko/research/sparkle/cefp2007/.

4.1 Loading a Program

The first step of formal reasoning with Sparkle is loading a Clean-program into its memory. This program provides the context information that is required for stating and proving properties. The fastest way of starting Sparkle and loading a program is by means of the standard IDE of Clean, in which access to Sparkle has been integrated:

Assignment 1. *(Loading a program into* Sparkle *automatically)*
(a) Open the Clean-project `primes.prj` in the `Examples\CEFP` folder.
(b) Examine the code of the main module (`primes.icl`) and attempt to predict the behavior of the program. Then, compile and run the program.
(c) Find the `Theorem Prover Project` option and use it to launch Sparkle.

Alternatively, programs (and individual modules) can also be loaded from within Sparkle, either by opening entire projects (Ctrl-O), or by opening the standard environment only (Ctrl-E), or by adding single modules (Ctrl-+).

Internally, Sparkle maintains its own representation of the program. In this representation, a program is simply considered to be a list of (interdependent) modules, and each module is considered to be a list of definitions. Sparkle does not distinguish between the definition (`.dcl`) and implementation (`.icl`) parts of a module and allows access to all components of a program at any time.

$$\begin{aligned} \mathit{Program} &:== \mathit{Module}^* \\ \mathit{Module} &:== \mathit{Definition}^* \\ \mathit{Definition} &:== \mathit{Algebraic\ Type} \mid \mathit{Record\ Type} \mid \mathit{Function} \mid \mathit{Class} \mid \mathit{Instance} \end{aligned}$$

Sparkle has a powerful graphical user interface that allows the structure of the loaded program to be inspected in detail:

Assignment 2. *(Browsing through the program structure)*
(a) Find the window that displays the list of modules that are currently loaded. In this list, find the `primes` module and open it.
(b) The opened window actually filters *all* available definitions with the formula 'functions from the `primes` module'. Change the filter to find all functions in `StdList` and `StdFunc` that begin with the letter 's'.

The user interface also allows each individual definition of the loaded program to be displayed in a separate window. Furthermore, these definition windows are interconnected by means of the symbols that are used within it:

Assignment 3. *(Browsing through the program components)*
(a) Open the definition of the function `isPrime` in the `primes` module.
(b) Follow the internal link to the `canBeDividedByAny` function.
(c) Follow the internal link to the predefined `rem` function.

SPARKLE is a *dedicated* proof assistant that aims to support reasoning on the level of the programming language. Unfortunately, reasoning on the level of CLEAN is not practical, because of the many different syntactical constructs that are allowed. Therefore SPARKLE uses CORE-CLEAN, which is basically the subset of CLEAN in which all syntactic sugar has been removed, as intermediate reasoning language. The only remaining definitions in CORE-CLEAN are algebraic types and global functions, and expressions may only be constructed by means of applications, case distinction and lets.

Even though CORE-CLEAN is a small language only, all CLEAN-programs can be represented in it. When a CLEAN-program is loaded into SPARKLE, it is always automatically converted to CORE-CLEAN. As a result, the program in the memory of SPARKLE differs from the original CLEAN version. Some important differences between the CLEAN-program and its CORE-CLEAN-equivalent are:

- All local functions have been lifted to the global level;
- All pattern matches have been transformed to case distinctions;
- All sharing has been expressed by means of recursive lets;
- All overloading has been expressed by means of dictionaries;
- All synonym types and macro's have been expanded fully;
- All list comprehensions and dot-dot-expressions have been transformed to function applications.

Fortunately, the differences between the internally loaded CORE-CLEAN program and the original CLEAN version only have a slight effect on reasoning, and are therefore hardly noticeable most of the time. Furthermore, the user interface of SPARKLE is able to optionally display parts of CORE-CLEAN programs in the syntax of their original CLEAN versions:

Assignment 4. *(Effect of the optional display options)*
(a) Open the function definitions `isPrime` and `canBeDividedByAny` from the `primes` module and `span` from the `StdList` module.
(b) Toggle the display options `Pattern Matching` and `Case/Let vs #/!`. The 'real' CORE-CLEAN program is displayed when the options are toggled off.
(c) There is one difference between the internal version of `isPrime` and the CLEAN version that cannot be hidden. What is this difference?

4.2 Undefinedness in Clean and Core-Clean

As in any other programming language, computations in Core-Clean and in Clean can terminate erroneously. This can happen in a number of situations, for example when dividing by zero, or when a partial function is applied to an argument for which it was not intended. Additionally, Clean even offers two standard functions that always terminate erroneously, namely `abort` and `undef`.

One of the features of lazy languages is that it is possible for a computation to produce a (partial) end result, even when it contains subcomputations that terminate erroneously. This is only possible, however, when the subcomputation is not needed for producing the end result at all.

Assignment 5. *(Partial undefinedness in practice)*
(a) Open the `undefined` project with the IDE. Run and compile it.
(b) Replace the body of `my_undefined` with another computation that also terminates erroneously.
(c) Cycle through the available `Start` bodies and examine the run-time results.

A formal model of Clean needs to be able to handle expressions that contain undefined subexpressions. For this purpose, Core-Clean defines the additional expression alternative '$\perp$'. This constant expression is treated as a base value of any type, because a computation of any type can terminate erroneously. All different kinds of errors are treated equally; therefore, only one $\perp$ suffices and it does not need additional arguments.

Note that $\perp$ is a special value with special characteristics. It cannot be used as a pattern, or in a case distinction. In fact, it is not possible at all in Clean to produce a defined result based on a successful check of undefinedness.

Assignment 6. *(Undefinedness cannot be detected)*
(a) What famous (unsolvable) problem would be solved if it was possible to detect undefinedness within a Clean program?

4.3 Stating a Property

A property in Sparkle is a logical statement, either true or false, that deals with the executional behavior of a Clean-program. Properties can be used to state that the program functions correctly with respect to its specification. Expressing the desired behavior of a program by means of properties is very useful.

Sparkle allows properties to be expressed in an extended first-order logic. The usual logic operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) and $\leftrightarrow$ (iff) are supported, as well as the quantors $\forall$ (for all) and $\exists$ (exists), and the constants `TRUE` and `FALSE`. Variables and quantors can range over propositions and over expressions of an arbitrary type, but not over predicates or relations of any kind. To state properties of programs, the logic also supports *equality* on expressions.

$$\begin{array}{rl} \mathit{Prop} ::== & \mathit{Var}^{\mathit{Prop}} \\ & \mid \mathtt{TRUE} \mid \mathtt{FALSE} \\ & \mid \neg\mathit{Prop} \mid \mathit{Prop} \wedge \mathit{Prop} \mid \mathit{Prop} \vee \mathit{Prop} \mid \mathit{Prop} \rightarrow \mathit{Prop} \mid \mathit{Prop} \leftrightarrow \mathit{Prop} \\ & \mid \forall_{\mathit{Var}^{\mathit{Prop}}}.\mathit{Prop} \mid \forall_{\mathit{Var}^{\mathit{Expr}}}.\mathit{Prop} \mid \exists_{\mathit{Var}^{\mathit{Prop}}}.\mathit{Prop} \mid \exists_{\mathit{Var}^{\mathit{Expr}}}.\mathit{Prop} \\ & \mid \mathit{Expr} = \mathit{Expr} \end{array}$$

Many concepts of the proposition level are also available on the expression level, which can be a little confusing. Note for instance the subtle differences between:

- `True` and `False`, which are expressions of type `Bool`, and `TRUE` and `FALSE`, which are propositions;
- `not`, `&&` and ||, which are CLEAN-functions that operate on values of type `Bool`, and ¬, ∧ and ∨, which are operators that connect propositions;
- `==`, which is an overloaded CLEAN-function that produces a `Bool` and must be defined manually for each type, and =, which produces a proposition and is available automatically for each type.
  (the CLEAN-*function* `==` *is computable and cannot compare undefined values, while the formal = is not computable and can compare undefined values; this additional expressiveness is really important, because many properties have definedness preconditions that could otherwise not be expressed)*

On the other hand, the availability of the expression level also allows boolean expressions to sometimes be used as predicates and relations (see Section 5.7).

Assuming the context of the `primes` project, examples of properties are:

1. $\forall_P \forall_Q.(P \wedge Q) \leftrightarrow (Q \wedge P)$
2. $17 > 12 = \mathtt{True}$
3. $\forall_f \forall_{xs} \forall_{ys}.\mathtt{map}\ f\ (xs\ \mathtt{++}\ ys) = \mathtt{map}\ f\ xs\ \mathtt{++}\ \mathtt{map}\ f\ ys$
4. $\forall_{xs}.\mathtt{reverse}\ (\mathtt{reverse}\ xs) = xs$
5. $\forall_n \forall_{xs}.(n < \mathtt{length}\ xs = \mathtt{True}) \rightarrow \mathtt{length}\ (\mathtt{take}\ n\ xs) = n$
6. $\forall_i \forall_j.(i > j = \mathtt{True} \wedge j > 0 = \mathtt{True}) \rightarrow \mathtt{primes}\ \mathtt{!!}\ i > \mathtt{primes}\ \mathtt{!!}\ j = \mathtt{True}$

Of these properties, the first does not refer to any component of the program; in fact, it is a *tautology* which is independent of any program. The second property refers to the function `>`, which is defined for integers in the module `StdInt`. The third, fourth and fifth properties refer to the functions `map`, `++`, `reverse`, `take` and `length`, which are all defined in the module `StdList`. The sixth property, finally, is the only property that is really specific for the `primes` project. It not only depends on the standard functions `>` and `!!`, but also on the `primes` function of the `primes` module.

Assignment 7. *(Validity of the example properties)*

(a) Of the six example properties, only five are true, and one is in fact false (it needs an additional precondition). Which one is false?
(*Hint:* lists may be infinite in CLEAN)

(b) What happens to the sixth property if either i or j is undefined?

The only way to enter properties in SPARKLE is by means of textual input. The parser allows the natural syntax to be used, with the following conventions:

- `~P` denotes $\neg P$;
- `P /\ Q` denotes $P \wedge Q$;
- `P \/ Q` denotes $P \vee Q$;
- `P -> Q` denotes $P \rightarrow Q$;
- `P <-> Q` denotes $P \leftrightarrow Q$;
- `_|_` denotes $\perp$;
- `[x]` denotes $\forall_x$; and
- `{x}` denotes $\exists_x$.

Type-checking of propositions is performed automatically by SPARKLE. During this check, the types of the variables are inferred as well. Alternatively, it is also possible to explicitly specify the type of a variable in a quantor. These explicit types may contain type variables, which are implicitly assumed to be bound by universal quantors. Typed quantors are denoted by:

- `[x::a]` denotes $\forall_{x::a}$; and
- `{x::a}` denotes $\exists_{x::a}$.

Assignment 8. *(Specify the example properties (1))*
(a) Use `New Theorem` to manually enter all six example properties. (*Hint:* in case of failure, attempt to add brackets)

Assignment 9. *(Specify properties with overloading)*
The manual specification of types is essential when making use of overloading:
(a) Without explicit types, attempt to specify $\forall_x \forall_y . x + y = y + x$.
(b) Use explicit types ($x :: \texttt{Int}, y :: \texttt{Int}$) to help SPARKLE solve the overloading in $\forall_x \forall_y . x + y = y + x$.

For the sake of convenience, SPARKLE offers two features to make the manual specification of properties easier:

- Each free symbol in the proposition is assumed to be a variable, and a universal quantor is created automatically for it. This feature allows universal quantors to be omitted when specifying properties. It also means, however, that mistyping the name of an identifier, or using an identifier that is not defined by the current program, does *not* lead to a bind error, but instead results in an incorrect universal quantor.
- When possible, boolean expressions are automatically lifted to propositions by implicitly adding '= `True`'. This feature shortens specifications, but may also lead to confusion between the expression and the proposition level. Note that the '= `True`' behind a lifted boolean expression is not even displayed by SPARKLE if the `Boolean Predicates` display option is turned on.

Assignment 10. *(Specify the example properties (2))*
(a) Specify the example properties again, using the features described above. Do *not* quit SPARKLE afterwards.

SPARKLE organizes theorems and proofs into sections, much in the same way as CLEAN organizes definitions into modules. Sections are stored in a semi-readable internal format in SPARKLE's `\Sections` subdirectory. Theorems and (parts of) proofs can be assigned to individual sections, which must then be saved explicitly. The special section `main` is always available, but it cannot be saved and should only be used for temporary properties. A warning for users: SPARKLE does not save sections automatically, and does not prompt you to do so either!

Assignment 11. *(Save properties into sections)*
(a) Create a new section with the name `temp`.
(b) Open both the `main` section and the `temp` section.
(c) Move the example properties from the `main` section into the `temp` section.
(d) Save the `temp` section and quit SPARKLE.

Of course, sections can be loaded into SPARKLE as well. Because the contents of a section may depend on various other components, the following actions are carried out when a section is loaded:

- First, it is verified if the symbols are available that are required for stating the properties of the section. If this is not the case, then the section is not loaded at all. Otherwise, theorems are created for the properties of the section. The proofs themselves, however, are not loaded yet.
- Then, the sections are loaded recursively that contain the theorems that are used within the proofs of the top-level section.
- Finally, the proofs of the section are loaded and carried out again, step by step. If a step fails, which may be the case if a definition within the program has been altered (but its name and type were unchanged), then the proof can be loaded partially until the error point.

After this process, it can be guaranteed that the internal state of SPARKLE is consistent, and that all proofs that were loaded successfully are valid.

Assignment 12. *(Load sections into memory)*
(a) Start SPARKLE manually (directly and not from within the IDE).
(b) Attempt to load the predefined section `lists`.
(c) Use Ctrl-O to open the `primes` project from within SPARKLE.
(d) Load the predefined section `lists`.
(e) Load the section `temp` of the previous assignment.

4.4 The Meaning of Properties

The meaning of properties is described by a formal algorithm that determines whether a given property, in the context of a given program, is true or false. This algorithm is expressed at the formal level only, and cannot be executed in practice, neither by a human nor by a computer. If it could be executed, formal reasoning would not have been necessary in the first place.

A meaning must be provided for all alternatives of SPARKLE's first-order logic, which was introduced in Section 4.3. This logic contains both standard elements (`TRUE`, `FALSE`, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\forall$ on propositions, $\exists$ on propositions) and customized ones ($=$, $\forall$ on expressions, $\exists$ on expressions). The meaning of the standard elements is the same as in standard logic, which we assume to be well-known. The meaning of the customized elements is as follows:

– The equality $e_1 = e_2$ holds if for all programs P the *observational* behavior stays the same if e_1 is interchanged with e_2 (or vice versa, e_2 with e_1). The observational behavior of a program is the visible output that is produced when it is executed. SPARKLE cannot deal with programs that perform I/O; therefore, only output that is displayed on the console is considered.

  To be able to determine the equality between observational behaviors, it has to be taken into account that programs may not terminate, and that the output that they produce may be infinite. On the formal level, observational behavior is therefore modeled by time indexed *streams*, and *bisimulation* is used to determine equality. On the intuitive level, this is equivalent to assuming that infinite time is available to programs, and that the resulting infinite streams are equal only if all their finite substreams are equal.

  Finally, note that it is not possible to determine if e_1 and e_2 are semantically equal based only on the observational behaviors of the programs `Start = `e_1 and `Start = `e_2. This is because e_1 and e_2 may be functions that only produce meaningful output when they are supplied with arguments.

– The universal quantification $\forall_x.P$ holds if for all wellformed expressions E the instantiated proposition $P[x \mapsto E]$ holds. An expression E is *wellformed* if the resulting $P[x \mapsto E]$ is both closed and welltyped.

  Note that the undefined expression $\perp$ is always a valid value for E, because it is closed and of any type. Furthermore, if the domain of x allows for it, infinite expressions are also valid values for E.

– The meaning of the existential quantification $\exists_x.P$ is defined in the same way as the universal quantification.

Assignment 13. *(Examples of (in)equality)*

(a) Are '`ones`' and '`let x = [1:x] in x`' equal? If so, argue; if not, give the program that distinguishes between them.

(b) Same question for '`ones`' and '`ones ++ ones`'.

(c) Same question for '`ones`' and '`[2] ++ ones`'.

(d) Same question for '`ones`' and '`ones ++ [2]`'.

(e) Same question for '$\perp$' and '`[1:`$\perp$`]`'.

(f) Same question for '$\perp$' and 'λ`x.`$\perp$'.
(*Hint*: make use of explicit strictness)

(g) Same question for '$\perp$' and '`let x = x in x`'.
(*Hint*: only basic values and constructors are meaningful output)

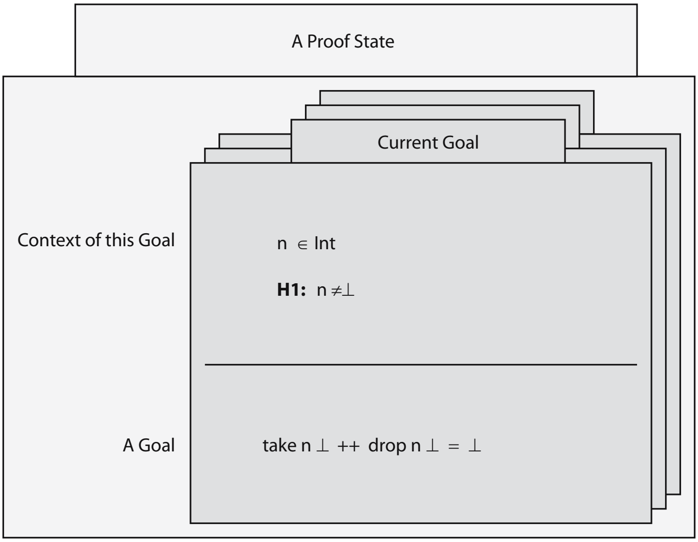


Fig. 1. A proof state

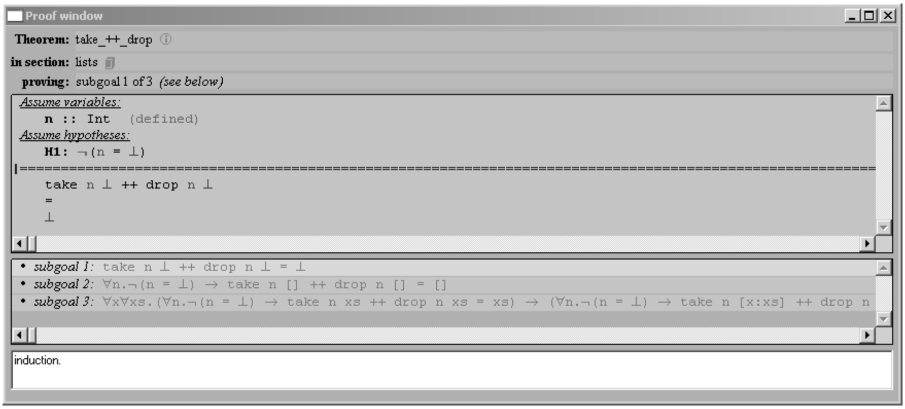
4.5 Reasoning Style in Sparkle

As most modern day proof assistants, Sparkle is based on the LCF-approach. This means that reasoning takes place by the repeated simplification of a list of goals by means of the application of tactics. This process of reasoning was first introduced by the LCF[18] proof assistant, and has since been named after it.

The theoretical background of this style of reasoning was already introduced in Sections 2.3 and 2.4. From the user's point of view, each theorem requires the repeated manipulation of a list of goals (=properties to be proved) by means of the application of tactics (=reasoning steps). The goals can be proved in any order; the goal currently being manipulated is called the *active* goal and the others are called *subgoals*. The tactics must be selected from a fixed library, and are guaranteed to be sound. The formal proof tree is maintained internally by Sparkle and can be browsed manually for an overview of the proof, but it is otherwise not needed for reasoning at all.

Assignment 14. *(Backwards proving)*
(a) Why is Sparkle's reasoning style sometimes also called *backwards proving*?

A goal corresponds to a property that still to be proved, but on the syntactic level it is broken into components which can be manipulated separately by the reasoning process. The components of a goal are introduced variables, introduced hypotheses and the 'to prove'. If $x_1, \ldots, x_n$ are the introduced variables, $H_1 : P_1, \ldots, H_m : P_m$ are the introduced hypotheses, and Q is the to prove, then the goal corresponds to the property $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$.



Theorem: take_++_drop

in section: lists

proving: subgoal 1 of 3 *(see below)*

Assume variables:

```
    n :: Int   (defined)
```

Assume hypotheses:

```
    H1: ¬(n = ⊥)
|==========================================================================================
    take n ⊥ ++ drop n ⊥
    =
    ⊥
```

- *subgoal 1:* `take n ⊥ ++ drop n ⊥ = ⊥`
- *subgoal 2:* `∀n.¬(n = ⊥) → take n [] ++ drop n [] = []`
- *subgoal 3:* `∀x∀xs.(∀n.¬(n = ⊥) → take n xs ++ drop n xs = xs) → (∀n.¬(n = ⊥) → take n [x:xs] ++ drop n`

```
induction.
```

Fig. 2. Screen shot of the SPARKLE proof window at the same proof state as in Fig. 1

Assignment 15. *(Decompose the property)*
The proof states in Fig. 1 and Fig. 2 are taken from an actual proof.
(a) Which property corresponds to the current goal in Fig. 1?
(b) Which property was the starting point of the proof?

4.6 Proving a Simple Property

In this section, we will use SPARKLE to prove a simple property which concerns the behavior of the `map` function from the standard environment of CLEAN.

Assignment 16. *(Specification of a property of* `map`*)*
(a) Open SPARKLE from scratch, then load the standard environment (Ctrl-E).
(b) Create a new section with the name `map_section`.
(c) In `map_section`, create a new theorem named `map_property`, stating:

$$\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys$$

(d) Open the proof window (Ctrl-P) that corresponds to the created theorem.

Building a proof is the repeated process of selecting tactics and applying them on the current goal. For this process, SPARKLE makes a total of 39 tactics available, which are all described briefly in Appendix A. The user interface of SPARKLE allows tactics to be applied by means of three different methods:

- The *hint mechanism*, which is activated by opening the Tactic Suggestion Window during proving. This window holds a dynamically updated list of suggestions for tactics that can be applied to the current goal. SPARKLE generates these suggestions automatically based on built-in heuristics. Each suggestion is assigned a score between 1 and 100 that indicates the likelihood of that tactic being helpful in the proof. Based on this score, the suggestions are ordered. A suggested tactic can be applied by either clicking on it, or by means of its associated hot-key (F1 for the first hint, F2 for the second, etc.).

It is also possible to configure Sparkle to apply the top hint automatically if it has a score higher than a manually set threshold.

The hint mechanism is mainly for *beginning* Sparkle users. It is fast and easy to use, and requires little expertise of the available tactics (simply trust Sparkle !). The hint mechanism is a valuable tool that can be used as a means of learning Sparkle, and with which many small proofs can be built fully. However, it is not very powerful and by no means failsafe. Sometimes the right tactic is not suggested, or several wrong tactics get high scores.

– The *tactic dialogs*. Each tactic has its own dialog that can be opened by clicking on its name in the Tactic List Window. This dialog has entries for all the arguments that can be given to the tactic. When possible, the current goal is used to restrict the input to valid values only. When all arguments have been entered, the tactic can be applied from the dialog directly.

  The tactic dialogs are for *intermediate* users. This method of proving is both powerful, because all tactics can be applied this way, and fairly easy, because one does not need to memorize the name or syntax of a tactic, nor the arguments that it requires.

– The *command line interface*. This is a textual interface, located at the bottom of the Proof Window, that is for *advanced users* only. It is powerful, but requires extensive expertise of Sparkle and its tactics. However, once mastered, it is the fastest way of building proofs, because all tactics can be applied this way and it does not require opening additional dialogs at all.

The property of `map` that was given above is very easy and can therefore be proved automatically with the hint mechanism:

Assignment 17. *(Proving the* `map` *property with the hint mechanism)*

(a) Open the Tactic Suggestions Window (Ctrl-H) and set the threshold to 1.

(b) Set the threshold back to 101. Why is this necessary prior to (c)?

(c) Enter ◀`Restart.`▶ at the command-line interface.
(From now on, ◀`cmd.`▶ *will be used to denote textual input to the command-line. For reasons of parsing, these commands have to end with a closing '.', otherwise* Sparkle *will not be able to recognize them.)*

(d) Redo the proof by applying suggestions manually with the hot-keys.

The complete proof tree of the example property has now been stored internally by Sparkle. By means of the Theorem Info Window, this proof tree can be browsed and inspected in detail:

Assignment 18. *(Browsing through the proof)*

(a) Open the Theorem Info Window of the completed proof.

(b) Click 'browse' after the first tactic and then browse through the proof using the 'previous' and 'next' buttons.

(c) Undo the first application of `Reflexive` only.

(d) Click on the brown star to return to the Proof Window.

(e) Use a different tactic to prove the goal.

The hint mechanism has succeeded in completing the proof automatically, and it did not require any expertise at all. The downside to this, unfortunately, is that no understanding of the tactics has been gained in the process. Therefore, below we will present the entire proof again, and this time we will explain each tactic that was applied too.

The initial goal is simply the property to be proved:

$$\boxed{\frac{\text{-}}{\forall_f \forall_{xs} \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys}}\ (1)$$

Because both `map` and `++` are tail-recursive, 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 ⊥; one for the case that *xs* is `Nil`; and one for the case that *xs* is an application of `Cons`. Note that ⊥ is a base value of any type and is therefore always treated by induction as a constructor case.

$$\boxed{\frac{\text{-}}{\forall_f \forall_{ys}.\texttt{map}\ f\ (\bot\ \texttt{++}\ ys) = \texttt{map}\ f\ \bot\ \texttt{++}\ \texttt{map}\ f\ ys}}\ (1.1)$$

The current proposition starts with two universal quantifications, on which it does not make sense to perform induction (on *f* it is not possible, and on *ys* it does not help because `++` is not tail-recursive in its second argument). It is therefore best to apply ◀`Introduce f ys.`▶, which removes the quantors and introduces the variables *f* and *xs* in the context of the goal. After this action, the main proposition can be accessed more easily.

$$\boxed{\frac{f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]}}{\texttt{map}\ f\ (\bot\ \texttt{++}\ ys) = \texttt{map}\ f\ \bot\ \texttt{++}\ \texttt{map}\ f\ ys}}\ (1.1')$$

Due to the strictness of `map` and `++` and the presence of ⊥ 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. With other parameters, the tactic `Reduce` can also be used for stepwise reduction, reduction to root normal form, reduction of one particular redex and reduction in the goal context.

$$\boxed{\frac{f :: \texttt{b} \rightarrow \texttt{a},\ ys :: \texttt{[b]}}{\bot = \bot}}\ (1.1'')$$

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

$$\boxed{\frac{\text{-}}{\forall_f \forall_{ys}.\texttt{map}\ f\ (\texttt{[]}\ \texttt{++}\ ys) = \texttt{map}\ f\ \texttt{[]}\ \texttt{++}\ \texttt{map}\ f\ ys}}\ (1.2)$$

This is the second goal of induction, created for the case that *xs* is the empty list. Again, induction makes no sense for *f* and *ys*, and they should therefore be introduced in the goal context by means of ◀`Introduce f ys.`▶.

$$\boxed{\dfrac{f :: \mathtt{b} \rightarrow \mathtt{a},\ ys :: \mathtt{[b]}}{\mathtt{map}\ f\ (\mathtt{[]}\ \mathtt{++}\ ys) = \mathtt{map}\ f\ \mathtt{[]}\ \mathtt{++}\ \mathtt{map}\ f\ ys}} \qquad (1.2')$$

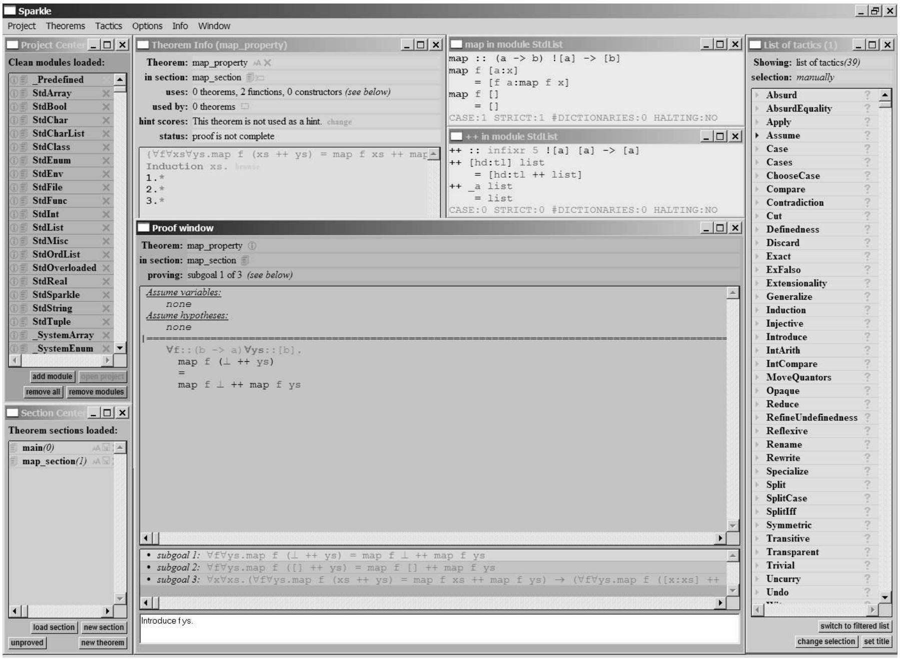
There are again redexes present in the current goal, because both `map` and `++` have patterns that match on the empty list `[]`. Therefore: ◀`Reduce NF All.`▶.

$$\boxed{\dfrac{f :: \mathtt{b} \rightarrow \mathtt{a},\ ys :: \mathtt{[b]}}{\mathtt{[]} = \mathtt{[]}}} \qquad (1.2'')$$

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

$$\boxed{\begin{array}{l} \hfill - \hfill \\ \hline \forall_x \forall_{xs}. \\ \quad (\forall_f \forall_{ys}.\mathtt{map}\ f\ (xs\ \mathtt{++}\ ys) = \mathtt{map}\ f\ xs\ \mathtt{++}\ \mathtt{map}\ f\ ys) \\ \quad \rightarrow (\forall_f \forall_{ys}.\mathtt{map}\ f\ (\mathtt{[}x\!:\!xs\mathtt{]}\ \mathtt{++}\ ys) = \mathtt{map}\ f\ \mathtt{[}x\!:\!xs\mathtt{]}\ \mathtt{++}\ \mathtt{map}\ f\ ys) \end{array}} \qquad (1.3)$$

This is the third goal created by induction for the case that *xs* is a composed list. The current goal looks quite complicated, but introduction can make things a lot clearer. Here, we will not only introduce variables from universal quantors, but we will also introduce hypotheses from implications. This can be performed in one go with ◀`Introduce x xs IH f ys.`▶.



Theorem: map_property
in section: map_section
uses: 0 theorems, 2 functions, 0 constructors *(see below)*
used by: 0 theorems
hint scores: This theorem is not used as a hint. change
status: proof is not complete

```
(∀f∀xs∀ys.map f (xs ++ ys) = map f xs ++ map
Induction xs.
1.*
2.*
3.*
```

```
map :: (a -> b) ![a] -> [b]
map f [a:x]
    = [f a:map f x]
map f []
    = []
CASE:1 STRICT:1 #DICTIONARIES:0 HALTING:NO
```

```
++ :: infixr 5 ![a] [a] -> [a]
++ [hd:tl] list
    = [hd:tl ++ list]
++ _a list
    = list
CASE:0 STRICT:0 #DICTIONARIES:0 HALTING:NO
```

Theorem: map_property
in section: map_section
proving: subgoal 1 of 3 *(see below)*

```
Assume variables:
    none
Assume hypotheses:
    none
|====================================================================
   ∀f::(b -> a)∀ys::[b].
     map f (⊥ ++ ys)
     =
     map f ⊥ ++ map f ys
```

- *subgoal 1:* `∀f∀ys.map f (⊥ ++ ys) = map f ⊥ ++ map f ys`
- *subgoal 2:* `∀f∀ys.map f ([] ++ ys) = map f [] ++ map f ys`
- *subgoal 3:* `∀x∀xs.(∀f∀ys.map f (xs ++ ys) = map f xs ++ map f ys) → (∀f∀ys.map f ([x:xs] ++`

Introduce f ys.

Showing: list of tactics*(39)*
selection: *manually*

Fig. 3. Screen shot of SPARKLE at proof state (1.1)

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \to \texttt{a},\ ys :: \texttt{[b]} \\ \mathit{IH} : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \\ \hline \texttt{map}\ f\ (\texttt{[}x\texttt{:}xs\texttt{]}\ \texttt{++}\ ys) = \texttt{map}\ f\ \texttt{[}x\texttt{:}xs\texttt{]}\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}}\ (1.3')$$

Again, the current goal contains redexes, because `map` and `++` have patterns that match on constructed lists of the form [x:xs]. Therefore, ◀`Reduce NF All.`▶.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \to \texttt{a},\ ys :: \texttt{[b]} \\ \mathit{IH} : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \\ \hline \texttt{[}f\ x\texttt{:}\texttt{map}\ f\ (xs\ \texttt{++}\ ys)\texttt{]} = \texttt{[}f\ x\texttt{:}\texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys\texttt{]} \end{array}}\ (1.3'')$$

The current proposition is now of the form `[X:Y] = [X:Z]`. Using the automatic injectivity of all *lazy* data constructors in Clean, we can simplify this to `X = X` ∧ `Y = Z`. Therefore, ◀`Injective.`▶.

Assignment 19. *(Injectivity and strictness)*
(a) Why does injectivity not hold for strict data constructors?

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \to \texttt{a},\ ys :: \texttt{[b]} \\ \mathit{IH} : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \\ \hline f\ x = f\ x \wedge \texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}}\ (1.3''')$$

The current proposition is now of the form `P` ∧ `Q`, and can obviously be split into subgoals `P` and `Q`. Therefore, ◀`Split.`▶, which creates subgoals 1.3.1 and 1.3.2.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \to \texttt{a},\ ys :: \texttt{[b]} \\ \mathit{IH} : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \\ \hline f\ x = f\ x \end{array}}\ (1.3.1)$$

This is a reflexive equality that can be proved immediately with ◀`Reflexive.`▶.

$$\boxed{\begin{array}{c} x :: \texttt{b},\ xs :: \texttt{[b]},\ f :: \texttt{b} \to \texttt{a},\ ys :: \texttt{[b]} \\ \mathit{IH} : \forall_f \forall_{ys}.\texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \\ \hline \texttt{map}\ f\ (xs\ \texttt{++}\ ys) = \texttt{map}\ f\ xs\ \texttt{++}\ \texttt{map}\ f\ ys \end{array}}\ (1.3.2)$$

The current proposition is now an instantiation of the induction hypothesis *IH*. It can therefore be proved immediately by applying *IH* with ◀`Apply IH.`▶.

$$\boxed{\text{Q.E.D.}}$$

There are no more subgoals, which means that the proof is complete!

Assignment 20. *(Manual proof of the* `map` *property)*
(a) Prove the `map` property again, using the tactic dialogs only.
(b) Prove the `map` property again, using the command interface only.
(*Hint:* ◀`Reduce.`▶ abbreviates ◀`Reduce NF All.`▶, and ◀`Intros.`▶ is a variant of introduction that comes up with suitable names on its own)

(c) The automatic proof consists of the application of 13 tactics. It is possible to prove the property in less steps (our shortest proof consists of 9 steps). Try to shorten the proof yourself.

Assignment 21. *(More small proofs)*
Try to prove the following properties, preferably without the hint mechanism:
(a) $\forall_{xs}\forall_{ys}\forall_{zs}.xs$ `++` $(ys$ `++` $zs) = (xs$ `++` $ys)$ `++` $zs.$
(b) $\forall_{xs}.\neg(xs = \perp) \rightarrow \neg(xs =$ `[]`$) \rightarrow$ `[hd` xs`:tl` xs`]` $= xs.$
(c) $\forall_{n}\forall_{xs}.\neg(n = \perp) \rightarrow$ `take` $n\ xs$ `++` `drop` $n\ xs = xs.$
(d) $\forall_{P}\forall_{Q}.(\neg P \leftrightarrow Q) \leftrightarrow (P \leftrightarrow \neg Q).$

5 Tutorial Part II: Specialized Features of SPARKLE

In this section, the tutorial will be continued with advanced information about the dedicated use of SPARKLE in practice, and the features that are specialized for reasoning about CLEAN will be described. The same explanatory style will be used as in part I of the tutorial, and various assignments will again be included.

First, in Section 5.1 the importance of sharing in proofs will be explained. Then, the specification of definedness conditions in properties will be described in Section 5.2. The specialized behavior of four tactics will be introduced next; for ‘Extensionality’ in Section 5.3, for ‘Induction’ in Section 5.4, for ‘Definedness’ in Section 5.5, and for ‘Reduce’ in Section 5.6. Finally, the specification of properties by means of CLEAN-functions will be discussed in Section 5.7.

5.1 The Influence of Sharing on Reasoning

Sharing is important for the efficiency of functional programs. In CLEAN sharing is explicit, because for every construct it is precisely defined what is shared and what is not shared [29]. The semantics of CLEAN are based on graph rewriting [2,3,27]. This means that during reduction of the `Start` expression to its result, sharing is maintained as much as possible.

In SPARKLE, reduction may be used at many points in proofs as well. This reduction should behave in a semantically equivalent way to reduction in CLEAN, but it does not have to be exactly the same. Note that reduction in SPARKLE is symbolic, because it may encounter free variables that are introduced by logic quantors. In CLEAN, reduction only operates on closed expressions.

Sharing has no influence on semantics, and reduction in SPARKLE is free to either preserve or break it. Currently, the following strategy is realized:

- *Within* the application of reduction sharing is always preserved;
- But *afterwards* sharing is always automatically broken.

The idea behind this strategy is twofold. Firstly, efficiency is important in proofs too, therefore sharing is preserved within reduction. Secondly, after full reduction sharing is often not meaningful anymore and only hinders reduction, therefore it is automatically broken.

Assignment 22. *(The effect of sharing during reduction in proofs)*

(a) Consider in Sparkle the trivial theorem (`let` n = 1+2+3 `in` n+n) = 12. Prove it using ◀`Reduce NF All.`▶, followed by ◀`Reflexive.`▶.

(b) Undo the proof with `Ctrl-Z` and prove the theorem again, this time using reduction with a fixed number of steps (◀`Reduce 4.`▶).

(c) Undo the proof with `Ctrl-Z` and prove the theorem again, this time using repeated single-step reduction (◀`Reduce 1.`▶).

(d) Explain why more reduction steps are needed in (c) than in (b).

Unfortunately, Sparkle's current strategy for handling sharing is not optimal. The main problem is that all meaningful sharing, such as for instance recursion that has been expressed by means of cyclic lets, cannot be dealt with at all. Moreover, the current behavior is not very intuitive, as was already demonstrated in the assignment above.

The way sharing is handled in Sparkle is currently being fixed according to the reduction mechanism described in [12]. In the next release, Sparkle will always preserve all sharing, and manual reasoning steps will be added that allow users to manipulate, and possibly break, shared expressions at will.

5.2 Definedness Conditions in Properties

Sparkle makes use of a *total* semantics in which undefinedness is taken into consideration explicitly. This has two consequences for the property language. Firstly, expressions are only equal if they either produce the same defined value, or both produce undefinedness. Secondly, the undefined value $\bot$ is a member of any type, and therefore a valid instantiation of any quantor.

In order to specify properties of Clean-programs correctly, one therefore has to know precisely how they behave in case some of their input becomes undefined. This behavior is determined by the lazy rewriting semantics of Clean, of which a thorough understanding is required for formal reasoning. Below we present a small example to illustrate the propagation of $\bot$-values through expressions. For a full explanation of computation in Clean we refer to [29] and [33].

Example. Consider the following definition of the well-known function `take`:

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

In Clean, patterns are evaluated from top to bottom, and right-hand-sides are only evaluated when their pattern matches. Consequently:

- `take` n $\bot = \bot$ for all n, because the first pattern always causes $\bot$ to be matched against `[]`, which fails;
- `take` $\bot$ `[]` = `[]`, because the successful match of the first pattern does not require $\bot$ to be evaluated;
- `take` $\bot$ `[`x`:`xs`]` $= \bot$ for all x and xs, because the second pattern matches, and its right-hand-side requires the computation of $\bot$ `> 0`, which fails.

It is very important that the starting point of formal reasoning is a logically correct property. Therefore, the specification of properties must always involve an analysis of behavior in the undefined case. In some cases, the property turns out to hold automatically for the undefined value, and nothing has to be changed. In other cases, however, the property actually turns out to be false:

Example. Consider the following intuitively true property of `drop` and `take`:

| $\forall_n \forall_{xs}.$`take` $n\ xs$ `++` `drop` $n\ xs = xs.$

This property is falsified by the case $n = \perp$, because then the left-hand-side may become undefined, while the right-hand-side remains xs:

- Assume $xs =$ `[1]`. Then the left-hand-side reduces to $\perp$, as follows:
  `take` $\perp$ `[1] ++ drop` $\perp$ `[1]` $= \perp$ `++ drop` $\perp$ `[1]` $= \perp$.
  But the right-hand-side is `[1]`, which is defined.

Assignment 23. *(More definedness analysis)*

(a) The example property $\forall_n \forall_{xs}.$`take` $n\ xs$ `++` `drop` $n\ xs = xs$ is *not* falsified in the case that $xs = \perp \wedge n \neq \perp$. Argue why this is the case.
(*Hint*: distinguish between $n = 0$ and $n \neq 0$.)

(b) Is the property $\forall_f \forall_{xs} \forall_{ys}.$`map` $f\ (xs$ `++` $ys) = ($`map` $f\ xs)$ `++` $($`map` $f\ ys)$ falsified in the undefined case? If so, give example values for f, xs and ys that break the property. If not, argue why.
(*Hint:* see also Section 4.6.)

If definedness analysis shows that a property is falsified by a set of variable values V, then it can be rectified simply by adding conditions that exclude V. These *definedness conditions* are often simple and of the form '$n \neq \perp$', but they can also be more intricate (see Section 5.7).

Rectified Example: The `take-drop` property can be corrected by means of:

| $\forall_n \forall_{xs}.n \neq \perp \rightarrow$ `take` $n\ xs$ `++` `drop` $n\ xs = xs.$

Assignment 24. *(Proving the rectified* `take-drop` *example)*

(a) In SPARKLE, prove $\forall_n \forall_{xs}.n \neq \perp \rightarrow$ `take` $n\ xs$ `++` `drop` $n\ xs = xs.$

Finally, note that CLEAN supports *strictness annotations*, with which the strict evaluation of certain expressions can be enforced explicitly. These annotations are often placed without much thought with the purpose of improving efficiency. However, strictness annotations change the definedness behavior of the program, and have an effect on properties and reasoning as well. In the context of formal reasoning, they should therefore only be used with care.

The precise effect of strictness annotations on properties is difficult to predict. Adding a strictness annotation can either: **(1)** not change a property at all; or **(2)** falsify a property, requiring additional definedness conditions to be formulated; or **(3)** allow existing definedness conditions to be removed. The third effect in particular is rather surprising.

Example of (1). Consider the following property:

| $\forall_{xs} \forall_{ys} \forall_{zs}.(xs$ `++` $ys)$ `++` $zs = xs$ `++` $(ys$ `++` $zs)$

This property holds for the standard definition of ++, which is strict in its first argument only. Adding strictness to the second argument does not effect the property, however; it remains valid in the strict case as well.

Example of (2). Consider the following property:

| $\forall_{f,g}\forall_{xs}.\texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)$

This property is valid for lazy lists, but invalid for element-strict lists.
Suppose $xs = \texttt{[12]}$, $g\ 12 = \bot$ and $f\ (g\ 12) = 7$.
Then $\texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{[7]}$, both in the lazy and in the strict case.
However, $\texttt{map}\ f\ (\texttt{map}\ g\ xs) = \texttt{[7]}$ in the lazy case, but $\bot$ in the strict case.
The property can be adapted to element-strict lists by explicitly enforcing that g produces a defined result for all elements x of xs:

| $\forall_{f,g,xs}.(\forall_{x \in xs}.g\ x \neq \bot) \rightarrow \texttt{map}\ (f\ \texttt{o}\ g)\ xs = \texttt{map}\ f\ (\texttt{map}\ g\ xs)$.

Example of (3). Consider the following property:

| $\forall_{xs}.\mathit{finite}\ xs \rightarrow \texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

This property is valid both for lazy lists and for spine-strict lists.
The condition *finite xs*, however, is satisfied automatically for spine-strict lists, because spine-strict lists can never be infinite. In the spine-strict case, the property can therefore safely be reformulated (or, rather, optimized) by removing the *finite xs* condition:

| $\forall_{xs}.\texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

Note that without the condition, the property is invalid in the lazy case: just choose any infinite list for xs.

5.3 Specialized Behavior of Extensionality

The property of *extensionality*, which states that two functions are equal iff they produce the same result for all possible arguments, is often considered to be universal. Unfortunately, there is a (rather obscure) example of two functions for which the property of extensionality does not hold unconditionally in the context of lazy evaluation:

```
H :: a -> b                F :: (a -> b)
H x = H x                  F = F
```

In the definitions above, H is a function of arity 1 that only reduces (to itself) when it is given an argument. F on the other hand is a function of arity 0 that always reduces to itself, regardless of whether it is applied or not. Obviously, $\texttt{F}\ x = \texttt{H}\ x$ now holds for all x, because they both reduce to themselves and are therefore both undefined.

Surprisingly, the property $\texttt{F} = \texttt{H}$ does not hold, because H is defined (it is a partial function application, and is thus in head normal form), while the meaning of F is undefined. It is therefore not safe to replace H by F (nor F by H); such a replacement could namely change the termination behavior of the program.

Fortunately, the problem can be corrected by weakening the property of extensionality as follows:

Definition 5.3.1. *(Revised version of extensionality)*

$\forall_f \forall_g.(f = \bot \leftrightarrow g = \bot) \rightarrow (\forall_x.f\ x = g\ x) \rightarrow f = g$

This revised version of extensionality is correct in the context of CLEAN. It can not be applied to prove $F = H$, because the condition $F = \bot \leftrightarrow H = \bot$ does not hold. SPARKLE defines a reasoning step for extensionality that makes use of the correct behavior.

Assignment 25. *(Extensionality)*

(a) Prove using extensionality that `sum` $\circ$ (`map` (`const` 1)) = `length` holds.

5.4 Specialized Behavior of Induction

An important reasoning step for dealing with recursive functions over algebraic datatypes is *structural induction.* Although induction is not always applicable, it is extremely useful in the context of functional programming, because it can be used successfully on many common data structures (such as for instance lists) and on many common kinds of recursive functions (such as for instance those defined by recursion on the results of pattern matching).

In order to deal with lazy evaluation, induction has to be customized in two different ways. Firstly, an extra base step is required for the undefined value $\bot$. Because $\bot$ is a member of each type, it must namely be treated as a constructor with no arguments. This behavior of induction is actually quite intuitive; for instance, if we want to prove $\forall_{x \in [A]}.P(x)$ with induction on the list structure, we would get the following proof obligations:

- $P(\bot)$;
- $P($`[]`$)$;
- $\forall_{x \in A} \forall_{xs \in [A]}.P(xs) \rightarrow P([x\!:\!xs])$

Note that without the case for undefinedness it is possible to prove properties that are not true. For instance, we could easily prove that every lazy list is finite: the empty list is finite, and the extension of a finite list with a single element is always finite as well. The undefined list, on the other hand, is not finite!

The second customization of induction extends it to infinite structures as well. Because an infinite structure does not end with a base case, the induction principle is in general not applicable to it. In [25], however, Paulson has shown that the results of induction may be applied to infinite structures as long as the induction predicate satisfies the criterion of *admissibility.* We claim that Paulson's results may be applied to the context of CLEAN as well.

The admissibility criterion can be lifted to lazy functional languages easily. The basic idea is that equalities on negative positions (behind a negation) within a proposition must be decidable. An equality on type α is decidable if all possible expressions of type α are finite. This can be approximated statically: if α does not contain any recursion, then all its members are certainly finite. An equality on `Bool` is for instance decidable, but an equality on lists is not.

Definition 5.4.1. *(Finite types)*
A type α is *finite* if the set E of all possible expressions of type α is finite.

Definition 5.4.2. *(Decidable equalities)*
An equality between values of type α is *decidable* if α is finite.
We will denote this (informally) with $\mathit{Decidable}(=)$.

Definition 5.4.3. *(Admissibility)*
A proposition P is admissible if $\mathit{Adm}(+1, P)$ holds, by means of:

$$\begin{array}{ll}
\mathit{Adm}(\mathit{sign},\ \mathit{True}) & = \mathit{True} \\
\mathit{Adm}(\mathit{sign},\ \mathit{False}) & = \mathit{True} \\
\mathit{Adm}(\mathit{sign},\ \neg P) & = \mathit{Adm}(-\mathit{sign}, P) \\
\mathit{Adm}(\mathit{sign},\ P \wedge Q) & = \mathit{Adm}(\mathit{sign}, P) \wedge \mathit{Adm}(\mathit{sign}, Q) \\
\mathit{Adm}(\mathit{sign},\ P \vee Q) & = \mathit{Adm}(\mathit{sign}, P) \wedge \mathit{Adm}(\mathit{sign}, Q) \\
\mathit{Adm}(\mathit{sign},\ P \rightarrow Q) & = \mathit{Adm}(-\mathit{sign}, P) \wedge \mathit{Adm}(\mathit{sign}, Q) \\
\mathit{Adm}(\mathit{sign},\ P \leftrightarrow Q) & = \mathit{Adm}(\mathit{sign}, P \rightarrow Q) \wedge \mathit{Adm}(\mathit{sign}, Q \rightarrow P) \\
\mathit{Adm}(\mathit{sign},\ \forall .P) & = \mathit{Adm}(\mathit{sign}, P) \\
\mathit{Adm}(\mathit{sign},\ \exists .P) & = \mathit{Adm}(\mathit{sign}, P) \\
\mathit{Adm}(\mathit{sign},\ E_1 = E_2) & = \mathit{Decidable}(=) \vee \mathit{sign} = +1
\end{array}$$

Assignment 26. *(Induction on lazy lists)*
For each of the theorems below: prove it or show that it is not admissible.
(a) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{take}\ (\texttt{length}\ xs)\ xs = xs$
(b) $\forall_{xs}.xs = \texttt{ones} \rightarrow \texttt{finite}\ xs$
(c) $\forall_{xs}\forall_{f}\forall_{p}.\texttt{all}\ p\ (\texttt{map}\ f\ xs) = \texttt{all}\ (p\ \texttt{o}\ f)\ xs$
(d) $\forall_{xs\in\texttt{[}a\texttt{]}}\forall_{ys\in\texttt{[}a\texttt{]}}.xs = ys \rightarrow xs\ \texttt{==}\ ys$

In order to reason about non-admissible predicates and/or non-inductive types several techniques have been developed. The most renowned of them are the take lemma and its improved version the approximation lemma [4] on one hand, and the class of techniques concerning co-induction based on bisimilarity[17] on the other hand. To treat them in further detail is outside the scope of this paper.

5.5 Definedness Analysis and the Special 'Definedness' Tactic

A consequence of the specialized behavior described in Sections 5.2-5.4 is that reasoning in Sparkle often involves properties of the form $E = \bot$ or $E \neq \bot$. Dealing with definedness is cumbersome, and should therefore be supported as much as possible. For this purpose, Sparkle derives definedness information automatically, and offers specialized tactics that make use of this information.

Definedness analysis is the process of deriving definedness information. It is carried out automatically by Sparkle each time a new goal is constructed. The results of definedness analysis are sets D and U, which contain expressions that have been determined to be defined and undefined respectively. The sets D and U are stored with each goal and can be used by various tactics.

The process of definedness analysis starts by assigning all occurring basic values to D and $\bot$ to U. It then repeatedly extends D and U by examining the hypotheses that have been introduced, and by making use of *strictness* and *totality* properties. The following derivation rules are used for this purpose:

- *Definedness by hypothesis equality.*
  If a hypothesis $E_0 = E_1$ is available, and $E_i \in D$, then add E_{1-i} to D.
  If a hypothesis $E_0 = E_1$ is available, and $E_i \in U$, then add E_{1-i} to U.
  If a hypothesis $E_0 \neq E_1$ is available, and $E_i \in U$, then add E_{1-i} to D.
- *Constructor definedness.*
  Assume that C is a constructor of arity n with strict arguments $S \subseteq \{1 \ldots n\}$.
  If the application $A = (C\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \subseteq D$, then add A to D.
  If the application $A = (C\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \cap U \neq \varnothing$, then add A to U.
- *Total function definedness.*
  Assume that F is a function of arity n which is known to be total.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \subseteq D$, then add A to D.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid 1 \leq i \leq n\} \cap U \neq \varnothing$, then add A to U.
- *Normal function definedness.*
  Assume that F is a function of arity n with strict arguments $S \subseteq \{1 \ldots n\}$.
  If the application $A = (F\ E_1 \ldots E_n)$ occurs as a subexpression in the goal, and $\{E_i \mid i \in S\} \cap U \neq \varnothing$, then add A to U.

Note that the strictness information for the definedness analysis is available explicitly in the source program, whereas the totality information is assumed to be made available externally (in Sparkle, many functions from `StdEnv` are hard-coded to be total). Furthermore, to maximize the effectiveness of the definedness analysis, the negation of the current goal is treated as a hypothesis as well.

An important tactic that makes use of definedness analysis is 'Definedness'. It immediately proves any goal that contains contradictory definedness, which is the case if D and U overlap. Note that because the negation of the current goal is treated as a hypothesis, it also proves any goal in which the definedness information implies the validity of the to prove. Although the rules of definedness analysis are relatively simple, it is surprisingly powerful. The Sparkle-tactic 'Definedness' is therefore extremely useful, and can be applied often in proofs.

Assignment 27. *(Using the Definedness-tactic)*
Prove each of the following properties in Sparkle with the Definedness-tactic.
(a) $\forall_f \forall_{xs}.\neg(\texttt{map}\ f\ xs = \bot) \rightarrow \neg(xs = \bot)$
(b) $\forall_n.\texttt{eval}\ (n\ \texttt{+}\ 12) \rightarrow \neg(n = \bot)$
(see Section 5.7 for an introduction of the `eval` function)
(c) $\forall_n \forall_m.(n\ \texttt{/}\ m = 42) \rightarrow \neg(n\ \texttt{+}\ m = \bot)$
(d) $\forall_n.(7\ \texttt{+}\ (12\ \texttt{*}\ (13\ \texttt{-}\ n)) = \bot) \rightarrow n = \bot$

More examples of the use of definedness can be found in [33].

5.6 Specialized Behavior of Reduction

Because of the presence of logic variables that are introduced by quantors on the property level, reduction in Sparkle is *symbolic*. A logic variable may be instantiated with an arbitrary well-typed expression, and its evaluation does not yield anything. Assuming termination, it is therefore no longer possible to reduce every expression to either a weak head normal form or to $\bot$.

It is important that reduction in Sparkle carries on as far as possible. For this purpose, Sparkle realizes two extensions in its reduction mechanism that allow reduction to continue, even when a logic variable is encountered on a strict position.The first extension involves ignoring unnecessary strictness annotations; the second extension involves using the results of definedness analysis.

The idea of the first extension is that some strictness annotations can safely be removed without changing the semantics of the program. To illustrate this, take a look at the following three Clean-functions:

```
id :: !a -> a    K :: !a !b -> a   length :: ![a] -> Int
id x = x         K x y = x         length [x:xs] = 1+length xs
                                   length []     = 0
```

An exclamation mark before the type of an argument indicates strictness. During evaluation, the strict arguments of a function will always be reduced to weak head normal form *before* the function is expanded, whereas the non-strict arguments will not. A strictness annotation always changes the reduction behavior of the program; however, it does not always change the semantics.

The strictness annotation in the function `id` does not change the semantics, because the evaluation of its body immediately requires the evaluation of its argument anyway. The same goes for the `length` function, because the pattern match enforces evaluation. In the function `K`, the first strictness annotation does not change the semantics, but the second one does. In fact, removing the second annotation would cause `K` $x \perp = x$, where in the current situation `K` $x \perp = \perp$.

The reduction system of Sparkle is able to recognize the different kinds of strictness annotations. In case a strict function argument is encountered like in `id` or in `K` (first annotation), it will be reduced first, but the function will *always* be expanded afterwards. This is different from reduction in Clean, but semantically sound, and much more user friendly for reasoning (not expanding '`id x`' would be really inconvenient). The behavior of Sparkle on annotations as in `K` (second annotation) is of course not changed, because that would be semantically unsound. The behavior on annotations as in `length` is not changed either, because the pending pattern match requires its argument to be reduced. Expanding the function therefore does not make much sense, because reduction would be stopped by the pattern match anyway.

Assignment 28. *(Reduction in* Sparkle *(1))*

(a) Build a Clean-module with the functions above and load it into Sparkle.

(b) Prove $\forall_x$.`id` $x = x$

(c) Prove $\forall_x$.`K` x $12 = x$

(d) Attempt to prove $\forall_x$.`K` 12 $x = 12$. Why does this property not hold?

The second extension of reduction is very straightforward: simply make use of the results of the definedness analysis. In case SPARKLE encounters a function argument whose strictness cannot be removed safely, and on which no pattern match is performed, then the function is allowed to be expanded anyway, as long as the argument expression is an element of D. Again, the argument will be reduced as much as possible first. The second extension allows users to influence the reduction mechanism by means of specifying (and later proving) additional definedness properties.

Assignment 29. *(Reduction in* SPARKLE *(2))*
(a) Prove $\forall_x \forall_y . \neg(y = \bot) \rightarrow \mathtt{K}\ x\ y = x$

5.7 Property Specification in CLEAN

The property language of SPARKLE is a simple first-order proposition logic only, in which predicates and relations cannot be expressed. However, the possibility to define higher-order functions in the programming language and use them as boolean predicates gives unexpected expressive power. The higher-order of the programming language can be combined with SPARKLE's first order logic.

A good example of a boolean predicate in CLEAN is the function `eval`. The purpose of `eval` is to *fully* reduce its argument and return `True` afterwards. Such an 'eval' function is usually used to express evaluation strategies in the context of parallelism [6,32]. We use `eval` for expressing definedness conditions.

In the module `StdSparkle` of SPARKLE's standard environment, the function `eval` is defined by means of overloading. The instance on `Char` is defined by:

```
class eval a :: !a -> Bool

instance eval Char
where    eval :: !Char -> Bool
         eval x = True
```

In a logical property, $(\mathtt{eval}\ x = \mathtt{True})$ can now be used as a manual definedness condition. The meaning of this condition is identical to $\neg(x = \bot)$, because:

- If $x = \bot$, then $(\mathtt{eval}\ x) = (\mathtt{eval}\ \bot) = \bot$ on the semantic level, because `eval` is strict in its argument. Therefore, $\mathtt{eval}\ x = \mathtt{True}$ is not satisfied.
- If $x \neq \bot$, then x must be equal to some defined basic character b. Therefore, $(\mathtt{eval}\ x) = (\mathtt{eval}\ b) = \mathtt{True}$ on the semantic level.
- Note that `eval` is defined in such a way that it is *never* equal to `False`.

On characters, `eval` is not so interesting. However, by means of overloading, it can easily be defined for lists, and all other kinds of data structures as well. The overloading is used to assume the presence of an `eval` on the element type:

```
instance eval [a] | eval a
where    eval :: ![a] -> Bool | eval a
         eval [x:xs] = eval x && eval xs
         eval []     = True
```

This instance of `eval` fully evaluates both the spine of the list and all its elements, and only returns `True` if this succeeds. It can therefore be used to express the intricate definedness condition that a list is finite and contains defined elements only. This condition cannot be expressed on the property level at all.

Assignment 30. *(Proofs of properties that use* `eval`*)*
Using the function `eval` from `StdSparkle`, prove the following properties:
(a) $\forall_x \forall_{xs}.\texttt{eval}\ xs \rightarrow \texttt{isMember}\ x\ xs \rightarrow \texttt{eval}\ x$
(b) $\forall_{xs}.\texttt{eval}\ xs \rightarrow \texttt{sum}\ (\texttt{map}\ (\texttt{K}\ 1)\ xs) = \texttt{length}\ xs$
(using the strict version of function `K`, see assignment 28)
(c) $\forall_x \forall_p \forall_{xs}.\texttt{eval}\ x \rightarrow \texttt{eval}\ xs \rightarrow \texttt{eval}\ (\texttt{map}\ p\ xs) \rightarrow$
$\texttt{isMember}\ x\ (\texttt{filter}\ p\ xs) = \texttt{isMember}\ x\ xs\ \texttt{\&\&}\ p\ x$

All instances of `eval` have to share certain properties. To prove properties of *all* members of a certain type class, the recently added tool support for general type classes can be used [21]. With this tool, the properties $\forall_x.\texttt{eval}\ x \rightarrow x \neq \bot$ and $\forall_x.\texttt{eval}\ x \neq \texttt{False}$ can be stated and proven in SPARKLE.

A useful variation of `eval` on lists is the function that evaluates the spine of the list only, but leaves the elements alone. This function expresses the condition that a list is finite. It is defined in `StdSparkle` as follows:

```
finite :: ![a] -> Bool
finite [x:xs]    = finite xs
finite []        = True
```

The boolean predicate `finite` allows several useful properties to be stated and proven in SPARKLE:

Assignment 31. *(Proofs of properties that use* `finite`*)*
Using the function `finite` from `StdSparkle`, prove the following properties:
(a) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{length}\ xs \geq 0$
(b) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{finite}\ (\texttt{reverse}\ xs)$
(c) $\forall_{xs}.\texttt{finite}\ xs \rightarrow \texttt{reverse}\ (\texttt{reverse}\ xs) = xs$

6 Related Work

Currently, well-known and widely used proof assistants are PVS [24], COQ [31] and ISABELLE [26]. They are all generic provers that are not tailored towards a specific programming language. It is very hard for programmers to reason in them, because they require using a different syntax and a different semantics. For instance, strictness annotations as in CLEAN are not supported by any existing proof assistant. On the other hand, these well established proof assistants offer features that are not available in SPARKLE. Most notably, the tactic language and the logic are much richer than in SPARKLE.

At Chalmers University of Technology, the proof assistant AGDA [1] has been developed in the context of the COVER [9] project. AGDA is dedicated to the lazy functional language HASKELL [20]. As in SPARKLE, the program is translated to a core-version on which the proofs are performed. Being geared towards facilitating the 'average' functional programmer, SPARKLE offers dedicated tactics and a dedicated semantics based on graph rewriting. AGDA uses standard constructive type theory on λ-terms, enabling independent proof checking.

Also as part of the COVER project, it is argued in [10] that "loose reasoning" is "morally correct", i.e. that the correctness of a theorem under the assumption that every subexpression is strict and terminating implies the correctness of the theorem in the lazy case under certain additional conditions. The conditions that are found in this way, however, may be too restrictive for the lazy case. SPARKLE offers good support for reasoning with definedness conditions directly.

Another proof assistant dedicated to HASKELL is ERA [34], which stands for Equational Reasoning Assistant. This proof assistant builds on earlier work initiated by Andy Gill [15]. It is intended to be used for equational reasoning, and not for theorem proving in general. Additional proving methods, such as induction or logical steps, are not supported. ERA is a stand-alone application. Unfortunately, it seems that work on this project has been discontinued for a while. Recently, Andy Gill took up the project again, producing a version with an Ajax based interface, under the name of HERA [16], short for HASKELL Equational Reasoning Assistant.

In [22], a description is given of an automated proof tool which is dedicated to HASKELL. It supports a subset of HASKELL, and needs no guidance of users in the proving process. Induction is only applied when the corresponding quantor has been marked explicitly in advance. The user, however, cannot further influence the proving process at all, and cannot suggest tactics to help the prover in constructing the proof.

Another proof assistant that is dedicated to a functional language is EVT [23], the Erlang Verification Tool. However, ERLANG differs from CLEAN, because it is a strict, untyped language which is mainly used for developing distributed applications. EVT has been applied in practice to larger examples.

The PROGRAMATICA project of the Pacific Software Research Center in Oregon (www.cse.ogi.edu/PacSoft/projects/programatica) is another project that aims to integrate programming and reasoning. They intend to support a wide range of validation techniques for programs written in different languages. For functional languages they use P-logic, which is based on a modal μ-calculus in which undefinedness can also be expressed. In the PROGRAMATICA project, properties are mixed with the HASKELL source.

Properties about functional programs are proved by hand in many textbooks, for instance in [4]. Also, several articles (for instance [8]) 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.

7 Conclusions

In this paper, we have presented a thorough description of the dedicated proof assistant SPARKLE, which is integrated in the distribution of the lazy functional programming language CLEAN. We have introduced SPARKLE in detail, both on the theoretical and on the practical level. On the theoretical level, we have explained the process of formal reasoning in general, and Sparkle's dedicated support for it in specific. On the practical level, we have provided an extensive tutorial of the actual use of Sparkle.

The tutorial not only covers the fundamental functionality of Sparkle, but also explains several of its advanced features that are specific for reasoning about lazy functional programs. Assignments are included at various points in the tutorial; they allow useful hands-on experience with Sparkle to be obtained in a guided way. After completion of the tutorial, anyone with a basic understanding of functional programming will be able to make effective use of Sparkle in practice, and will be able to prove small to medium properties with little effort.

Furthermore, we also hope to have sparked an interest in making use of formal reasoning to show important properties of functional programs. With the right tool support, this is already feasible for many smaller examples, and provides an enjoyable challenge for bigger programs too!

References

1. Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Leijen, D. (ed.) Proceedings of the ACM SIGPLAN 2005 Haskell Workshop, Tallinn, Estonia, pp. 62–74. ACM Press, New York (2005)
2. Barendregt, H.P., van Eekelen, M.C.J.D., Glauert, J.R.W., Kennaway, R., Plasmeijer, M.J., Sleep, M.R.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
3. Barendsen, E., Smetsers, S.: Graph rewriting aspects of functional programming. In: Handbook of Graph Grammars and Computing by Graph Transformation, pp. 63–102. World Scientific, Singapore (1999)
4. Bird, R.S.: Introduction to Functional Programming using Haskell, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
5. Brus, T.H., van Eekelen, M.C.J.D., van Leer, M.O., Plasmeijer, M.J.: Clean: A language for functional graph writing. In: Proceedings of the Functional Programming Languages and Computer Architecture, pp. 364–384. Springer, Heidelberg (1987)
6. Burn, G.L.: Evaluation transformers a model for the parallel evolution of functional languages. In: Proc. of a conference on Functional programming languages and computer architecture, pp. 446–470. Springer, Heidelberg (1987)
7. Butterfield, A., Strong, G.: Proving Correctness of Programs with I/O - a paradigm comparison. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 72–88. Springer, Heidelberg (2002)
8. Butterfield, A., Strong, G.: Proving correctness of programs with io - a paradigm comparison. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 72–87. Springer, Heidelberg (2002)
9. Coquand, T., Dybjer, P., Hughes, J., Sheeran, M.: Combining verification methods in software development. Project proposal, Chalmers Institute of Techology, Sweden (December 2001)

10. Danielsson, N.A., Hughes, J., Jansson, P., Gibbons, J.: Fast and loose reasoning is morally correct. SIGPLAN Not. 41(1), 206–217 (2006)
11. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem proving for functional programmers - Sparkle: A functional theorem prover. In: Arts, T., Mohnen, M. (eds.) IFL 2002. LNCS, vol. 2312, pp. 55–72. Springer, Heidelberg (2002)
12. de Mol, M., van Eekelen, M., Plasmeijer, R.: A single-step term-graph reduction system for proof assistants. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) Proceedings of Selected and Invited Papers of Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, pp. 181–197 (2007)
13. de Mol, M., van Eekelen, M., Plasmeijer, R.: The Mathematical Foundation of the Proof Assistant Sparkle. Technical Report ICIS–R07025, Radboud University Nijmegen (November 2007)
14. Dowse, M., Butterfield, A., van Eekelen, M.C.J.D.: Reasoning About Deterministic Concurrent Functional I/O. In: Grelck, C., Huch, F., Michaelson, G., Trinder, P.W. (eds.) IFL 2004. LNCS, vol. 3474, pp. 177–194. Springer, Heidelberg (2004)
15. Gill, A.: The technology behind a graphical user interface for an equational reasoning assistant. In: Turner, D.N. (ed.) Functional Programming. Workshops in Computing, p. 4. Springer, Heidelberg (1995)
16. Gill, A.: Introducing the haskell equational reasoning assistant. In: Haskell 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, pp. 108–109. ACM Press, New York (2006)
17. Gordon, A.: Bisimilarity as a theory of functional programming. In: Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier Science B.V, Amsterdam (1995)
18. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF. LNCS, vol. 78. Springer, Berlin (1979)
19. Horváth, Z., Kozsik, T., Tejfel, M.: Proving invariants of functional programs. In: Kilpeläinen, P., Päivinen, N. (eds.) SPLST. Department of Computer Science, pp. 115–126. University of Kuopio (2003)
20. Hudak, P., Jones, S.L.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J.H., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R.B., Nikhil, R.S., Partain, W., Peterson, J.: Report on the Programming Language Haskell, A Non-strict, Purely Functional Language. SIGPLAN Notices 27(5), R1–R164 (1992)
21. van Kesteren, R., van Eekelen, M., de Mol, M.: Proof support for general type classes. In: Loidl, H.-W. (ed.) Trends in Functional Programming 5: Selected papers from the Fifth International Symposium on Trends in Functional Programming, TFP 2004, München, Germany, Intellect, pp. 1–16 (2004)
22. Mintchev, S.: Mechanized reasoning about functional programs. In: Hammond, K., Turner, D., Sansom, P. (eds.) Proceedings of the Glasgow Functional Progamming Workshop, pp. 151–167. Springer, Heidelberg (1994)
23. Noll, T., Fredlund, L., Gurov, D.: The evt erlang verification tool. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 582–585. Springer, Heidelberg (2001)
24. Owre, S., Shankar, N., Rushby, J., Stringer-Calvert, D.: PVS Language Reference (version 2.4) (2001), `http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf`
25. Paulson, L.C.: Logic and Computation. Cambridge University Press, Cambridge (1987)
26. Paulson, L.C.: The Isabelle Reference Manual. University of Cambridge (2007), `http://isabelle.in.tum.de/doc/ref.pdf`

27. Plasmeijer, R., van Eekelen, M.: Functional Programming and Parallel Graph Rewriting. Addison-Wesley Publishing Company, Reading (1993)
28. Plasmeijer, R., van Eekelen, M.: Keep it clean: a unique approach to functional programming. SIGPLAN Not. 34(6), 23–31 (1999)
29. Plasmeijer, R., van Eekelen, M.: Concurrent CLEAN Language Report (version 2.0) (December 2001), `http://www.cs.ru.nl/~clean/`
30. Tejfel, M., Horváth, Z., Kozsik, T.: Extending the sparkle core language with object abstraction. Acta Cybern. 17(2) (2006)
31. The Coq Development Team. The Coq Proof Assistant Reference Manual (version 7.3). Inria (2002), `http://pauillac.inria.fr/cdrom/www/coq/doc/main.html`
32. Trinder, P.W., Hammond, K., Loidl, H.-W., Jones, S.L.P.: Algorithm + strategy = parallelism. J. Funct. Program. 8(1), 23–60 (1998)
33. van Eekelen, M., de Mol, M.: Proof tool support for explicit strictness. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 37–54. Springer, Heidelberg (2006)
34. Winstanley, N.: Era User Manual, version 2.0. University of Glasgow (March 1998), `http://www.dcs.gla.ac.uk/~nww/Era/Era.html`

A Appendix: Short Description of all Sparkle Tactics

This appendix provides a short description of the tactics that can be used in Sparkle proofs. In total, Sparkle makes a library of 39 tactics available. In the description below, each tactic is briefly categorized as follows:

Equivalence/Strengthening - an equivalence tactic creates new goals that are logically equivalent to the original goal; a strengthening tactic creates goals that are logically stronger.

Forwards/Backwards - a forwards tactic brings hypotheses closer to the current goal; a backwards tactic brings the current goal closer to the hypotheses.

Instantaneous - an instantaneous tactic proves a goal in one single step (and will not be categorized as equivalence/strengthening or forwards/backwards).

Programming/Logic - a programming tactic is based on the semantics of Clean; a logic tactic is based on the semantics of the logical connectives.

Besides the type, for each tactic some *information* about its inner working is stated, and a small *example* is given of its use.

```
Absurd <Hyp1> <Hyp2>.
```

Type: Instantaneous; logic.
Info: Proves a goal that contains contradictory (absurd) hypotheses.
Details: Hypotheses are contradictory if they are each other's exact negation.
Example: $p, \langle \mathsf{H1}{:}\neg(p = 12)\rangle, \langle \mathsf{H2}{:}p = 12\rangle \vdash \texttt{FALSE}$
◀`Absurd H1 H2.`▶
Q.E.D.

```
AbsurdEquality <Hyp>.
```

Type: Instantaneous; programming.
Info: Proves a goal that contains a hypothesis stating an absurd equality.

Details: An equality between two different basic values is absurd, as well as an equality between applications of different lazy constructors.
Example: $\langle \mathsf{H1}{:}\mathtt{True} = \mathtt{False}\rangle \vdash \mathtt{FALSE}$
◀`AbsurdEquality H1.`▶
Q.E.D.
Notes: `True` and `False` are constructors; `FALSE` is a constant proposition.

`Apply <Fact>.`

Type: Usually strengthening, depends on fact; backwards; logic.
Info: Applies a fact to the current goal.
Details: A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$. It is only valid if $r_1 \ldots r_n$ can be found such that $Q[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ equals the current goal. If this is the case, then the current goal is replaced with the conjunction $P_1[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}] \wedge \ldots \wedge P_m[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$.
Example: $p, \langle \mathsf{H1}{:}\forall_x \forall_y \forall_z.x > 0 \rightarrow y < z \rightarrow x + y < x + z\rangle \vdash 7 + p < 7 + 12$
◀`Apply H1.`▶
$p, \langle \mathsf{H1}{:}\forall_x \forall_y \forall_z.x > 0 \rightarrow y < z \rightarrow x + y < x + z\rangle \vdash 7 > 0 \wedge p < 12$
Notes: This tactic can also be applied in a forwards manner. In that case, P_1 must match on a hypothesis R, which is then replaced by $P_2 \rightarrow \ldots P_n \rightarrow Q$.

`Assume <Prop>.`

Type: Equivalence; forwards; logic.
Info: Assumes the validity of a manually stated proposition.
Details: Two goals are created: one with the assumption as new hypothesis, and one with the hypothesis as goal itself.
Example: $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle \vdash R$
◀`Assume` $P \vee \neg P$.▶
(1) $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle, \langle \mathsf{H3}{:}P \vee \neg P\rangle \vdash R$
(2) $P, Q, R, \langle \mathsf{H1}{:}P \rightarrow R\rangle, \langle \mathsf{H2}{:}\neg P \rightarrow R\rangle \vdash P \vee \neg P$
Notes: A name for the new hypothesis is generated automatically.

`Case <Hyp>.`

Type: Equivalence; backwards; logic.
Info: Breaks down an introduced disjunction.
Details: The hypothesis must be of the form $P \vee Q$. Two goals are created: one in which the hypothesis is replaced by P, and one in which it is replaced by Q.
Example: $P, Q, \langle \mathsf{H1}{:}P \vee \neg P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$
◀`Case H1.`▶
(1) $P, Q, \langle \mathsf{H1}{:}P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$
(2) $P, Q, \langle \mathsf{H1}{:}\neg P\rangle, \langle \mathsf{H2}{:}P \rightarrow Q\rangle, \langle \mathsf{H3}{:}\neg P \rightarrow Q\rangle \vdash Q$

`Cases <Expr>.`

Type: Equivalence; programming.
Info: Performs a case distinction on a given expression.
Details: The expression must be of an algebraic type. New goals are created

for each of its constructors, and one for $\bot$ as well. Each new goal is obtained by replacing *all* occurrences (also in the hypotheses) of the indicated expression with a generic application of the constructor.
Example: $xs, ys, \langle \mathsf{H1}{:}\texttt{length}\ (xs\ \texttt{++}\ ys) > 0\rangle \vdash \neg(xs\ \texttt{++}\ ys = \texttt{[]})$
◀`Cases (xs ++ ys).`▶
(1) $\langle \mathsf{H1}{:}\texttt{length}\ \bot > 0\rangle \vdash \neg(\bot = \texttt{[]})$
(2) $\langle \mathsf{H1}{:}\texttt{length}\ \texttt{[]} > 0\rangle \vdash \neg(\texttt{[]} = \texttt{[]})$
(3) $x_1, x_2, \langle \mathsf{H1}{:}\texttt{length}\ \texttt{[}x_1{:}x_2\texttt{]} > 0\rangle \vdash \neg(\texttt{[}x_1{:}x_2\texttt{]} = \texttt{[]})$
Notes: Names for the newly introduced variables are generated automatically.

`ChooseCase.`

Type: Equivalence; programming.
Info: Simplifies a case distinction in which only one pattern is valid.
Details: The goal must be of the form $E_1 = E_2$, where E_1 is a case distinction and E_2 is a basic value. A pattern is valid if its result is not statically unequal to E_2. The tactic succeeds only if there is exactly one valid pattern. The case is then simplified to the result of the single valid pattern, and its condition is introduced as a conjunction in the goal.
Example: $n \vdash \texttt{case}\ n\ \texttt{of}\ (7 \mapsto 13; 13 \mapsto 7; n \mapsto 11) = 13$
◀`ChooseCase.`▶
$n \vdash n = 7 \wedge 13 = 13$

`Compare <Expr1> with <Expr2>.`

Type: Equivalence; backwards; logic.
Info: Distinguishes between the possible compare results of two expressions.
Details: The expressions must both be of type `Int`. Five new goals are created; one for $E_1 = \bot$, one for $E_2 = \bot$, one for $E_1 < E_2$, one for $E_1 = E_2$ (provided that E_1 and E_2 are not $\bot$), and one for $E_2 < E_1$.
Example: $m, n \vdash \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$
◀`Compare m with n.`▶
(1) $m, n \vdash m = \bot \rightarrow \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$
(2) $m, n \vdash n = \bot \rightarrow \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$
(3) $m, n \vdash m < n \rightarrow \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$
(4) $m, n \vdash \neg(m = \bot) \rightarrow \neg(n = \bot) \rightarrow m = n \rightarrow \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$
(5) $m, n \vdash n < m \rightarrow \texttt{min}\ m\ n \leq \texttt{max}\ m\ n$

`Contradiction.`

Type: Equivalence; backwards; logic.
Info: Builds a proof by contradiction.
Details: 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: $P, \langle \mathsf{H1}{:}P \rightarrow \texttt{FALSE}\rangle \vdash \neg P$
◀`Contradiction.`▶
$P, \langle \mathsf{H1}{:}P \rightarrow \texttt{FALSE}\rangle, \langle \mathsf{H2}{:}P\rangle \vdash \texttt{FALSE}$
Notes: A name for the new hypothesis is generated automatically. This tactic

can also be applied in a forwards manner on a hypothesis. In that case, the negation of the hypothesis simply becomes the new goal to prove.

`Cut <Fact>.`

Type: Equivalence; backwards; logic.
Info: Duplicates a fact.
Details: A fact is either an earlier proved theorem or an introduced hypothesis. It is added to the to prove by means of a new implication.
Example: $\langle \mathsf{H1}{:}\forall_P.P \lor \neg P\rangle \vdash \texttt{FALSE}$
◀`Cut H1.`▶
$\langle \mathsf{H1}{:}\forall_P.P \lor \neg P\rangle \vdash (\forall_P.P \lor \neg P) \rightarrow \texttt{FALSE}$

`Definedness.`

Type: Instantaneous; logic.
Info: Uses contradictory definedness information to prove a goal.
Details: Two sets of expressions are determined: (1) those that are statically known to be *equal* to $\bot$; (2) those that are statically known to be *unequal* 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. If an overlap between the two sets is found, the goal is proved immediately.
Example: $xs, ys, zs, \langle \mathsf{H1}{:}xs = \bot\rangle, \langle \mathsf{H2}{:}xs \texttt{ ++ } ys = \texttt{[}1 : zs\texttt{]}\rangle \vdash \texttt{FALSE}$
◀`Definedness.`▶
Q.E.D.
Notes: In the example, $xs = \bot$ due to H1, and $\neg(xs = \bot)$ due to the strictness of `++` and the definedness of the result of $xs \texttt{ ++ } ys$ by means of H2.

`Discard <Hyp>.`

Type: Strengthening; logic.
Info: Deletes an introduced hypothesis.
Example: $x, xs, \langle \mathsf{H1}{:}\texttt{reverse [] = []}\rangle \vdash \texttt{reverse } [x : xs] = \texttt{reverse } xs \texttt{ ++[}x\texttt{]}$
◀`Discard H1.`▶
$x, xs \vdash \texttt{reverse } [x : xs] = \texttt{reverse } xs \texttt{ ++[}x\texttt{]}$

`Exact <Hyp>.`

Type: Instantaneous; logic.
Info: Proves a goal that is identical to an introduced hypothesis.
Example: $\langle \mathsf{H1}{:}\forall_P \forall_Q.(P \land Q) \rightarrow P\rangle \vdash \forall_P \forall_Q.(P \land Q) \rightarrow P$
◀`Exact H1.`▶
Q.E.D.

`ExFalso <Hyp>.`

Type: Instantaneous; logic.
Info: Proves a goal that contains a hypothesis stating `FALSE`.

Example: $\langle \texttt{H1:FALSE} \rangle \vdash 5 = 6$
◀`ExFalso H1.`▶
Q.E.D.

`Extensionality <Name>.`

Type: Equivalence; backwards; logic.
Info: Proves equality of functions by means of extensionality.
Details: The current goal must of the form $E_1 = E_2$, and both E_1 and E_2 must be functions. The goal is then replaced with $\forall_{Name}.(E_1\ Name) = (E_2\ Name)$.
Example: $\vdash (\texttt{++ []}) = \texttt{id}$
◀`Extensionality xs.`▶
$\vdash \forall_{xs}.\texttt{[] ++ } xs = \texttt{id } xs$
Notes: To prevent proving $\bot = \lambda x.\bot$, which is not valid, additional definedness conditions are created under certain conditions.

`Generalize <Expr> to <Name>.`

Type: Strengthening; backwards; logic.
Info: Generalizes an arbitrary subexpression.
Details: In the to prove, replaces all occurrences of the indicated expression with the variable *Name*. Then, adds the quantor $\forall_{Name}$ in front of it.
Example: $xs \vdash (\texttt{reverse } xs) \texttt{ ++ [] } = \texttt{reverse } xs$
◀`Generalize (reverse xs) to ys.`▶
$\vdash \forall_{ys}.ys \texttt{ ++ [] } = ys$

`Induction <Var>.`

Type: Strengthening; backwards; programming.
Info: Performs structural induction on a variable
Details: The type of the indicated variable must be `Int`, `Bool` or algebraic. A goal is created for each root normal form(RNF) the variable may have, which includes $\bot$. The RNFs of an algebraic type are determined by its constructors. In each created goal, the variable is replaced by its corresponding RNF. Universal quantors are created for new variables. Additionally, induction hypotheses are added (as implications) for all recursive variables.
Example: $\vdash \forall_{xs}.xs \texttt{ ++ [] } = xs$
◀`Induction xs.`▶
(1) $\vdash \bot \texttt{ ++ [] } = \bot$
(2) $\vdash \texttt{[]++ [] } = \texttt{[]}$
(3) $\vdash \forall_x \forall_{xs}.(xs \texttt{ ++ [] } = xs) \rightarrow \texttt{[}x\texttt{:}xs\texttt{] ++ [] } = \texttt{[}x\texttt{:}xs\texttt{]}$

`Injective.`

Type: Strengthening; backwards; logic.
Info: Proves equality of applications by making use of injectivity.
Details: Replaces a goal of the form $(S\ E_1 \ldots E_n) = (S\ E'_1 \ldots E'_n)$, where S is either a function or a constructor, with the conjunction $E_1 = E'_1 \wedge \ldots \wedge E_n = E'_n$.

Example: $xs, ys \vdash xs$ `++` `[]` $= xs$ `++` ys

◀`Injective.`▶

$xs, ys \vdash xs = xs \wedge$ `[]` $= ys$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

```
IntArith.
```

Type: Equivalence; backwards; logic.

Info: Built-in simplification of arithmetic expressions.

Details: This tactic operates on expressions containing applications of $+$, $-$ and $*$ on integers. It performs three simplifications: (1) $a * (b + c)$ is replaced with $a * b + a * c$; (2) constants are moved to the right as much as possible; and (3) computations on constants are carried out statically.

Example: $x, y \vdash 3 + 7 * (12 + x) - 100 = y$

◀`IntArith.`▶

$x, y \vdash 7 * x - 13 = y$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

```
IntCompare.
```

Type: Instantaneous; logic.

Info: Proves goals with contradictory integer comparisons.

Details: Only hypotheses of the exact form $x < y$ are used as input. If a chain $x < y < \ldots < x$ can be found, then the goal is proved immediately.

Example: $x, y, z, \langle \mathsf{H1}{:}y < x\rangle, \langle \mathsf{H2}{:}z < y\rangle, \langle \mathsf{H3}{:}x < z\rangle \vdash$ `FALSE`

◀`IntCompare.`▶

Q.E.D.

```
Introduce <Name1> <Name2> ... <Namen>.
```

Type: Equivalence; backwards; logic.

Info: Introduces universally quantified variables and hypotheses in the goal.

Details: The current goal must be of the form $\forall_{x_1 \ldots x_a}.P_1 \rightarrow \ldots P_b \rightarrow Q$, 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: $\vdash \forall_x.(x = 7 \rightarrow \forall_y.(y = 7 \rightarrow x = y))$

◀`Introduce p H1 q H2.`▶

$p, q, \langle \mathsf{H1}{:}p = 7\rangle, \langle \mathsf{H2}{:}q = 7\rangle \vdash p = q$

```
MoveQuantors <Dir>.
```

Type: Equivalence; backwards; logic.

Info: Swaps implications and universal quantifications.

Details: The direction argument is either 'In' or 'Out'. When moving inwards, goals of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$ are transformed to $P_1 \rightarrow \ldots P_m \rightarrow \forall_{x_1 \ldots x_n}.Q$, provided that none of the x_i occur in any of the P_j. The outwards move is the opposite of the inwards move.

Example: $R \vdash \forall_P \forall_Q . R \to \neg R \to P \wedge Q$
◀`MoveQuantors In.`▶
$R \vdash R \to \neg R \to \forall_P \forall_Q . P \wedge Q$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

```
Opaque <Fun>.
```

Type: Special.
Info: Marks a function as non-expandable.
Details: When a function is marked opaque, it will not be expanded by the reduction mechanism. Instead, reduction will stop.
Example: $\vdash$ `zip` $($`[]`$,$ `[]`$) =$ `[]`
◀`Opaque zip2; Reduce NF All.`▶
$\vdash$ `zip2 [] []` $=$ `[]`

```
Reduce NF All.
```

Type: Equivalence; backwards; programming.
Info: Reduces all expressions in the current goal to normal form.
Details: All redexes in the current goal are replaced by their reducts. This full reduction is accomplished by first using standard reduction to root normal form, and then continuing recursively on the top-level arguments.
Example: $\vdash$ `reverse` `[`$7 * 12, 100 - 12$`]` $=$ `[`$89 - 1, 83 + 1$`]`
◀`Reduce NF All.`▶
$\vdash$ `[`$88, 84$`]` $=$ `[`$88, 84$`]`
Notes(1): An artificial limit is imposed on the maximum number of reduction steps in order to safely handle non-terminating reductions.
Notes(2): This tactic can also be configured to reduce n steps; or to reduce to root normal form; or to reduce a specific redex; or to reduce within a hypothesis.

```
RefineUndefinedness.
```

Type: Equivalence; backwards; logic.
Info: Refines undefinedness equalities.
Details: Attempts to refine all undefinedness equalities in the current goal of the form $(S\ E_1 \ldots E_n) = \perp$, where S is either a constructor or a halting function. Replaces the equality with the disjunction of all $E_i = \perp$ where E_i is on a strict position and not statically known to be defined.
Example: $x, y \vdash (x + y) - 13 = \perp$
◀`RefineUndefinedness.`▶
$x, y \vdash (x + y) = \perp$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

```
Reflexive.
```

Type: Instantaneous; logic.
Info: Utilizes the reflexivity of the built-in operators $=$ and $\leftrightarrow$.
Details: Immediately proves any goal of the form $\forall_{x_1 \ldots x_n} . P_1 \to \ldots P_m \to Q$, where Q is either $E = E$ or $P \leftrightarrow P$.

Example: $\vdash \forall_x \exists_y . x < y \rightarrow x + y = x + y$
◀`Reflexive.`▶
Q.E.D.

`Rename <Name1> to <Name2>.`

Type: Special.
Info: Renames an introduced variable or an introduced hypothesis.
Example: $x, y \vdash x < y \rightarrow \neg(x = y)$
◀`Rename x to z.`▶
$z, y \vdash z < y \rightarrow \neg(z = y)$

`Rewrite <fact>.`

Type: Usually strengthening, depends on fact; backwards; logic.
Info: Rewrites the current goal using an equality in a fact.
Details: A fact is either an earlier proved theorem or an introduced hypothesis, and must be of the form $\forall_{x_1 \ldots x_n} . P_1 \rightarrow \ldots P_m \rightarrow Q$, where Q is either $L = R$ or $L \leftrightarrow R$. It is only valid if $r_1 \ldots r_n$ can be found such that $L[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ occurs within the to prove. If this is the case, then all occurrences of $L[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$ are replaced with $R[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$. Furthermore, goals are created for each condition of the fact; the i-th states $P_i[\overrightarrow{x_i} \mapsto \overrightarrow{r_i}]$.
Example: $p, \langle \mathsf{H1}{:}\forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash (p - 7) * 0 = 0$
◀`Rewrite H1.`▶
(1) $p, \langle \mathsf{H1}{:}\forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash 0 = 0$
(2) $p, \langle \mathsf{H1}{:}\forall_x . \neg(x = \perp) \rightarrow x * 0 = 0 \rangle \vdash \neg(p - 7) = \perp$
Notes: This tactic can also be configured to rewrite from right to left; or to rewrite at one specific location only; or to rewrite within a hypothesis.

`Specialize <Hyp> with <Expr>/<Prop>.`

Type: Strengthening; forwards; logic.
Info: Specializes a universally quantified hypothesis.
Details: The hypothesis must be $\forall_x . P$, and the given expression/proposition r must have the same type as x. Then, the hypothesis is replaced with $P[x \mapsto r]$.
Example: $x, y, z, \langle \mathsf{H1}{:}x < y \rangle, \langle \mathsf{H2}{:}y < z \rangle, \langle \mathsf{H3}{:}\forall_a . x < a \rightarrow a < z \rightarrow x < z \rangle \vdash x < z$
◀`Specialize H3 with y.`▶
$x, y, z, \langle \mathsf{H1}{:}x < y \rangle, \langle \mathsf{H2}{:}y < z \rangle, \langle \mathsf{H3}{:}x < y \rightarrow y < z \rightarrow x < z \rangle \vdash x < z$

`Split.`

Type: Equivalence; backwards; logic.
Info: Splits a conjunction into separate goals.
Example: $P, Q, \langle \mathsf{H1}{:}P \rangle, \langle \mathsf{H2}{:}Q \rangle \vdash P \wedge Q$
◀`Split.`▶
(1) $P, Q, \langle \mathsf{H1}{:}P \rangle, \langle \mathsf{H2}{:}Q \rangle \vdash P$
(2) $P, Q, \langle \mathsf{H1}{:}P \rangle, \langle \mathsf{H2}{:}Q \rangle \vdash Q$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`SplitCase <Num>.`

Type: Strengthening; backwards; programming.
Info: Splits a case expression into its alternatives.
Details: The case expression that will be split is indicated by means of an index (cases are numbered from left to right starting with 1). A new goal is created for each of the alternatives of the case, including one for ⊥ and one for the default. In each goal, the case expression is replaced by the result of the alternative. Hypotheses are introduced to indicate which alternative was chosen.
Example: $xs, \langle \mathsf{H1}{:}\neg(xs = \perp)\rangle \vdash \texttt{case}\ xs\ \texttt{of}\ (\texttt{[}y\,{:}\,ys\texttt{]} \mapsto y; \_ \mapsto 12) > 0$
◀`SplitCase 1.`▶
(1) $xs, \langle \mathsf{H1}{:}\neg(xs = \perp)\rangle, \langle \mathsf{H2}{:}xs = \perp\rangle \vdash \perp > 0$
(2) $xs, y, ys, \langle \mathsf{H1}{:}\neg(xs = \perp)\rangle, \langle \mathsf{H2}{:}xs = \texttt{[}y\,{:}\,ys\texttt{]}\rangle \vdash y > 0$
(3) $xs, \langle \mathsf{H1}{:}\neg(xs = \perp)\rangle, \langle \mathsf{H2}{:}xs = \texttt{[]}\rangle \vdash 12 > 0$

`SplitIff.`

Type: Equivalence; backwards; logic.
Info: Splits a ↔ into a → and a ←.
Details: The current goal must be of the form $P \leftrightarrow Q$. Two goals are created, one for with $P \rightarrow Q$ and one for $Q \rightarrow P$.
Example: $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash P \leftrightarrow Q$
◀`SplitIff.`▶
(1) $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash P \rightarrow Q$
(2) $P, Q, \langle \mathsf{H1}{:}P \rightarrow Q\rangle, \langle \mathsf{H2}{:}Q \rightarrow P\rangle \vdash Q \rightarrow P$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Symmetric.`

Type: Equivalence; backwards; logic.
Info: Utilizes the symmetry of the built-in operators = and ↔.
Details: The current goal must be of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow Q$, where Q is either $E_1 = E_2$ or $Q_1 \leftrightarrow Q_2$. If this is the case, then Q is replaced with $E_2 = E_1$ if it was a =, and with $Q_2 \leftrightarrow Q_1$ if it was a ↔.
Example: $x, \langle \mathsf{H1}{:}x = y\rangle \vdash y = x$
◀`Symmetric.`▶
$x, \langle \mathsf{H1}{:}x = y\rangle \vdash x = y$
Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Transitive <Expr>/<Prop>.`

Type: Equivalence; backwards; logic.
Info: Utilizes the transitivity of the built-in operators = and ↔.
Details: If the argument T is an expression, then the current goal must be of the form $E_1 = E_2$; if T is a proposition, then it must be of the form $P_1 \leftrightarrow P_2$. Two goals are then created, one stating $E_1 = T$ (or $P_1 \leftrightarrow T$), and the other stating $T = E_2$ (or $T \leftrightarrow P_2$).

Example: $P \vdash P \leftrightarrow ((P \wedge P) \wedge P)$

◀`Transitive` $(P \wedge P)$`.`▶

(1) $P \vdash P \leftrightarrow (P \wedge P)$

(2) $P \vdash (P \wedge P) \leftrightarrow ((P \wedge P) \wedge P)$

`Transparent.`

Type: Special.
Info: Marks a function as expandable.
Details: Undos the effect of Opaque.
Example: $\vdash$ `zip ([], []) = []`

◀`Opaque zip2; Transparent zip2; Reduce NF All.`▶

$\vdash$ `[] = []`

`Trivial.`

Type: Instantaneous; logic.
Info: Proves the trivial proposition `TRUE`.
Details: Immediately proves any goal of the form $\forall_{x_1 \ldots x_n}.P_1 \rightarrow \ldots P_m \rightarrow$ `TRUE`.
Example: $\vdash \forall_P.P \rightarrow \neg P \rightarrow$ `TRUE`

◀`Trivial.`▶

Q.E.D.

`Uncurry.`

Type: Equivalence; backwards; programming.
Info: Uncurries all applications in the current goal.
Details: Forces all curried applications $(f\ x_1 \ldots x_i)\ x_{i+1} \ldots x_n$ in the current goal to be uncurried to $f\ x_1 \ldots x_n$.
Example: $\vdash$ `[`$((+)\ 1)\ 1 :$ `map` $((+)\ 1)$ `[]] = [2]`

◀`Uncurry.`▶

$\vdash$ `[`$1 + 1 :$ `map` $((+)\ 1)$ `[]] = [2]`

Notes: This tactic can also be applied in a forwards manner on a hypothesis.

`Undo <num>.`

Type: Special.
Info: Undos the last n steps of the proof.
Details: Sparkle does not memorize the last actions of the user. Instead, n upwards steps in the *proof tree* are made.
Example: $\vdash \forall_{xs}.xs$ `++ [] = []`

◀`Induction xs; Reduce. Undo 2.`▶

$\vdash \forall_{xs}.xs$ `++ [] = []`

`Witness <Expr>/<Prop>.`

Type: Strengthening; backwards; logic.
Info: Chooses a witness for an existentially quantified goal.
Details: The current goal must be of the form $\exists_x.P$, and $P[x \mapsto T]$ (where T

is the term argument) must be welltyped. If this is the case, then the goal is replaced with $P[x \mapsto T]$.

Example: $\vdash \exists_x . x * x = x$
◀`Witness 1.`▶
$\vdash 1 * 1 = 1$

Notes: This tactic can also be applied in a forwards manner on a hypothesis.